Chapter Three: Requirements Elicitation

3.1. 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 to produce an **analysis model**. 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.

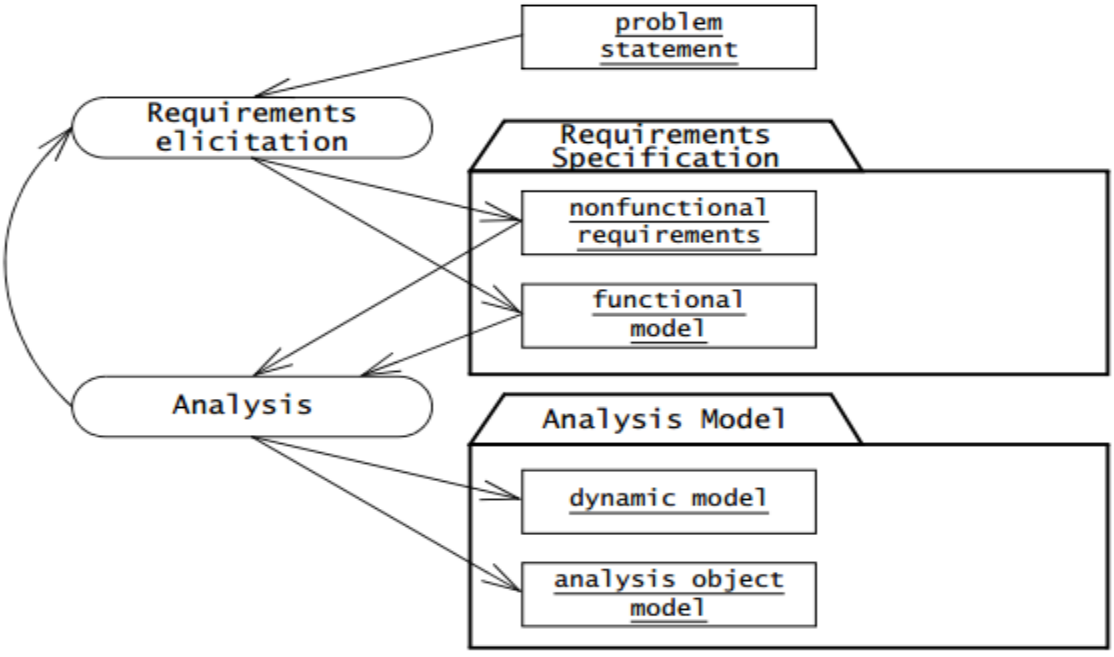

Figure 3-1 Products of requirements elicitation and analysis (UML activity diagram).

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.
- **Traceability** focuses on recording, structuring, linking, grouping, and maintaining dependencies among requirements and between requirements and other work products.

3.2. Requirements Elicitation Concepts

3.2.1. 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 Satellite Watch 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.

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)

3.2.2 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+ model2 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).

3.2.3. 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).

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

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.

3.2.4. 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 non verifiable 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.

3.3. **Requirements Elicitation Activities**

This section describes the requirements elicitation activities. These map a problem statement into a requirements specification that represent as a set of actors, scenarios, and use cases (see Chapter 2, *Modeling with UML*). It also discusses heuristics and methods for eliciting requirements from users and modeling the system in terms of these concepts. Requirements elicitation activities include:

- Identifying Actors
- Identifying Scenarios
- Identifying Use Cases
- Refining Use Cases

- Identifying Relationships Among Actors and Use Cases
- Identifying Initial Analysis Objects.
- Identifying Nonfunctional Requirements.

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]

3.3.1 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 3-2). They all exchange information with the SatWatch. Note, however, that they all have specific interactions with SatWatch: the watch

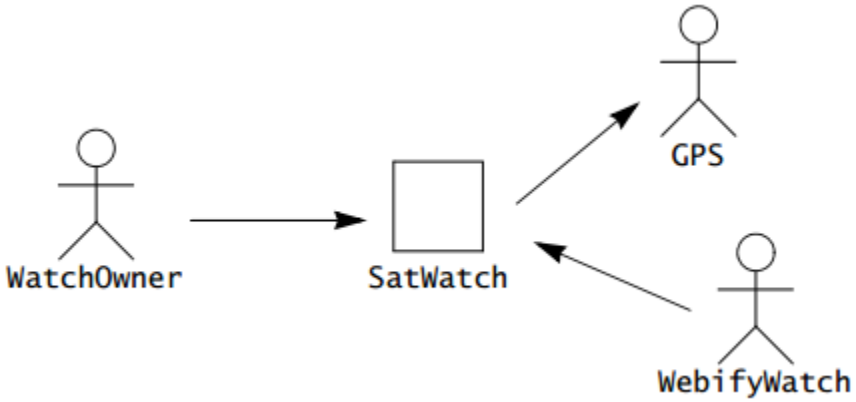

Figure 3.2. Actors for the SatWatch system. WatchOwner moves the watch (possibly across time zones) and consults it to know what time it is. SatWatch interacts with GPS to compute its position. WebifyWatch upgrades the data contained in the watch to reflect changes in time policy (e.g., changes in daylight savings time start and end dates).

3.3.2 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. 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.

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 3.3 warehouseOnFire scenario for the ReportEmergency use case.

Note that this scenario is concrete, in the sense that it describes a single 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.

3.3.3 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 3.4 depicts the use case ReportEmergency of which the scenario warehouseOnFire (see Figure 3-3) 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 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.

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 3.4 An example of a use case, ReportEmergency. Under ReportEmergency, the left column denotes actor actions, and the right column denotes system responses.

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. 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.

3.3.4 Refining Use Cases

Figure 3-5 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 3-5 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.

3.3.5 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.

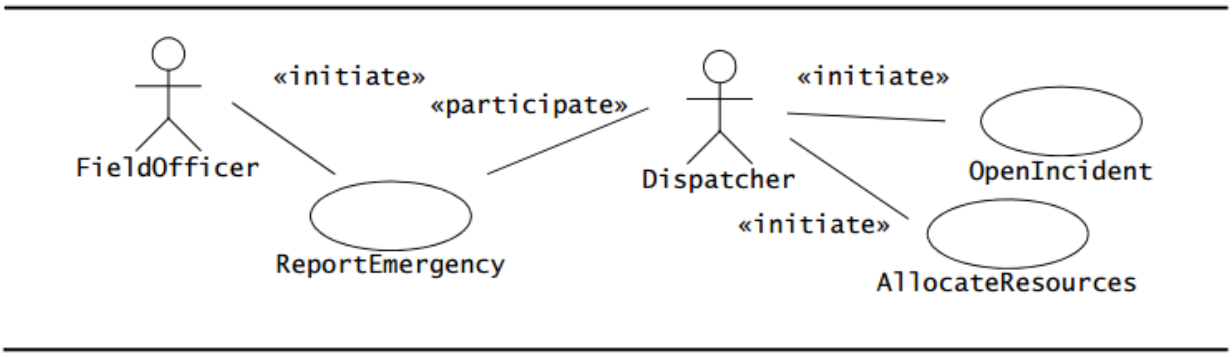

Figure 3-6 Example of communication relationships among actors and use cases in FRIEND (UML use case diagram). The FieldOfficer initiates the ReportEmergency use case, and the Dispatcher initiates the OpenIncident and AllocateResources use cases. FieldOfficers cannot directly open an incident or allocate resources on their own.

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 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.

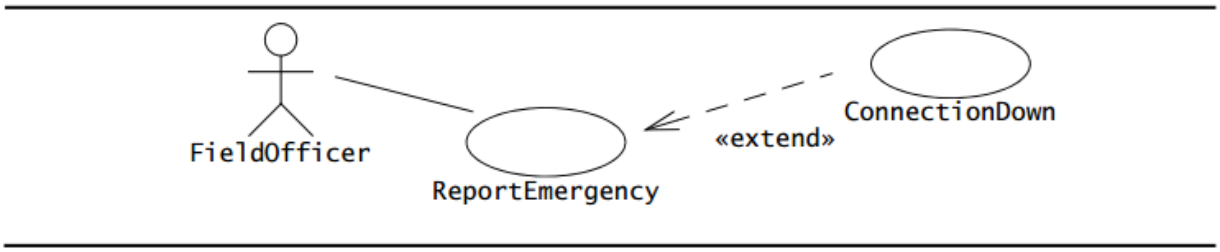

Figure 3-7 Example of use of extend relationship (UML use case diagram). ConnectionDown extendsthe ReportEmergency use case. The ReportEmergency use case becomes shorter and solely focused onemergency reporting.

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

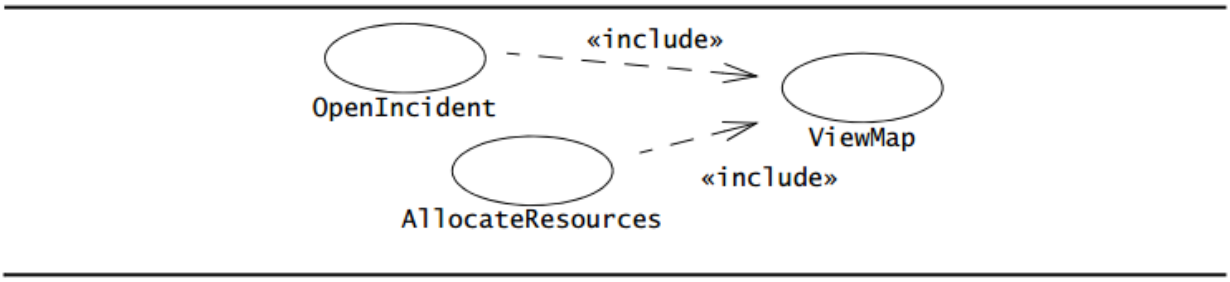

Figure 3-8 Example of include relationships among use cases. ViewMap describes the flow of events for viewing a city map (e.g., scrolling, zooming, query by street name) and is used by both OpenIncident and AllocateResources use cases.

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. 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).

3.3.6 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. 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 many fold: 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

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.

3.3.7 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.

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.

3.4. Managing Requirements Elicitation

The previous section, 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:

3.4.1 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.

3.4.2 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 person power, and it is error prone. However, for small projects, developers can observe benefits early.

3.4.3 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.

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

of the current system. Otherwise, this section describes how the tasks supported by the new system are accomplished now.

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.

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 4-9Outline of the Requirements Analysis Document (RAD). Sections in *italics* are completed during analysis (see next chapter).