SYSC-3020 — Introduction to Software Engineering

Software Requirements Elicitation and Specifications

Quotes from a disgruntled customer:

- They are not listening to what I want for my system. They are giving me the system that they want to build, not the one that I want.
- We often don't know how to say what we want – we know it but don't know that we do – so it's their job to listen to us, organize it and say it back to us, better than how we ourselves could say it.

Software Requirements Elicitation and Specifications

- Fundamentals
  - Motivation and Goals
  - Requirement Engineering
  - Functional vs. Non-Functional
  - Defining Software System Scope
  - Specifying a Use Case
  - Use case relationships
  - Pitfalls
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))
- Documentation

What is a Requirement ?

- [Lethbridge] A requirement is a statement about what the proposed system will do that all stakeholders agree must be made true in order for the customer's problem to be adequately solved.
  - Statement: brief, concise, fact-based.
    - Collection of requirements = requirements document
  - Do: What tasks the system will perform.
    - Does not describe implementation (the "how").
  - All stakeholders: Fundamental purpose is communication vis-à-vis agreements/contracts
  - Problem: System must be focused on customer's problem.
- [Dutoit] A requirement is a feature that the system must have or a constraint that it must satisfy to be accepted by the customer.

Stating requirements

Typical statements…

- ATM: the ATM shall allow a customer to access (i.e., view the balance of) his/her accounts.
- Cruise control: the cruise control shall be automatically and immediately disengaged when the driver uses the break pedal.
- Air traffic control: the air traffic control software shall run 24h a day, 7 days a week.
- Windows application: the software shall be configurable to any language supported by the operating system.
- Anti-collision software: the software shall detect potential colliding aircraft trajectories within 5mn before that can actually happen.

Motivations and Goals (I)

- Requirements describes the expected behavior of a system (functional, non-functional)
- Every nontrivial engineering system must be specified, based on user requirements
- Requirements need to be explicitly stated and documented for system implementation
  - e.g., used for design decisions, verification and validation, and a reference point during maintenance
- SE is about developing software solutions to problems
  - Good solutions can only be developed if software engineers understand the problems.

Motivations and Goals (II)

- Defects are cheaper when detected earlier
- For safety-critical systems, requirements problems are more likely to be safety-related
- Failure to understand and manage requirements is the biggest single cause of cost and schedule slippage
- Requirements documentation treats the software system as a black-box
- **Separation of concerns: what vs. How**

Surveys

- Standish Group surveyed 350 companies, over 8000 projects, in 1994
  - 31% cancelled before completed, 9-16% were delivered within cost and budget
  - Causes of failed projects:
    - Incomplete requirements (13%)
    - Changing requirements and specifications (9%)
    - Unrealistic expectations (9%)
    - Lack of user involvement (12%) …
- Source: Lutz, 1993, IEEE Int. Symp. On Requirements Engineering
  - NASA Voyager (87 faults) and Galileo (122 faults)
  - Safety-related interface faults overwhelmingly caused by communication errors between development teams (93%, 72%)
  - Functional faults, especially safety-related ones, primarily caused by misunderstanding requirements (62%, 79%)

Software Requirements Elicitation and Specifications

- Fundamentals
  - Motivation and Goals
  - Requirement Engineering
  - Functional vs. Non-Functional
  - Defining Software System Scope
  - Specifying a Use Case
  - Use case relationships
  - Pitfalls
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))
- Documentation

Requirements Engineering

- Requirements Engineering is the process of defining the requirements for the system under construction
- Dutoit: Requirements engineering has two main activities:
  1. Elicitation : results in requirements specification that the customer understands
  2. Analysis: results in analysis model that developer can unambiguously understand

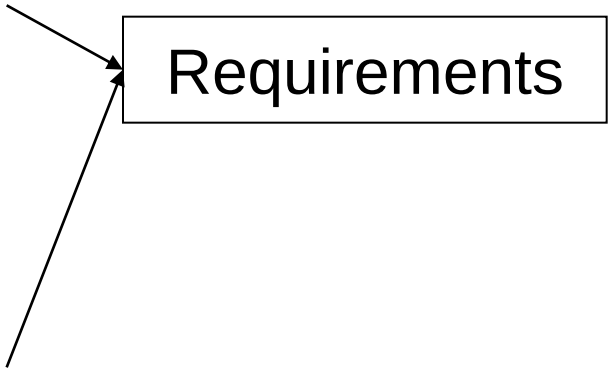
  Both represent the same information
  - Specification: communication with customer (informal notation)
  - Analysis: communication among developers (formal notation)
- Traditional terminology:
  - Requirements specification = = Requirements Definition
  - Analysis model = = Requirements Specification

Sources of Requirements

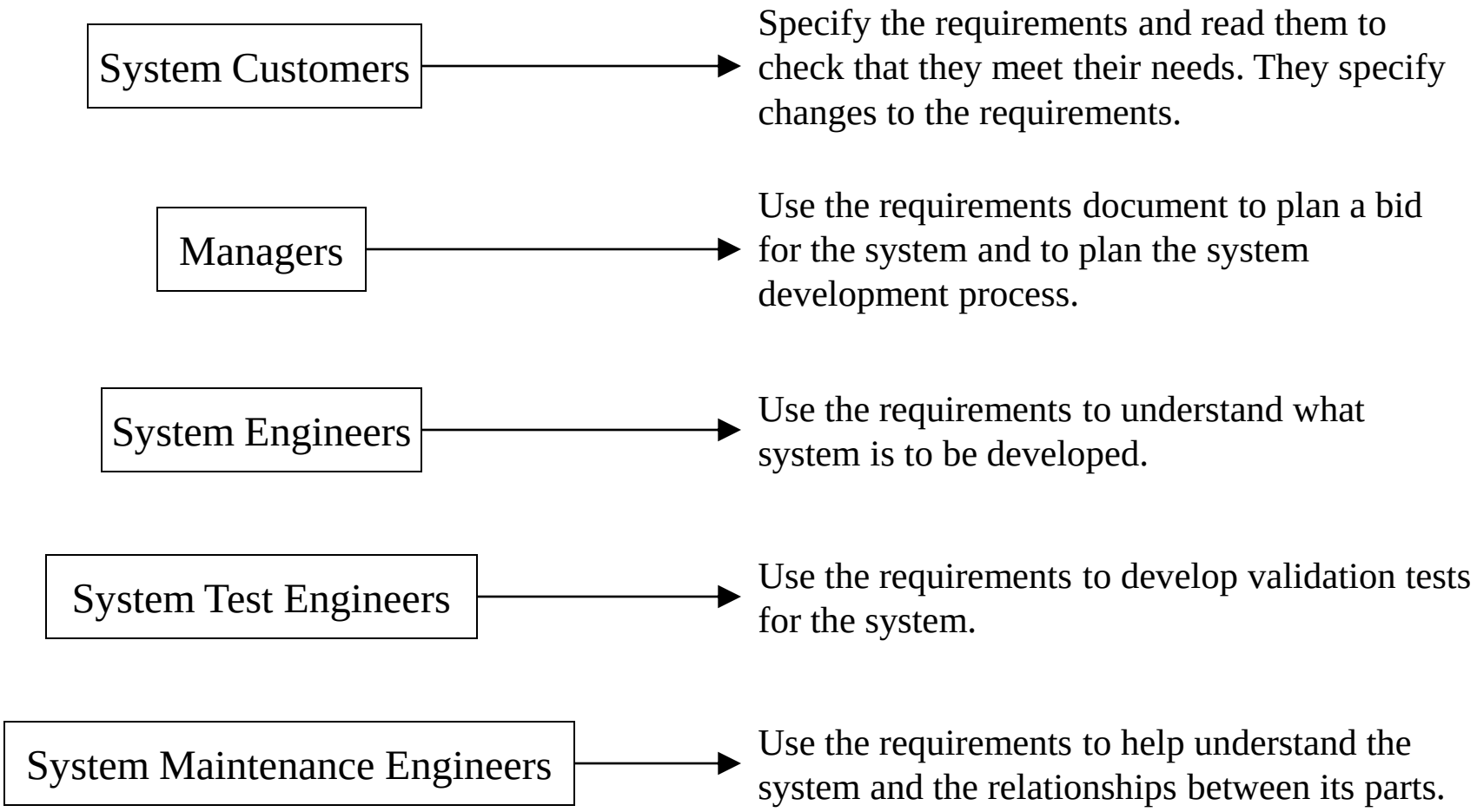
Stakeholders wants and needs

Current organization and systems
- Best practices
- Existing documents
- Requirement templates
- Domain models
- …

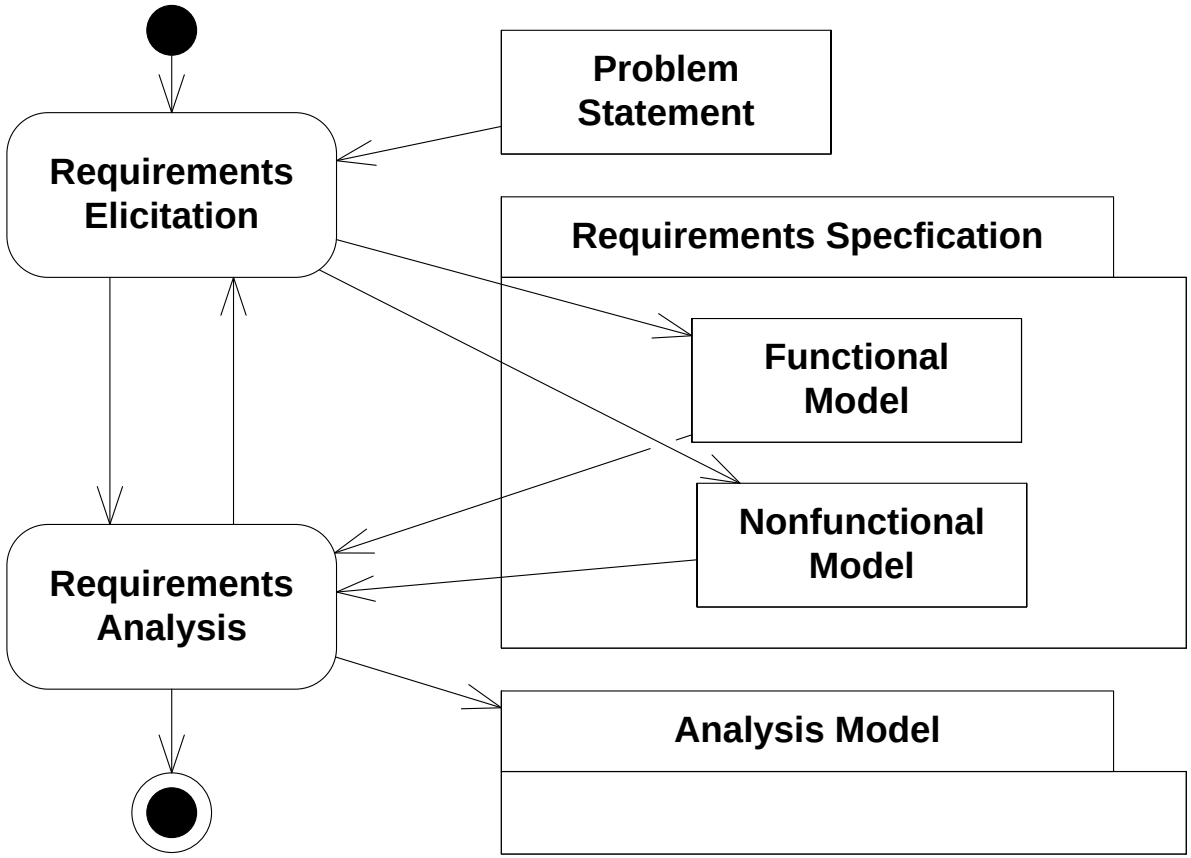
Current organization and systems
- Standards
- Legal issues
- Certification bodies
- …



Users of the Requirements (Sommerville, 2000)



Products of Requirements Process



Requirements Elicitation - Objectives

- Understand the processes, people, and resources involved
- Determine the coverage and boundary of the future system (scope)
  - VERY important decision, huge consequences if wrong
- Separate requirements according to level of priority
- No implementation decisions, unless mandated by customer

Why is Requirements Elicitation hard?

- Customers / Users are not always good at describing what they want or need
- Software Engineers are not always good at understanding someone else's concerns
- In certain application domains, software engineers and customers have completely different backgrounds and use a different terminology

Techniques for Requirement Elicitation

- Observation
  - Observe users at work
  - Obtain subtle information not told by customer (forgotten, didn't think it was important, didn't understand implication)
- Interviewing
  - Requires skill, preparation, listening
  - Ask specific details: boundaries, exceptions, anticipated changes
  - Ask vision of future
  - Ask alternatives
  - Ask minimally acceptable solution
  - Ask other sources of information
- Brainstorming : Moderated meeting with trigger questions
- Prototyping : To stimulate reaction by user.

Qualities of Specifications I

- Clear, Unambiguous, Understandable
  => Need for rigor (and perhaps formality)
  => But rigor and understandability may be contradictory goals
- Realistic (late changes to requirements are expensive)
- Correspond to real needs (Valid)
- Verifiable (to ease testing), e.g., use metrics

Qualities of Specifications II

- Consistency: the specification is inconsistent if it is self-contradictory

  => More likely to be inconsistent as complexity grows, and modifications are performed over time
- Completeness:
  - Internally complete, self-contained
  - Complete set of requirements (captures all needs)
  - Includes things the system must *not* do …

Quality Example: Lethbridge

Restaurant Advisor System: “This system will allow people to choose a restaurant in a city. Users enter one or more of the following criteria, and then the system searches its database for suitable restaurants: food type, price range, neighbourhood, size, service type (fast food, cafeteria, buffer, full service), smoking arrangements (none allowed, separately ventilated section ,non-separately-ventilated section). The user can also specify a desired day and time period, and the number of people in their party. They system will tap into the reservation database (of participating restaurants) and only display restaurants that have available space. After entering the criteria, the user clicks on “search” and the system displays a list of matching restaurants. For restaurants that participate in the automated reservation system, the user can click on “reserve” next to a selection in order to make a reservation.

Point out problems that you find in this “short statement of functional requirements”

The importance of organization and priority

1. Every character in the Encounter video game shall have a name
2. Every game character has the same set of qualities, each with a floating point value
3. Encounter shall take less than a second to compute the results of an engagement.
4. Each area has a specified set of “qualities needed”
5. When two Encounter game characters are in the same area at the same time, they may either choose or be obliged by the game to engage each other.
6. Every game character shall have an amount of life points.
7. The sum of the value of qualities of a game character relevant to the area in question shall be referred to as the character’s area value. In an engagement the system compares the area values of the characters and compute the result of the engagement.
8. The name of any character shall have no more than 15 letters.

The importance of precision

Before: Every area shall have a name of up to 15 characters.

After: Every area will have a unique name consisting of 1 to 15 characters. Acceptable characters shall consist of blanks, 0 through 9, a through z, and A through Z only.

Before: Every game character has the same set of qualities, each have a floating point value. These are initialized to 100/n where n is the number of qualities. The qualities are attention span, endurance, intelligence, patience and strength.

After: Every game character has the same set of qualities. Each quality shall be a nonnegative floating point number with at least one decimal of precision. These are all initialized equally so that the sum of their values is 100. The value of a quality cannot be both greater than 0 and less than 0.5. For the first release, the default qualities will be concentration, intelligence, patience, stamina and strength. Qualities may be added or removed during configuration, before any characters are created.

Software Requirements Elicitation and Specifications

- Fundamentals
  - Motivation and Goals
  - Requirement Engineering
  - Functional vs. Non-Functional
  - Defining Software System Scope
  - Specifying a Use Case
  - Use case relationships
  - Pitfalls
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))
- Documentation

Functional vs. Non-Functional

- Functional requirement:
  - interaction between a system and its environment (e.g., UML actors)
  - (independent from its implementation)
- Non-Functional requirement:
  - restriction on the system that limits our choices for constructing a solution
  - e.g., memory, platform, real-time constraints
- Non-Functional requirements have as much impact on the system cost and development as functional requirements

Types of NF Requirements

- Usability: the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system.
  - Relates to the user interface—number of nested levels in menus, color schemes …, online help, level of documentation …
- Dependability: the property of a system such that reliance can justifiably be placed on the service it delivers. Includes reliability, robustness, and safety.
- Reliability: the ability of a system to perform its required functions under stated conditions for a specified period of time.
  - Includes acceptable mean time to failure, the ability to detect specified faults or withstand specified security attacks
- Robustness: the degree to which a system can function correctly in the presence of invalid inputs or stressful environment conditions.
- Safety: A measure of the absence of catastrophic consequences to the environment.

Types of NF Requirements (cont.)

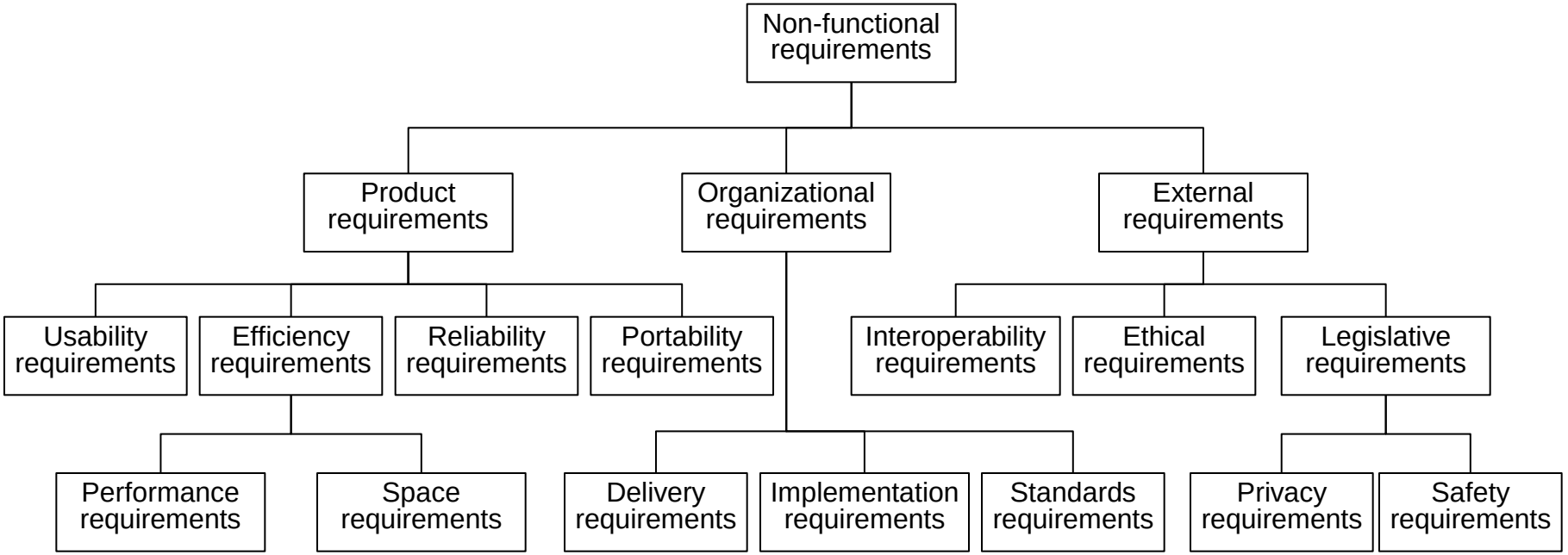
- Performance: Quantifiable attributes of the system such as response time, throughput, availability, accuracy.
- Response time: how quickly the system reacts to a user input.
- Throughput: how much work the system can accomplish within a specified amount of time.
- Availability: the degree to which a system is operational and accessible when required for use.
  - E.g., an availability of 0.998 means that in every 1000 time units, the system is likely to be available for 998 of these.
- Accuracy: a quantitative measure of the magnitude of error.

Types of NF Requirements (cont.)

- Supportability: Requirements concerned with the ease of changes to the system after deployment. Includes adaptability, maintainability.
- Adaptability: the ability to change the system to deal with additional application domain concepts.
- Maintainability: the ability to change the system to deal with new technology or to fix defects.

- In practice, NF requirements have to be prioritized by importance. Some of them *need* to be met for the system to operate correctly.

Sommerville's Classification



Examples

- The product should identify an aircraft within 0.25 seconds
- The product should be used with poor lighting conditions and the users will wear gloves
- The product should be easy to use with only one hand free
- The system shall not disclose any personal information about customers
- The product should be readily portable to the Linux operating system

What is usually not in the Requirements?

- System structure, implementation technology
- Development methodology
- Development environment
- Implementation language
- Reusability

It is desirable that none of these be constrained by the client

- But in certain application domains, like airborne systems, military systems, there are (international) standards to follow.

Realistically

- Many requirements can only be clearly identified after some experience with the system => incrementality
- Some amount of imprecision (“common knowledge”) is accepted
  - e.g., what is a saving bank account in a given banking environment, a sensor of a certain type
- Responsibility of users and software engineers to determine what is acceptable

NF Requirements Metrics

Property	Metric
• Speed	• Process transactions per second • User/event response time • Screen refresh time
• Size	• K Bytes • Number of RAM chips
• Ease of use	• Training time • Number of help frames
• Reliability	• Mean time between failure (MTBF) • Probability of unavailability • Rate of failure occurrence • Availability
• Robustness	• Time to restart after failure • Percentage of events causing failure • Probability of data corruption on failure
• Portability	• Percentage of target dependent statements • Number of target systems

Notice how each metric is a quantifiable amount – a number to be verified! Even portability

Software Requirements Elicitation and Specifications

- Fundamentals
  - Motivation and Goals
  - Requirement Engineering
  - Functional vs. Non-Functional
  - Defining Software System Scope
  - Specifying a Use Case
  - Use case relationships
  - Pitfalls
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))
- Documentation

Defining the Software System Scope

- VERY important decision, huge consequences if wrong
- System Boundary: define activities and data IN the system
- Fundamental questions:
  - What/who triggers the behaviour expected from the software?
  - Should we implement the requirements or is the requested functionality a responsibility of another system or a human?
- We need to know the context in which a system operates
- External entities:
  - other systems, organizations, people, machines, device (sensor, actuator), etc. that expect services/data from us or provide services to us
- Input/Output data flows from/to external entities

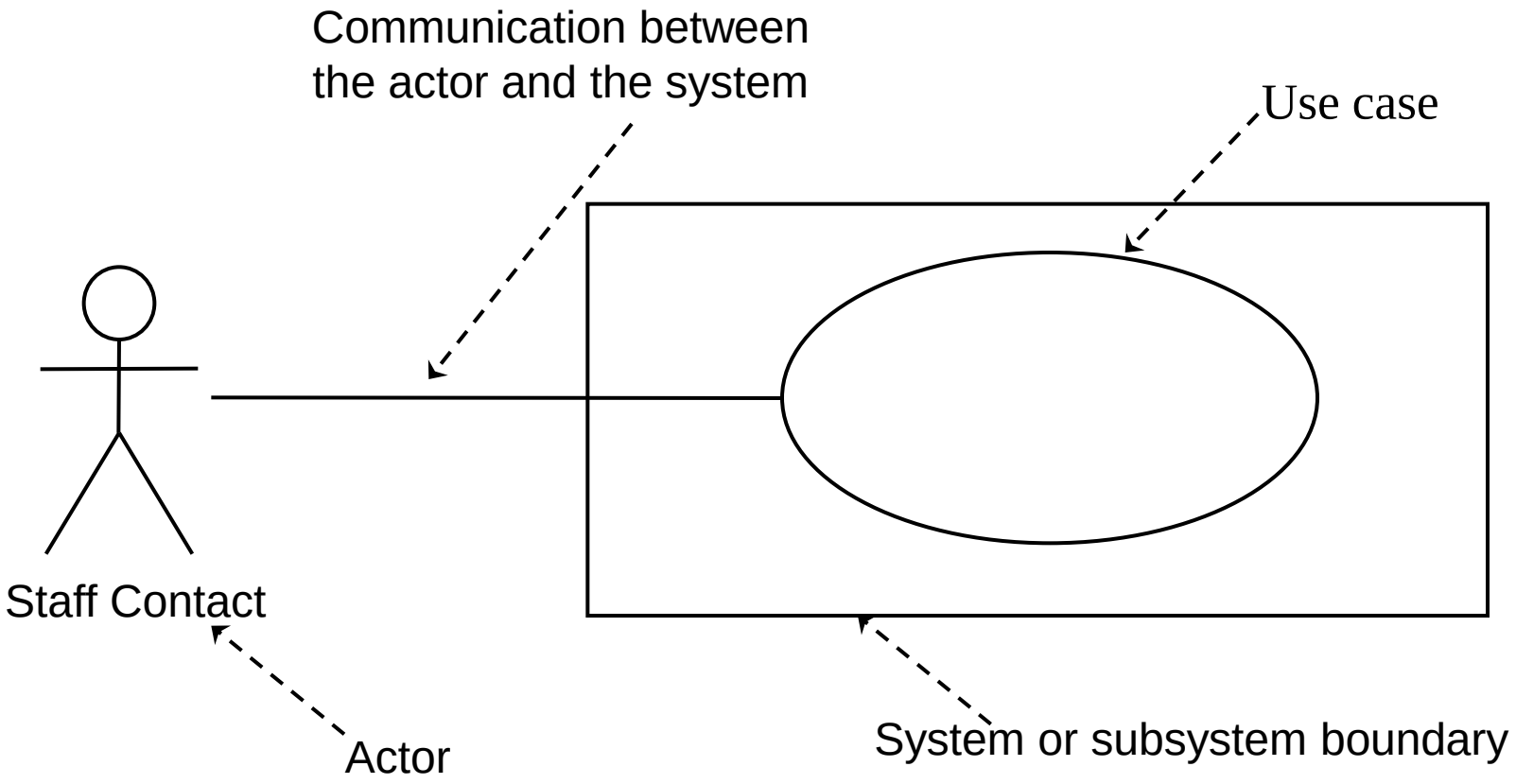
Scope Example

- Web-based store: The system allows the purchase of items over the web. When a purchase it made, inventory is checked and updated and the total cost is computed. Because it is web-based, foreign purchases may be made, requiring the cost to be computed in the foreign currency using that day's currency exchange rate.

- Two databases can be envisioned:
  - Inventory/price database: For each item, number available and price
  - Google's currency exchange database

- Question: Define the scope of the system. In particular, are the databases inside or outside the system ?

Software System Scope in UML—Use Case Diagram



Defining Software System Scope

- Defines what is IN the software system
  - Defines what is OUT of the software system
- Defines what is the responsibility of the software
- Defines what the engineers have to build
- Can be a binding (legal) description of the software system

Software Requirements Elicitation and Specifications

- Fundamentals
  - Motivation and Goals
  - Requirement Engineering
  - Functional vs. Non-Functional
  - Defining Software System Scope
  - Specifying a Use Case
  - Use case relationships
  - Pitfalls
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))
- Documentation

Use Case [Cockburn]

- A use case captures a contract between the stakeholders of a system about the system behaviour.
  - What the actor expects from, sends to the system
  - What the system ensures to, expects form the actor
- A use case describes the system's behaviour under various conditions as the system responds to a request from one of the stakeholders, called the **primary actor**.
  - The primary actor has a goal with the software system
- The system responds, protecting the interests of all the stakeholders (not all stakeholders are actors)
- Different behaviours (i.e., scenarios) can unfold
  - Including descriptions of what may go wrong

Use Cases and Actors

- A **primary actor** has a goal with the software system
- The system has a responsibility: to achieve the goal of the primary actor
- The system formulates sub-goals to carry out its responsibility
  - Some sub-goals can become other use cases (carried out internally)
  - Some sub-goals can be achieved with the help of another (**secondary**) **actor** (carried our externally)
    - Recall the fundamental question: Should we implement the requirements or is the requested functionality a responsibility of another system or a human?

Use Case Scope/Extent

- How much does a use case describe?
  - How many things are done in a use case
  - How many scenarios?
  - How many steps?
- Answer: what is the actor's goal? [Cockburn]
- A use case usually passes the one person, one sitting test
  - Can the primary actor go away happy after the use case finishes?
  - *Coffee break test*: "after I get done with this, I can take a coffee break"
  - 2-20mn long to follow the steps of a use case.

Use Case Scope/Extent (cont.)

- Some (potential) use cases do not pass this test, and should not count as user goals:
  - Use Case "Complete an on-line auction purchase“
    - On-line auctions take several days, so fail the single-sitting test.
    - This long “goal” should be split: e.g., making the auction publicly available, making a bid, changing a bid, selecting the “winner”
  - Use Case "Log on“
    - Logging on 42 times in a row does not (usually) satisfy the person’s job responsibilities or purpose in using the system.
- Instead
  - Use Case "Register a new customer"
    - Registering 42 new customers has some significance to a sales agent.
  - Use Case "Buy a book"
    - A book purchase can be completed in a single sitting.

Login in is not a system behaviour that is that much interesting to stakeholders (from a functional point of view). However, this is a behaviour the system will need to exhibit, and this will therefore be a use case.

Specifying a Use Case (template)

Use case are described by following template descriptions, which typically include the following sections (many different templates exist):

- Use Case Name
- Brief Description
- Precondition
- Primary Actor
- Secondary Actors
- Dependencies to other use cases
- Basic Flow
- Alternative Flows: Specific, Bounded, Global Alternative Flows
- Special requirements
- Technology and data variations
- Open issues

Specifying a Use Case (cont.)

Use case name:

- Should be a verb phrase denoting what the actor is trying to accomplish (goal)
- Should reflect the perspective of the actor
  - E.g., “perform withdrawal” instead of “record withdrawal”

Precondition

- States what must always be true before any scenario of the use case begins.

Primary Actor

- The principal actor that initiates the use case.

Secondary Actors

- The secondary actors that the system relies on to accomplish some of the sub-goals.

Dependencies to other use cases (see later)

- <<include>> and <<exclude>> relationships between use cases.
- Generalization relationship between use cases.

Specifying a Use Case (cont.)

Basic Flow

- Describes a typical success path that satisfies the interests of the stakeholders. It often does not include any conditions or branching.
- A step can be one of the following five interactions:
  1. Primary actor → system: the primary actor sends a request and data to the system.
  2. System → system: the system validates a request and data.
  3. System → system: the system alters its internal state (e.g., recording or modifying something)
  4. System → secondary actor: the system sends requests to a secondary actor.
- The first step (outside this classification) often indicates the trigger event that starts the scenario.
- All steps are numbered sequentially:
  1. description of step.
  2. …
- Use **active voice only**
- Postcondition: what should be true after the basic flow has executed.

Specifying a Use Case (cont.)

Alternative flows

- Describe all the other scenarios or branches, both success and failure. An alternative flow always depends on a condition occurring in a specific step in a flow of reference, referred to as *reference flow step (RFS)*, and that reference flow is either the basic flow or an alternative flow itself.
- All action steps are numbered sequentially.
- Each alternative flow must have a postcondition.

Three types of alternative flows:

- *Specific alternative flow:* an alternative flow that refers to a specific step in the reference flow (either a main flow or another alternative flow).
- *Bounded alternative flow:* an alternative flow that refers to more than one step in the reference flow–consecutive steps or not.
- *Global alternative flow:* an alternative flow that refers to any step in the reference flow.

Specifying a Use Case (cont.)

Special Requirements

- If a non-functional requirement, quality attribute, or constraint relates specifically to the use case, list it here.

Technology and Data Variation

- Foreseeable technology changes are listed
  - E.g., providing credit account input using a card reader and the keyboard
- Foreseeable input type variations
  - E.g., metric vs. imperial
- The list can refer to steps in the basic or alternate flows.

Open issues

- Lists what remains to be clarified with stakeholders
  - E.g., terminology.

Specifying a Use Case (cont.)—possible layout

Use Case Name	The name of the use case. It usually starts with a verb.	
Brief Description	Summarizes the use case in a short paragraph.	
Precondition	What should be true before the use case is executed.	
Primary Actor	The actor which initiates the use case.	
Secondary Actors	Other actors the system relies on to accomplish the services of the use case.	
Dependency	Include and extend relationships to other use cases.	
Generalization	Generalization relationships to other use cases.	
Basic Flow	Specifies the main successful path, also called "happy path".	
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the basic flow executes.
Specific Alternative Flows	Applies to one specific step of the basic flow.	
	RFS	A reference flow step number where flow branches from.
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.
Global Alternative Flows	Applies to all the steps of the basic flow.	
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.
Bounded Alternative Flows	Applies to more than one step of the basic flow, but not all of them.	
	RFS	A list of reference flow steps where flow branches from.
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.

Specifying a Use Case (cont.)

- See example of the report emergency use case (pdf file)…
- See example of the withdraw funds use case (pdf file)…

Specifying a Use Case (restricting the use of English)

- The subject of a sentence should be “the system” or an actor.
  - The card has been ejected.
  - The system ejects the ATM card.
- Describe the flow of events sequentially (a use case describes what should happen).
- Actor-to-actor interactions are not allowed (these interactions are not supported by the software you specify, are they?).
  - The customer gives the teller the ATM card.
  - The customer inserts the ATM card into the card reader.
- Describe one action per sentence.
- Use present tense only.
  - The system *ejected* the card.
  - The system ejects the card.
- Use active voice rather than passive voice.
  - The card *is ejected*.
  - The system ejects the card.

Specifying a Use Case (cont.)

- Clearly describe the interaction between the system and actors without omitting its sender and receiver.
  - Customer enters PIN.
  - ATM customer enters PIN number to the system.
- Use declarative sentence only. “Is the system idle?” is a non-declarative sentence.
  - Ejects the card.
  - The system ejects the card.
- Use words in a consistent way. Keep one term to describe one thing.
  - Customer inserts the ATM card…
  - ATM customer inserts the ATM card…
- Don’t use modal verbs (e.g., *might*) nor adverbs (e.g., very
  - The system might eject the card. The system likely ejects the card.
  - The system ejects the card. The system ejects the card

Specifying a Use Case (cont.)

- Use simple sentences only. A simple sentence must contain only one subject and one predicate.

  System displays customer accounts and prompts customer for transaction type…

  1. The system displays ATM customer accounts.
  2. The system prompts ATM customer for …
- Don't use negative adverb and adjective (e.g., *hardly*, *never*), but it is allowed to use *not* or *no*.
  - The PIN number has never been validated.
  - The PIN number has not been validated
- Don't use pronouns (e.g. *he*, *this*)
  - …it reads the card number.
  - …the system reads the card number.
- Don't use participle phrases as adverbial modifier.
  - ATM is idle, displaying a Welcome message.
  - The system is idle. The system is displaying a Welcome message.

Specifying a Use Case (cont.)

- INCLUDE USE CASE = including another use cases.
  - Grammar
    - INCLUDE USE CASE <included use case name>
  - Example:
    - Include ValidatePIN use case.
    - INCLUDE USE CASE ValidatePIN
- EXTENDED BY USE CASE = extension by another use case.
  - Grammar
    - EXTENDED BY USE CASE <extending use case>
  - Example:
    - Use case CreateIncident extends the current use case.
    - EXTENDED BY USE CASE CreateIncident

Specifying a Use Case (cont.)

- RFS = reference flow step (number(s))
  - Grammar
    - RFS <reference flow step #> (specific alternative flow)
    - RFS <reference flow step numbers> (bounded alternative flow)
    - Not required for global alternative flow.
  - Explanation
    - One specific or bounded alternative flow must correspond to exactly one or more than one reference flow steps.
  - Example:
    - RFS Basic Flow 5 …
    - RFS Basic Flow 5-7, 10, 14 …
- IF, THEN, ELSE, ELSEIF, and ENDIF = conditional logic.
  - Grammar
    - IF <condition> THEN <steps> ENDIF
    - IF <condition> THEN <steps> – ELSE <steps> ENDIF
    - IF <condition> THEN <steps> – ELSEIF <condition> THEN <steps> ENDIF
  - Example:

    IF the system recognizes the ATM card, THEN the system reads the ATM card number, ENDIF.

Specifying a Use Case (cont.)

- MEANWHILE = concurrency.
  - Grammar
    - <action> MEANWHILE <action>
  - Example:
    - the system cancels the transaction and ejects the card.
    - the system cancels the transaction MEANWHILE the system ejects the card.
- VALIDATES THAT = a condition is evaluated.
  - Grammar
    - VALIDATES THAT <condition>
  - Explanation
    - a condition is evaluated and must be true to proceed to the next step.
    - the alternative case (the condition does not hold) must be described in a corresponding alternative flow (BFS).
  - Example:
    - the system checks whether the user-entered PIN…
    - the system VALIDATES THAT the user-entered PIN…

Specifying a Use Case (cont.)

- DO … UNTIL = iteration.
  - Grammar
    - DO <steps> UNTIL <condition >
  - Explanation
    - Following keyword DO is a sequence of steps. Following keyword UNTIL is a loop ending condition.
  - Example:
    1. DO
    2. action1
    3. action2
    4. UNTIL condition
- ABORT = an exceptionally exit action.
  - Grammar
    - ABORT
  - Explanation
    - Used in alternative flows, iterative, and conditional logic sentences. It means the ending of a use case.
    - An alternative flow ends either with ABORT or RESUME STEP.

Specifying a Use Case (cont.)

- RESUME STEP = an alternative flow goes back to its corresponding basic flow.
  - Grammar
    - RESUME STEP <basic flow step #>
  - Explanation
    - Used in alternative flows.

Specifying a Use Case (summary)

- Template + Restrictions =
  - Facilitates communication between stakeholders (fewer ambiguities)
  - Facilitates subsequent phases in development
    - Requirements are used in many phases and ought to be correct and understandable
  - Facilitates automation!
    - Checking consistency
    - Producing documentation
    - Producing glossary of terms (domain)
    - Generating analysis document (first draft)
    - …

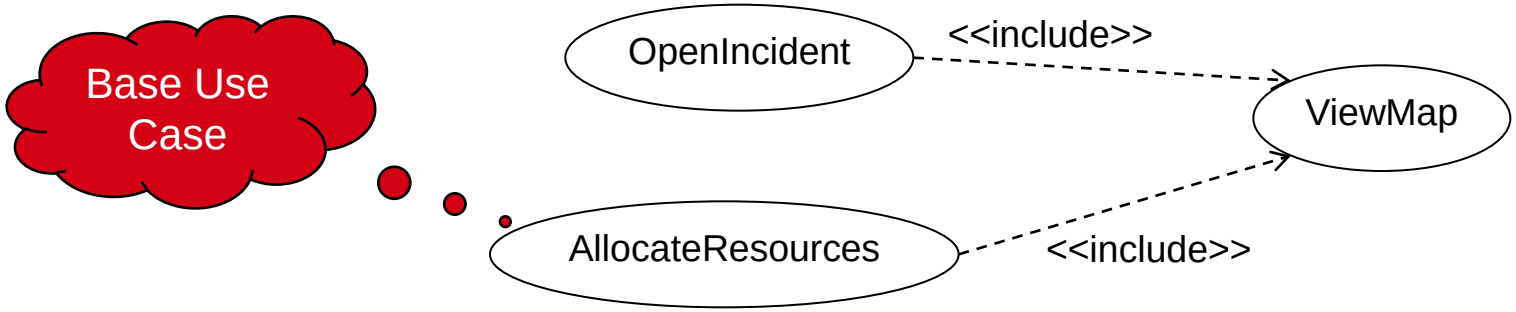
Software Requirements Elicitation and Specifications

- Fundamentals
  - Motivation and Goals
  - Requirement Engineering
  - Functional vs. Non-Functional
  - Defining Software System Scope
  - Specifying a Use Case
  - Use case relationships
  - Pitfalls
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))
- Documentation

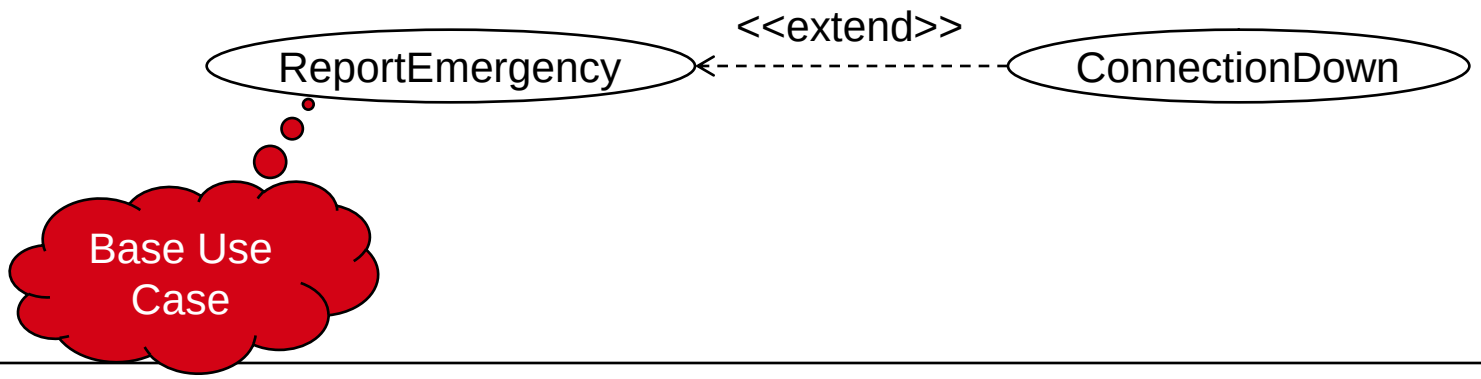
<<include>>

- Used to:
  - Decompose larger/longer use case
  - Share functionalities
  - The included use case is always triggered when the base use case is.
  - The base use case delegates (sub)goals to the included use case
- Example:
  - The use case ViewMap describes behaviour that can be used by the use case OpenIncident and the use case AllocateResources



<<extend>>

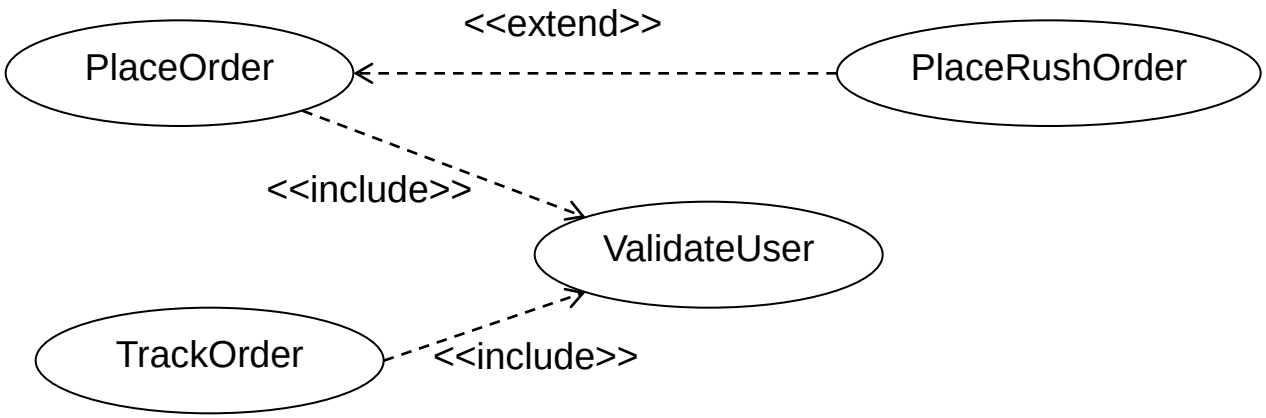
- Problem:
  - The functionality in the original problem statement needs to be extended to account for exceptional flow of events.
- Solution:
  - An `extend` association from (direction of the arrow head) a use case A to a use case B indicates that use case A is an extension of use case B.
  - This specifies that use case A is triggered when use case B executes **only** under some condition



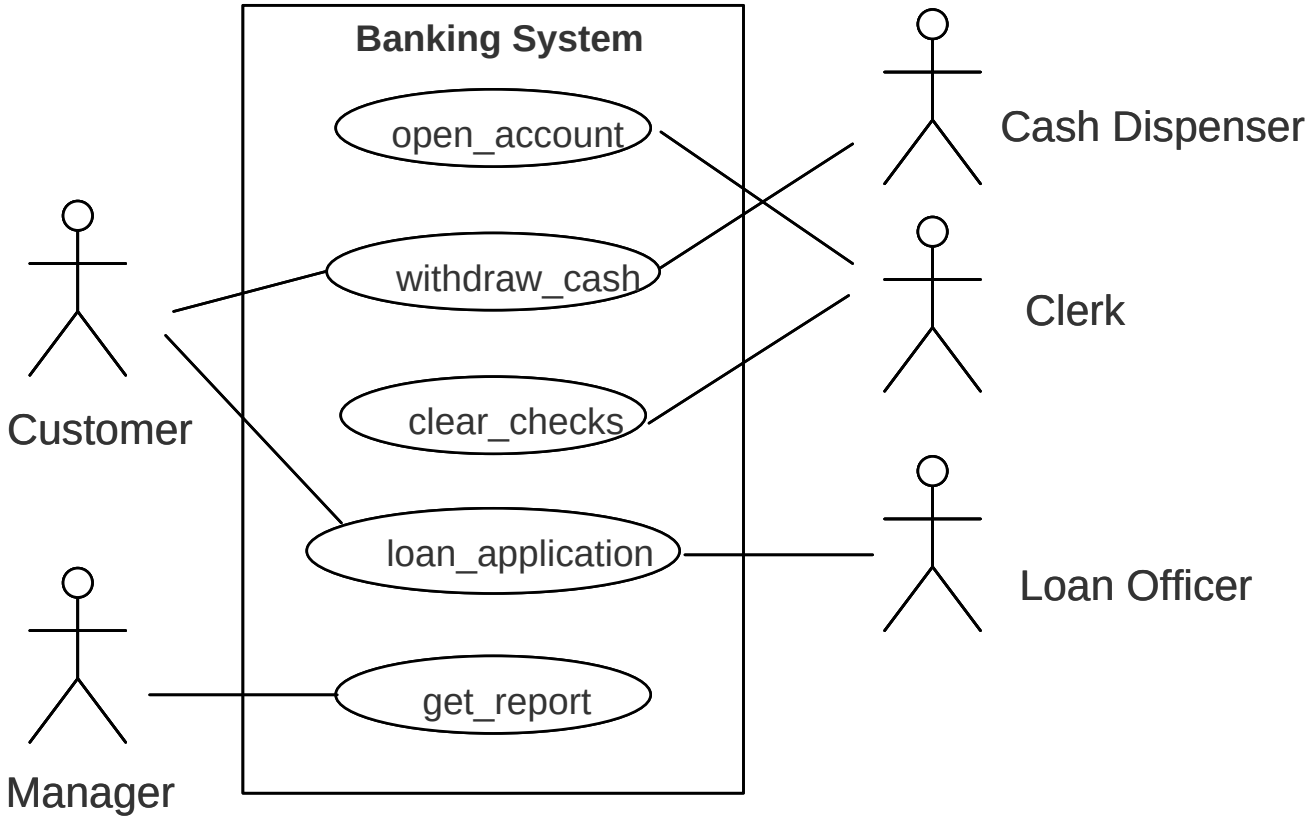
Use case generalization

- A (base) use case can be specialized by another use case
- No much use!
  - “Use case experts have been successfully doing use case work without this optional relationship […] and there is not yet agreement by practitioners on the best-practice guidelines of how to get value from this idea.” [Larman]

Example I


<<extend>>
PlaceOrder
PlaceRushOrder
<<include>>
ValidateUser
TrackOrder
<<include>>

Example II


Banking System
open_account
withdraw_cash
clear_checks
loan_application
get_report
Customer
Manager
Cash Dispenser
Clerk
Loan Officer

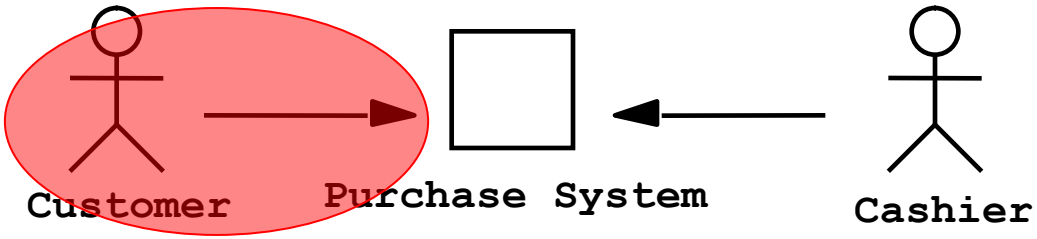
Software Requirements Elicitation and Specifications

- Fundamentals
  - Motivation and Goals
  - Requirement Engineering
  - Functional vs. Non-Functional
  - Defining Software System Scope
  - Specifying a Use Case
  - Use case relationships
  - Pitfalls
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))
- Documentation

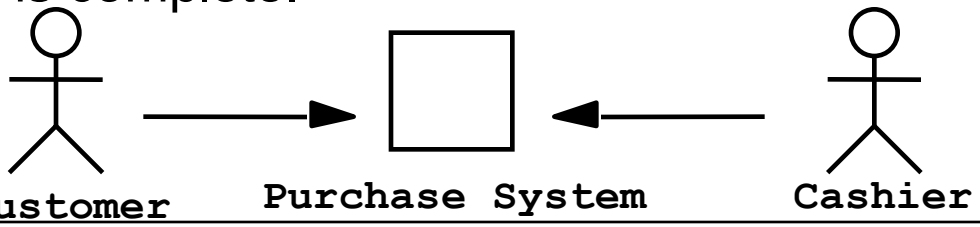
Common mistake : Identifying wrong actor

- Consider a purchase system for any kind of store (Sears, Leons, AMC Theatre, MacDonald's). Customers give the cashier their order. The cashier enters the selection (the item's bar code, the selected movie, the number of Big Macs) and the system calculates the total.



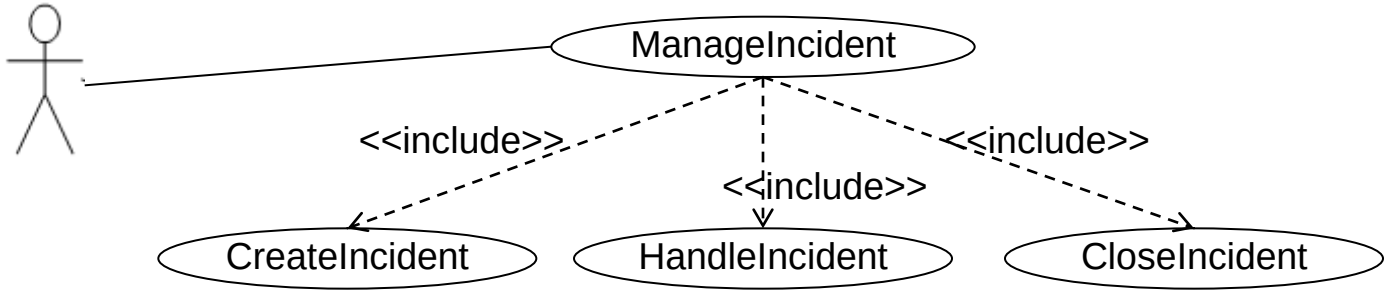
Nowhere in the description above the customer is said to interact with the system

- The customer gives payment (debit/credit card) and the transaction is complete.

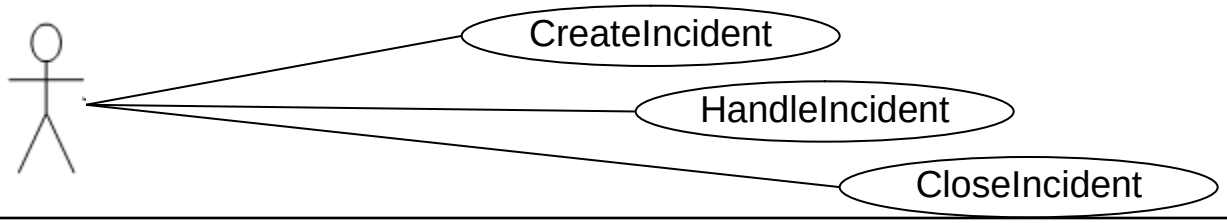


<<Include>>: Functional Decomposition

- Problem:
  - A function in the original problem statement is too complex to be solvable immediately
- Solution:
  - Describe the function as the aggregation of a set of simpler functions. The associated use case is decomposed into smaller use cases

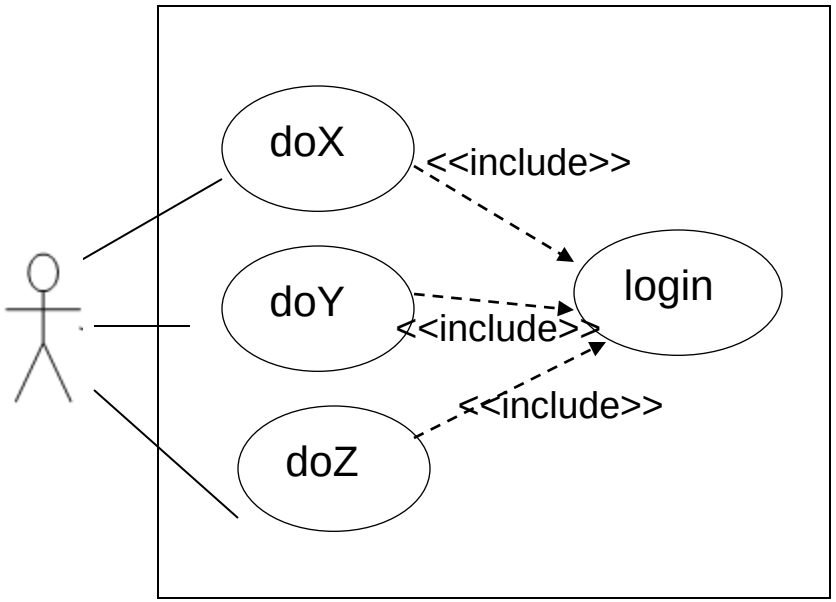


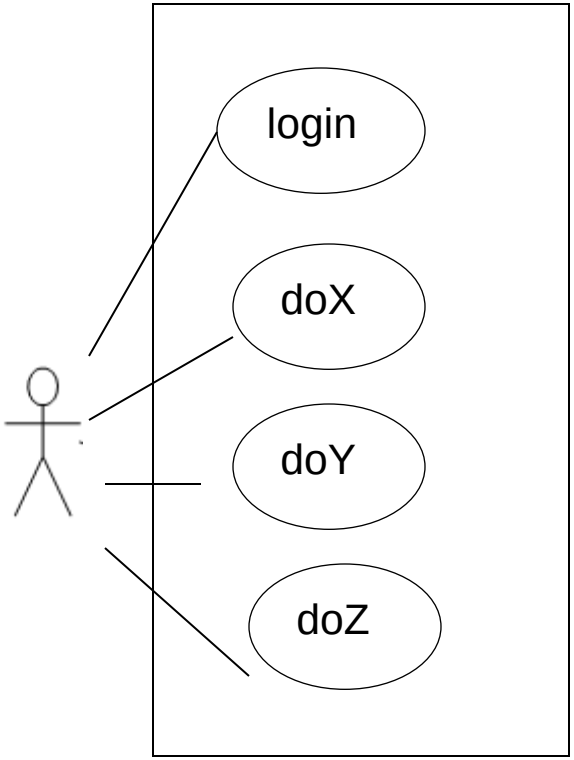
- However: remember the “one person, one sitting” test?



Discussion : <<include>> as Functional Composition

A typical login situation





Neither is right or wrong.
Instead, they mean different things.

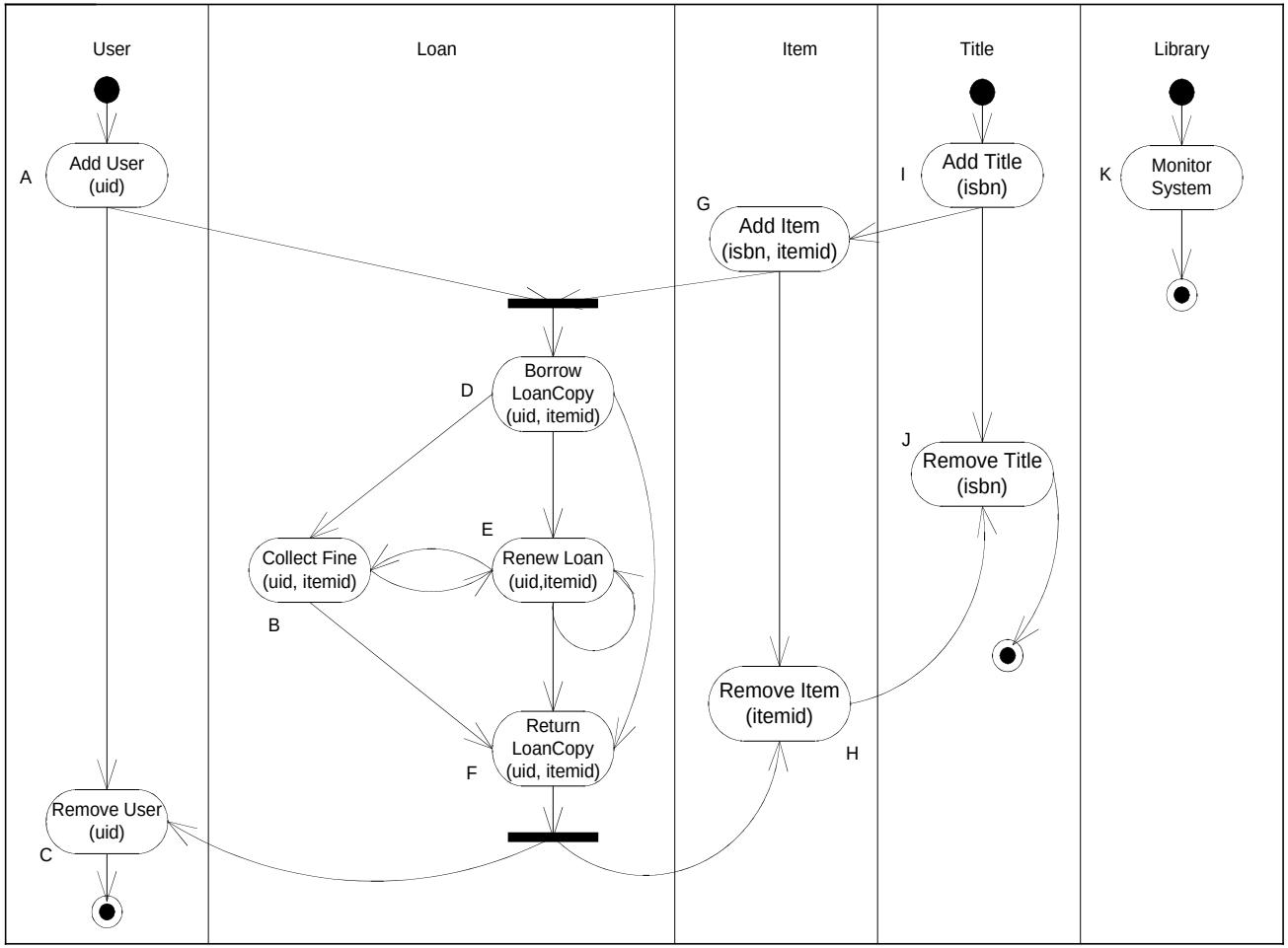
In What Order Do Use Cases Execute?

- For each actor, we identify what that actor wants to do with the system.
  - Each of these things that the actor wants to do with the system become a Use Case.
  - E.g., an actor wants to perform tasks A, B, C, and D with the system.
    - This leads to 4 different actors, all triggered by the actor.
    - But perhaps, A is always being triggered before the other three?

- This **cannot** be modeled by a use case diagram!

- Solution: UML Activity diagram (mentioned in Bruegge&Dutoit)
  - Activities are use cases.
  - Swimlanes can be actors or domain objects (data manipulated in use cases)
  - Activity diagram notation: conditions, loops, fork/join …

Use Case Order: The Library Example



Software Requirements Elicitation and Specifications

- Fundamentals
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))

  - Identifying actors
  - Identifying scenarios
  - Identifying use cases
  - Identifying non-functional requirements
  - Refining use cases
  - Relationships between use cases
  - Summary
- Documentation

Use Case Model

- Define system functional requirements in terms of Actors and Use Cases
  - Each use case defined in terms of sequences of interactions between Actor and System
    - Structured narrative description, sequence diagram
  - Basic sequences
    - Most common sequences
  - Alternative sequences
    - Error conditions
- Use case associations (include, extend)

Requirements Elicitation Activities

1. Identify actors
   - Identify the different types of users of the future system
2. Identify scenarios
   - Identify scenarios for typical functionalities
3. Identify use cases
   - Abstract scenarios into use cases
4. Identify nonfunctional requirements
   - Identify aspects visible to the user but not directly related to functionalities
5. Refine use cases
   - Is the system specification complete (e.g., exceptional conditions)
6. Identify relationships among use cases
   - Consolidate the use case model by eliminating redundancies

1. Identify Actors

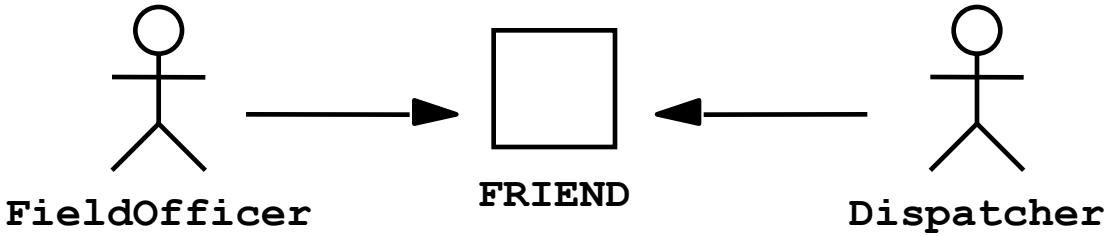
- Can be human or external system, device
- Define system boundaries
- Find all stakeholders
- May correspond to roles in an organization
- Need to differentiate roles only when they access different functionality
- Classes of functionality

Questions to Ask

- 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?
- Will the system interact with any external hardware or software system?

FRIEND: Accident Management System

- It is a distributed information system for managing accidents. It allows dispatchers and authorities to communicate and allocate resources to an emergency.



- Problem: Long list of potential actors
  - Firefighters, police officers, dispatchers, investigators, …
- We need to consolidate the list into a small number of actors who are different from the point of view of the system usage
  - A firefighter and a field police officer share the same interface, both involved with a single incident on the field
  - Dispatcher manages multiple concurrent incidents and requires access to more information

2. Identify scenarios

Bridging the gap between the user and the developer

- *Scenarios*: Example of the use of the system in terms of a series of interactions between an actor and the system
- *Use cases*: Abstraction that describes a class of scenarios (e.g., end user functionalities)

Scenarios

- “A narrative description of what people do and experience as they try to make use of computer systems and applications” [M. Carrol, Scenario-based Design, Wiley, 1995]
- A concrete, focused, informal description of a single feature of the system used by a single actor.
- Readily understandable by clients and users

- Developers and users write and refine a series of scenarios in order to gain a shared understanding of what the system should be.
  - Iterative process.

Heuristics for finding Scenarios

- Ask yourself or the client the following questions:
  - What are the primary tasks that the system needs to perform?
  - What data will the actor create, store, change, remove or add in the system?
  - What external changes / events does the system need to know about?
  - What changes or events will the actor of the system need to be informed about?
- Insist on task observation if a system already exists
  - Ask to speak to the end user, not just to the software contractor
  - Expect resistance and try to overcome it
- Sources of information:
  - User manuals of previous systems, procedure manuals, company standards, user and client interviews

FRIEND Scenario: Warehouse on Fire

- Scenario:
  - A fire is detected in a warehouse; two field officers arrive at the scene and request resources
- Source of information:
  - Observation or discussions with actual, future users of the system about how they would use the system (or are using the current system) in certain circumstances
- Details:
  - Bob, driving down main street in his patrol car notices smoke coming out of a warehouse. His partner, Alice, reports the emergency from her car.
  - 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, 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.
  - 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.
  - Alice received the acknowledgment and the ETA.

Observations about Warehouse on Fire Scenario

- Concrete scenario
  - Describes a single instance of reporting a fire incident.
  - Does not describe all possible situations in which a fire can be reported.
- Participating actors
  - Bob, Alice: Field officer (Primary Actor)
  - and John, Dispatcher (Other stakeholder)

FRIEND: Other Scenarios

- 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.
- Earthquake:
  - An unprecedented earthquake seriously damages buildings and roads, spanning multiple accidents and triggering the activation of the statewide emergency operations plan.

3. Identify Use Cases

- A scenario is an instance of a use case
- A use case specifies all possible scenarios for a given piece of functionality
  - Find hints for a use case in the scenario descriptions, e.g., “Report Emergency “ in the first paragraph of the scenario is a candidate for a use case
  - Report Emergency accounts for all possible scenarios, i.e., Warehouse on fire, FenderBender, Earthquake etc.
- A use case is always initiated by an actor but may interact with other actors as well
- A use case is a complete flow of events through the system

Steps in Formulating a Use Case (I)

- First name the use case
  - Use case name: ReportEmergency
- Then find the actors
  - Generalize the concrete names (“Bob”) to participating actors (“Field officer”)
  - Participating Actors: ReportEmergency
    - Field Officer (Initiator)
    - Dispatcher
- Then concentrate on the flow of events
- Pre and post conditions

Formulate the Flow of Events

1. The `FieldOfficer` activates the “Report Emergency” function on her terminal.
2. FRIEND responds by presenting a form to the officer.
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
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` acknowledges the emergency report and selects a response.

Entry and Exit Conditions

- *Precondition:*
  - The `FieldOfficer` is logged into FRIEND
- *Post-condition:*
  - The `FieldOfficer` has received an acknowledgement and the selected response from the `Dispatcher`, OR
  - The `FieldOfficer` has received an explanation indicating why the transaction could not be processed

Steps in formulating a use case (II)

- Write down the exceptions:
  - The `FieldOfficer` is notified immediately if the connection between her terminal and the central is lost.
  - The `Dispatcher` is notified immediately if the connection between any logged in `FieldOfficer` and the central is lost.
- Identify and write down any quality (NF) requirement:
  - The `FieldOfficer`'s report is acknowledged within 30 seconds.
  - The selected response arrives no later than 2 minutes after it is sent by the `Dispatcher`.

Writing Guidelines (I)

- 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.
- Use case steps in the flow of events should be phrased in the active voice. This makes it explicit who accomplished the step.

Writing Guidelines (II)

- 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 acknowledgement).
- Exceptions and alternative flows 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 mockups (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 extends relationships to decompose it in smaller use cases.

4. Identify Non-Functional (NF) Requirements

- User-visible aspects of the system that are not directly related to the functional behavior of the system
- It is important to be systematic when eliciting quality requirements.
- Use template questions.
  - Here are examples.
  - Standards (IEEE, DoD) also help with this.

- Template questions for:
  - Usability
  - Performance
  - Reliability

Template Questions for NF Requirements

Usability

- What is the level of expertise of the user?
- What are the user interface standards familiar to the user?
- What documentation should be provided to the user?

Template Questions for NF Requirements (II)

Performance

- How responsive should the system be?
- Are there user tasks that are 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 for users?

Template Questions for NF Requirements (III)

Reliability

- How reliable, available, 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 on the system?
- Are there security requirements on the system?

5. Refining Use Cases

- Precision, correctness and completeness and consistency
- Expect Use Cases to change a lot and many iterations
- Scenarios and user interface mock-ups can be used to help exploration and validation
- Links to other use cases (`AllocateResources` in FRIEND)
- `ReportEmergency`:
  - Include details about the type of incident known to FRIEND
  - Detail how the `Dispatcher` acknowledges the report of the `FieldOfficer`

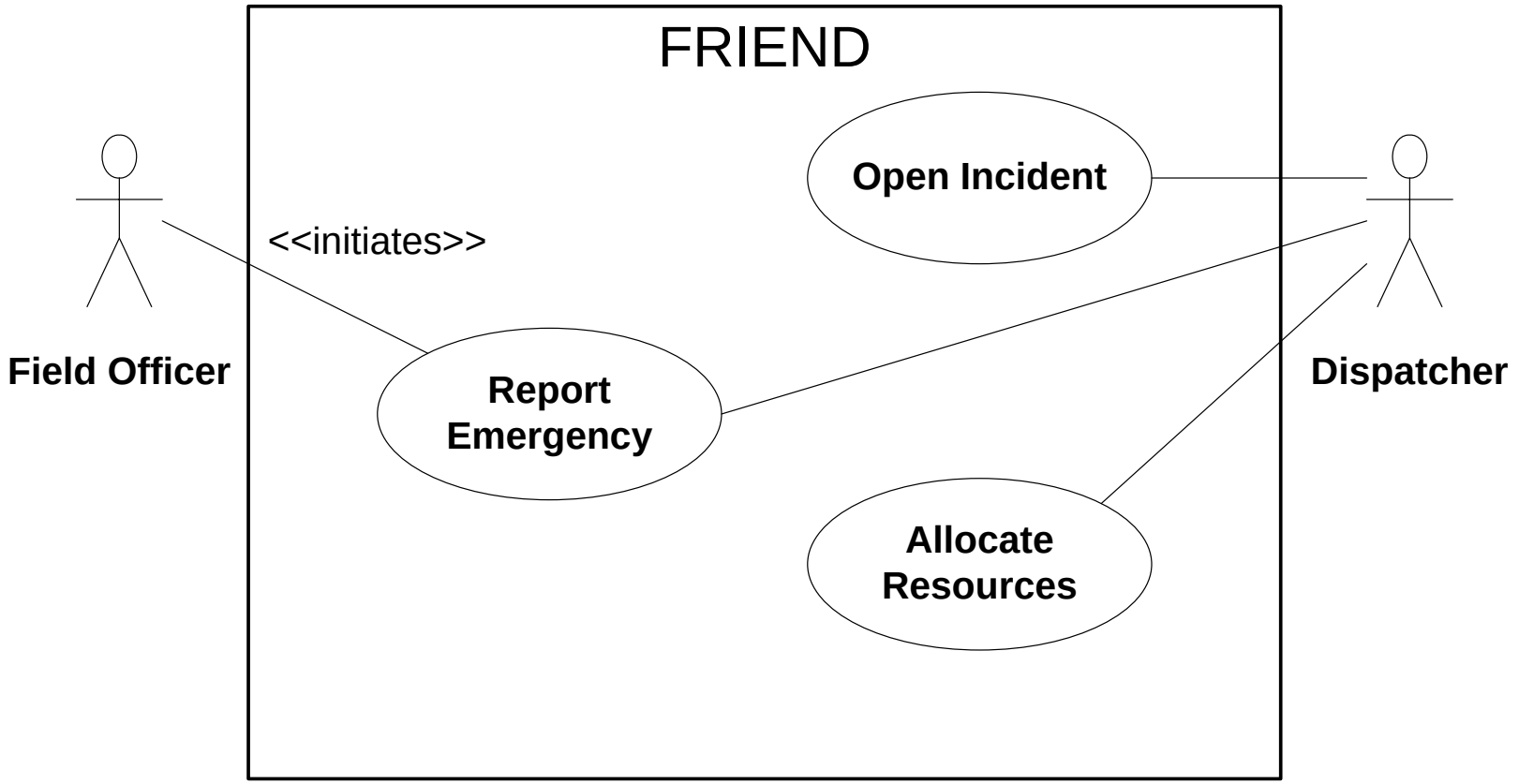
Refinement Example (I)

1. The `FieldOfficer` activates the "Report Emergency" function on her terminal.
2. FRIEND responds by presenting a form to the `FieldOfficer`. The form includes an emergency type menu (general emergency, fire,transportation), a location, incident description, resource request, and hazardous material fields.
3. The `FieldOfficer` fills the form 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`.

Refinement Example (II)

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

Use Case Diagram for FRIEND



6. Identify relationships among use cases

- Even simple systems have a lot of use cases
- Use case association = relationship between use cases
- Goal: Reduce complexity, Increase understandability, maximize reuse of use cases
- Important types:
  - Extend
    - A use case extends another use case (separate exceptional and common flows of events)
  - Include
    - A use case uses another use case (“functional decomposition”)
  - Generalization
    - “Use case experts have been successfully doing use case work without this optional relationship […] and there is not yet agreement by practitioners on the best-practice guidelines of how to get value from this idea.” [Larman]

ConnectionDown

The ConnectionDown 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 occur (e.g., 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 though other means and the Dispatcher initiates *ReportEmergency* from the Dispatcher station

Summary

- Scenarios and Use Cases is one way to document and formalize requirements in a form which is understandable by users and clients
- Scenarios are the basis to derive Use Cases
- Use Cases associations can be used to simplify the use case model: `extend`, `include`
- They are modeled as stereotypes of UML dependencies
- Other associations could be imagined: sequential dependencies …

Tips from Authors

- Lethbridge:
  - A use case should describe the user's interaction with the system, not the computations the system performs.
  - A use case should be written to be as independent as possible from any particular user interface design.
    - No: Push the Open button
    - Yes: Choose the Open command.
  - In general, a use-case should cover the full sequence of steps from the beginning of a task until the end.
- Fowler: Although use cases have been around for a while, there's been little standardization on their use. The UML is silent on the important contents of a use case and has standardized only the much less important diagrams. As a result, you can find a divergent range of opinions on use cases.
- Breugge & Dutoit : Do not overstructure the use case model. A few longer use cases (eg. 2 pages long) are easier to understand and review than many short ones (eg. ten lines long)

Advantages of using Use-Cases

- Helps define the scope of the system (what it does and does not do)
- Can be used as part of the development plan
  - # use cases is an indicator of project size
  - progress can be measured by % use cases completed
- Form the basis of definition of test cases
- Can be used to structure user manuals

Disadvantages of Use-Cases

1) Use cases themselves must be validated

2) Some aspects of functional requirements are not covered by use case analysis, only those triggered by an actor.

3) You say !

Software Requirements Elicitation and Specifications

- Fundamentals
- Requirements Elicitation Process

  (Requirements Elicitation Based on Use Cases and Scenarios (from Bruegge and Dutoit, 2000))
- Documentation

Requirements Definition document

- Defines system to be built from the customer's perspective
- Customer needs to understand the document
- Basis of contract between customer and system developer
- Requirements ought to be complete and consistent!!!

Software requirements document

- Cover page
- Introduction
- Functional requirements specification
- Nonfunctional requirements (e.g. standards to be met, platform, memory requirements)
- Glossary

Software requirements document cover page

- Name of the project/product
- Date
- Version number
- Author(s)
- Responsibilities of every author
- Key changes since last version

Format and Style

- Modifiability
  - Well-structured, indexed, cross-referenced
  - Little and explicit redundancy
- Traceability
  - Backwards, e.g., stakeholder, document
  - Forwards, e.g., to design, test plan
- Useful annotations: levels of necessity, stability

Review - questions

- Important interfaces described?
- Major functions within scope?
- Design constraints realistic?
- Technological risk considered?
- Clear validation criteria stated?
- Do inconsistencies, omissions, redundancy exist?

Review - guidelines

- Lookout for persuasive connectors (certainly, therefore, obviously,….)
- Watch for vague terms (some, often, usually,….)
- Lists complete (no etc., Such as,…)
- Check the use of terms: always the same meaning in document?
- Beware ambiguity and vague statements (e.g., undefined terminology)

Standards

- IEEE-STD-830-1993.
  - IEEE Recommended Practice for Software Requirements Specifications
- MIL-STD-498.
  - Military Standard for Software Development and Documentation.
- DO-178B.
  - Promotes traceability between high-level requirements down to code statements (airborne systems)

Similar standards exist for public transportation (train), power plants …