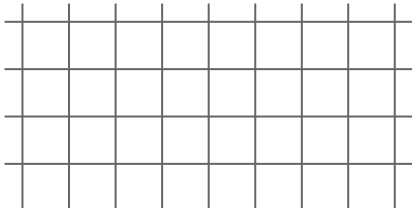
ICT - 4231
OBJECT ORIENTED SOFTWARE ENGINEERING

LECTURE - 4

CHAPTER – 4 : REQUIREMENTS ELICITATION

Moinul Islam, IIT, JU



CONTENTS

- Introduction
- An Overview of Requirements Elicitation
- Requirements Elicitation Concepts
- Requirements Elicitation Activities
- Managing Requirements Elicitation

TEXT BOOK

- Object-oriented Software Engineering: Using UML, Patterns, and Java; 3rd Edition; Bernd Bruegge, Allen H. Dutoit

INTRODUCTION

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

-----Douglas Adams, in Mostly Harmless

INTRODUCTION

- A **requirement** is a feature that the system must have or a constraint that it must satisfy to be accepted by the client. **Requirements engineering** aims at defining the requirements of the system under construction.
- Requirements engineering includes two main activities;
  - **requirements elicitation**, which results in the specification of the system that the client understands, and
  - **analysis**, which results in an analysis model that the developers can unambiguously interpret.

INTRODUCTION (CONT.)

- Requirements elicitation is challenging due to the collaboration of diverse groups with varying expertise. Clients and users understand their domain but lack software development experience, while developers have system-building expertise but may not grasp user environments.
- Scenarios and use cases bridge this gap, describing system use in natural language. The focus is on scenario-based elicitation, where developers observe and interview users, creating as-is scenarios and visionary scenarios. The client and users validate these through reviews and prototype testing.
- As system definition matures, consensus is reached on a requirements specification encompassing functional and nonfunctional requirements, use cases, and scenarios. Effective communication is crucial to avoid costly errors, such as missing functionality or misleading interfaces.
- Elicitation methods aim to enhance communication, involving observation, prototype construction, and user feedback. Prototypes offer a glimpse into system usage without full functionality implementation.

AN OVERVIEW OF REQUIREMENTS ELICITATION

- Requirements elicitation collaboratively defines the system's purpose, creating a contract between clients and developers known as a requirements specification. Concurrently, during analysis, a formal or semiformal notation results in an analysis model representing the same information.
- While the requirements specification aids communication with clients and users in natural language, the analysis model facilitates communication among developers using a more formal approach.
- Both aim to accurately represent external aspects of the system. The process, iterative and concurrent, focuses solely on the user's perspective, covering functionality, interaction, error handling, and environmental conditions. Aspects like system structure, implementation technology, and design are excluded from the requirements.

AN OVERVIEW OF REQUIREMENTS ELICITATION

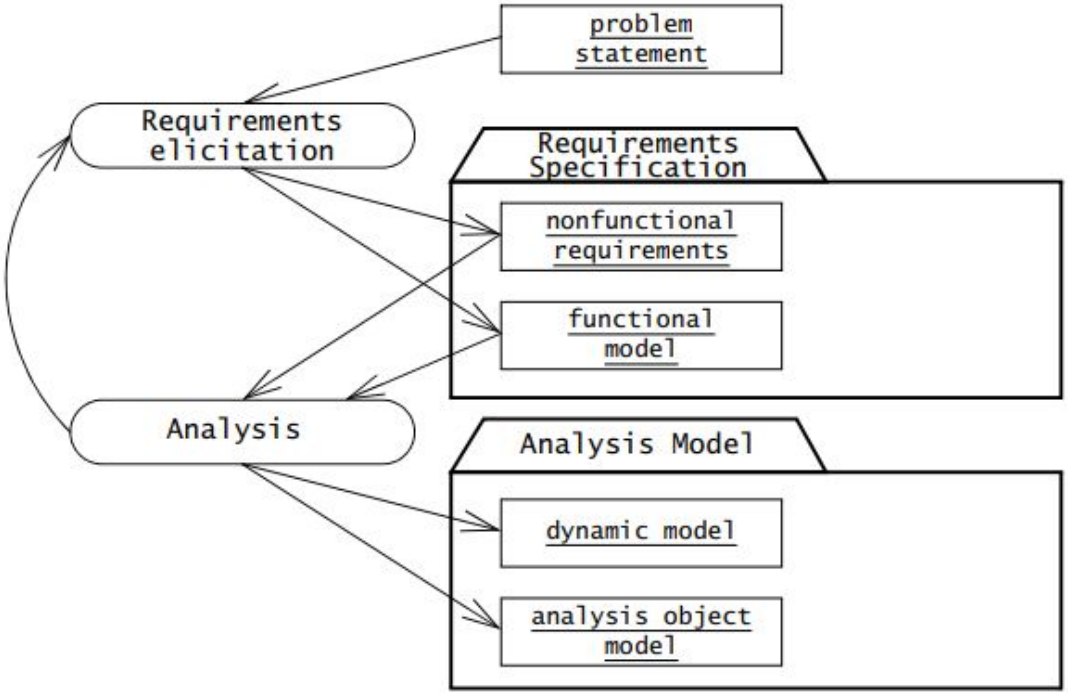


Figure 4-1 Products of requirements elicitation and analysis (UML activity diagram).

AN OVERVIEW OF REQUIREMENTS ELICITATION

- Requirements elicitation includes the following activities:
  - Identifying actors.
  - Identifying scenarios.
  - Identifying use cases.
  - Refining use cases.
  - Identifying relationships among use cases.
  - Identifying nonfunctional requirements.
- 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.
- 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

REQUIREMENTS ELICITATION CONCEPTS

Requirements elicitation concepts includes the following concepts:

- Functional Requirements
- Nonfunctional Requirements
- Completeness, Consistency, Clarity, and Correctness
- Realism, Verifiability, and Traceability
- Greenfield Engineering, Reengineering, and Interface Engineering

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,

SatWatch is a wrist watch that displays the time based on its current location. SatWatch uses GPS satellites (Global Positioning System) to determine its location and internal data structures to convert this location into a time zone.
The information stored in SatWatch and its accuracy measuring time is such that the watch owner never needs to reset the time. SatWatch adjusts the time and date displayed as the watch owner crosses time zones and political boundaries. For this reason, SatWatch has no buttons or controls available to the user.
SatWatch determines its location using GPS satellites and, as such, suffers from the same limitations as all other GPS devices (e.g., inability to determine location at certain times of the day in mountainous regions). During blackout periods, SatWatch assumes that it does not cross a time zone or a political boundary. SatWatch corrects its time zone as soon as a blackout period ends.
SatWatch has a two-line display showing, on the top line, the time (hour, minute, second, time zone) and on the bottom line, the date (day, date, month, year). The display technology used is such that the watch owner can see the time and date even under poor light conditions.
When political boundaries change, the watch owner may upgrade the software of the watch using the WebifyWatch device (provided with the watch) and a personal computer connected to the Internet.

- 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).

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 provides the following categories of nonfunctional requirements:

- **Usability** is centered on how easily users can learn, operate, and interpret outputs within a system or component. This encompasses factors like adhering to user interface conventions, determining the scope of online help, and specifying the level of user documentation.
- **Reliability** focuses on a system's ability to perform its necessary functions under specified conditions for a predetermined period. This involves aspects such as mean time to failure, the system's capacity to detect specified faults, and its resilience against security attacks.
- **Performance** requirements are concerned with quantifiable attributes, including how quickly a system reacts to user inputs (response time), the amount of work it can accomplish within a specified time (throughput), its operational and accessible degree (availability), and the precision of its operations.
- **Supportability** requirements address the ease of implementing changes to the system post-deployment. This encompasses adaptability to handle additional application domain concepts, maintainability for incorporating new technology or fixing defects, and internationalization to accommodate various international conventions such as languages, units, and number formats. The ISO 9126 standard, akin to the FURPS+ model, replaces supportability with maintainability and portability, focusing on the ease of system transferability and adaptability across different environments.

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).

COMPLETENESS, CONSISTENCY, CLARITY, AND CORRECTNESS

Table 4-1 Specification properties checked during validation.

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

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).
- 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.

GREENFIELD ENGINEERING, REENGINEERING, AND INTERFACE ENGINEERING

Requirements elicitation activities can be classified into three categories, depending on the source of the requirements.

- In **greenfield engineering**, the development starts from scratch—no prior system exists—so the requirements are extracted from the users and the client. A greenfield engineering project is triggered by a user need or the creation of a new market. SatWatch is a greenfield engineering project.
- A **reengineering** project is the redesign and reimplementation of an existing system triggered by technology enablers or by business processes. Sometimes, the functionality of the new system is extended, but the essential purpose of the system remains the same. The requirements of the new system are extracted from an existing system.
- An **interface engineering** project is the redesign of the user interface of an existing system. The legacy system is left untouched except for its interface, which is redesigned and reimplemented. This type of project is a reengineering project in which the legacy system cannot be discarded without entailing high costs

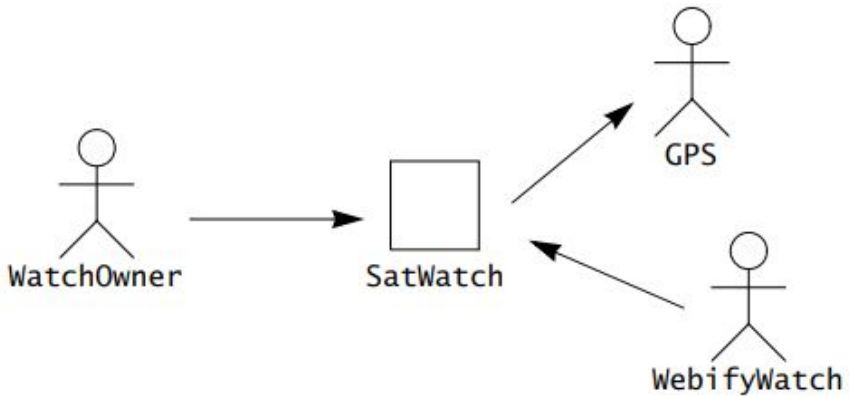
REQUIREMENTS ELICITATION ACTIVITIES

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

IDENTIFYING ACTORS

- **Actors** represent external entities that interact with the system. An actor can be human or an external system. In the SatWatch example, the watch owner, the GPS satellites, and the WebifyWatch serial device are actors (see Figure 4-4). They all exchange information with the SatWatch. Note, however, that they all have specific interactions with SatWatch: the watch owner wears and looks at her watch; the watch monitors the signal from the GPS satellites; the WebifyWatch downloads new data into the watch. Actors define classes of functionality.



IDENTIFYING ACTORS

- Consider a more complex example, FRIEND, a distributed information system for accident management. It includes many actors, such as FieldOfficer, who represents the police and fire officers who are responding to an incident, and Dispatcher, the police officer responsible for answering 911 calls and dispatching resources to an incident. FRIEND supports both actors by keeping track of incidents, resources, and task plans. It also has access to multiple databases, such as a hazardous materials database and emergency operations procedures. The FieldOfficer and the Dispatcher actors interact through different interfaces: FieldOfficers access FRIEND through a mobile personal assistant, Dispatchers access FRIEND through a workstation (see Figure 4-5).
- Actors are role abstractions and do not necessarily directly map to persons. The same person can fill the role of FieldOfficer or Dispatcher at different times. However, the functionality they access is substantially different. For that reason, these two roles are modeled as two different actors.
- The first step of requirements elicitation is the identification of actors. This serves both to define the boundaries of the system and to find all the perspectives from which the developers need to consider the system. When the system is deployed into an existing organization (such as a company), most actors usually exist before the system is developed: they correspond to roles in the organization.

IDENTIFYING ACTORS

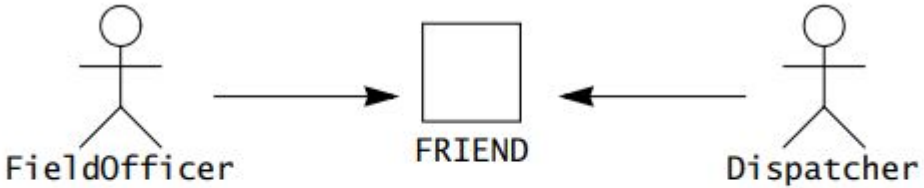


Figure 4-5 Actors of the FRIEND system. FieldOfficers not only have access to different functionality, they use different computers to access the system.

Questions for identifying actors

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

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

IDENTIFYING SCENARIOS

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

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

IDENTIFYING SCENARIOS

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

IDENTIFYING SCENARIOS

Questions for identifying scenarios

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

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

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

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.

IDENTIFYING USE CASES

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

IDENTIFYING USE CASES

Simple Use Case Writing Guide

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

Figure 4-8 Example of use case writing guide.

IDENTIFYING USE CASES

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

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

REFINING USE CA

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

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

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

REFINING USE CASES

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.

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 AMONG ACTORS AND USE CASES

- Communication relationships between actors and use cases represent the flow of information during the use case. The actor who initiates the use case should be distinguished from the other actors with whom the use case communicates.
- By specifying which actor can invoke a specific use case, we also implicitly specify which actors cannot invoke the use case. Similarly, by specifying which actors communicate with a specific use case, we specify which actors can access specific information and which cannot.
- Thus, by documenting initiation and communication relationships among actors and use cases, we specify access control for the system at a coarse level. The relationships between actors and use cases are identified when use cases are identified.
- Figure 4-11 depicts an example of communication relationships in the case of the FRIEND system. The «initiate» stereotype denotes the initiation of the use case by an actor, and the «participate» stereotype denotes that an actor (who did not initiate the use case) communicates with the use case.

COMMUNICATION RELATIONSHIPS AMONG ACTORS AND USE CASES

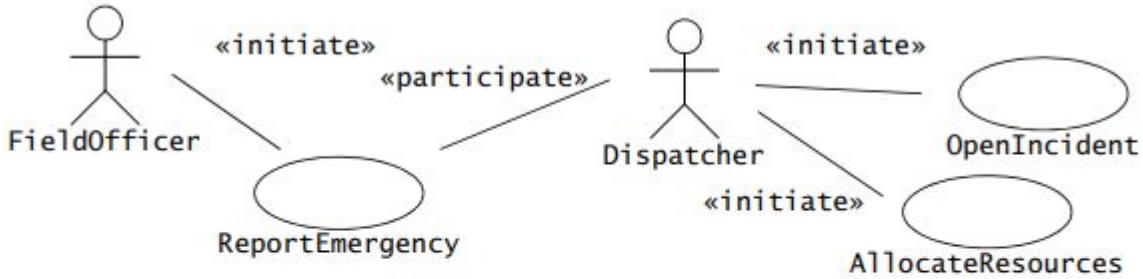


Figure 4-11 Example of communication relationships among actors and use cases in FRIEND (UML use case diagram). The FieldOfficer initiates the ReportEmergency use case, and the Dispatcher initiates the OpenIncident and AllocateResources use cases. FieldOfficers cannot directly open an incident or allocate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

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.
- The ConnectionDown use case is modeled as an extension of ReportEmergency (see Figure 4-12). The conditions under which the ConnectionDown use case is initiated are described in ConnectionDown as opposed to ReportEmergency. Separating exceptional and optional flows of events from the base use case has two advantages.
- First, the base use case becomes shorter and easier to understand. Second, the common case is distinguished from the exceptional case, which enables the developers to treat each type of functionality differently (e.g., optimize the common case for response time, optimize the exceptional case for robustness). Both the extended use case and the extensions are complete use cases of their own. They each must have entry and end conditions and be understandable by the user as an independent whole.

EXTEND RELATIONSHIPS BETWEEN USE CASES

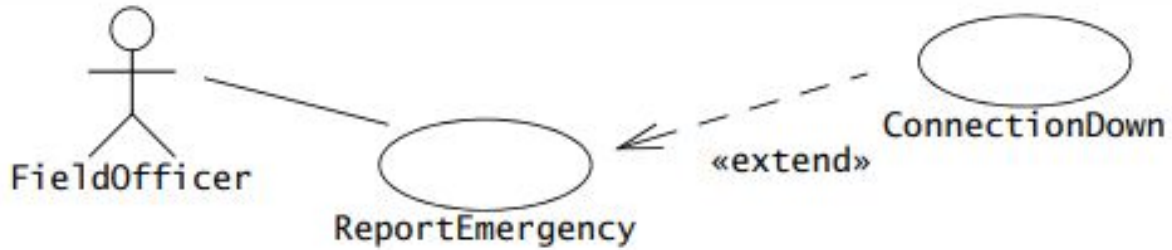


Figure 4-12 Example of use of extend relationship (UML use case diagram). `ConnectionDown` extends the `ReportEmergency` use case. The `ReportEmergency` use case becomes shorter and solely focused on emergency reporting.

INCLUDE RELATIONSHIPS BETWEEN USE CASES

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

INCLUDE RELATIONSHIPS BETWEEN USE CASES

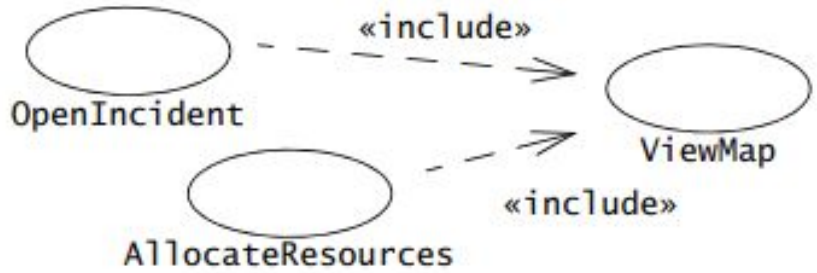


Figure 4-13 Example of include relationships among use cases. `ViewMap` describes the flow of events for viewing a city map (e.g., scrolling, zooming, query by street name) and is used by both `OpenIncident` and `AllocateResources` use cases.

EXTEND VERSUS INCLUDE RELATIONSHIPS

Heuristics for extend and include relationships

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

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

IDENTIFYING INITIAL ANALYSIS OBJECTS

`ReportEmergency` (include relationship)	`ReportEmergency` (extend relationship)
1. ...	1. ...
2. ...	2. ...
3. The `FieldOfficer` completes the form by selecting the emergency level, type, location, and brief description of the situation. The `FieldOfficer` also describes possible responses to the emergency situation. Once the form is completed, the `FieldOfficer` submits the form, at which point, the `Dispatcher` is notified. *If the connection with the `Dispatcher` is broken, the `ConnectionDown` use case is used.*	3. The `FieldOfficer` completes the form by selecting the emergency level, type, location, and brief description of the situation. The `FieldOfficer` also describes possible responses to the emergency situation. Once the form is completed, the `FieldOfficer` submits the form, at which point, the `Dispatcher` is notified.
4. If the connection is still alive, the `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. The `Dispatcher` selects a response and acknowledges the emergency report. *If the connection is broken, the `ConnectionDown` use case is used.*	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. ...

IDENTIFYING INITIAL ANALYSIS OBJECTS

ConnectionDown (include relationship)	ConnectionDown (extend relationship)
	The ConnectionDown use case extends any use case in which the communication between the FieldOfficer and the Dispatcher can be lost.
1. The FieldOfficer and the Dispatcher are notified that the connection is broken. They are advised of the possible reasons why such an event would occur (e.g., "Is the FieldOfficer station in a tunnel?"). 2. The situation is logged by the system and recovered when the connection is reestablished. 3. The FieldOfficer and the Dispatcher enter in contact through other means and the Dispatcher initiates ReportEmergency from the Dispatcher station.	1. The FieldOfficer and the Dispatcher are notified that the connection is broken. They are advised of the possible reasons why such an event would occur (e.g., "Is the FieldOfficer station in a tunnel?"). 2. The situation is logged by the system and recovered when the connection is reestablished. 3. The FieldOfficer and the Dispatcher enter in contact through other means and the Dispatcher initiates ReportEmergency from the Dispatcher station.

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

IDENTIFYING INITIAL ANALYSIS OBJECTS

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.

IDENTIFYING INITIAL ANALYSIS OBJECTS

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

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

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

IDENTIFYING INITIAL ANALYSIS OBJECTS

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

Heuristics for cross-checking use cases and participating objects

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

IDENTIFYING NONFUNCTIONAL REQUIREMENTS

- Nonfunctional requirements are aspects of a system not directly tied to its functional behavior. These requirements encompass various issues, including user interface aesthetics, response times, and security considerations. They are defined alongside functional requirements, given their significant impact on system development and cost.
- For instance, in the context of an air traffic controller using a mosaic display to track planes, the system's ability to handle a specific number of aircraft influences performance and costs. Balancing the capacity to display multiple aircraft against system complexity and cost is crucial for effective air traffic control.
- Nonfunctional requirements can have unexpected effects on user tasks. Effective collaboration between clients and developers is essential to identify critical attributes that may be challenging to achieve but are vital for user functionality. In the mosaic display example, the number of aircraft handled affects various aspects such as icon size, aircraft identification features, and data refresh rate.
- The collection of nonfunctional requirements often involves conflicting demands. For instance, in the case of the SatWatch depicted in Figure 4-3, conflicting requirements include the need for accuracy without requiring time resets and a low unit cost, as high accuracy typically correlates with increased costs. Resolving such conflicts involves prioritizing nonfunctional requirements collaboratively between the client and developer, ensuring consistent addressing during system realization.

IDENTIFYING NONFUNCTIONAL REQUIREMENTS

Table 4-3 Example questions for eliciting nonfunctional requirements.

Category	Example questions
Usability	• What is the level of expertise of the user? • What user interface standards are familiar to the user? • What documentation should be provided to the user?
Reliability *(including robustness, safety, and security)*	• How reliable, available, and robust should the system be? • Is restarting the system acceptable in the event of a failure? • How much data can the system loose? • How should the system handle exceptions? • Are there safety requirements of the system? • Are there security requirements of the system?
Performance	• How responsive should the system be? • Are any user tasks time critical? • How many concurrent users should it support? • How large is a typical data store for comparable systems? • What is the worse latency that is acceptable to users?
Supportability *(including maintainability and portability)*	• What are the foreseen extensions to the system? • Who maintains the system? • Are there plans to port the system to different software or hardware environments?

IDENTIFYING NONFUNCTIONAL REQUIREMENTS

Implementation	• Are there constraints on the hardware platform? • Are constraints imposed by the maintenance team? • Are constraints imposed by the testing team?
Interface	• Should the system interact with any existing systems? • How are data exported/imported into the system? • What standards in use by the client should be supported by the system?
Operation	• Who manages the running system?
Packaging	• Who installs the system? • How many installations are foreseen? • Are there time constraints on the installation?
Legal	• How should the system be licensed? • Are any liability issues associated with system failures? • Are any royalties or licensing fees incurred by using specific algorithms or components?

THANK YOU