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

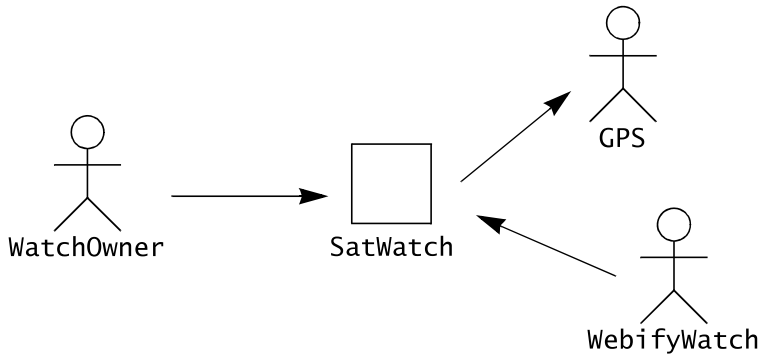

Figure 4-4 Actors for the SatWatch system. WatchOwner moves the watch (possibly across time zones) and consults it to know what time it is. SatWatch interacts with GPS to compute its position. WebifyWatch upgrades the data contained in the watch to reflect changes in time policy (e.g., changes in daylight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

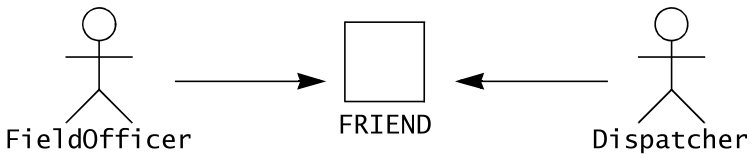

Figure 4-5 Actors of the FRIEND system. FieldOfficers not only have access to different functionality, they use different computers to access the system.

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

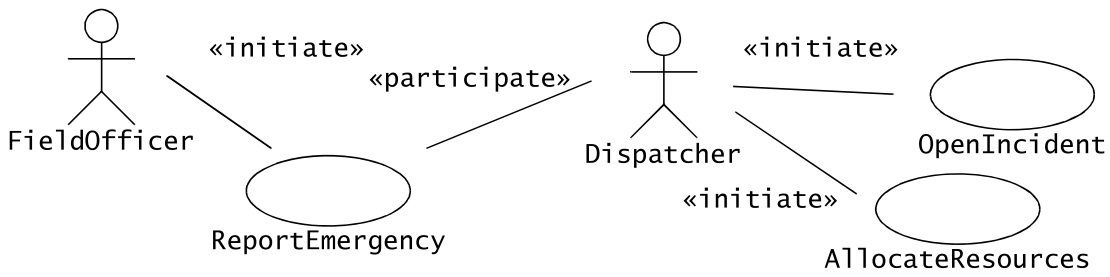

Figure 4-11 Example of communication relationships among actors and use cases in `FRIEND` (UML use case diagram). The `FieldOfficer` initiates the `ReportEmergency` use case, and the `Dispatcher` initiates the `OpenIncident` and `AllocateResources` use cases. `FieldOfficers` cannot directly open an incident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

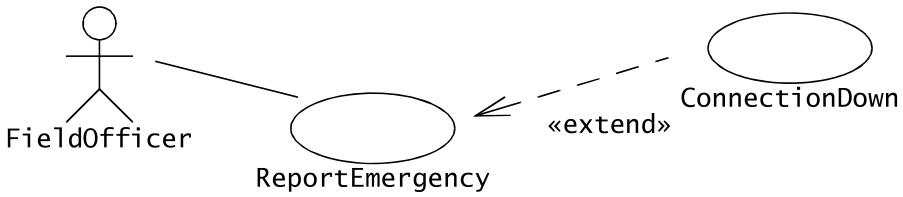

Figure 4-12 Example of use of extend relationship (UML use case diagram). ConnectionDown extends the ReportEmergency use case. The ReportEmergency use case becomes shorter and solely focused on emergency reporting.

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).

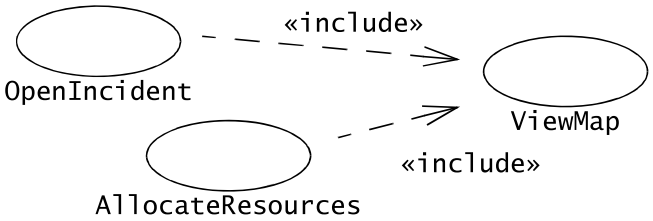

Figure 4-13 Example of include relationships among use cases. ViewMap describes the flow of events for viewing a city map (e.g., scrolling, zooming, query by street name) and is used by both OpenIncident and AllocateResources use cases.

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):