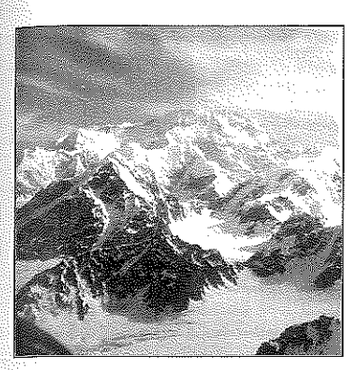
4



Requirements Elicitation

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

—Douglas Adams, in *Mostly Harmless*

A **requirement** is a feature that the system must have or a constraint that it must satisfy to be accepted by the client. **Requirements engineering** aims at defining the requirements of the system under construction. Requirements engineering includes two main activities; *requirements elicitation*, which results in the specification of the system that the client understands, and *analysis*, which results in an analysis model that the developers can unambiguously interpret. Requirements elicitation is the more challenging of the two because it requires the collaboration of several groups of participants with different backgrounds. On the one hand, the client and the users are experts in their domain and have a general idea of what the system should do, but they often have little experience in software development. On the other hand, the developers have experience in building systems, but often have little knowledge of the everyday environment of the users.

Scenarios and use cases provide tools for bridging this gap. A *scenario* describes an example of system use in terms of a series of interactions between the user and the system. A *use case* is an abstraction that describes a class of scenarios. Both scenarios and use cases are written in natural language, a form that is understandable to the user.

In this chapter, we focus on scenario-based requirements elicitation. Developers elicit requirements by observing and interviewing users. Developers first represent the user's current work processes as as-is scenarios, then develop visionary scenarios describing the functionality to be provided by the future system. The client and users validate the system description by reviewing the scenarios and by testing small prototypes provided by the developers. As the definition of the system matures and stabilizes, developers and the client agree on a requirements specification in the form of functional requirements, nonfunctional requirements, use cases, and scenarios.

4.1 Introduction: Usability Examples

Feet or miles?[a]

During a laser experiment, a laser beam was directed at a mirror on the Space Shuttle Discovery. The test called for the laser beam to be reflected back toward a mountain top. The user entered the elevation of the mountain as "10,023," assuming the units of the input were in feet. The computer interpreted the number in miles and the laser beam was reflected away from Earth, toward a hypothetical mountain 10,023 miles high.

Decimal point versus thousand separator

In the United States, decimal points are represented by a period (".") and thousand separators are represented by a comma (","). In Germany, the decimal point is represented by a comma and the thousand separator by a period. Assume a user in Germany, aware of both conventions, is viewing an online catalog with prices listed in dollars. Which convention should be used to avoid confusion?

Standard patterns

In the Emacs text editor, the command <Control-x><Control-c> exits the program. If any files need to be saved, the editor will ask the user, "Save file myDocument.txt? (y or n)". If the user answers y, the editor saves the file prior to exiting. Many users rely on this pattern and systematically type the sequence <Control-x><Control-c> followed by a "y" when exiting an editor. Other editors, however, ask when exiting the question: "Are you sure you want to exit? (y or n)". When users switch from Emacs to such an editor, they will fail to save their work until they manage to break this pattern.

a. Examples from [Nielsen, 1993] and [Neumann, 1995].

Requirements elicitation is about communication among developers, clients, and users to define a new system. Failure to communicate and understand each others' domains results in a system that is difficult to use or that simply fails to support the user's work. Errors introduced during requirements elicitation are expensive to correct, as they are usually discovered late in the process, often as late as delivery. Such errors include missing functionality that the system should have supported, functionality that was incorrectly specified, user interfaces that are misleading or unusable, and obsolete functionality. Requirements elicitation methods aim at improving communication among developers, clients, and users. Developers construct a model of the application domain by observing users in their environment. Developers select a representation that is understandable by the clients and users (e.g., scenarios and use cases). Developers validate the application domain model by constructing simple prototypes of the user interface and collecting feedback from potential users. An example of a simple prototype is the layout of a user interface with menu items and buttons. The potential user can manipulate the menu items and buttons to get a feeling for the usage of the system, but there is no actual response after buttons are clicked, because the required functionality is not implemented.

Section 4.2 provides an overview of requirements elicitation and its relationship to the other development activities. Section 4.3 defines the concepts used in this chapter. Section 4.4 discusses the activities of requirements elicitation. Section 4.5 discusses the management activities related to requirements elicitation. Section 4.6 discusses the ARENA case study.

4.2 An Overview of Requirements Elicitation

Requirements elicitation focuses on describing the purpose of the system. The client, the developers, and the users identify a problem area and define a system that addresses the problem. Such a definition is called a **requirements specification** and serves as a contract between the client and the developers. The requirements specification is structured and formalized during analysis (Chapter 5, *Analysis*) to produce an **analysis model** (see Figure 4-1). Both requirements specification and analysis model represent the same information. They differ only in the language and notation they use; the requirements specification is written in natural language, whereas the analysis model is usually expressed in a formal or semiformal notation. The requirements specification supports the communication with the client and users. The analysis model supports the communication among developers. They are both models of the system in the sense that they attempt to represent accurately the external aspects of the system. Given that both models represent the same aspects of the system, requirements elicitation and analysis occur concurrently and iteratively.

Requirements elicitation and analysis focus only on the user's view of the system. For example, the system functionality, the interaction between the user and the system, the errors that the system can detect and handle, and the environmental conditions in which the system functions are part of the requirements. The system structure, the implementation technology selected to build the system, the system design, the development methodology, and other aspects not directly visible to the user are not part of the requirements.

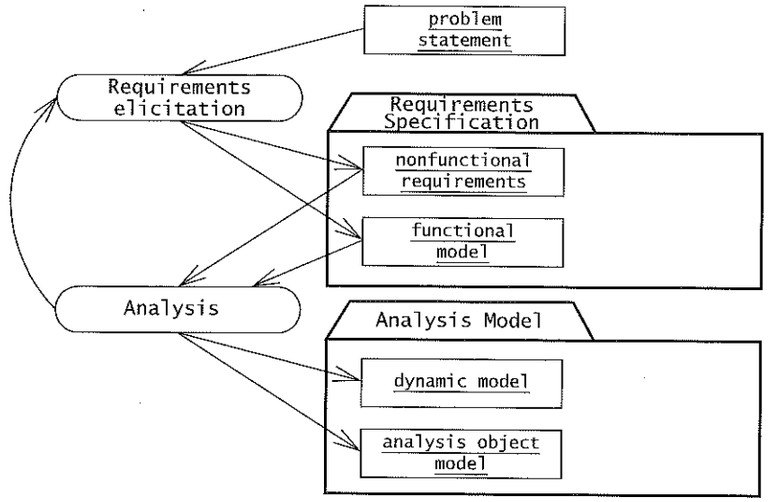


Figure 4-1 Products of requirements elicitation and analysis (UML activity diagram).

Requirements elicitation includes the following activities:

- *Identifying actors.* During this activity, developers identify the different types of users the future system will support.
- *Identifying scenarios.* During this activity, developers observe users and develop a set of detailed scenarios for typical functionality provided by the future system. Scenarios are concrete examples of the future system in use. Developers use these scenarios to communicate with the user and deepen their understanding of the application domain.
- *Identifying use cases.* Once developers and users agree on a set of scenarios, developers derive from the scenarios a set of use cases that completely represent the future system. Whereas scenarios are concrete examples illustrating a single case, use cases are abstractions describing all possible cases. When describing use cases, developers determine the scope of the system.
- *Refining use cases.* During this activity, developers ensure that the requirements specification is complete by detailing each use case and describing the behavior of the system in the presence of errors and exceptional conditions.
- *Identifying relationships among use cases.* During this activity, developers identify dependencies among use cases. They also consolidate the use case model by factoring out common functionality. This ensures that the requirements specification is consistent.
- *Identifying nonfunctional requirements.* During this activity, developers, users, and clients agree on aspects that are visible to the user, but not directly related to functionality. These include constraints on the performance of the system, its documentation, the resources it consumes, its security, and its quality.

During requirements elicitation, developers access many different sources of information, including client-supplied documents about the application domain, manuals and technical documentation of legacy systems that the future system will replace, and most important, the users and clients themselves. Developers interact the most with users and clients during requirements elicitation. We focus on two methods for eliciting information, making decisions with users and clients, and managing dependencies among requirements and other artifacts:

- **Joint Application Design (JAD)** focuses on building consensus among developers, users, and clients by jointly developing the requirements specification.[1]
- **Traceability** focuses on recording, structuring, linking, grouping, and maintaining dependencies among requirements and between requirements and other work products.

1. Note that the use of the term "design" in JAD is a misnomer: it has nothing to do with our use of the term in the subsequent chapters on system and object design.

4.3 Requirements Elicitation Concepts

In this section, we describe the main requirements elicitation concepts used in this chapter. In particular, we describe

- Functional Requirements (Section 4.3.1)
- Nonfunctional Requirements (Section 4.3.2)
- Completeness, Consistency, Clarity, and Correctness (Section 4.3.3)
- Realism, Verifiability, and Traceability (Section 4.3.4)
- Greenfield Engineering, Reengineering, and Interface Engineering (Section 4.3.5).

We describe the requirements elicitation activities in Section 4.4.

4.3.1 Functional Requirements

Functional requirements describe the interactions between the system and its environment independent of its implementation. The environment includes the user and any other external system with which the system interacts. For example, Figure 4-2 is an example of functional requirements for `SatWatch`, a watch that resets itself without user intervention:

`SatWatch` is a wrist watch that displays the time based on its current location. `SatWatch` uses GPS satellites (Global Positioning System) to determine its location and internal data structures to convert this location into a time zone.

The information stored in `SatWatch` and its accuracy measuring time is such that the watch owner never needs to reset the time. `SatWatch` adjusts the time and date displayed as the watch owner crosses time zones and political boundaries. For this reason, `SatWatch` has no buttons or controls available to the user.

`SatWatch` determines its location using GPS satellites and, as such, suffers from the same limitations as all other GPS devices (e.g., inability to determine location at certain times of the day in mountainous regions). During blackout periods, `SatWatch` assumes that it does not cross a time zone or a political boundary. `SatWatch` corrects its time zone as soon as a blackout period ends.

`SatWatch` has a two-line display showing, on the top line, the time (hour, minute, second, time zone) and on the bottom line, the date (day, date, month, year). The display technology used is such that the watch owner can see the time and date even under poor light conditions.

When political boundaries change, the watch owner may upgrade the software of the watch using the WebifyWatch device (provided with the watch) and a personal computer connected to the Internet.

Figure 4-2 Functional requirements for `SatWatch`.

The above functional requirements focus only on the possible interactions between `SatWatch` and its external world (i.e., the watch owner, GPS, and WebifyWatch). The above description does not focus on any of the implementation details (e.g., processor, language, display technology).

4.3.2 Nonfunctional Requirements

Nonfunctional requirements describe aspects of the system that are not directly related to the functional behavior of the system. Nonfunctional requirements include a broad variety of requirements that apply to many different aspects of the system, from usability to performance. The FURPS+ model[2] used by the Unified Process [Jacobson et al., 1999] provides the following categories of nonfunctional requirements:

- **Usability** is the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component. Usability requirements include, for example, conventions adopted by the user interface, the scope of online help, and the level of user documentation. Often, clients address usability issues by requiring the developer to follow user interface guidelines on color schemes, logos, and fonts.
- **Reliability** is the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Reliability requirements include, for example, an acceptable mean time to failure and the ability to detect specified faults or to withstand specified security attacks. More recently, this category is often replaced by **dependability**, which is the property of a computer system such that reliance can justifiably be placed on the service it delivers. Dependability includes reliability, **robustness** (the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions), and **safety** (a measure of the absence of catastrophic consequences to the environment).
- **Performance** requirements are concerned with quantifiable attributes of the system, such as **response time** (how quickly the system reacts to a user input), **throughput** (how much work the system can accomplish within a specified amount of time), **availability** (the degree to which a system or component is operational and accessible when required for use), and **accuracy**.
- **Supportability** requirements are concerned with the ease of changes to the system after deployment, including for example, **adaptability** (the ability to change the system to deal with additional application domain concepts), **maintainability** (the ability to change the system to deal with new technology or to fix defects), and internationalization (the ability to change the system to deal with additional international conventions, such as languages, units, and number formats). The ISO 9126 standard on software quality [ISO Std. 9126], similar to the FURPS+ model, replaces this category with two categories: **maintainability** and **portability** (the ease with which a system or component can be transferred from one hardware or software environment to another).

2. FURPS+ is an acronym using the first letter of the requirements categories: Functionality, Usability, Reliability, Performance, and Supportability. The + indicates the additional subcategories. The FURPS model was originally proposed by [Grady, 1992]. The definitions in this section are quoted from [IEEE Std. 610.12-1990].

The FURPS+ model provides additional categories of requirements typically also included under the general label of nonfunctional requirements:

- **Implementation requirements** are constraints on the implementation of the system, including the use of specific tools, programming languages, or hardware platforms.
- **Interface requirements** are constraints imposed by external systems, including legacy systems and interchange formats.
- **Operations requirements** are constraints on the administration and management of the system in the operational setting.
- **Packaging requirements** are constraints on the actual delivery of the system (e.g., constraints on the installation media for setting up the software).
- **Legal requirements** are concerned with licensing, regulation, and certification issues. An example of a legal requirement is that software developed for the U.S. federal government must comply with Section 508 of the Rehabilitation Act of 1973, requiring that government information systems must be accessible to people with disabilities.

Nonfunctional requirements that fall into the URPS categories are called **quality requirements** of the system. Nonfunctional requirements that fall into the implementation, interface, operations, packaging, and legal categories are called **constraints** or **pseudo requirements**. Budget and schedule requirements are usually not treated as nonfunctional requirements, as they constrain attributes of the projects (see Chapter 14, *Project Management*). Figure 4-3 depicts the nonfunctional requirements for `SatWatch`.

Quality requirements for `SatWatch`

- Any user who knows how to read a digital watch and understands international time zone abbreviations should be able to use `SatWatch` without the user manual. [Usability requirement]
- As the `SatWatch` has no buttons, no software faults requiring the resetting of the watch should occur. [Reliability requirement]
- `SatWatch` should display the correct time zone within 5 minutes of the end of a GPS blackout period. [Performance requirement]
- `SatWatch` should measure time within 1/100th second over 5 years. [Performance requirement]
- `SatWatch` should display time correctly in all 24 time zones. [Performance requirement]
- `SatWatch` should accept upgrades to its onboard via the Webify Watch serial interface. [Supportability requirement]

Constraints for `SatWatch`

- All related software associated with `SatWatch`, including the onboard software, will be written using Java, to comply with current company policy. [Implementation requirement]
- `SatWatch` complies with the physical, electrical, and software interfaces defined by WebifyWatch API 2.0. [Interface requirement]

Figure 4-3 Nonfunctional requirements for `SatWatch`.

4.3.3 Completeness, Consistency, Clarity, and Correctness

Requirements are continuously validated with the client and the user. Validation is a critical step in the development process, given that both the client and the developer depend on the requirements specification. Requirement validation involves checking that the specification is complete, consistent, unambiguous, and correct. It is **complete** if all possible scenarios through the system are described, including exceptional behavior (i.e., all aspects of the system are represented in the requirements model). The requirements specification is **consistent** if it does not contradict itself. The requirements specification is **unambiguous** if exactly one system is defined (i.e., it is not possible to interpret the specification two or more different ways). A specification is **correct** if it represents accurately the system that the client needs and that the developers intend to build (i.e., everything in the requirements model accurately represents an aspect of the system to the satisfaction of both client and developer). These properties are illustrated in Table 4-1.

The correctness and completeness of a requirements specification are often difficult to establish, especially before the system exists. Given that the requirements specification serves as a contractual basis between the client and the developers, the requirements specification must be

Table 4-1 Specification properties checked during validation.

Complete—All features of interest are described by requirements. *Example of incompleteness:* The `SatWatch` specification does not specify the boundary behavior when the user is standing within GPS accuracy limitations of a state's boundary. *Solution:* Add a functional requirement stating that the time depicted by `SatWatch` should not change more often than once very 5 minutes.
Consistent—No two requirements of the specification contradict each other. *Example of inconsistency:* A watch that does not contain any software faults need not provide an upgrade mechanism for downloading new versions of the software. *Solution:* Revise one of the conflicting requirements from the model (e.g., rephrase the requirement about the watch not containing any faults, as it is not verifiable anyway).
Unambiguous—A requirement cannot be interpreted in two mutually exclusive ways. *Example of ambiguity:* The `SatWatch` specification refers to time zones and political boundaries. Does the `SatWatch` deal with daylight saving time or not? *Solution:* Clarify the ambiguous concept to select one of the mutually exclusive phenomena (e.g., add a requirement that `SatWatch` should deal with daylight saving time).
Correct—The requirements describe the features of the system and environment of interest to the client and the developer, but do not describe other unintended features. *Example of fault:* There are more than 24 time zones, Several countries and territories (e.g, India) are half an hour ahead of a neighboring time zone.

carefully reviewed by both parties. Additionally, parts of the system that present a high risk should be prototyped or simulated to demonstrate their feasibility or to obtain feedback from the user. In the case of `SatWatch` described above, a mock-up of the watch would be built using a traditional watch and users surveyed to gather their initial impressions. A user may remark that she wants the watch to be able to display both American and European date formats.

4.3.4 Realism, Verifiability, and Traceability

Three more desirable properties of a requirements specification are that it be realistic, verifiable, and traceable. The requirements specification is **realistic** if the system can be implemented within constraints. The requirements specification is **verifiable** if, once the system is built, repeatable tests can be designed to demonstrate that the system fulfills the requirements specification. For example, a mean time to failure of a hundred years for `SatWatch` would be difficult to verify (assuming it is realistic in the first place). The following requirements are additional examples of nonverifiable requirements:

- *The product shall have a good user interface.*—Good is not defined.
- *The product shall be error free.*—Requires large amount of resources to establish.
- *The product shall respond to the user with 1 second for most cases.*—"Most cases" is not defined.

A requirements specification is **traceable** if each requirement can be traced throughout the software development to its corresponding system functions, and if each system function can be traced back to its corresponding set of requirements. Traceability includes also the ability to track the dependencies among requirements, system functions, and the intermediate design artifacts, including system components, classes, methods, and object attributes. Traceability is critical for developing tests and for evaluating changes. When developing tests, traceability enables a tester to assess the coverage of a test case, that is, to identify which requirements are tested and which are not. When evaluating changes, traceability enables the analyst and the developers to identify all components and system functions that the change would impact.

4.3.5 Greenfield Engineering, Reengineering, and Interface Engineering

Requirements elicitation activities can be classified into three categories, depending on the source of the requirements. In **greenfield engineering**, the development starts from scratch—no prior system exists—so the requirements are extracted from the users and the client. A greenfield engineering project is triggered by a user need or the creation of a new market. `SatWatch` is a greenfield engineering project.

A **reengineering** project is the redesign and reimplementation of an existing system triggered by technology enablers or by business processes [Hammer & Champy, 1993]. Sometimes, the functionality of the new system is extended, but the essential purpose of the

system remains the same. The requirements of the new system are extracted from an existing system.

An **interface engineering** project is the redesign of the user interface of an existing system. The legacy system is left untouched except for its interface, which is redesigned and reimplemented. This type of project is a reengineering project in which the legacy system cannot be discarded without entailing high costs.

In both reengineering and greenfield engineering, the developers need to gather as much information as possible from the application domain. This information can be found in procedures manuals, documentation distributed to new employees, the previous system's manual, glossaries, cheat sheets and notes developed by the users, and user and client interviews. Note that although interviews with users are an invaluable tool, they fail to gather the necessary information if the relevant questions are not asked. Developers must first gain a solid knowledge of the application domain before the direct approach can be used.

Next, we describe the activities of requirements elicitation.

4.4 Requirements Elicitation Activities

In this section, we describe the requirements elicitation activities. These map a problem statement (see Chapter 3, *Project Organization and Communication*) into a requirements specification that we represent as a set of actors, scenarios, and use cases (see Chapter 2, *Modeling with UML*). We discuss heuristics and methods for eliciting requirements from users and modeling the system in terms of these concepts. Requirements elicitation activities include

- Identifying Actors (Section 4.4.1)
- Identifying Scenarios (Section 4.4.2)
- Identifying Use Cases (Section 4.4.3)
- Refining Use Cases (Section 4.4.4)
- Identifying Relationships Among Actors and Use Cases (Section 4.4.5)
- Identifying Initial Analysis Objects (Section 4.4.6)
- Identifying Nonfunctional Requirements (Section 4.4.7).

The methods described in this section are adapted from OOSE [Jacobson et al., 1992], the Unified Software Development Process [Jacobson et al., 1999], and responsibility-driven design [Wirfs-Brock et al., 1990].

4.4.1 Identifying Actors

Actors represent external entities that interact with the system. An actor can be human or an external system. In the `SatWatch` example, the watch owner, the GPS satellites, and the WebifyWatch serial device are actors (see Figure 4-4). They all exchange information with the `SatWatch`. Note, however, that they all have specific interactions with `SatWatch`: the watch

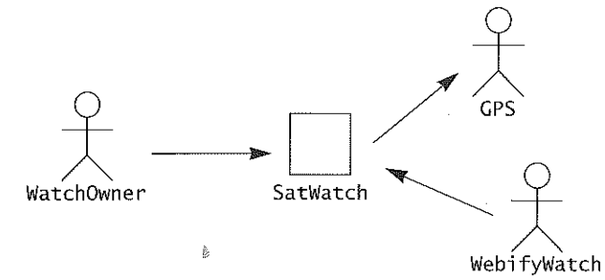


Figure 4-4 Actors for the `SatWatch` system. `WatchOwner` moves the watch (possibly across time zones) and consults it to know what time it is. `SatWatch` interacts with `GPS` to compute its position. `WebifyWatch` upgrades the data contained in the watch to reflect changes in time policy (e.g., changes in daylight savings time start and end dates).

owner wears and looks at her watch; the watch monitors the signal from the GPS satellites; the WebifyWatch downloads new data into the watch. Actors define classes of functionality.

Consider a more complex example, `FRIEND`, a distributed information system for accident management [Bruegge et al., 1994]. It includes many actors, such as `FieldOfficer`, who represents the police and fire officers who are responding to an incident, and `Dispatcher`, the police officer responsible for answering 911 calls and dispatching resources to an incident. `FRIEND` supports both actors by keeping track of incidents, resources, and task plans. It also has access to multiple databases, such as a hazardous materials database and emergency operations procedures. The `FieldOfficer` and the `Dispatcher` actors interact through different interfaces: `FieldOfficers` access `FRIEND` through a mobile personal assistant, `Dispatchers` access `FRIEND` through a workstation (see Figure 4-5).

Actors are role abstractions and do not necessarily directly map to persons. The same person can fill the role of `FieldOfficer` or `Dispatcher` at different times. However, the functionality they access is substantially different. For that reason, these two roles are modeled as two different actors.

The first step of requirements elicitation is the identification of actors. This serves both to define the boundaries of the system and to find all the perspectives from which the developers

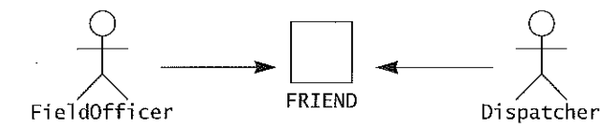


Figure 4-5 Actors of the `FRIEND` system. `FieldOfficers` not only have access to different functionality, they use different computers to access the system.

need to consider the system. When the system is deployed into an existing organization (such as a company), most actors usually exist before the system is developed: they correspond to roles in the organization.

During the initial stages of actor identification, it is hard to distinguish actors from objects. For example, a database subsystem can at times be an actor, while in other cases it can be part of the system. Note that once the system boundary is defined, there is no trouble distinguishing between actors and such system components as objects or subsystems. Actors are outside of the system boundary; they are external. Subsystems and objects are inside the system boundary; they are internal. Thus, any external software system using the system to be developed is an actor. When identifying actors, developers can ask the following questions:

Questions for identifying actors

- Which user groups are supported by the system to perform their work?
- Which user groups execute the system's main functions?
- Which user groups perform secondary functions, such as maintenance and administration?
- With what external hardware or software system will the system interact?

In the FRIEND example, these questions lead to a long list of potential actors: fire fighter, police officer, dispatcher, investigator, mayor, governor, an EPA hazardous material database, system administrator, and so on. We then need to consolidate this list into a small number of actors, who are different from the point of view of the usage of the system. For example, a fire fighter and a police officer may share the same interface to the system, as they are both involved with a single incident in the field. A dispatcher, on the other hand, manages multiple concurrent incidents and requires access to a larger amount of information. The mayor and the governor will not likely interact directly with the system, but will use the services of a trained operator instead.

Once the actors are identified, the next step in the requirements elicitation activity is to determine the functionality that will be accessible to each actor. This information can be extracted using scenarios and formalized using use cases.

4.4.2 Identifying Scenarios

A scenario is "a narrative description of what people do and experience as they try to make use of computer systems and applications" [Carroll, 1995]. A scenario is a concrete, focused, informal description of a single feature of the system from the viewpoint of a single actor. Scenarios cannot (and are not intended to) replace use cases, as they focus on specific instances and concrete events (as opposed to complete and general descriptions). However, scenarios enhance requirements elicitation by providing a tool that is understandable to users and clients.

Figure 4-6 is an example of scenario for the FRIEND system, an information system for incident response. In this scenario, a police officer reports a fire and a Dispatcher initiates the incident response. Note that this scenario is concrete, in the sense that it describes a single

Scenario name	warehouseOnFire
Participating actor instances	bob, alice:FieldOfficer john:Dispatcher
Flow of events	1. Bob, driving down main street in his patrol car, notices smoke coming out of a warehouse. His partner, Alice, activates the "Report Emergency" function from her FRIEND laptop. 2. Alice enters the address of the building, a brief description of its location (i.e., northwest corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene, given that the area appears to be relatively busy. She confirms her input and waits for an acknowledgment. 3. John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the Incident site and sends their estimated arrival time (ETA) to Alice. 4. Alice receives the acknowledgment and the ETA.

Figure 4-6 warehouseOnFire scenario for the ReportEmergency use case.

instance. It does not attempt to describe all possible situations in which a fire incident is reported. In particular, scenarios cannot contain descriptions of decisions. To describe the outcome of a decision, two scenarios would be needed, one for the "true" path, and another one for the "false" path.

Scenarios can have many different uses during requirements elicitation and during other activities of the life cycle. Below is a selected number of scenario types taken from [Carroll, 1995]:

- **As-is scenarios** describe a current situation. During reengineering, for example, the current system is understood by observing users and describing their actions as scenarios. These scenarios can then be validated for correctness and accuracy with the users.
- **Visionary scenarios** describe a future system. Visionary scenarios are used both as a point in the modeling space by developers as they refine their ideas of the future system and as a communication medium to elicit requirements from users. Visionary scenarios can be viewed as an inexpensive prototype.
- **Evaluation scenarios** describe user tasks against which the system is to be evaluated. The collaborative development of evaluation scenarios by users and developers also improves the definition of the functionality tested by these scenarios.
- **Training scenarios** are tutorials used for introducing new users to the system. These are step-by-step instructions designed to hand-hold the user through common tasks.

In requirements elicitation, developers and users write and refine a series of scenarios in order to gain a shared understanding of what the system should be. Initially, each scenario may be high level and incomplete, as the `warehouseOnFire` scenario is. The following questions can be used for identifying scenarios.

Questions for identifying scenarios

- What are the tasks that the actor wants the system to perform?
- What information does the actor access? Who creates that data? Can it be modified or removed? By whom?
- Which external changes does the actor need to inform the system about? How often? When?
- Which events does the system need to inform the actor about? With what latency?

Developers use existing documents about the application domain to answer these questions. These documents include user manuals of previous systems, procedures manuals, company standards, user notes and cheat sheets, user and client interviews. Developers should always write scenarios using application domain terms, as opposed to their own terms. As developers gain further insight into the application domain and the possibilities of the available technology, they iteratively and incrementally refine scenarios to include increasing amounts of detail. Drawing user interface mock-ups often helps to find omissions in the specification and to build a more concrete picture of the system.

In the `FRIEND` example, we identify four scenarios that span the type of tasks the system is expected to support:

- `warehouseOnFire` (Figure 4-6): A fire is detected in a warehouse; two field officers arrive at the scene and request resources.
- `fenderBender`: A car accident without casualties occurs on the highway. Police officers document the incident and manage traffic while the damaged vehicles are towed away.
- `catInATree`: A cat is stuck in a tree. A fire truck is called to retrieve the cat. Because the incident is low priority, the fire truck takes time to arrive at the scene. In the meantime, the impatient cat owner climbs the tree, falls, and breaks a leg, requiring an ambulance to be dispatched.
- `earthQuake`: An unprecedented earthquake seriously damages buildings and roads, spanning multiple incidents and triggering the activation of a statewide emergency operations plan. The governor is notified. Road damage hampers incident response.

The emphasis for developers during actor identification and scenario identification is to understand the application domain. This results in a shared understanding of the scope of the system and of the user work processes to be supported. Once developers have identified and described actors and scenarios, they formalize scenarios into use cases.

4.4.3 Identifying Use Cases

A **scenario** is an instance of a **use case**; that is, a use case specifies all possible scenarios for a given piece of functionality. A use case is initiated by an actor. After its initiation, a use case may interact with other actors, as well. A use case represents a complete flow of events through the system in the sense that it describes a series of related interactions that result from its initiation.

Figure 4-7 depicts the use case `ReportEmergency` of which the scenario `warehouseOnFire` (see Figure 4-6) is an instance. The `FieldOfficer` actor initiates this use case by activating the "Report Emergency" function of `FRIEND`. The use case completes when the `FieldOfficer` actor receives an acknowledgment that an incident has been created. The steps in the flow of events are indented to denote who initiates the step. Steps 1 and 3 are initiated by the actor, while steps

Use case name	`ReportEmergency`
Participating actors	Initiated by `FieldOfficer` Communicates with `Dispatcher`
Flow of events	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. 2. `FRIEND` responds by presenting a form to the `FieldOfficer`. 3. The `FieldOfficer` completes the form by selecting the emergency level, type, location, and brief description of the situation. The `FieldOfficer` also describes possible responses to the emergency situation. Once the form is completed, the `FieldOfficer` submits the form. 4. `FRIEND` receives the form and notifies the `Dispatcher`. 5. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. The `Dispatcher` selects a response and acknowledges the report. 6. `FRIEND` displays the acknowledgment and the selected response to the `FieldOfficer`.
Entry condition	• The `FieldOfficer` is logged into `FRIEND`.
Exit conditions	• The `FieldOfficer` has received an acknowledgment and the selected response from the `Dispatcher`, OR • The `FieldOfficer` has received an explanation indicating why the transaction could not be processed.
Quality requirements	• The `FieldOfficer`'s report is acknowledged within 30 seconds. • The selected response arrives no later than 30 seconds after it is sent by the `Dispatcher`.

Figure 4-7 An example of a use case, `ReportEmergency`. Under `ReportEmergency`, the left column denotes actor actions, and the right column denotes system responses.

2 and 4 are initiated by the system. This use case is general and encompasses a range of scenarios. For example, the `ReportEmergency` use case could also apply to the `fenderBender` scenario. Use cases can be written at varying levels of detail as in the case of scenarios.

Generalizing scenarios and identifying the high-level use cases that the system must support enables developers to define the scope of the system. Initially, developers name use cases, attach them to the initiating actors, and provide a high-level description of the use case as in Figure 4-7. The name of a use case should be a verb phrase denoting what the actor is trying to accomplish. The verb phrase "Report Emergency" indicates that an actor is attempting to report an emergency to the system (and hence, to the `Dispatcher` actor). This use case is not called "Record Emergency" because the name should reflect the perspective of the actor, not the system. It is also not called "Attempt to Report an Emergency" because the name should reflect the goal of the use case, not the actual activity.

Attaching use cases to initiating actors enables developers to clarify the roles of the different users. Often, by focusing on who initiates each use case, developers identify new actors that have been previously overlooked.

Describing a use case entails specifying four fields. Describing the entry and exit conditions of a use case enables developers to understand the conditions under which a use case is invoked and the impact of the use case on the state of the environment and of the system. By examining the entry and exit conditions of use cases, developers can determine if there may be missing use cases. For example, if a use case requires that the emergency operations plan dealing with earthquakes should be activated, the requirements specification should also provide a use case for activating this plan. Describing the flow of events of a use case enables developers and clients to discuss the interaction between actors and system. This results in many decisions about the boundary of the system, that is, about deciding which actions are accomplished by the actor and which actions are accomplished by the system. Finally, describing the quality requirements associated with a use case enables developers to elicit nonfunctional requirements in the context of a specific functionality. In this book, we focus on these four fields to describe use cases as they describe the most essential aspects of a use case. In practice, many additional fields can be added to describe an exceptional flow of events, rules, and invariants that the use case must respect during the flow of events.

Writing use cases is a craft. An analyst learns to write better use cases with experience. Consequently, different analysts tend to develop different styles, which can make it difficult to produce a consistent requirements specification. To address the issue of learning how to write use cases and how to ensure consistency among the use cases of a requirements specification, analysts adopt a use case writing guide. Figure 4-8 is a simple writing guide adapted from [Cockburn, 2001] that can be used for novice use case writers. Figure 4-9 provides an example of a poor use case that violates the writing guideline in several ways.

The `ReportEmergency` use case in Figure 4-7 may be illustrative enough to describe how `FRIEND` supports reporting emergencies and to obtain general feedback from the user, but it does not provide sufficient detail for a requirements specification. Next, we discuss how use cases are refined and detailed.

Simple Use Case Writing Guide

- Use cases should be named with verb phrases. The name of the use case should indicate what the user is trying to accomplish (e.g., `ReportEmergency`, `OpenIncident`).
- Actors should be named with noun phrases (e.g., `FieldOfficer`, `Dispatcher`, `Victim`).
- The boundary of the system should be clear. Steps accomplished by the actor and steps accomplished by the system should be distinguished (e.g., in Figure 4-7, system actions are indented to the right).
- Use case steps in the flow of events should be phrased in the active voice. This makes it explicit who accomplished the step.
- The causal relationship between successive steps should be clear.
- A use case should describe a complete user transaction (e.g., the `ReportEmergency` use case describes all the steps between initiating the emergency reporting and receiving an acknowledgment).
- Exceptions should be described separately.
- A use case should not describe the user interface of the system. This takes away the focus from the actual steps accomplished by the user and is better addressed with visual mock-ups (e.g., the `ReportEmergency` only refers to the "Report Emergency" function, not the menu, the button, nor the actual command that corresponds to this function).
- A use case should not exceed two or three pages in length. Otherwise, use include and extend relationships to decompose it in smaller use cases, as explained in Section 4.4.5.

Figure 4-8 Example of use case writing guide.

Use case name	`Accident`	*Bad name: What is the user trying to accomplish?*
Initiating actor	Initiated by `FieldOfficer`	
Flow of events	1. The `FieldOfficer` reports the accident.	
	2. An ambulance is dispatched.	*Causality: Which action caused the `FieldOfficer` to receive an acknowledgment?*
		Passive voice: Who dispatches the ambulance?
	3. The `Dispatcher` is notified when the ambulance arrives on site.	*Incomplete transaction: What does the `FieldOfficer` do after the ambulance is dispatched?*

Figure 4-9 An example of a poor use case. Violations of the writing guide are indicated in *italics* in the right column.

4.4.4 Refining Use Cases

Figure 4-10 is a refined version of the `ReportEmergency` use case. It has been extended to include details about the type of incidents known to `FRIEND` and detailed interactions indicating how the `Dispatcher` acknowledges the `FieldOfficer`.

Use case name	`ReportEmergency`
Participating actors	Initiated by `FieldOfficer` Communicates with `Dispatcher`
Flow of events	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. 2. `FRIEND` responds by presenting a form to the officer. *The form includes an emergency type menu (general emergency, fire, transportation) and location, incident description, resource request, and hazardous material fields.* 3. The `FieldOfficer` completes the form by *specifying minimally the emergency type and description fields*. The `FieldOfficer` may also describe possible responses to the emergency situation *and request specific resources*. Once the form is completed, the `FieldOfficer` submits the form. 4. `FRIEND` receives the form and notifies the `Dispatcher` *by a pop-up dialog*. 5. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. *All the information contained in the `FieldOfficer's` form is automatically included in the `Incident`. The `Dispatcher` selects a response by allocating resources to the `Incident` (with the `AllocateResources` use case) and acknowledges the emergency report by sending a short message to the `FieldOfficer`.* 6. `FRIEND` displays the acknowledgment and the selected response to the `FieldOfficer`.
Entry condition	• ...

Figure 4-10 Refined description for the `ReportEmergency` use case. Additions emphasized in *italics*.

The use of scenarios and use cases to define the functionality of the system aims at creating requirements that are validated by the user early in the development. As the design and implementation of the system starts, the cost of changing the requirements specification and adding new unforeseen functionality increases. Although requirements change until late in the development, developers and users should strive to address most requirements issues early. This entails many changes and much validation during requirements elicitation. Note that many use cases are rewritten several times, others substantially refined, and yet others completely

dropped. To save time, much of the exploration work can be done using scenarios and user interface mock-ups.

The following heuristics can be used for writing scenarios and use cases:

Heuristics for developing scenarios and use cases

- Use scenarios to communicate with users and to validate functionality.
- First, refine a single scenario to understand the user's assumptions about the system. The user may be familiar with similar systems, in which case, adopting specific user interface conventions would make the system more usable.
- Next, define many not-very-detailed scenarios to define the scope of the system. Validate with the user.
- Use mock-ups as visual support only; user interface design should occur as a separate task after the functionality is sufficiently stable.
- Present the user with multiple and very different alternatives (as opposed to extracting a single alternative from the user). Evaluating different alternatives broadens the user's horizon. Generating different alternatives forces developers to "think outside the box."
- Detail a broad vertical slice when the scope of the system and the user preferences are well understood. Validate with the user.

The focus of this activity is on completeness and correctness. Developers identify functionality not covered by scenarios, and document it by refining use cases or writing new ones. Developers describe seldom occurring cases and exception handling as seen by the actors. Whereas the initial identification of use cases and actors focused on establishing the boundary of the system, the refinement of use cases yields increasingly more details about the features provided by the system and the constraints associated with them. In particular, the following aspects of the use cases, initially ignored, are detailed during refinement:

- The elements that are manipulated by the system are detailed. In Figure 4-10, we added details about the attributes of the emergency reporting form and the types of incidents.
- The low-level sequence of interactions between the actor and the system are specified. In Figure 4-10, we added information about how the `Dispatcher` generates an acknowledgment by selecting resources.
- Access rights (which actors can invoke which use cases) are specified.
- Missing exceptions are identified and their handling specified.
- Common functionality among use cases are factored out.

In the next section, we describe how to reorganize actors and use cases with relationships, which addresses the last three bullets points above.

4.4.5 Identifying Relationships among Actors and Use Cases

Even medium-sized systems have many use cases. Relationships among actors and use cases enable the developers and users to reduce the complexity of the model and increase its understandability. We use communication relationships between actors and use cases to describe the system in layers of functionality. We use extend relationships to separate exceptional and common flows of events. We use include relationships to reduce redundancy among use cases.

Communication relationships between actors and use cases

Communication relationships between actors and use cases represent the flow of information during the use case. The actor who initiates the use case should be distinguished from the other actors with whom the use case communicates. By specifying which actor can invoke a specific use case, we also implicitly specify which actors cannot invoke the use case. Similarly, by specifying which actors communicate with a specific use case, we specify which actors can access specific information and which cannot. Thus, by documenting initiation and communication relationships among actors and use cases, we specify access control for the system at a coarse level.

The relationships between actors and use cases are identified when use cases are identified. Figure 4-11 depicts an example of communication relationships in the case of the FRIEND system. The «initiate» stereotype denotes the initiation of the use case by an actor, and the «participate» stereotype denotes that an actor (who did not initiate the use case) communicates with the use case.

Extend relationships between use cases

A use case extends another use case if the extended use case may include the behavior of the extension under certain conditions. In the FRIEND example, assume that the connection between the FieldOfficer station and the Dispatcher station is broken while the

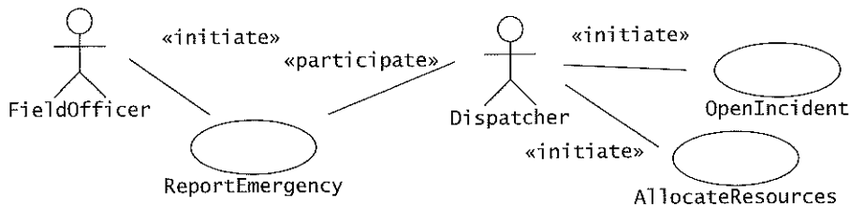


Figure 4-11 Example of communication relationships among actors and use cases in FRIEND (UML use case diagram). The FieldOfficer initiates the ReportEmergency use case, and the Dispatcher initiates the OpenIncident and AllocateResources use cases. FieldOfficers cannot directly open an incident or allocate resources on their own.

FieldOfficer is filling the form (e.g., the FieldOfficer's car enters a tunnel). The FieldOfficer station needs to notify the FieldOfficer that his form was not delivered and what measures he should take. The ConnectionDown use case is modeled as an extension of ReportEmergency (see Figure 4-12). The conditions under which the ConnectionDown use case is initiated are described in ConnectionDown as opposed to ReportEmergency. Separating exceptional and optional flows of events from the base use case has two advantages. First, the base use case becomes shorter and easier to understand. Second, the common case is distinguished from the exceptional case, which enables the developers to treat each type of functionality differently (e.g., optimize the common case for response time, optimize the exceptional case for robustness). Both the extended use case and the extensions are complete use cases of their own. They each must have entry and end conditions and be understandable by the user as an independent whole.

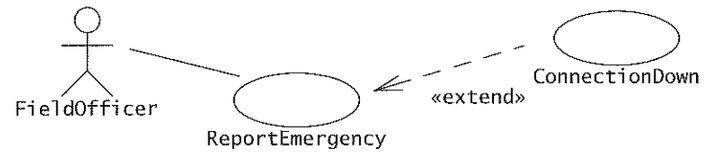


Figure 4-12 Example of use of extend relationship (UML use case diagram). ConnectionDown extends the ReportEmergency use case. The ReportEmergency use case becomes shorter and solely focused on emergency reporting.

Include relationships between use cases

Redundancies among use cases can be factored out using include relationships. Assume, for example, that a Dispatcher needs to consult the city map when opening an incident (e.g., to assess which areas are at risk during a fire) and when allocating resources (e.g., to find which resources are closest to the incident). In this case, the ViewMap use case describes the flow of events required when viewing the city map and is used by both the OpenIncident and the AllocateResources use cases (Figure 4-13).

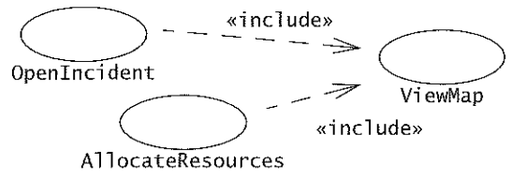


Figure 4-13 Example of include relationships among use cases. ViewMap describes the flow of events for viewing a city map (e.g., scrolling, zooming, query by street name) and is used by both OpenIncident and AllocateResources use cases.

Factoring out shared behavior from use cases has many benefits, including shorter descriptions and fewer redundancies. Behavior should *only* be factored out into a separate use case if it is shared across two or more use cases. Excessive fragmentation of the requirements specification across a large number of use cases makes the specification confusing to users and clients.

Extend versus include relationships

Include and extend are similar constructs, and initially it may not be clear to the developer when to use each one [Jacobson et al., 1992]. The main distinction between these constructs is the direction of the relationship. For include relationships, the event triggering the target (i.e., included) use case is described in the flow of event of the source use case. For extend relationships, the event triggering the source (i.e., extending) use case is described in the source use case as a precondition. In other words, for include relationships, every including use case must specify where the included use case should be invoked. For extend relationships, only the extending use case specifies which use cases are extended. Hence, a behavior that is strongly tied to an event and that occurs only in a relatively few use cases should be represented with an included relationship. These types of behavior usually include common system functions that can be used in several places (e.g., viewing a map, specifying a filename, selecting an element). Conversely, a behavior that can happen anytime or whose occurrence can be more easily specified as an entry condition should be represented with an extend relationship. These types of behavior include exceptional situations (e.g., invoking the online help, canceling a transaction, dealing with a network failure).

Figure 4-14 shows the `ConnectionDown` example described with an include relationship (left column) and with an extend relationship (right column). In the left column, we need to insert text in two places in the event flow where the `ConnectionDown` use case can be invoked. Also, if additional exceptional situations are described (e.g., a help function on the `FieldOfficer` station), the `ReportEmergency` use case will have to be modified and will become cluttered with conditions. In the right column, we need to describe only the conditions under which the exceptional use case is invoked, which can include a large number of use cases (e.g., "any use case in which the connection between the `FieldOfficer` and the `Dispatcher` is lost"). Moreover, additional exceptional situations can be added without modifying the base use case (e.g., `ReportEmergency`). The ability to extend the system without modifying existing parts is critical, as it allows us to ensure that the original behavior is left untouched. The distinction between include and extend is a documentation issue: using the correct type of relationship reduces dependencies among use cases, reduces redundancy, and lowers the probability of introducing errors when requirements change. However, the impact on other development activities is minimal.

In summary, the following heuristics can be used for selecting an extend or an include relationship.

Heuristics for extend and include relationships

- Use extend relationships for exceptional, optional, or seldom-occurring behavior. An example of seldom-occurring behavior is the breakdown of a resource (e.g., a fire truck). An example of optional behavior is the notification of nearby resources responding to an unrelated incident.
- Use include relationships for behavior that is shared across two or more use cases.
- However, use discretion when applying the above two heuristics and do not overstructure the use case model. A few longer use cases (e.g., two pages long) are easier to understand and review than many short ones (e.g., ten lines long).

In all cases, the purpose of adding include and extend relationships is to reduce or remove redundancies from the use case model, thus eliminating potential inconsistencies.

4.4.6 Identifying Initial Analysis Objects

One of the first obstacles developers and users encounter when they start collaborating with each other is differing terminology. Although developers eventually learn the users' terminology, this problem is likely to be encountered again when new developers are added to the project. Misunderstandings result from the same terms being used in different contexts and with different meanings.

To establish a clear terminology, developers identify the **participating objects** for each use case. Developers should identify, name, and describe them unambiguously and collate them into a glossary.[3] Building this glossary constitutes the first step toward analysis, which we discuss in the next chapter.

The glossary is included in the requirements specification and, later, in the user manuals. Developers keep the glossary up to date as the requirements specification evolves. The benefits of the glossary are manyfold: new developers are exposed to a consistent set of definitions, a single term is used for each concept (instead of a developer term and a user term), and each term has a precise and clear official meaning.

The identification of participating objects results in the initial analysis object model. The identification of participating objects during requirements elicitation only constitutes a first step toward the complete analysis object model. The complete analysis model is usually not used as a means of communication between users and developers, as users are often unfamiliar with object-oriented concepts. However, the description of the objects (i.e., the definitions of the terms in the glossary) and their attributes are visible to the users and reviewed. We describe in detail the further refinement of the analysis model in Chapter 5, *Analysis*.

3. The glossary is also called a "data dictionary" [Rumbaugh et al., 1991].

ReportEmergency (include relationship)	ReportEmergency (extend relationship)
1. ... 2. ... 3. The FieldOfficer completes the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form, at which point, the Dispatcher is notified. *If the connection with the Dispatcher is broken, the ConnectionDown use case is used.* 4. If the connection is still alive, the Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the emergency report. *If the connection is broken, the ConnectionDown use case is used.* 5. ...	1. ... 2. ... 3. The FieldOfficer completes the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form, at which point, the Dispatcher is notified. 4. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the emergency report. 5. ...
ConnectionDown (include relationship)	**ConnectionDown (extend relationship)**
1. The FieldOfficer and the Dispatcher are notified that the connection is broken. They are advised of the possible reasons why such an event would occur (e.g., "Is the FieldOfficer station in a tunnel?"). 2. The situation is logged by the system and recovered when the connection is reestablished. 3. The FieldOfficer and the Dispatcher enter in contact through other means and the Dispatcher initiates ReportEmergency from the Dispatcher station.	*The ConnectionDown use case extends any use case in which the communication between the FieldOfficer and the Dispatcher can be lost.* 1. The FieldOfficer and the Dispatcher are notified that the connection is broken. They are advised of the possible reasons why such an event would occur (e.g., "Is the FieldOfficer station in a tunnel?"). 2. The situation is logged by the system and recovered when the connection is reestablished. 3. The FieldOfficer and the Dispatcher enter in contact through other means and the Dispatcher initiates ReportEmergency from the Dispatcher station.

Figure 4-14 Addition of ConnectionDown exceptional condition to ReportEmergency. An extend relationship is used for exceptional and optional flow of events because it yields a more modular description.

Many heuristics have been proposed in the literature for identifying objects. Here are a selected few:

Heuristics for identifying initial analysis objects

- Terms that developers or users must clarify to understand the use case
- Recurring nouns in the use cases (e.g., Incident)
- Real-world entities that the system must track (e.g., FieldOfficer, Resource)
- Real-world processes that the system must track (e.g., EmergencyOperationsPlan)
- Use cases (e.g., ReportEmergency)
- Data sources or sinks (e.g., Printer)
- Artifacts with which the user interacts (e.g., Station)
- *Always* use application domain terms.

During requirements elicitation, participating objects are generated for each use case. If two use cases refer to the same concept, the corresponding object should be the same. If two objects share the same name and do not correspond to the same concept, one or both concepts are renamed to acknowledge and emphasize their difference. This consolidation eliminates any ambiguity in the terminology used. For example, Table 4-2 depicts the initial participating objects we identified for the ReportEmergency use case.

Table 4-2 Participating objects for the ReportEmergency use case.

Dispatcher	Police officer who manages Incidents. A Dispatcher opens, documents, and closes incidents in response to EmergencyReports and other communication with FieldOfficers. Dispatchers are identified by badge numbers.
EmergencyReport	Initial report about an Incident from a FieldOfficer to a Dispatcher. An EmergencyReport usually triggers the creation of an Incident by the Dispatcher. An EmergencyReport is composed of an emergency level, a type (fire, road accident, other), a location, and a description.
FieldOfficer	Police or fire officer on duty. A FieldOfficer can be allocated to at most one Incident at a time. FieldOfficers are identified by badge numbers.
Incident	Situation requiring attention from a FieldOfficer. An Incident may be reported in the system by a FieldOfficer or anybody else external to the system. An Incident is composed of a description, a response, a status (open, closed, documented), a location, and a number of FieldOfficers.

Once participating objects are identified and consolidated, the developers can use them as a checklist for ensuring that the set of identified use cases is complete.

Heuristics for cross-checking use cases and participating objects

- Which use cases create this object (i.e., during which use cases are the values of the object attributes entered in the system)?
- Which actors can access this information?
- Which use cases modify and destroy this object (i.e., which use cases edit or remove this information from the system)?
- Which actor can initiate these use cases?
- Is this object needed (i.e., is there at least one use case that depends on this information?)

4.4.7 Identifying Nonfunctional Requirements

Nonfunctional requirements describe aspects of the system that are not directly related to its functional behavior. Nonfunctional requirements span a number of issues, from user interface look and feel to response time requirements to security issues. Nonfunctional requirements are defined at the same time as functional requirements because they have as much impact on the development and cost of the system.

For example, consider a mosaic display that an air traffic controller uses to track planes. A mosaic display system compiles data from a series of radars and databases (hence the term "mosaic") into a summary display indicating all aircraft in a certain area, including their identification, speed, and altitude. The number of aircraft such a system can display constrains the performance of the air traffic controller and the cost of the system. If the system can only handle a few aircraft simultaneously, the system cannot be used at busy airports. On the other hand, a system able to handle a large number of aircraft is more costly and more complex to build and to test.

Nonfunctional requirements can impact the work of the user in unexpected ways. To accurately elicit all the essential nonfunctional requirements, both client and developer must collaborate so that they identify (minimally) which attributes of the system that are difficult to realize are critical for the work of the user. In the mosaic display example above, the number of aircraft that a single mosaic display must be able to handle has implications on the size of the icons used for displaying aircraft, the features for identifying aircraft and their properties, the refresh rate of the data, and so on.

The resulting set of nonfunctional requirements typically includes conflicting requirements. For example, the nonfunctional requirements of the `SatWatch` (Figure 4-3) call for an accurate mechanism, so that the time never needs to be reset, and a low unit cost, so that it is acceptable to the user to replace the watch with a new one when it breaks. These two nonfunctional requirements conflict as the unit cost of the watch increases with its accuracy. To deal with such conflicts, the client and the developer prioritize the nonfunctional requirements, so that they can be addressed consistently during the realization of the system.

Table 4-3 Example questions for eliciting nonfunctional requirements.

Category	Example questions
Usability	• What is the level of expertise of the user? • What user interface standards are familiar to the user? • What documentation should be provided to the user?
Reliability *(including robustness, safety, and security)*	• How reliable, available, and robust should the system be? • Is restarting the system acceptable in the event of a failure? • How much data can the system loose? • How should the system handle exceptions? • Are there safety requirements of the system? • Are there security requirements of the system?
Performance	• How responsive should the system be? • Are any user tasks time critical? • How many concurrent users should it support? • How large is a typical data store for comparable systems? • What is the worse latency that is acceptable to users?
Supportability *(including maintainability and portability)*	• What are the foreseen extensions to the system? • Who maintains the system? • Are there plans to port the system to different software or hardware environments?
Implementation	• Are there constraints on the hardware platform? • Are constraints imposed by the maintenance team? • Are constraints imposed by the testing team?
Interface	• Should the system interact with any existing systems? • How are data exported/imported into the system? • What standards in use by the client should be supported by the system?
Operation	• Who manages the running system?
Packaging	• Who installs the system? • How many installations are foreseen? • Are there time constraints on the installation?
Legal	• How should the system be licensed? • Are any liability issues associated with system failures? • Are any royalties or licensing fees incurred by using specific algorithms or components?

There are unfortunately few systematic methods for eliciting nonfunctional requirements. In practice, analysts use a taxonomy of nonfunctional requirements (e.g., the FURPS+ scheme described previously) to generate check lists of questions to help the client and the developers focus on the nonfunctional aspects of the system. As the actors of the system have already been identified at this point, this check list can be organized by role and distributed to representative users. The advantage of such check lists is that they can be reused and expanded for each new system in a given application domain, thus reducing the number of omissions. Note that such check lists can also result in the elicitation of additional functional requirements. For example, when asking questions about the operation of the system, the client and developers may uncover a number of use cases related with the administration of the system. Table 4-3 depicts example questions for each of the FURPS+ category.

Once the client and the developers identify a set of nonfunctional requirements, they can organize them into refinement and dependency graphs to identify further nonfunctional requirements and identify conflicts. For more material on this topic, the reader is referred to the specialized literature (e.g., [Chung et al., 1999]).

4.5 Managing Requirements Elicitation

In the previous section, we described the technical issues of modeling a system in terms of use cases. Use case modeling by itself, however, does not constitute requirements elicitation. Even after they become expert use case modelers, developers still need to elicit requirements from the users and come to an agreement with the client. In this section, we describe methods for eliciting information from the users and negotiating an agreement with a client. In particular, we describe:

- Negotiating Specifications with Clients: Joint Application Design (Section 4.5.1)
- Maintaining Traceability (Section 4.5.2)
- Documenting Requirements Elicitation (Section 4.5.3).

4.5.1 Negotiating Specifications with Clients: Joint Application Design

Joint Application Design (JAD) is a requirements method developed at IBM at the end of the 1970s. Its effectiveness lies in that the requirements elicitation work is done in one single workshop session in which all stakeholders participate. Users, clients, developers, and a trained session leader sit together in one room to present their viewpoints, listen to other viewpoints, negotiate, and come to a mutually acceptable solution. The outcome of the workshop, the final JAD document, is a complete requirements specification document that includes definitions of data elements, work flows, and interface screens. Because the final document is jointly developed by the stakeholders (that is, the participants who not only have an interest in the success of the project, but also can make substantial decisions), the final JAD document represents an agreement among users, clients, and developers, and thus minimizes requirements changes later in the development process. JAD is composed of five activities (Figure 4-15):

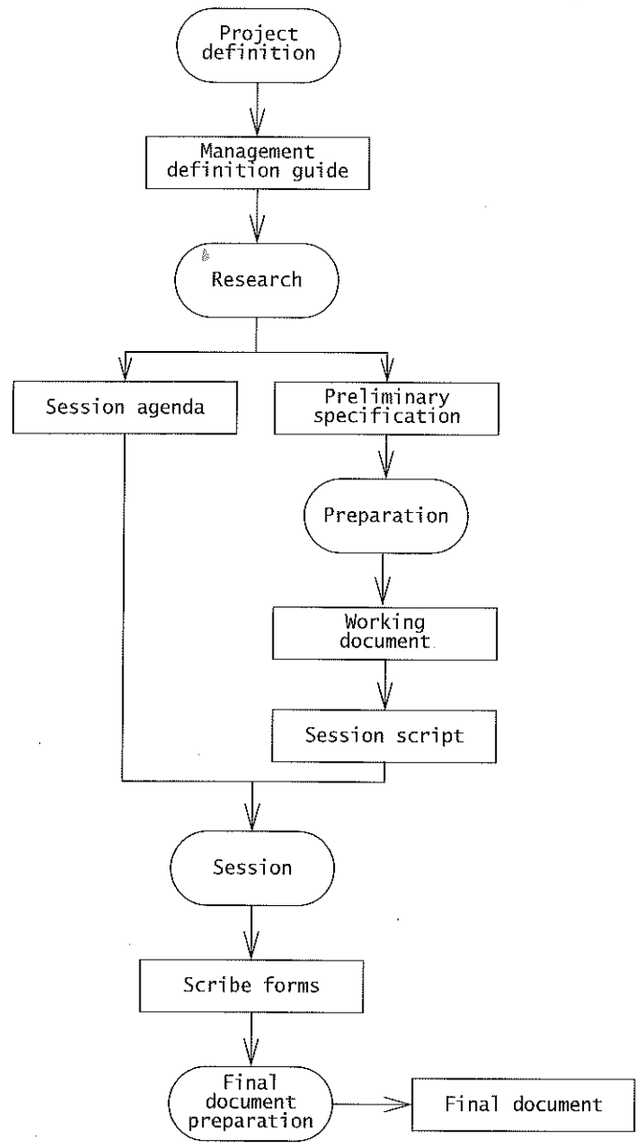


Figure 4-15 Activities of JAD (UML activity diagram). The heart of JAD is the `Session` activity during which all stakeholders design and agree to a requirements specification. The activities prior to the `Session` maximize its efficiency. The production of the final document captures the decisions made during the `Session`.

1. *Project definition.* During this activity, the JAD facilitator interviews the project manager and the client to determine the objectives and the scope of the project. The findings from the interviews are collected in the *Management Definition Guide*.
2. *Research.* During this activity, the JAD facilitator interviews present and future users, gathers information about the application domain, and describes a first set of high-level use cases. The JAD facilitator also starts a list of problems to be addressed during the session. The results of this activity are a *Session Agenda* and a *Preliminary Specification* listing work flow and system information.
3. *Preparation.* During this activity, the JAD facilitator prepares the session. The JAD facilitator creates a *Working Document*, which is the first draft of the final document, an agenda for the session, and any overhead slides or flip charts representing information gathered during the Research activity. The JAD facilitator also selects a team composed of the client, the project manager, selected users, and developers. All stakeholders are represented, and the participants are able to make binding decisions.
4. *Session.* During this activity, the JAD facilitator guides the team in creating the requirements specification. A JAD session lasts for 3 to 5 days. The team defines and agrees on the scenarios, use cases, and user interface mock-ups. All decisions are documented by a scribe.
5. *Final document.* The JAD facilitator prepares the *Final Document*, revising the working document to include all decisions made during the session. The Final Document represents a complete specification of the system agreed on during the session. The Final Document is distributed to the session participants for review. The participants then attend a 1- to 2-hour meeting to discuss the reviews and finalize the document.

JAD has been used by IBM and other companies. JAD leverages group dynamics to improve communication among participants and to accelerate consensus. At the end of a JAD session, developers are more knowledgeable of user needs, and users are more knowledgeable of development trade-offs. Additional gains result from a reduction of redesign activities downstream. Because of its reliance on social dynamics, the success of a JAD session often depends on the qualifications of the JAD facilitator as a meeting facilitator. For a detailed overview of JAD, the reader is referred to [Wood & Silver, 1989].

4.5.2 Maintaining Traceability

Traceability is the ability to follow the life of a requirement. This includes tracing where the requirements came from (e.g., who originated it, which client need does it address) to which aspects of the system and the project it affects (e.g., which components realize the requirement, which test case checks its realization). Traceability enables developers to show that the system is complete, testers to show that the system complies with its requirements, designers to record the rationale behind the system, and maintainers to assess the impact of change.

Consider the `SatWatch` system we introduced at the beginning of the chapter. Currently, the specification calls for a two-line display that includes time and date. After the client decides that the digit size is too small for comfortable reading, developers change the display requirement to a single-line display combined with a button to switch between time and date. Traceability would enable us to answer the following questions:

- Who originated the two-line display requirement?
- Did any implicit constraints mandate this requirement?
- Which components must be changed because of the additional button and display?
- Which test cases must be changed?

The simplest approach to maintaining traceability is to use cross-references among documents, models, and code artifacts. Each individual element (e.g., requirement, component, class, operation, test case) is identified by a unique number. Dependencies are then documented manually as a textual cross-reference containing the number of the source element and the number of the target element. Tool support can be as simple as a spreadsheet or a word processing tool. This approach is expensive in time and personpower, and it is error prone. However, for small projects, developers can observe benefits early.

For large-scale projects, specialized database tools enable the partial automation of the capture, editing, and linking of traceability dependencies at a more detailed level (e.g., DOORS [Telelogic] or RequisitePro [Rational]). Such tools reduce the cost of maintaining traceability, but they require the buy-in and training of most stakeholders and impose restrictions on other tools in the development process.

4.5.3 Documenting Requirements Elicitation

The results of the requirements elicitation and the analysis activities are documented in the **Requirements Analysis Document (RAD)**. This document completely describes the system in terms of functional and nonfunctional requirements. The audience for the RAD includes the client, the users, the project management, the system analysts (i.e., the developers who participate in the requirements), and the system designers (i.e., the developers who participate in the system design). The first part of the document, including use cases and nonfunctional requirements, is written during requirements elicitation. The formalization of the specification in terms of object models is written during analysis. Figure 4-16 is an example template for a RAD used in this book.

The first section of the RAD is an *Introduction*. Its purpose is to provide a brief overview of the function of the system and the reasons for its development, its scope, and references to the development context (e.g., reference to the problem statement written by the client, references to existing systems, feasibility studies). The introduction also includes the objectives and success criteria of the project.

The second section, *Current system,* describes the current state of affairs. If the new system will replace an existing system, this section describes the functionality and the problems

Requirements Analysis Document

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Scope of the system
   - 1.3 Objectives and success criteria of the project
   - 1.4 Definitions, acronyms, and abbreviations
   - 1.5 References
   - 1.6 Overview
2. Current system
3. Proposed system
   - 3.1 Overview
   - 3.2 Functional requirements
   - 3.3 Nonfunctional requirements
     - 3.3.1 Usability
     - 3.3.2 Reliability
     - 3.3.3 Performance
     - 3.3.4 Supportability
     - 3.3.5 Implementation
     - 3.3.6 Interface
     - 3.3.7 Packaging
     - 3.3.8 Legal
   - 3.4 System models
     - 3.4.1 Scenarios
     - 3.4.2 Use case model
     - *3.4.3 Object model*
     - *3.4.4 Dynamic model*
     - 3.4.5 User interface—navigational paths and screen mock-ups
4. Glossary

Figure 4-16 Outline of the Requirements Analysis Document (RAD). Sections in *italics* are completed during analysis (see next chapter).

of the current system. Otherwise, this section describes how the tasks supported by the new system are accomplished now. For example, in the case of SatWatch, the user currently resets her watch whenever she travels across a time zone. Because of the manual nature of this operation, the user occasionally sets the wrong time and occasionally neglects to reset. In contrast, the SatWatch will continually ensure accurate time within its lifetime. In the case of FRIEND, the current system is paper based: dispatchers keep track of resource assignments by filling out forms. Communication between dispatchers and field officers is by radio. The current system requires a high documentation and management cost that FRIEND aims to reduce.

The third section, *Proposed system,* documents the requirements elicitation and the analysis model of the new system. It is divided into four subsections:

- *Overview* presents a functional overview of the system.

- *Functional requirements* describes the high-level functionality of the system.
- *Nonfunctional requirements* describes user-level requirements that are not directly related to functionality. This includes usability, reliability, performance, supportability, implementation, interface, operational, packaging, and legal requirements.
- *System models* describes the scenarios, use cases, object model, and dynamic models for the system. This section contains the complete functional specification, including mock-ups illustrating the user interface of the system and navigational paths representing the sequence of screens. The subsections *Object model* and *Dynamic model* are written during the Analysis activity, described in the next chapter.

The RAD should be written after the use case model is stable, that is, when the number of modifications to the requirements is minimal. The requirements, however, are updated throughout the development process when specification problems are discovered or when the scope of the system is changed. The RAD, once published, is baselined and put under configuration management.[4] The revision history section of the RAD will provide a history of changes include the author responsible for each change, the date of the change, and a brief description of the change.

4.6 ARENA Case Study

In this section, we apply the concepts and methods described in this chapter to the ARENA system. We start with the initial problem statement provided by the client, and develop a use case model and an initial analysis object model. In previous sections, we selected examples for their illustrative value. In this section, we focus on a realistic example, describe artifacts as they are created and refined. This enables us to discuss more realistic trade-offs and design decisions and focus on operational details that are typically not visible in illustrative examples. In this discussion, "ARENA" denotes the system in general, whereas "arena" denotes a specific instantiation of the system.

4.6.1 Initial Problem Statement

After an initial meeting with the client, the problem statement is written (Figure 4-17).

Note that this brief text describes the problem and the requirements at a high level. This is not typically the stage at which we commit to a budget or a delivery date. First, we start developing the use case model by identifying actors and scenarios.

4. A **baseline** is a version of a work product that has been reviewed and formally approved. **Configuration management** is the process of tracking and approving changes to the baseline. We discuss configuration management in Chapter 13, *Configuration Management.*

ARENA Problem Statement

1. Problem

The popularity of the Internet and the World Wide Web has enabled the creation of a variety of virtual communities, groups of people sharing common interests, but who have never met each other in person. Such virtual communities can be short lived (e.g., a group of people meeting in a chat room or playing a tournament) or long lived (e.g., subscribers to a mailing list). They can include a small group of people or many thousands.

Many multi-player computer games now include support for the virtual communities that are players of the given game. Players can receive news about game upgrades, new game maps and characters; they can announce and organize matches, compare scores and exchange tips. The game company takes advantage of this infrastructure to generate revenue or to advertise its products.

Currently, however, each game company develops such community support in each individual game. Each company uses a different infrastructure, different concepts, and provides different levels of support. This redundancy and inconsistency results in many disadvantages, including a learning curve for players when joining each new community, for game companies who need to develop the support from scratch, and for advertisers who need to contact each individual community separately. Moreover, this solution does not provide much opportunity for cross-fertilization among different communities.

2. Objectives

The objectives of the ARENA project are to:

- provide an infrastructure for operating an arena, including registering new games and players, organizing tournaments, and keeping track of the players scores.
- provide a framework for league owners to customize the number and sequence of matches and the accumulation of expert rating points.
- provide a framework for game developers for developing new games, or for adapting existing games into the ARENA framework.
- provide an infrastructure for advertisers.

3. Functional requirements

ARENA supports five types of users:

- The *operator* should be able to define new games, define new tournament styles (e.g., knock-out tournaments, championships, best of series), define new expert rating formulas, and manage users.
- *League owners* should be able to define a new league, organize and announce new tournaments within a league, conduct a tournament, and declare a winner.
- *Players* should be able to register in an arena, apply for a league, play the matches that are assigned to the player, or drop out of the tournament.
- *Spectators* should be able to monitor any match in progress and check scores and statistics of past matches and players. Spectators do not need to register in an arena.
- The *advertiser* should be able to upload new advertisements, select an advertisement scheme (e.g., tournament sponsor, league sponsor), check balance due, and cancel advertisements.

Figure 4-17 Initial ARENA problem statement.

4. Nonfunctional requirements

- *Low operating cost.* The operator must be able to install and administer an arena without purchasing additional software components and without the help of a full-time system administrator.
- *Extensibility.* The operator must be able to add new games, new tournament styles, and new expert rating formulas. Such additions may require the system to be temporarily shut down and new modules (e.g., Java classes) to be added to the system. However, no modifications of the existing system should be required.
- *Scalability.* The system must support the kick-off of many parallel tournaments (e.g., 10), each involving up to 64 players and several hundreds of simultaneous spectators.
- *Low-bandwidth network.* Players should be able to play matches via a 56K analog modem or faster.

5. Target environment

- All users should be able to access any arena with a web browser supporting cookies, Javascript, and Java applets. Administration functions (e.g., adding new games, tournament styles, and users) used by the operator should not be available through the web.
- ARENA should run on any Unix operating system (e.g., MacOS X, Linux, Solaris).

Figure 4-17 *Continued.*

4.6.2 Identifying Actors and Scenarios

We identify five actors, one for each type of user in the problem statement (`Operator`, `LeagueOwner`, `Player`, `Spectator`, and `Advertiser`). As the core functionality of the system is to organize and play tournaments, we first develop an example scenario, `organize-TicTacToeTournament` (Figure 4-18) to elicit and explore this functionality in more detail. By first focusing on a narrow vertical slice of the system, we understand better the client's expectation of the system, including the boundary of the system and the kinds of interactions between the user and the system. Using the `organizeTicTacToeTournament` scenario of Figure 4-18, we produce a series of questions for the client depicted in (Figure 4-19). Based on the answers from the client, we refine the scenario accordingly.

Note that when asking questions of a client, our primary goal is to understand the client's needs and the application domain. Once we understand the domain and produce a first version of the requirements specification, we can start trading off features and cost with the client and prioritizing requirements. However, intertwining elicitation and negotiation too early is usually counterproductive.

After we refine the first scenario to the point that both we agree with the client on the system boundary (for that scenario), we focus on the overall scope of the system. This is done by identifying a number of shorter scenarios for each actor. Initially, these scenarios are not detailed, but instead, cover a broad range of functionality (Figure 4-20).

When we encounter disagreements or ambiguities, we detail specific scenarios further. In this example, the scenarios `defineKnockOutStyle` and `installTicTacToeGame` would be refined to a comparable level of detail as the `organizeTicTacToeTournament` (Figure 4-18).

Scenario name	`organizeTicTacToeTournament`
Participating actor instances	`alice:Operator, joe:LeagueOwner, bill:Spectator, mary:Player`
Flow of events	1. Joe, a friend of Alice, is a Tic Tac Toe aficionado and volunteers to organize a tournament. 2. Alice registers Joe in the arena as a league owner. 3. Joe first defines a Tic Tac Toe beginners league, in which any players can be admitted. This league, dedicated to Tic Tac Toe games, stipulates that tournaments played in this league will follow the knockout tournament style and "Winner Takes All" formula. 4. Joe schedules the first tournament in the league for 16 players starting the next day. 5. Joe announces the tournament in a variety of forums over the Web and sends mail to other Tic Tac Toe community members. 6. Bill and Mary receive the E-mail notification. 7. Mary is interested in playing the tournament and registers. 19 others apply. 8. Joe schedules 16 players for the tournament and rejects the 4 that applied last. 9. The 16 players, including Mary, receive an electronic token for entering the tournament and the time of their first match. 10. Other subscribers to the Tic Tac Toe mailing list, including Bill, receive a second notice about the Tournament, including the name of the players and the schedule of matches. 11. As Joe kicks off the tournament, the players have a limited amount of time to enter the match. If a player fails to show up, he loses the game. 12. Mary plays her first match and wins. She advances in the tournament and is scheduled for the next match against another winner of the first round. 13. After visiting the Tic Tac Toe Tournament's home page, Bill notices Mary's victory and decides to watch her next match. He selects the match, and sees the sequence of moves of each player as they occur. He also sees an advertisement banner at the bottom of his browser, advertising other tournaments and tic tac toe products. 14. The tournament continues until the last match, at which point the winner of the tournament is declared and his league record is credited with all the points associated with the tournament. 15. Also, the winner of the tournament accumulates expert rating points. 16. Joe can choose to schedule more tournaments in the league, in which case, known players are notified about the date and given priority over new players.

Figure 4-18 `organizeTicTacToeTournament` scenario for ARENA.

Typical scenarios, once refined, span several pages of text. We also start to maintain a glossary of important terms, to ensure consistency in the specification and to ensure that we use the client's terms. We quickly realize that the terms `Match`, `Game`, `Tournament`, and `League` represent application domain concepts that need to be defined precisely, as these terms could have a different interpretation in other gaming contexts. To accomplish this, we maintain a working glossary and revise our definitions as our exploratory work progresses (Table 4-4).

Steps 2, 7: Different actors register with the system. In the first case, the administrator registers Joe as a league owner; in the second case, a player registers herself with the system.

- Registration of users should follow the same paradigm. Who provides the registration information and how is the information reviewed, validated, and accepted?
- *Client: Two processes are confused in steps 2 & 7, the registration process, during which new users (e.g., a player or a league owner) establish their identity, and the application process, during which players indicate they want to take part in a specific tournament. During the registration process, the user provides information about themselves (name, nickname, E-mail) and their interests (types of games and tournaments they want to be informed about). The information is validated by the operator. During the application process, players indicate which tournament they want to participate in. This is used by the league owner during match scheduling.*
- Since the player information has already been validated by the operator, should the match scheduling be completely automated?
- *Client: Yes, of course.*

Step 5: Joe sends mail to the Tic Tac Toe community members:

- Does ARENA provide the opportunity to users to subscribe to individual mailing lists?
- *Client: Yes. There should be mailing lists for announcing new games, new leagues, new tournaments, etc.*
- Does ARENA store a user profile (e.g., game watched, games played, interests specified by a user survey) for the purpose of advertisement?
- *Client: Yes, but users should still be able to register without completing a user survey, if they want to. They should be encouraged to enter the survey, but this should not prevent them from entering. They will be exposed to advertisements anyway.*
- Should the profile be used to automatically subscribe to mailing lists?
- *Client: No, we think users in our community would prefer having complete control over their mailing list subscriptions. Guessing subscriptions would not give them the impression they are in control.*

Step 13: Bill browses match statistics and decides to see the next match in real time.

- How are players identified to the spectators? By real name, by E-mail, by nickname?
- *Client: This should be left to the user during the registration.*
- Can a spectator replay old matches?
- *Client: Games should be able to provide this ability, but some games (e.g., real-time, 3D action games) may choose not to do so because of resource constraints.*
- ARENA should support real-time games?
- *Client: Yes, these represent the largest share of our market. In general, ARENA should support as broad a range of games as possible.*
- ...

Figure 4-19 Questions generated from the scenario of Figure 4-18. Answers from the client emphasized in *italics*. The interviewer can ask follow-up questions as new knowledge is accidentally stumbled upon.

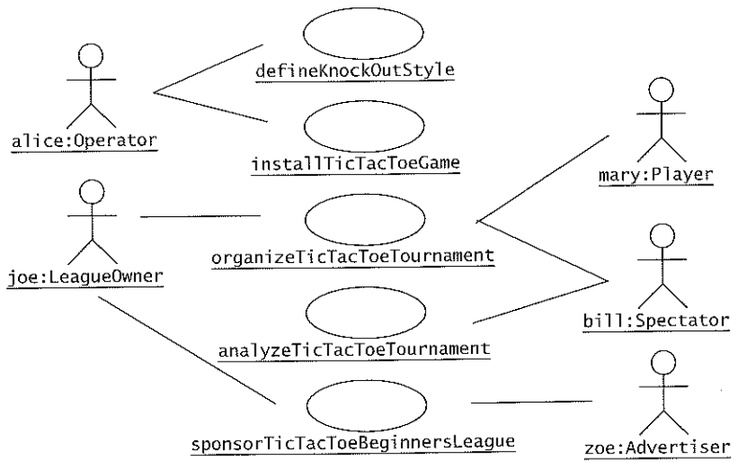


Figure 4-20 High-level scenarios identified for ARENA. Clients and developers initially briefly describe scenarios. They refine them further to clarify ambiguities or uncover disagreements.

Table 4-4 Working glossary for ARENA. Keeping track of important terms and their definitions ensures consistency in the specification and ensures that developers use the language of the client.

Game	A Game is a competition among a number of Players that is conducted according to a set of rules. In ARENA, the term Game refers to a piece of software that enforces the set of rules, tracks the progress of each Player, and decides the winner. For example, tic tac toe and chess are Games.
Match	A Match is a contest between two or more Players following the rules of a Game. The outcome of a Match can be a single winner and a set of losers or a tie (in which their are no winners or losers). Some Games may disallow ties.
Tournament	A Tournament is a series of Matches among a set of Players. Tournaments end with a single winner. The way Players accumulate points and Matches are scheduled is dictated by the League in which the Tournament is organized.
League	A League represents a community for running Tournaments. A League is associated with a specific Game and TournamentStyle. Players registered with the League accumulate points according to the ExpertRating defined in the League. For example, a novice chess League has a different ExpertRating formula than an expert League.
TournamentStyle	The TournamentStyle defines the number of Matches and their sequence for a given set of Players. For example, Players face all other Players in the Tournament exactly once in a round robin TournamentStyle.

Once we agree with the client on a general scope of the system, we formalize the knowledge acquired so far in the form of high-level use cases.

4.6.3 Identifying Use Cases

Generalizing scenarios into use cases enables developers to step back from concrete situations and consider the general case. Developers can then consolidate related functionality into single use cases and split unrelated functionality into several use cases.

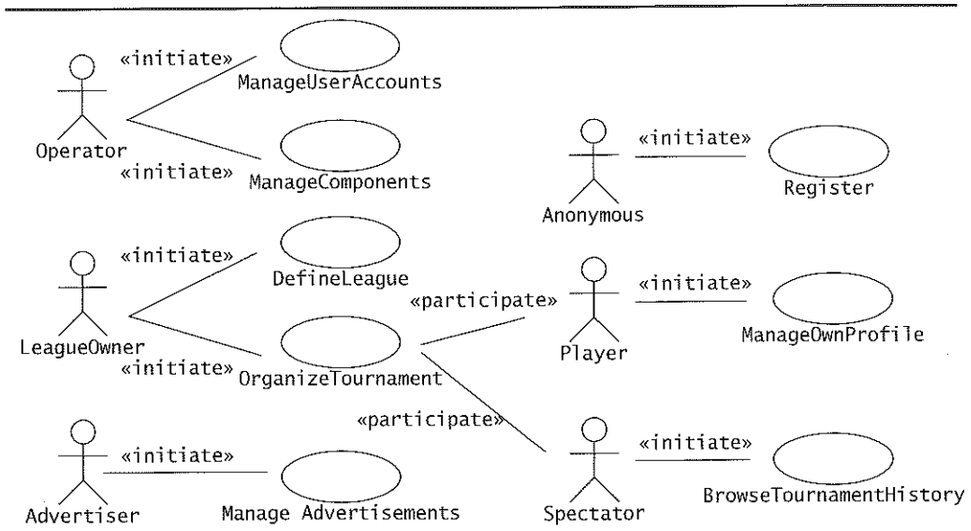
When inspecting the `organizeTicTacToeTournament` scenario closely, we realize that it covers a broad range of functionality initiated by many actors. We anticipate that generalizing this scenario would result in a use case of several dozen pages long, and attempt to split it into self-contained and independent use cases initiated by single actors. We first decide to split the functionality related to user accounts into two use cases, `ManageUserAccounts`, initiated by the `Operator`, and `Register`, initiated by potential players and league owners (Figure 4-21). We identify a new actor, `Anonymous`, representing these potential users who do not yet have an account. Similarly, we split the functionality with browsing past matches and with managing user profiles into separate use cases (`BrowseTournamentHistory` and `ManageOwnProfile`, initiated by the `Spectator` and the `Player`, respectively). Finally, to further shorten the use case `OrganizeTournament`, we split off the functionality for creating new leagues into the `DefineLeague` use case, as a `LeagueOwner` may create many tournaments within the scope of a single league. Conversely, we anticipate that the installation of new games and new styles requires similar steps from the `Operator`. Hence, we consolidate all functionality related to installing new components into the `ManageComponents` use case initiated by the `Operator`.

We capture these decisions by drawing an overview use case diagram and by briefly describing each use case (Figure 4-21). Note that a use case diagram alone does not describe much functionality. Instead, it is an index into the many descriptions produced during this phase.

Next, we describe the fields of each high-level use case, including the participating actors, entry and exit conditions, and a flow of events. Figure 4-22 depicts the high-level `OrganizeTournament` use case.

Note that all steps in this flow of events describe actor actions. High-level use cases focus primarily on the task accomplished by the actor. The detailed interaction with the system, and decisions about the boundaries of the system, are initially postponed to the refinement phase. This enables us to first describe the application domain with use cases, capturing, in particular, how different actors collaborate to accomplish their goals.

In Figure 4-22, we describe the sequence of actions that are performed by four actors to organize a tournament: the `LeagueOwner`, who facilitates the complete activity, the `Advertiser`, to resolve exclusive sponsorship issues, the potential `Players` who want to participate, and the `Spectators`. In the first step, we describe the handling of the sponsorship issue, thus making clear that any sponsorship issue needs to be resolved before the tournament is advertised and before the players apply for the tournament. Originally, the sponsorship issue was not described clearly in the scenarios of Figure 4-20 (which only described the sponsorships of leagues). After



Register — Anonymous users register with an Arena for a Player or a League-Owner account. User accounts are required before applying for a tournament or organizing a league. Spectators do not need accounts.

ManageUserAccounts — The Operator accepts registrations from LeagueOwners and for Players, cancels existing accounts, and interacts with users about extending their accounts.

ManageComponents — The Operator installs new games and defines new tournament styles (generalizes defineKnockOutStyle and installTicTacToeGame).

DefineLeague — The LeagueOwner defines a new league (generalizes the first steps of the scenario organizeTicTacToeTournament).

OrganizeTournament — The LeagueOwner creates and announces a new tournament, accepts player applications, schedules matches, and kicks off the tournament. During the tournament, players play matches and spectators follow matches. At the end of the tournament, players are credited with points (generalizes the scenario organizeTicTacToeTournament).

ManageAdvertisements — The Advertiser uploads banners and sponsors league or tournaments (generalizes sponsorTicTacToeBeginnersLeague).

ManageOwnProfile — The Players manage their subscriptions to mailing lists and answer a marketing survey.

BrowseTournamentHistory — Spectators examine tournament statistics and player statistics, and replay matches that have already been concluded (generalizes the scenario analyzeTicTacToeTournament).

Figure 4-21 High-level use cases identified for ARENA.

Use case name	OrganizeTournament
Participating actors	Initiated by LeagueOwner Communicates with Advertiser, Player, and Spectator
Flow of events	1. The LeagueOwner creates a Tournament, solicits sponsorships from Advertisers, and announces the Tournament (include use case AnnounceTournament). 2. The Players apply for the Tournament (include use case ApplyForTournament). 3. The LeagueOwner processes the Player applications and assigns them to matches (include use case ProcessApplications). 4. The LeagueOwner kicks off the Tournament (include use case KickoffTournament). 5. The Players compete in the matches as scheduled and Spectators view the matches (include use case PlayMatch). 6. The LeagueOwner declares the winner and archives the Tournament (include use case ArchiveTournament).
Entry condition	• The LeagueOwner is logged into ARENA.
Exit conditions	• The LeagueOwner archived a new tournament in the ARENA archive and the winner has accumulated new points in the league, OR • The LeagueOwner cancelled the tournament and the players' standing in the league is unchanged.

Figure 4-22 An example of a high-level use case, OrganizeTournament.

discussions with the client, we decided to handle also tournament sponsorship, and to handle it at the beginning of each tournament. On the one hand, this enables new sponsors to be added to the system, and on the other hand, it allows the sponsor, in exchange, to advertise the tournament using his or her own resources. Finally, this enables the system to better select advertisement banners during the application process.

In this high-level use case, we boiled down the essentials of the organizeTicTacToeTournament scenario into six steps and left the details to the detailed use case. By describing each high-level use case in this manner, we capture all relationships among actors that the system must be aware of. This also results in a summary description of the system that is understandable to any newcomer to the project.

Next, we write the detailed use cases to specify the interactions between the actors and the system.

4.6.4 Refining Use Cases and Identifying Relationships

Refining use cases enables developers to define precisely the information exchanged among the actors and between the actors and the system. Refining use cases also enables the discovery of alternative flows of events and exceptions that the system should handle.

To keep the case study manageable, we do not show the complete refinement. We start by identifying one detailed use case for each step of the flow of events in the high-level `OrganizeTournament` use case. The resulting use case diagram is shown in Figure 4-23. We then focus on the use case, `AnnounceTournament`: Figure 4-24 contains a description of the flow of events, and Figure 4-25 identifies the exceptions that could occur in `AnnounceTournament`. The remaining use cases will be developed similarly.

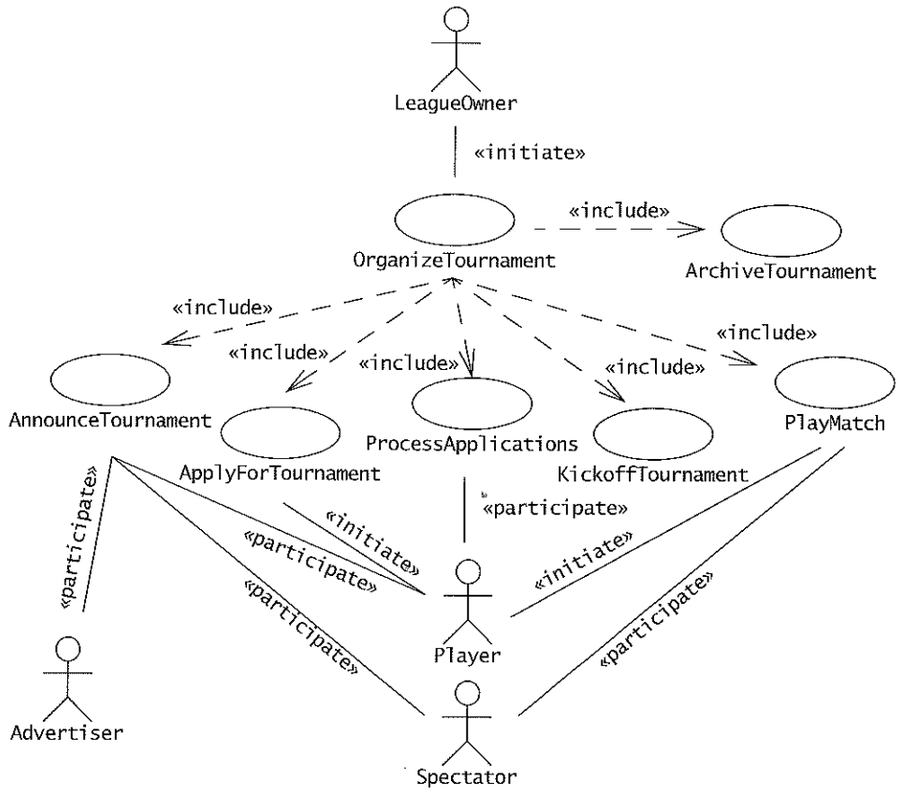


Figure 4-23 Detailed use cases refining the `OrganizeTournament` high-level use case.

All of the use cases in Figure 4-23 are initiated by the `LeagueOwner`, except that the `ApplyForTournament` and `PlayMatch` are initiated by the `Player`. The `Advertiser` participates in `AnnounceTournament` and the `Spectator` participates in `AnnounceTournament` and `PlayMatch` use cases. The `Player` participates in all use cases that refine `OrganizeTournament`. To keep the use case diagram readable, we omitted the «`initiate`» relationships between the `LeagueOwner` and the refined use cases. When using a UML modeling tool, we would include those

Name	`AnnounceTournament`
Participating actors	Initiated by `LeagueOwner` Communicates with `Player`, `Advertiser`, `Spectator`
Flow of events	1. The `LeagueOwner` requests the creation of a tournament. 2. The system checks if the `LeagueOwner` has exceeded the number of tournaments in the league or in the arena. If not, the system presents the `LeagueOwner` with a form. 3. The `LeagueOwner` specifies a name, application start and end dates during which `Players` can apply to the tournament, start and end dates for conducting the tournament, and a maximum number of `Players`. 4. The system asks the `LeagueOwner` whether an exclusive sponsorship should be sought and, if yes, presents a list of `Advertisers` who expressed the desire to be exclusive sponsors. 5. If the `LeagueOwner` decides to seek an exclusive sponsor, he selects a subset of the names of the proposed sponsors. 6. The system notifies the selected sponsors about the upcoming tournament and the flat fee for exclusive sponsorships. 7. The system communicates their answers to the `LeagueOwner`. 8. If there are interested sponsors, the `LeagueOwner` selects one of them. 9. The system records the name of the exclusive sponsor and charges the flat fee for sponsorships to the `Advertiser`'s account. From now on, all advertisement banners associated with the tournament are provided by the exclusive sponsor only. 10. Otherwise, if no sponsors were selected (either because no `Advertiser` was interested or the `LeagueOwner` did not select one), the advertisement banners are selected at random and charged to the `Advertiser's` account on a per unit basis. 11. Once the sponsorship issue is closed, the system prompts the `LeagueOwner` with a list of groups of `Players`, `Spectators`, and `Advertisers` that could be interested in the new tournament. 12. The `LeagueOwner` selects which groups to notify. 13. The system creates a home page in the arena for the tournament. This page is used as an entry point to the tournament (e.g., to provide interested `Players` with a form to apply for the tournament, and to interest `Spectators` in watching matches). 14. On the application start date, the system notifies each interested user by sending them a link to the main tournament page. The `Players` can then apply for the tournament with the `ApplyForTournament` use case until the application end date.

Figure 4-24 An example of a detailed use case, `AnnounceTournament`.

Entry condition	• The LeagueOwner is logged into ARENA.
Exit conditions	• The sponsorship of the tournament is settled: either a single exclusive Advertiser paid a flat fee or banners are drawn at random from the common advertising pool of the Arena. • Potential Players received a notice concerning the upcoming tournament and can apply for participation. • Potential Spectators received a notice concerning the upcoming tournament and know when the tournament is about to start. • The tournament home page is available for any to see, hence, other potential Spectators can find the tournament home page via web search engines, or by browsing the Arena home page.
Quality requirements	• Offers to and replies from Advertisers require secure authentication, so that Advertisers can be billed solely on their replies. • Advertisers should be able to cancel sponsorship agreements within a fixed period, as required by local laws.

Figure 4-24 *Continued.*

relationships as well. We start by writing out the flow of events for the AnnounceTournament use case (Figure 4-24).

The steps in Figure 4-24 describe in detail the information exchanged between the actor and the system. Note, however, that we did not describe any details of the user interface (e.g., forms, buttons, layout of windows or web pages). It is much easier to design a usable user interface later, after we know the intent and responsibility of each actor. Hence, the focus on the refinement phase is to assign (or discover) the detailed intent and responsibilities of each actor.

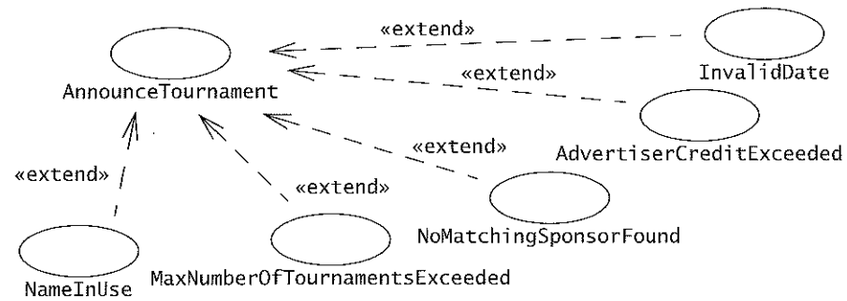
When describing the steps of the detailed AnnounceTournament use case, we and the client made more decisions about the boundaries of the system:

- We introduced start and end dates for the application process and for executing the tournament (Step 3 in Figure 4-24). This enables us to communicate deadlines to all actors involved to ensure that the tournament happens within a reasonable time frame.
- We decided that advertisers indicate in their profile whether they are interested in exclusive sponsorships or not. This enables the LeagueOwner to target Advertisers more specifically (Step 4 in Figure 4-24).
- We also decided to enable advertisers to commit to sponsorship deals through the system and automated the accounting of advertisement and the billing. This entails security and legal requirements on the system, which we document in the "quality requirements" field of the use case (Step 9 and 10 in Figure 4-24).

Note that these decisions are validated with the client. Different clients and environments can lead to evaluating trade-offs differently for the same system. For example, the decision about soliciting Advertisers and obtaining a commitment through the system results in a more complex and expensive system. An alternative would have been to solicit Advertisers via E-mail, but obtain their commitment via phone. This would have resulted in a simpler system, but more work on the part of the LeagueOwner. The client is the person who decides between such alternatives, understanding, of course, that these decisions have an impact on the cost and the delivery date of the system.

Next, we identify the exceptions that could occur during the detailed use case. This is done by reviewing every step in the use case and identifying all the events that could go wrong. We briefly describe the handling of each exception and depict the exception handling use cases as extensions of the AnnounceTournament use case (Figure 4-25).

Note that not all exceptions are equal, and different kinds of exceptions are best addressed at different stages of development. In Figure 4-25, we identify exceptions caused by resource constraints (MaxNumberOfTournamentsExceeded), invalid user input (InvalidDate, NameInUse), or application domain constraints (AdvertiserCreditExceeded, NoMatchingSponsorFound). Exceptions associated with resource constraints are best handled during system design. Only during system design will it become clear which resources are limited and how to best share



AdvertiserCreditExceeded	The system removes the Advertiser from the list of potential sponsors.
InvalidDate	The system informs the LeagueOwner and prompts for a new date.
MaxNumberOfTournaments Exceeded	The AnnounceTournament use case is terminated.
NameInUse	The system informs the LeagueOwner and prompts for a new name.
NoMatchingSponsorFound	The system skips the exclusive sponsor steps and chooses random advertisements from the advertisement pool.

Figure 4-25 Exceptions occurring in AnnounceTournament represented as extending use cases. (Note that AnnounceTournament in this figure is the same as the use case in Figure 4-23).

them among different users which may, in turn, trigger further requirements activities during system design to validate with the client the handling of such exceptions. Exceptions associated with invalid user input are best handled during user interface design, when developers will be able to decide at which point to check for invalid input, how to display error messages, and how to prevent invalid inputs in the first place. The third category of exceptions—application domain constraints—should receive the focus of the client and developer early. These are exceptions that are usually not obvious to the developer. When missed, they require substantial rework and changes to the system. A systematic way to elicit those exceptions is to walk through the use case step by step with the client or a domain expert.

Many exceptional events can be represented either as an exception (e.g., `AdvertiserCreditExceeded`) or as a nonfunctional requirement (e.g., "An `Advertiser` should not be able to spend more advertisement money than a fixed limit agreed beforehand with the `Operator` during the registration"). The latter representation is more appropriate for global constraints that apply to several use cases. Conversely, the former is more appropriate for events that can occur only in one use case (e.g., "`NoMatchingSponsorFound`").

Writing each detailed use case, including their exceptions, constitutes the lion's share of the requirements elicitation effort. Ideally, developers write every detailed use case and address all application domain issues before committing to the project and initiating the realization of the system. In practice, this never happens. For large systems, the developers produce a large amount of documentation in which it is difficult, if not impossible, to maintain consistency. Worse, the requirements elicitation activity of large projects should already be financed, as this phase requires a lot of resources from both the client and the development organization. Moreover, completeness at an early stage can be counterproductive: use case steps change during development as new domain facts are discovered. The decision about how many use cases to detail and how much to leave implicit is as much a question of trust as of economics: the client and the developers should share a sufficiently good understanding of the system to be ready to commit to a schedule, a budget, and a process for handling future changes (including changes in requirements, schedule, and budget).

In ARENA, we focus on specifying in detail the interactions that involve the `Advertisers` and the `Players`, since they have critical roles in generating revenue. Use cases associated with the administration of the system or the installation of new games or tournament styles are left for later, since they also include more technical issues that are dependent on the solution domain.

4.6.5 Identifying Nonfunctional Requirements

Nonfunctional requirements come from a variety of sources during the elicitation. The problem statement we started with in Figure 4-17 already specified performance and implementation requirements. When detailing the `AnnounceTournament` use case, we identified further legal requirements for billing `Advertisers`. When reviewing exceptions in the previous section, we identified a constraint on the amount of money `Advertisers` can spend. Although we encounter many nonfunctional requirements while writing use cases and refining them, we cannot ensure

that we identify all the essential nonfunctional requirements. To ensure completeness, we use the FURPS+ categories we described in Section 4.3.2 (or any other systematic taxonomy of nonfunctional requirements) as a checklist for asking questions of the client. Table 4-5 depicts the nonfunctional requirements we identified in ARENA after detailing the `AnnounceTournament` use case.

Table 4-5 Consolidated nonfunctional requirements for ARENA, after the first version of the detailed `AnnounceTournament` use case.

Category	Nonfunctional requirements
Usability	• `Spectators` must be able to access games in progress without prior registration and without prior knowledge of the `Game`.
Reliability	• Crashes due to software bugs in game components should interrupt at most one `Tournament` using the `Game`. The other `Tournaments` in progress should proceed normally. • When a `Tournament` is interrupted because of a crash, its `LeagueOwner` should be able to restart the `Tournament`. At most, only the last move of each interrupted `Match` can be lost.
Performance	• The system must support the kick-off of many parallel `Tournaments` (e.g., 10), each involving up to 64 `Players` and several hundreds of simultaneous `Spectators`. • `Players` should be able to play matches via an analog modem.
Supportability	• The `Operator` must be able to add new `Games` and new `TournamentStyles`. Such additions may require the system to be temporarily shut down and new modules (e.g., Java classes) to be added to the system. However, no modifications of the existing system should be required.
Implementation	• All users should be able to access an `Arena` with a web browser supporting cookies, Javascript, and Java applets. Administration functions used by the operator are not available through the web. • ARENA should run on any Unix operating system (e.g., MacOS X, Linux, Solaris).
Operation	• An `Advertiser` should not be able to spend more advertisement money than a fixed limit agreed beforehand with the `Operator` during the registration.
Legal	• Offers to and replies from `Advertisers` require secure authentication, so that agreements can be built solely on their replies. • `Advertisers` should be able to cancel sponsorship agreements within a fixed period, as required by local laws.

4.6.6 Lessons Learned

In this section, we developed an initial use case and analysis object model based on a problem statement provided by the client. We used scenarios and questions as elicitation tools to clarify ambiguous concepts and uncover missing information. We also elicited a number of nonfunctional requirements. We learned that

- Requirements elicitation involves constant switching between perspectives (e.g., high-level vs. detailed, client vs. developer, activity vs. entity).
- Requirements elicitation requires a substantial involvement from the client.
- Developers should not assume that they know what the client wants.
- Eliciting nonfunctional requirements forces stakeholders to make and document trade-offs.

4.7 Further Readings

The concept of use case was made popular by Ivar Jacobson in his landmark book, *Object-Oriented Software Engineering: A Use Case Approach* [Jacobson et al., 1992]. For an account of the early research on scenario-based requirements and, more generally, on participatory design, *Scenario-Based Design* [Carroll, 1995] includes many papers by leading researchers about scenarios and use cases. This book also describes limitations and pitfalls of scenario-based requirements and participatory design, which are still valid today.

For specific method guidance, *Software for Use* [Constantine & Lockwood, 1999] contains much material on specifying usable systems with use cases, including eliciting imprecise knowledge from users and clients, a soft topic that is usually not covered in software engineering text books. *Writing Effective Use Cases* [Cockburn, 2001] and its accompanying website `http://www.usecases.org` provide many practical heuristics for writing use cases textually (as opposed to just drawing them).

End users play a critical role during requirements elicitation. Norman illustrates this by using examples from everyday objects such as doors, stoves, and faucets [Norman, 2002]. He argues that users should not be expected to read a user manual and learn new skills for every product to which they are exposed. Instead, knowledge about the use of the product, such as hints indicating in which direction a door opens, should be embedded in its design. He takes examples from everyday objects, but the same principles are applicable to computer systems and user interface design.

The world of requirements engineering is much poorer when it comes to dealing with nonfunctional requirements. The NFR Framework, described in *Non-Functional Requirements in Software Engineering* [Chung et al., 1999], is one of the few methods that addresses this topic systematically and thoroughly.

The RAD template introduced in this chapter is just one example of how to organize a requirements document. IEEE published the documentation standard IEEE-Std 830-1998 for

software requirements specifications [IEEE Std. 830-1998]. The appendix of the standard contains several sample outlines for the description of specific requirements.

The examples in this chapter followed a dialectic approach to requirements elicitation, a process of discussion and negotiation among developers, the client, and the end users. This approach works well when the client is the end user, or when the client has a sufficiently detailed knowledge of the application domain. In large systems, such as an air traffic control system, no single user or client has a complete perspective of the system. In these situations, the dialectic approach breaks down, as much implicit knowledge about the users' activities is not encountered until too late. In the past decade, ethnography, a field method from anthropology, has gained popularity in requirements engineering. Using this approach, analysts immerse themselves in the world of users, observe their daily work, and participate in their meetings. Analysts record their observations from a neutral point of view. The goal of such an approach is to uncover implicit knowledge. The coherence method, reported in *Social analysis in the requirements engineering process: from ethnography to method* [Viller & Sommerville, 1999], provides a practical example of ethnography applied to requirements engineering.

Managing traceability beyond requirements is still a research topic, the reader is referred to the specialized literature [Jarke, 1998].

Finally, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices* [Jackson, 1995] is a concise, incisive, and entertaining piece that provides many insights into principles and methods of requirements engineering.

4.8 Exercises

4-1 Consider a microwave oven as a system. Heat a cup of water in the microwave for one minute. Write down each interaction between you and the microwave. Record all interactions including setting the time, setting the power level, whether you interrupt the cycle early because the water boils before the minute is up. Record any feedback the microwave provides you.

4-2 Consider the scenario you wrote in Exercise 4-1. Identify the actor in the scenario. Next, write the corresponding use case `HeatWater`. Include all cases, and include setting the time, setting the power, interrupting the cycle.

4-3 Assume the microwave system you described in Exercises 4-1 and 4-2 supports a delayed timer feature, which allows the microwave to be started automatically at a later time. Describe setting the timer as a self-contained use case named `SetTimer`.

4-4 Examine the `HeatWater` and `SetTimer` use cases you wrote in Exercise 4-2 and 4-3. Eliminate any redundancy by using an include relationship. Justify why an include relationship is preferable to an extend relationship in this case.

4-5 Assume the `FieldOfficer` can invoke a Help feature when filling an `EmergencyReport`. The `HelpReportEmergency` feature provides a detailed description for each field and specifies which fields are required. Modify the `ReportEmergency` use case (described

in Figure 4-10) to include this help functionality. Which relationship should you use to relate the `ReportEmergency` and `HelpReportEmergency`?

4-6 Below are examples of nonfunctional requirements. Specify which of these requirements are verifiable and which are not:

- "The system must be usable."
- "The system must provide visual feedback to the user within one second of issuing a command."
- "The availability of the system must be above 95 percent."
- "The user interface of the new system should be similar enough to the old system that users familiar with the old system can be easily trained to use the new system."

4-7 The need for developing a complete specification may encourage an analyst to write detailed and lengthy documents. Which competing quality of specification (see Table 4-1) may encourage an analyst to keep the specification short?

4-8 Maintaining traceability during requirements and subsequent activities is expensive, because of the additional information that must be captured and maintained. What are the benefits of traceability that outweigh this overhead? Which of those benefits are directly beneficial to the analyst?

4-9 Explain why multiple-choice questionnaires, as a primary means of extracting information from the user, are not effective for eliciting requirements.

4-10 From your point of view, describe the strengths and weaknesses of users during the requirements elicitation activity. Describe also the strengths and weaknesses of developers during the requirements elicitation activity.

4-11 Briefly define the term "menu." Write your answer on a piece of paper and put it upside down on the table together with the definitions of four other students. Compare all five definitions and discuss any substantial difference.

4-12 Write the high-level use case `ManageAdvertisement` initiated by the `Advertiser`, and write detailed use cases refining this high-level use case. Consider features that enable an `Advertiser` to upload advertisement banners, to associate keywords with each banner, to subscribe to notices about new tournaments in specific leagues or games, and to monitor the charges and payments made on the advertisement account. Make sure that your use cases are also consistent with the ARENA problem statement provided in Figure 4-17.

4-13 Considering the `AnnounceTournament` use case in Figure 4-24, write the event flow, entry conditions, and exit conditions for the use case `ApplyForTournament`, initiated by a `Player` interested in participating in the newly created tournament. Consider also the ARENA problem statement provided in Figure 4-17. Write a list of questions for the client when you encounter any alternative.

4-14 Write the event flows, entry conditions, and exit conditions for the exceptional use cases for `AnnounceTournament` depicted in Figure 4-25. Use include relationships if necessary to remove redundancy.

References

[Bruegge et al., 1994] B. Bruegge, K. O'Toole, & D. Rothenberger, "Design considerations for an accident management system," in M. Brodie, M. Jarke, M. Papazoglou (eds.), *Proceedings of the Second International Conference on Cooperative Information Systems*, pp. 90–100, University of Toronto Press, Toronto, Canada, May 1994.

[Carroll, 1995] J. M. Carroll (ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development.* Wiley, New York, 1995.

[Chung et al., 1999] L. Chung, B. A. Nixon, E. Yu & J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic, Boston, 1999.

[Cockburn, 2001] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, Reading, MA, 2001.

[Constantine & Lockwood, 1999] L. L Constantine & L. A. D. Lockwood, *Software for Use*, Addison-Wesley, Reading, MA, 1999.

[Grady, 1992] R. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[Hammer & Champy, 1993] M. Hammer & J. Champy, *Reengineering The Corporation: a Manifesto For Business Revolution*, Harper Business, New York, 1993.

[IEEE Std. 610.12-1990] IEEE, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, New York, NY, 1990.

[IEEE Std. 830-1998] *IEEE Standard for Software Requirements Specification*, IEEE Standards Board, 1998.

[ISO Std. 9126] International Standards Organization. *Software engineering—Product quality.* ISO/IEC-9126, Geneva, Switzerland, 2001.

[Jackson, 1995] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, Reading, MA, 1995.

[Jacobson et al., 1992] I. Jacobson, M. Christerson, P. Jonsson, & G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach*, Addison-Wesley, Reading, MA, 1992.

[Jacobson et al., 1999] I. Jacobson, G. Booch, & J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

[Jarke, 1998] M. Jarke, "Requirements tracing," *Communications of the ACM*, Vol. 41, No. 12, December 1998.

[Neumann, 1995] P. G. Neumann, *Computer-Related Risks*, Addison-Wesley, Reading, MA, 1995.

[Nielsen, 1993] J. Nielsen, *Usability Engineering*, Academic, New York, 1993.

[Norman, 2002] D. A. Norman, *The Design of Everyday Things*, Basic Books, New York, 2002.

[Rational] Rationale, `http://www.rational.com`.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Telelogic] Telelogic, `http://www.telelogic.se`.

[Viller & Sommerville, 1999] S. Viller & I. Sommerville, "Social analysis in the requirements engineering process: from ethnography to method," *International Symposium on Requirements Engineering (ISRE'99)*, Limerick, Ireland, June 1999.

[Wirfs-Brock et al., 1990] R. Wirfs-Brock, B. Wilkerson, & L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Wood & Silver, 1989] J. Wood & D. Silver, *Joint Application Design®*, Wiley, New York, 1989.