# 6. Requirements Elicitation

*If you can't write it down in English, you can't code it.*

**- Peter Halpern**

A *Requirement* is a feature that the system must have or a constraint that it must satisfy to be acceptable to the client. The *Requirements Process* is aimed at defining the requirements of the system under construction. The Requirements Process can be viewed as two main activities, *Requirements Elicitation*, which results in the specification of the system that the customer understands, and *Requirements Analysis*, which results into an analysis model that the developers can unambiguously interpret. Requirements elicitation is the most challenging of the two given that it requires the collaboration of several groups of participants who have different backgrounds. On the one hand, the client and the users have a solid background in their domain and have a general idea of what the system should do. However, they may have little experience in software development or interface design. On the other hand, the developers have experience in building systems but may have little knowledge of the everyday environment of the users. Moreover, each group may be using incompatible terminologies.

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

In this chapter, we describe requirements elicitation. We then focus the development of scenarios and use cases for defining a system. We then survey a number of requirements and problem analysis methods. Requirements analysis methods are presented in the next chapter, Chapter 7, *Requirements Analysis*.

## 6.1. Introduction: a watch example

Requirements elicitation focuses on describing *what* the system should be. The client, the developers, and the users identify a problem area and define a system that would address the problem. Such a system definition is called a *system specification* and often serves as a contract between the client and the developers. The system specification is structured and formalized during requirements analysis (see Chapter 7, *Requirements Analysis*) to produce an analysis model (see Figure 69). Both system specification and analysis model represent the same information. They differ only the language and notation they use. The system specification is written in natural language while the analysis model is usually expressed in a formal or semi-formal notation. The system specification serves as a vehicle for communication with the client and users. The analysis model serves as a vehicle for communication among developers and for validation. 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 requirements analysis usually occur concurrently and iteratively.

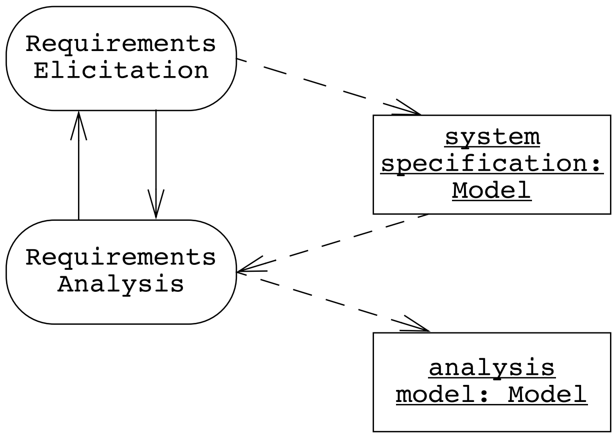


**FIGURE 69.** Products of requirements elicitation and requirements analysis (UML activity diagram).

Requirements elicitation and requirements analysis focus only on the user's view of the system and the constraints imposed by the client (e.g., the environment in which the system will operate). For example, the system functionality, the interaction between the user and the system, the errors which the system can detect and handle, the environmental conditions 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.

*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 of interest interacts. For example, the following is an example of functional requirements for SatWatch, a watch that resets itself automatically:

**Functional requirements for SatWatch.**

SatWatch is a wrist watch that displays the time based on its location using GPS satellites (Global Positioning System). The information stored in the watch and its accuracy to measure time is such (one hundredth of second uncertainty over five years) that the watch owner never needs to reset the time. SatWatch adjusts the time and date displayed as the watch owner crosses time lines and political boundaries (e.g., standard time vs. daylight saving time). For this reason, SatWatch has no buttons or controls available to the user.

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

When a new country or state institutes different rules for daylight saving time, the watch owner may upgrade the software of its watch using the WebifyWatch serial device (provided when the watch is purchased) and a personal computer connected to the Internet. SatWatch complies with the physical, electrical, and software interfaces defined by WebifyWatch API 2.0.

Note that the above functional requirements only focus 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).

*Nonfunctional requirements* describe user visible aspects of the system that are not directly related with the functional behavior of the system. Nonfunctional requirements include quantitative constraints such as response time (i.e., how fast the system reacts to user commands) or accuracy (i.e., how precise are the system's numerical answers). The following are the nonfunctional requirements for SatWatch:

**Nonfunctional requirements for SatWatch.**

SatWatch determines its location using GPS satellites, and as such, suffers from the same limitations as all other GPS devices (e.g., ~ 100 feet accuracy, inability to determine location at certain times of the day in mountainous regions). During black out periods, SatWatch assumes that it does not cross a time line or a political boundary. SatWatch corrects its time zone as soon as a black out period ends.

The battery life of SatWatch is limited to five years, which is the estimated life cycle of the housing of SatWatch. The SatWatch housing is not designed to be opened once manufactured, preventing battery replacement and repairs. Instead, SatWatch is priced such that the watch owner is expected to buy a new SatWatch to replace a defective or old SatWatch.

*Pseudo requirements* are requirements imposed by the client that restrict the implementation of the system. Typical pseudo requirements are the implementation language and the platform on which the system is to be implemented. For life critical developments, pseudo requirements often include process and documentation requirements (e.g, the use of a formal specification method, the complete release of all work products). Pseudo requirements have usually no direct effect on the users' view of the system. The following are the pseudo functional requirements for SatWatch:

**Pseudo requirement for SatWatch.**

All related software associated with SatWatch, including the onboard software, will be written using Java, to comply with current company policy.

Requirements is a modeling activity. The developer constructs a model describing the reality as seen from a user's point of view. Modeling consists of identifying and classifying real world phenomena (e.g., aspects of the system under construction) into concepts Figure 70 is a UML class diagram representing the relationships between models and reality. In this diagram, a model is said to be correct if each concept in the model corresponds to a relevant phenomenon. The model is complete if all relevant phenomena are represented by at least one concept. The model is consistent if all concepts represent phenomena of the same reality (i.e., if a model is inconsistent, it must represent aspects of two different realities).

Requirements, both functional and nonfunctional, 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 dependent on the system specification. Requirement validation checks minimally if the specification is correct, complete, consistent, unambiguous, and realistic. A specification is ***correct*** if it represents the client and the developers view of the system (i.e., everything in the requirements model represent accurately an aspect of the

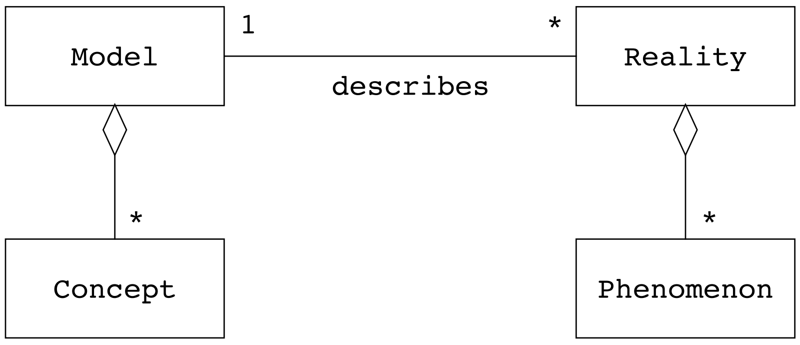


**FIGURE 70.** A system is a collection of real world phenomena. A model is a collection of concepts that represent the system's phenomena. Many models can represent different aspects of the same system. An unambiguous model corresponds to only one system.

system). It is ***complete*** if all possible scenarios through the system are described, included the behavior of the system in case of exceptional behavior from the part of the user or the external environment (i.e., all aspects of the system are represented in the requirements model). The system specification is ***consistent*** if it does not contradict itself. The system specification is ***unambiguous*** if exactly one system is defined (i.e., it is not possible to interpret the specification two or more different ways). Finally, it is ***realistic*** if the system can be implemented. This properties are illustrated with UML instance diagrams in Table 28.

**Table 28 Specification properties checked during validation.**

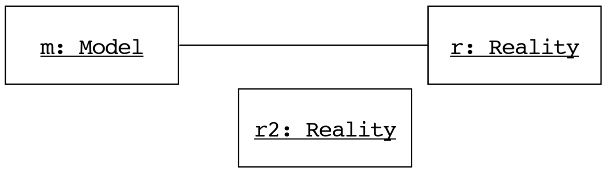
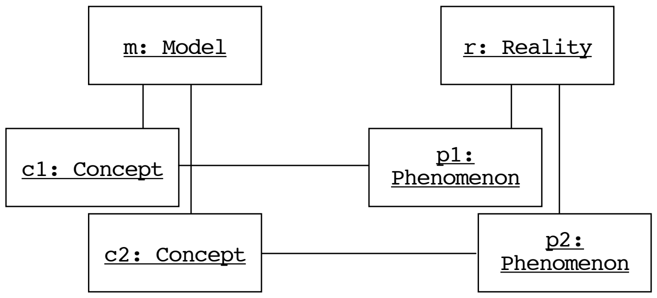
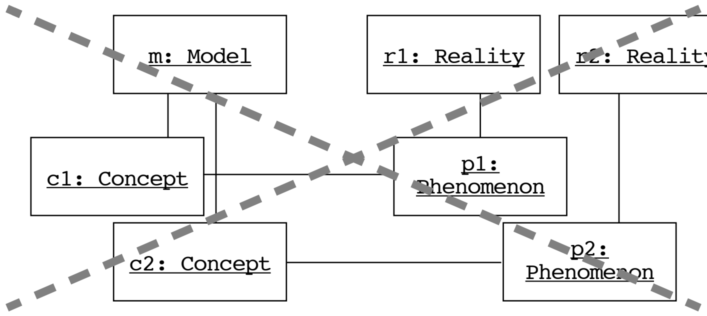
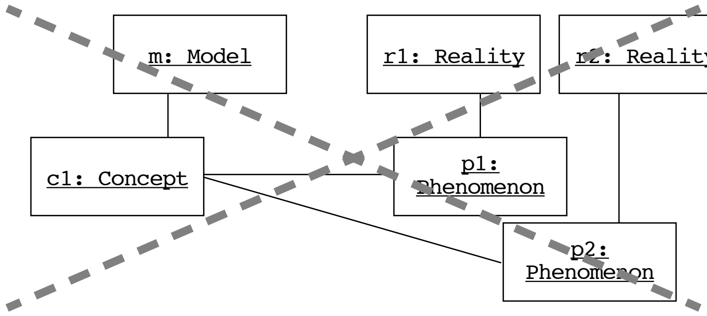
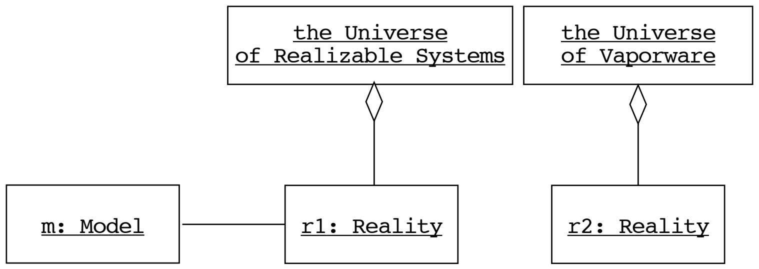
| | |
|---|---|
| ***Correct ness***- The model describes the reality of interest to the client, not another reality. |  |
| ***Complete ness***- Every phenomenon of interest is described in the model by a concept. |   m: Model<br>r: Reality<br>c1: Concept<br>p1: Phenomenon<br>c2: Concept<br>p2: Phenomenon |

**Table 28 Specification properties checked during validation.**

| | |
|---|---|
| ***Consistency***- All concepts in the model correspond to phenomena of the same reality. |  m: Model<br>r1: Reality<br>r2: Reality<br>c1: Concept<br>p1: Phenomenon<br>c2: Concept<br>p2: Phenomenon |
| ***Unambiguous***- All concepts in the model correspond to exactly one phenomenon. |  m: Model<br>r1: Reality<br>r2: Reality<br>c1: Concept<br>p1: Phenomenon<br>p2: Phenomenon |
| ***Realism*** The model describes a reality that can exist. |  the Universe of Realizable Systems<br>the Universe of Vaporware<br>m: Model<br>r1: Reality<br>r2: Reality |

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

Two more desirable properties of a system specification is that it is verifiable and traceable. The specification is ***verifiable*** if, once the system is built, a repeatable test can be designed to demonstrate that the system fulfills the requirement. For example, a mean time to failure of a hundred years for SatWatch would be difficult to achieve (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 system specification is ***traceable*** if each system function can be traced to its corresponding set of requirements. Traceability is not a constraint on the content of the specification, but rather, on its organization. Traceability facilitates the development of tests and the systematic validation of the design against the requirements.

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

A ***re-engineering*** project is the re-design and re-implementation of an existing system triggered by technology enablers or by new information flows [Hammer & Champy, 1993]. Sometimes, the functionality of the new system is extended, however, the essential purpose of the system remains the same. The requirements of the new system are extracted from an existing system.

An ***interface engineering*** project is the re-design of the user interface of an existing system. The legacy system is left untouched, except for its interface which is re-designed and re-implemented. This type of project is a re-engineering project in which the legacy system cannot be discarded without entailing high costs. In this section, we examine how requirements elicitation is performed in both situations.

In both re-engineering 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, user and client interviews. Note that 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.

In Section 6.3,we survey different approaches to requirements elicitation. In Section 6.2, we describe different representations provided by UML, such as scenarios and use cases, which can be used during requirements elicitation and requirements analysis.

## 6.2. Scenario and use cases in requirements elicitation.

In this section, we revisit the concepts of actor, scenario, and use case, which were introduced in Chapter 2, *Introduction to UML*. These are the basic representations that we use during requirements elicitation. We discuss heuristics and methods for extracting requirements from users and modeling the system in terms of these concepts. Unless mentioned otherwise, the methods described in this section have been adapted from Objectory [Objectory, 1993] and Responsibility-driven design [Wirfs-Brock et al., 1990].

### 6.2.1. Identifying actors

Actors represent external entities which interact with the system. An actor can be human or an external system. In the SatWatch example described in the introduction of this chapter (see Section 6.1), the watch owner, the GPS satellites, and the WebifyWatch serial device are actors (see Figure 71). They all interact and exchange information with the SatWatch. Note, however, they all have specific interactions with the SatWatch: the watch owner wears and looks at her watch; the GPS satellites are queried by the watch and return a signal; the WebifyWatch downloads new data into the watch. Actors define classes of functionality.

Consider a more complex example, the FRIEND system [FRIEND, 1994] we mentioned in Chapter 2, *Introduction to UML*. FRIEND is a distributed information system for incident response. It has many actors, including `FieldOfficer`, which 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. The FRIEND system supports both classes of actors by keeping track of incidents, resources, and task plans. It also has access to various databases, such as a hazardous materials database and emergency operations procedures. Both actors interact through different interfaces: `FieldOfficers` access FRIEND through a Newton personal assistant, `Dispatchers` access FRIEND through a workstation (see Figure 72).

Note that 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 definition of the actors. This serves both to define the boundaries of the system and to find all the perspectives from which the

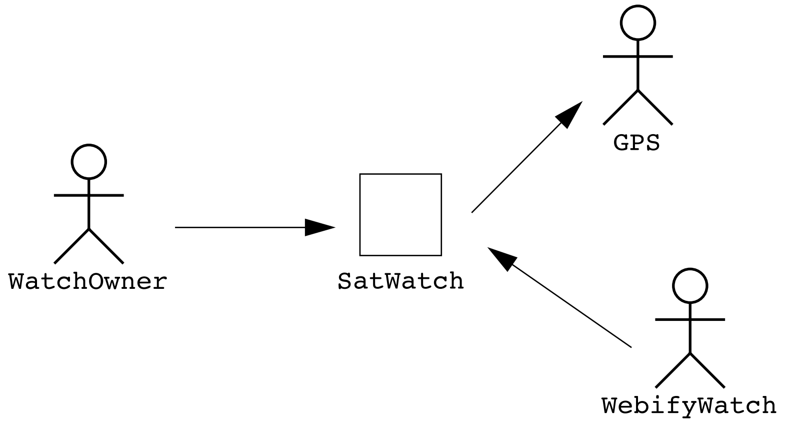


**FIGURE 71.** Actors for the SatWatch system. `WatchOwner` moves the watch (possibly across time zones) and consults it to know what time it is. SatWatch interacts with `GPS` to compute its position. `WebifyWatch` upgrades the data contained in the watch to reflect changes in time policy (e.g., changes in daylight saving time start and end dates).

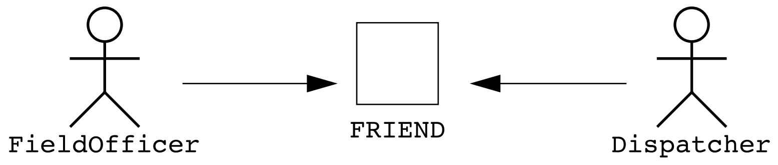


**FIGURE 72.** Actors of the FRIEND system. FieldOfficers not only have access to different functionality, they used different computers to access the system.

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

To find actors, the following heuristics can be used:[a]

- Which user groups require help from the system to perform their tasks?
- Which user groups are needed to execute the system's most obvious main functions?
- Which user groups are required to perform secondary functions, such as maintenance and administration?
- Will the system interact with any external hardware or software system?

a. These heuristics are taken from [Objectory, 1993].

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

### 6.2.2. 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."[1] A scenario is a concrete, focused, informal description of a single feature of the system used from the viewpoint of a single actor. The use of scenarios in requirements elicitation is a conceptual departure from the traditional representations which are generic and abstract. Traditional representations are centered around the system as opposed to the work that the system supports. Finally, their focus is on completeness, consistency, and accuracy, whereas scenarios are open ended and informal. A scenario-based approach cannot (and is not intended to) replace completely traditional approaches. It does, however, enhance requirements elicitation by providing a tool that is readily understandable to users and clients.

Figure 73 is an example of scenario for the FRIEND system [FRIEND, 1994], 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 instance. It does not attempt to describe all possible situations in which a fire incident is reported.

| | |
|---|---|
| *Scenario name* | `warehouseOnFire` |
| *Participating actor instances* | `bob`, `alice`: `FieldOfficer`<br>`john`: `Dispatcher` |

1. [Carroll, 1995], p 3.

| | |
|---|---|
| *Description* | 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.<br>2. Alice enters the address of the building, a brief description of its location (i.e., north west corner), and an emergency level. In addition to a fire unit, he requests several paramedic units on the scene given that area appear to be relatively busy. He confirms his input and waits for an acknowledgment.<br>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 creates allocates a fire unit and two paramedic units to the `Incident` site and sends their estimated arrival time (ETA) to Alice.<br>4. Alice received the acknowledgment and the ETA. |

**FIGURE 73.** `warehouseOnFire` scenario for the `ReportEmergency` use case.

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

- *As-is scenario* are used to described a current situation. During re-engineering for example, the current system can be 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* are used to described a future system, either re-engineered or designed from scratch. Visionary scenarios are used both as a design representation by developers as they refine their idea 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* are descriptions of 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 the case of requirements elicitation, developers and users may 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.

**Heuristics for finding scenarios and use cases:**[a]

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

a. Adapted from [Objectory, 1993].

Existing documents about the application domain should be used to answer these questions. These include user manuals of previous systems, procedures manuals, company standards, user notes and cheat sheets, user and client interviews. Scenarios should always be written using application domain terms, as opposed to the developers terms. As further insight in the application domain and the possibilities of the available technology are gained, scenarios are iteratively and incrementally refined to include sufficient detail for a complete system specification to be written. Drawing user interface mock-ups often help find omissions in the specification and help the users build a more concrete picture of the system. Note that at this stage, the user interface mock-ups should be used to define the functionality first, before resolving user interface issues. Putting too much emphasis on user interface details early may often result in functional issues being overlooked.

Once the users and developers have a good understanding of the system, scenarios are formalized into use cases.

### 6.2.3. Identifying use cases

In UML, a scenario is an instance of a use case, that is, a use case specifies all possible scenarios for a given class 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 resulted from the initiation of the use case.

Figure 74 depicts the use case `ReportEmergency` of which the scenario `warehouseOnFire` (see Figure 73) 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. This use case is general and encompasses a range of scenarios. For example, the

ReportEmergency use case could also apply to a road incident. Note also, however, that use cases can be written at varying levels of detail as in the case of scenarios. The ReportEmergency use case may be illustrative enough to describe how FRIEND supports reporting emergencies and obtain general feedback from the user, it does not provide sufficient detail for this function to be completely specified.

| | |
|---|---|
| *Use case name* | ReportEmergency |
| *Participating actor* | initiated by FieldOfficer<br>communicates with Dispatcher |
| *Entry condition* | 1. The FieldOfficer activates the "Report Emergency" function of her terminal. |
| *Description* | 2. FRIEND responds by presenting a form to the officer.<br>3. The FieldOfficer fills 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.<br>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. |
| *Exit condition* | 5. The FieldOfficer receives the acknowledgment and the selected response. |
| *Special 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 74.** An example of use case: ReportEmergency.

### 6.2.4. Refining use cases

Figure 75 is a refined version of the ReportEmergency use case. It has been extended to include details about the type of incidents that are known to FRIEND, detailed interactions indicating how the Dispatcher acknowledges the FieldOfficer (i.e., by sending as FRIENDgram), and illustrated with user interface mock-ups.

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

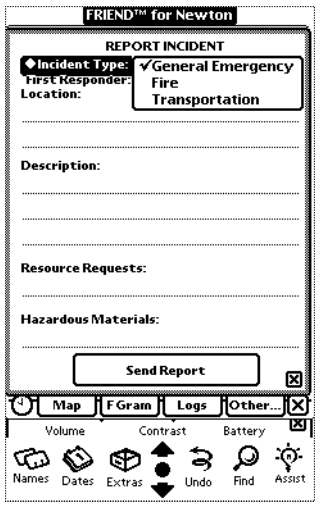
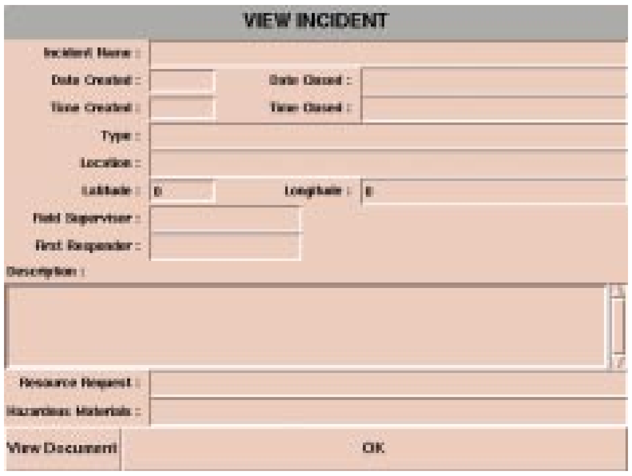
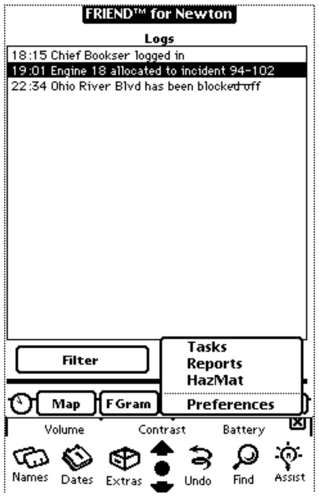
| *User interface mock-ups* | *Use case description* |
|---|---|
| *FieldOfficer station*   FRIEND™ for Newton<br>REPORT INCIDENT<br>◆Incident Type:<br>✓General Emergency<br>Fire<br>Transportation<br>First Responder:<br>Location:<br>Description:<br>Resource Requests:<br>Hazardous Materials:<br>Send Report<br>Map<br>FGram<br>Logs<br>Other...<br>Volume<br>Contrast<br>Battery<br>Names<br>Dates<br>Extras<br>Undo<br>Find<br>Assist | 1. The `FieldOfficer` activates the “Report Emergency” function of her terminal.<br>2. FRIEND responds by presenting a form to the officer. The form includes an emergency type menu (General emergency, fire, transportation), a location, incident description, resource request, and hazardous material fields.<br>3. The `FieldOfficer` fills the form, by specifying minimally the emergency type and description fields. The `FieldOfficer` may also describes possible responses to the emergency situation and request specific resources. Once the form is completed, the `FieldOfficer` submits the form by pressing the “Send Report” button, at which point, the `Dispatcher` is notified. |
| *Dispatcher station*   VIEW INCIDENT<br>Incident Name :<br>Date Created :<br>Date Closed :<br>Time Created :<br>Time Closed :<br>Type :<br>Location :<br>Latitude :<br>0<br>Longitude :<br>0<br>Field Supervisor :<br>First Responder :<br>Description :<br>Resource Request :<br>Hazardous Materials :<br>View Document<br>OK | 4. 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 `AllocateResource` use case) and acknowledges the emergency report by sending a FriendGRAM to the `FieldOfficer`. |

**FIGURE 75.** Refined description for the `ReportEmergency` use case.

*FieldOfficer station*



```
18:15 Chief Bookser logged in
19:01 Engine 18 allocated to incident 94-102
22:34 Ohio River Blvd has been blocked off
```

5. The `FieldOfficer` receives the acknowledgment and the selected response.

**FIGURE 75.** Refined description for the `ReportEmergency` use case.

implementation of the system starts, the cost of changing the system specification and adding new unforeseen functionality increases. Although it is generally not possible to freeze the requirements of the system until late in the development, developers and users should strive to address most requirements issues early. This entails lots of changes and experimentation during requirements elicitation. Note that many use cases are rewritten several times, others substantially refined, and yet others completely dropped. In order to save time, a lot of the exploration work can be done using scenarios and user interface mock-ups. Once the system specification becomes stable, traceability and redundancy issues are addressed by consolidating and reorganizing the actors and use cases.

**Heuristics for writing scenarios and use cases:**

- Use scenarios to communicate with users and to validate functionality.
- Refine a narrow vertical slice (i.e., one scenario) to understand the user's preferred style of interaction.
- Define a horizontal slice (i.e., many not very detailed scenarios) to define the scope of the system. Validate with the user.
- Use mock-ups as a visual support only, user interface design should occur once the functionality is sufficiently stable.
- Present the user with multiple alternatives (as opposed to extracting a single alternative from the user).
- Detail a broad vertical slice when the scope of the system and the user preferences are well understood. Validate with the user.

### 6.2.5. Identifying relationships among actors and use cases

Even a medium size system may have many use cases. Relationships among actors and use cases enable the developers and users to produce a intelligible model. Communication relationships between actors and use cases enable the system to be described in layers. Extends relationships allow exceptional and common flows of events to be described independently. Uses relationships help to suppress redundancy across use cases.

#### *Communication relationships between actors and use cases*

Communication relationships between actors and use cases denote 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. Thus, access control (i.e., which actor has access to which class functionality) can be represented at this level. The relationships between actors and use cases are usually identified at the same time as use cases are identified.

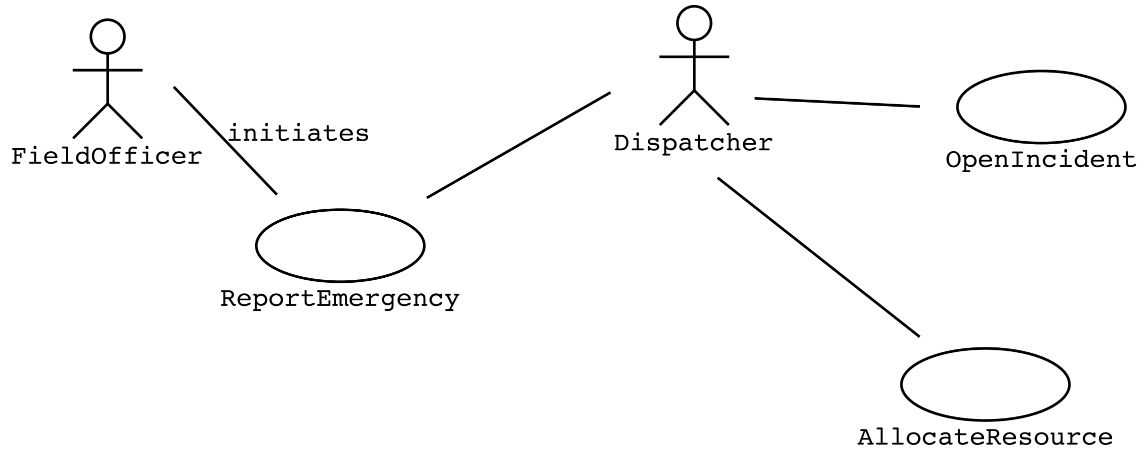


**FIGURE 76.** Example of communication relationships among actors and use cases in FRIEND. The `FieldOfficer` initiates the `ReportEmergency` use case and the `Dispatcher` initiates the `OpenIncident` and `AllocateResource` use cases. `FieldOfficers` cannot directly open an incident or allocate resources on their own.

### *Extends relationships between use cases*

A use case is said to extend another use case if the extended use case may include the behavior of the extension under certain conditions. For 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` is in a tunnel. The `FieldOfficer` station needs to notify the `FieldOfficer` that his form was not delivered and what measures he should take. The `HandleConnectionDown` use case is modeled as an extension of `ReportEmergency` (see Figure 77). The conditions under which the `HandleConnectionDown` use case is initiated are described in `HandleConnectionDown` as opposed to `ReportEmergency`. Separating exceptional and optional flow 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 speed, optimize the exceptional case for clarity). Note that both the extended use case and the extensions are complete use cases of their own. They must have a beginning and an end condition, and be understandable by the user as an independent whole.

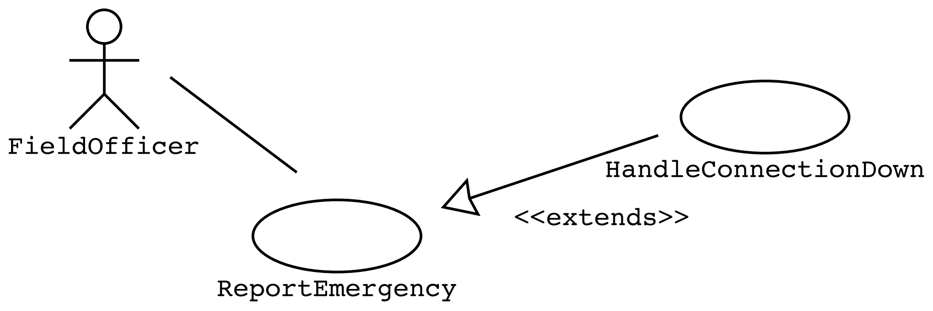


**FIGURE 77.** Example of use of extends relationship. `HandleConnectionDown` extends the `ReportEmergency` use case. The `ReportEmergency` use case becomes shorter, clearer, and solely focused on emergency reporting.

### *Uses relationships between use cases*

Shared behavior between use cases can be factored out using `uses` relationships. Assume for example that a Dispatcher needs to consult the city map when opening an incident (e.g, in order to assess which areas are at risk during a fire) and when allocating resources (e.g., to find which resources are closer 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 `AllocateResource` use cases.

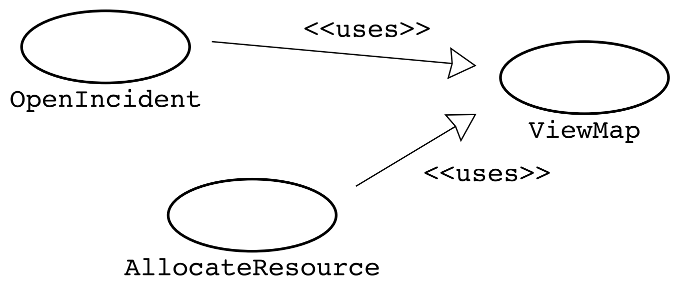


**FIGURE 78.** Example of uses relationships among use cases. `ViewMap` describes the flow of event for viewing a city map (e.g., scrolling, zooming, query by street name) and is used by both `OpenIncident` and `AllocateResource` use cases.

As in coding, factoring out shared behavior from use cases has many benefits, including shorter, clearer descriptions and decreased redundancies. Note that, unlike coding, 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 system specification across a large number of use cases makes the specification confusing to everyday users.

### *Extend vs. uses relationships*

`Uses` and `extends` are similar constructs, and initially, it may not be clear to the developer when to use each construct [Jacobson, 1995]. The main distinction between these constructions is the direction of the relationship. In the case of `uses`, the conditions under which the target use case is initiated are described in the initiating use case. In the case of `extends`, the conditions under which the extension is initiated are described in the extension. Figure 79 shows the `HandleConnectionDown` example described with a `uses` relationship (left column) and with an `extends` relationship (right column). In the left column, we need to insert text in two places in the event flow where the `HandleConnectionDown` 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 only need to describe the conditions under which the use case is invoked. Moreover, additional exceptional situations can be added with 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.

| ReportEmergency (uses relationship) | ReportEmergency (extends relationship) |
|---|---|
| 1. … | 1. … |
| 2. … | 2. … |
| 3. The `FieldOfficer` fills 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 HandleConnectionDown use case is used.* | 3. The `FieldOfficer` fills 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. 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 HandleConnectionDown use case is used.* | 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. … | 5. … |

**FIGURE 79.** Addition of `HandleConnectionDown` refinement to `ReportEmergency`. An extends relationship should be used for exceptional and optional flow of events as its yields a more modular description.

| HandleConnectionDown (uses relationship) | HandleConnectionDown (extends relationship) |
|---|---|
|  | *The HandleConnectionDown use case extends ReportEmergency when the connection between the FieldOfficer and the Dispatcher is 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?"). | 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 re-established. | 2. The situation is logged by the system and recovered when the connection is re-established. |
| 3. The `FieldOfficer` and the `Dispatcher` enter in contact through other means (e.g., telephone) and the `Dispatcher` initiates `ReportEmergency` from the `Dispatcher` station. | 3. The `FieldOfficer` and the `Dispatcher` enter in contact through other means (e.g., telephone) and the `Dispatcher` initiates `ReportEmergency` from the `Dispatcher` station. |

**FIGURE 79.** Addition of `HandleConnectionDown` refinement to `ReportEmergency`. An extends relationship should be used for exceptional and optional flow of events as its yields a more modular description.

**Heuristics for extends and uses relationships:**[a]

- Use extends for exceptional, optional, or seldom occurring behavior.
- Use uses for behavior that is shared across two or more use cases.

a. From [Objectory, 1993].

### 6.2.6. Identifying participating objects

Once use cases have been consolidated, developers start identifying the ***participating objects*** for each use cases. Participating objects form the basis for the analysis model, described in Chapter 7, *Requirements Analysis*.

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

System specifications, and later, user manuals, include a glossary section defining the terms of art used in the application domain. This glossary should be kept up-to-date as the system specification is expanded and revised. The benefits of a glossary are multiple: 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), terms have precise and clear official meanings.

The glossary of a system specification represents the initial version of the analysis model (described in Chapter 7, *Requirements Analysis*.). The analysis model itself is usually not used as means of communication between the users and the developers. However, the description of the objects (i.e., the definitions of the terms in the glossary) and their attributes are reviewed with the users.

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

**Heuristics for identifying participating objects:**

- Terms that developers or users need to clarify in order to understand the use case,
- Recurring nouns in the use cases (e.g., `Incident`),
- Real world entities that the system needs to keep track of (e.g., `FieldOfficer`, `Dispatcher`, `Resource`),
- Real world processes and procedures that the system needs to keep track of (e.g., `EmergencyOperationsPlan`),
- Use cases (e.g., `ReportEmergency`),
- Data sources or sinks (e.g., `Printer`),
- Interface artifacts (e.g., `Station`).
- *Always* use the user's terms.

During requirements elicitation, candidate objects should be generated for each use case. These are called the participating objects of the use case. If two use case 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 should be renamed to emphasize the difference. This process of consolidation aims eliminating any ambiguity in the terminology used.

Once candidate objects are identified and consolidated, the developers can use it as a check list for ensuring the set of identified use cases is complete.

**Heuristics for cross checking use cases and participating objects:**

- Which use cases creates 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 modifies and destroys this object (i.e., during 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 depend on this information?)

If new use cases are identified, they should be described, integrated in the model, and reviewed following the process we described before. Note that often in the requirements elicitation process, shifting perspectives introduces modifications in the system specification (e.g., finding new participating objects triggers the addition of new use cases; the addition of new use cases triggers the addition or refinement of new participating objects). This instability should be anticipated and encourage shifting perspectives. For the same reasons, time consuming tasks such as the description of exceptional cases and refinements of the user interfaces should be postponed until the set of use cases becomes stable.

## 6.3. Requirements methods survey

In this section, we briefly survey three methods that have been proposed for requirements or a subset thereof. These include:

- Joint Application Design (Section 6.3.1), a group session method that has been successfully used in IBM and elsewhere. The originality of the method lies in a team of users, clients, and developers developing requirements during a single workshop session.
- Quality Function Deployment (Section 6.3.2), a method that originated in the japanese car industry, emphasize the relationship between customer requirements and product features. The explicit focus on these relationships results in highly traceable requirements.
- Knowledge Analysis of Tasks (Section 6.3.3), a task analysis method that focuses on describing the problem domain in terms of tasks. Although KAT is not a requirements method per se, it results in critical information that can be used to improve the usability of a system and reduce the redesign efforts.

We describe these method to illustrate a broad variety of approaches to requirements and problem domain analysis.

### 6.3.1. Joint Application Design® (JAD)

Joint Application Design (JAD) is a requirements method developed in IBM at the end of the seventies. Its originality lies in that the requirements work is done in one single workshop session including all stakeholders. Users, clients, developers, and a trained session leader sit together in one room to present their view point, listen to other viewpoint, negotiate, and agree to a mutually acceptable solution. The outcome of the workshop, the final JAD document, is a system complete specification document including definitions of data elements, work flows, screens, and reports. In addition, the final JAD document represents an agreement between users, clients, and developers, and thus minimizes requirements changes late in the development process.

JAD is composed of five phases (summarized in Figure 80):

1. ***Project definition.*** During this phase, the JAD leader interviews managers and clients to determine the objectives and the scope of the project. The findings from the interviews are collected in the *Management Definition Guide*. During this phase, the JAD leader forms a team composed of users, clients, and developers. All stakeholders are represented and the participants are able to make binding decisions.
2. ***Research***. During this phase, the JAD leader interviews present and future users, gathers domain information, describes the work flows. The JAD leader also starts a list of issues that will need to be addressed during the session. The primary results of the Research phase are a *Session Agenda* and a *Preliminary Specification* listing work flow and system information.
3. ***Preparation.*** During this phase, the JAD leader prepares the session. The JAD leader creates a *Working Document*, first draft of the final document, an agenda for the session, and any number of overhead slides or flip charts representing information gathered during the Research phase.
4. ***Session***. During this phase, the JAD leader guides the team in creating the system specification. A JAD session lasts for three to five days. The team defines and agrees on the work flow, the data elements, the screens, and the reports of the system. All decisions are documented by a scribe filling JAD forms.
5. ***Final document***. The JAD leader prepares the *Final Document*, revising the working document to include all decisions made during the session. The Final Document represents a complete specification of the system as agreed during the session. The Final Document is distributed to the session's participants for review. The participants then meet for a one to two hour meeting to discuss the reviews and finalize the document.

JAD has been successfully used in IBM and other companies since the mid eighties. JAD leverages off group dynamics to improve communication among participants and accelerate

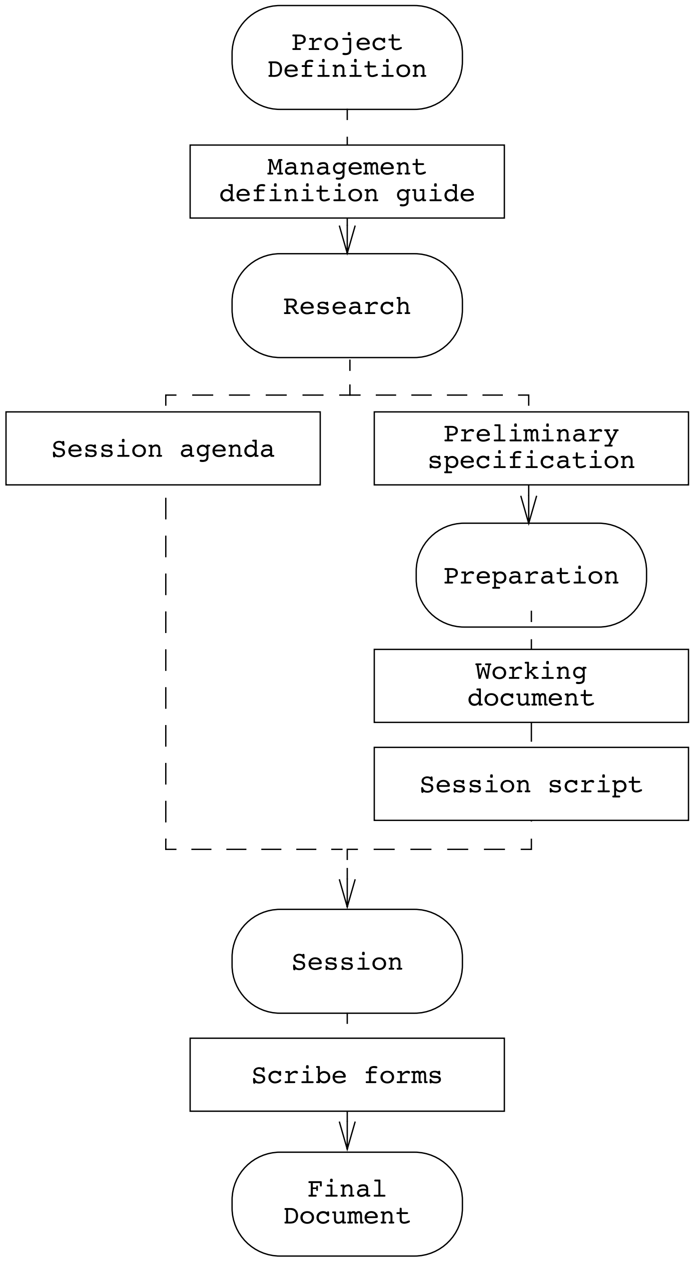


**FIGURE 80.** Phases of JAD (UML activity diagram). The heart of JAD is the Session phase during which all stakeholders design and agree to a system specification. The phases prior to the Session maximizes its efficiency. The production of the final document ensures that the decisions made during the Session are captured.

consensus. At the end of a JAD session, developers are more knowledgeable of users needs, and users are more knowledgeable of development trade-offs. Additional gains result from a reduction of re-design activities downstream. Because of its reliance on social dynamics, the success of a JAD session often depend on the qualifications of the JAD leader as a meeting facilitator.

### 6.3.2. Quality Function Deployment (QFD)

Quality Function Deployment (QFD) is a method developed in the japanese car industry in the sixties as a means for translating customer requirements into specific technical features [Sullivan, 1986]. Its use in software engineering has been encouraged at the end of the 1980s [Zultner, 1993]. As JAD, QFD is based on group sessions during which stakeholders trade-off different product features to meet customer requirements. QFD session focus on constructing a “House of Quality” for the system Figure 81.

The customer requirements are first listed as rows in the House of Quality. Customer requirements are gathered from a variety of sources such as existing products, user interviews, focus groups. QFD does not provide any guidance for this phase. Features are then proposed to satisfy one or more customer requirements. Proposed product features are listed across the top of the matrix. They should be listed as a quantifiable property of the product. The cells of the matrix are then filled with symbols indicating how strongly a feature supports a requirements: a ● indicates a strong relationship between a requirements and feature, ❍ indicates a medium relationship, a ▲ indicates a weak relationship, no symbol indicates no relationship. If any single row have no symbols or only a series of weak symbols, a customer requirements is not met and product features have to be added or replaced. The roof of the house of quality is filled with symbols to indicate relationships between features. Features that are strongly related need to be designed together. Features that are not related can be added or removed from the product independently.

The market evaluation of the product (i.e., right hand side columns) is then added to the matrix. The first column is the customer rating of each requirements. Requirements that are necessary or highly desirable are given a high grade. Requirements that represent bells and whistles are given low grades. The second column of the market evaluation is the evaluation of the product against competing products, requirement by requirement. The information from the first and second column is used to identify selling points for the proposed products.

QFD is not limited to requirements. A similar process can be repeated for each phase of design, during which the columns of a previous phase become the rows of the next. The roof of the house of quality can be used to identify independent features and partition the design into several subsystems.

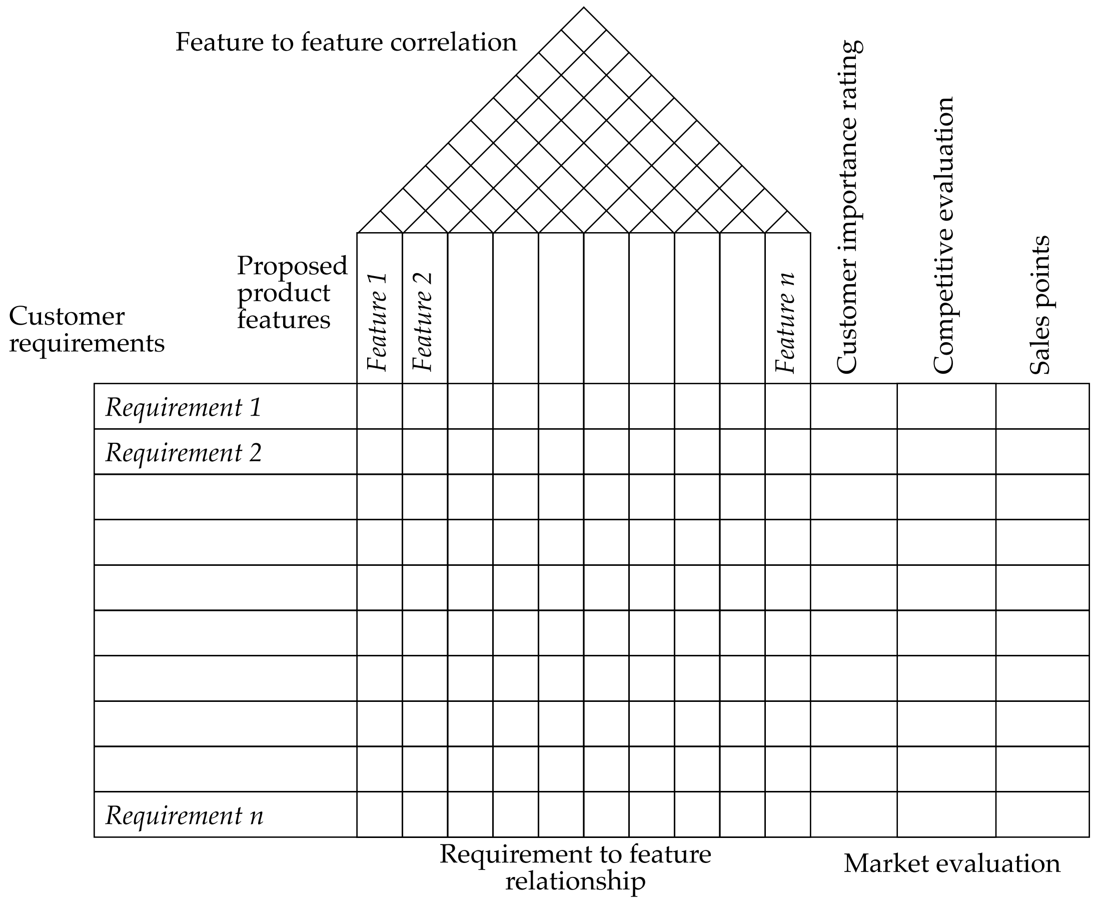


**FIGURE 81.** QFD House of Quality. Customer requirements are listed as rows. Product features are listed as columns. An important activity in QFD is to fill the cells of the matrix indicating how features support requirements.

Although QFD is much older than JAD, its introduction and use in software engineering is recent and still evolving.

### 6.3.3. Knowledge Analysis of Tasks (KAT)

Task analysis originated in the United States and the United Kingdom in the fifties and sixties [Johnson, 1992]. Initially, task analysis was not concerned with requirements or system design. Task analysis was used to identify how people should be trained. In the U.S., the military was primarily interested in task analysis. In the U.K., the Department of Trade and Industry were interested in task analysis for developing methods to enable people to move across industries. More recently, task analysis become important in the field of Human Computer Interaction (HCI) for identifying and describing the user tasks that a system should support.

The Knowledge Analysis of Tasks (KAT) is a task analysis method proposed by [Johnson, 1992]. It is concerned with collecting data from a variety of sources (e.g., interviews, protocol analysis, textbooks, standard procedures), analyzing these data to identify individual elements involved in the task (e.g., objects, actions, procedures, goals, and subgoals), and constructing a model of the overall knowledge used by people accomplishing the task of interest. KAT is similar to object-oriented analysis in that it represents the problem domain in terms of objects and actions on them. KAT is different in that it represents explicitly the goal and subgoals of tasks and procedures.

KAT can be summarized by the five following steps:

1. ***Identifying objects and actions.*** Object and actions associated with them are identified using similar techniques as object identification in object-oriented analysis, such as analyzing textbooks, manuals, rule books, reports, interviewing the task performer, observing the task performer.
2. ***Identifying procedures.*** A procedure is a set of actions, a pre-condition necessary to triggering the procedure, and a post condition. Actions may be partially ordered. Procedures can be identified by writing scenarios, observing the task performer, asking the task performer to select and order cards on which individual actions are written.
3. ***Identifying goals and subgoals.*** A goal is a state to be achieved for the task to be successful. Goals can be identified through interview during the performance of a task or afterwards. Subgoals are identified by decomposing goals.
4. ***Identifying typicality and importance.*** Each identified element is rated according to how frequently it is encountered and to whether it is necessary for accomplishing a goal.
5. ***Constructing a model of the task.*** The information gathered above is generalized to account for common features across tasks. Corresponding goals, procedures, and objects are related using a textual notation or a graph. Finally, the model is validated with the task performer.

Although task analysis and KAT are not requirements methods per se, they can greatly benefit the requirements process in several ways:

- During elicitation, they provide techniques for eliciting and describing problem domain knowledge, including information such as typicality and importance of specific actions; the end result is understandable by the task performer.

- When defining the boundaries of a system, task models assist in determining which parts of the task should remain manual and which parts should be automated; moreover, the task model may reveal problem areas in the current system.
- When designing the interface of the system, task models may serve as a source of inspiration for metaphors understandable by the user [Nielsen, 1994].

## 6.4. Summary

Requirements elicitation is the most difficult part of the software development process in general and of requirements analysis in particular. The main difficulty lies in that knowledge relevant to the system development is distributed across several different groups of participants. Moreover, each group has a different background (i.e., users know about the application domain; developers know about system development) and may use incompatible terminologies.

In this chapter, we presented scenarios and use cases as a bridge between users and developers for representing application domain knowledge. We have also surveyed a number of methods that explicitly address the user developer gap. In practice, a project may select combinations of requirements methods to maximize its benefits.

In the next chapter (Chapter 7, *Requirements Analysis*), we examine methods for analyzing and formalizing requirements. These techniques are used for clarifying requirements and ensuring their completeness and consistency. We also address issues of documentation and management in the next chapter.

## 6.5. Exercises

1. Modify the `ReportEmergency` use case (described in Figure 79) to include Help functionality. Justify your choices.
2. Write a scenario and its corresponding use case describing how the `WatchOwner` actor interacts with her SatWatch (see Section 6.1).
3. Consider this book as a system. Draw a UML use case diagram depicting the actors and selected use cases of this system. Consider also past interactions with the book.
4. Explain why multiple choice questionnaires for extracting information from the user is not effective or desirable in the scope of requirements elicitation.
5. From your point of view, describe the strengths and weaknesses of users during the requirements elicitation process. Describe also the strengths and weaknesses of developers during the requirements elicitation process.

## 6.6. References

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

[FRIEND, 1994] FRIEND Project Documentation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 1992-95.

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

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

[Jacobson, 1995] I. Jacobson. "The Use-Case Construct in Object-Oriented Software Engineering," In *Scenario-Based Design: Envisioning Work and Technology in System Development,* J. M. Carroll (Ed), John Wiley & Sons, Inc., New York, 1995.

[Johnson, 1992] P. Johnson, *Human Computer Interaction: psychology, task analysis and software engineering,* McGraw-Hill International, London, 1992.

[Macaulay, 1996] L. Macaulay, *Requirements Engineering,* Springer Verlag, London, U.K., 1996.

[Nielsen, 1994] J. Nielsen, *Usability Engineering,* Academic Press, Boston, MA, 1994.

[Objectory, 1993] *Objectory 3.3,* Objective Systems SF AB, Kista, Sweden, 1993.

[Sullivan, 1986] L.P. Sullivan, "Quality Function Deployment," *Quality Progress,* vol. 19, no. 6, pp. 39-50, 1986.

[UML Summary, 1997] *UML Summary,* http://www.rational.com/uml, Rational Software Corporation, 1997.

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

[Wirfs-Brock, 1995] R. Wirfs-Brock, "Design Objects and Their Interactions: A Brief Look at Responsibility-Driven Design," In *Scenario-Based Design: Envisioning Work and Technology in System Development,* J. M. Carroll (Ed), John Wiley & Sons, Inc., New York, 1995.

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

[Zultner, 1993] R. E. Zultner, "TQM for Technical Teams," *Communications of the ACM,* vol 36, no. 10, pp. 79-91, 1993.