Modern methods in Software Engineering

Requirements Elicitation

Introduction Content

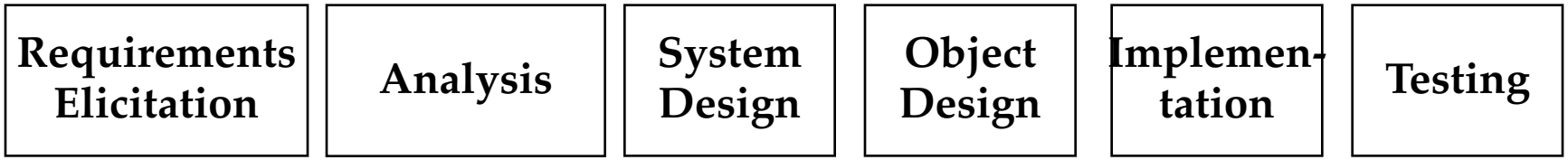
- Reminding Software lifecycle
  - requirements elicitation
  - requirements analysis
- Problem statement
- Requirements
  - functional
  - non-functional
- Methods of requirements elicitation
- Identifying scenario and use cases

Literature used

- Text book
  - Chapter 4

Activities of a Software Project

Activities we use



Each activity produces one or more models

Requirements process

- The requirements process consists of two activities:
  - Requirements Elicitation:
    - Definition of the system in terms understood by the customer (“Problem Description”)
  - Requirements Analysis:
    - Technical specification of the system in terms understood by the developer (“Problem Specification”)

Requirements

- Requirement: A statement about the proposed system that all stakeholders agree must be made true in order for the customer's problem to be adequately solved.
  - Short and concise piece of information
  - Says something about the system
  - All the stakeholders have agreed that it is valid
  - It helps solve the customer's problem

- A collection of requirements is a *requirements document*.

Principles of requirements determination

- Requirements determination:
  - The least technical phase of system development
  - Demands social, communication and managerial skills
  - Delivers a (mostly narrative) definition of user requirements
- Requirements definition
  - System services → functional requirements
    - scope of the system
    - necessary business functions
    - required data structures
  - System constraints → nonfunctional requirements
    - regarding 'look and feel', performance, security, etc.
    - also known as supplementary requirements

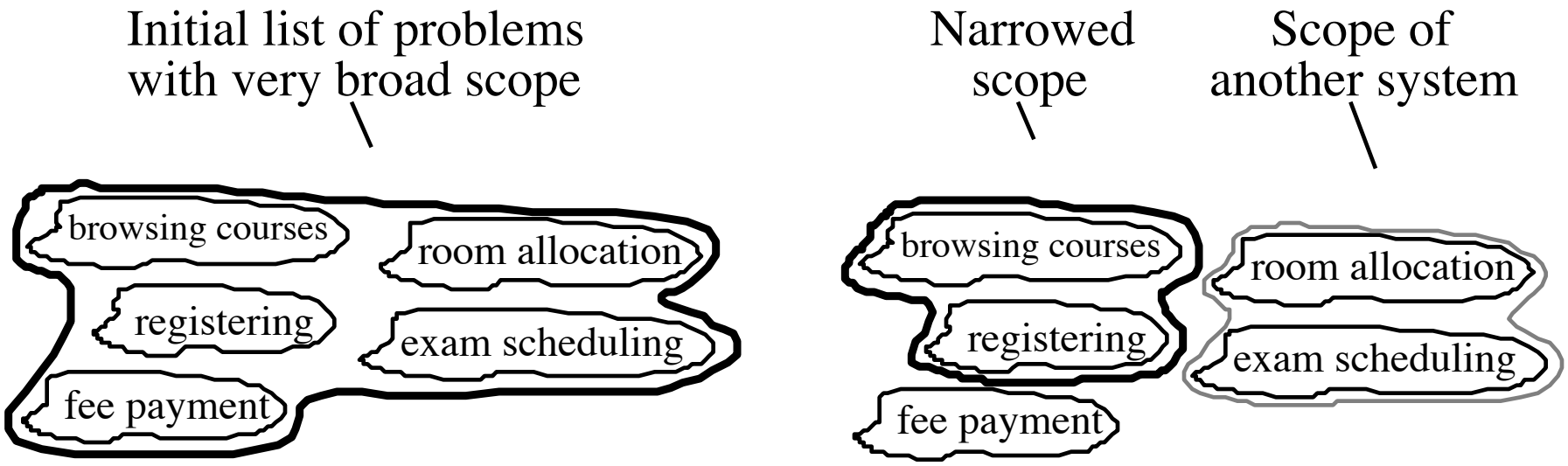
System Identification

- The development of a system is not just done by taking a snapshot of a scene (domain)
- Two questions need to be answered:
  - How can we identify the purpose of a system?
  - Crucial is the definition of the system boundary: What is inside, what is outside the system?
- These two questions are answered in the requirements process

What do you see here?

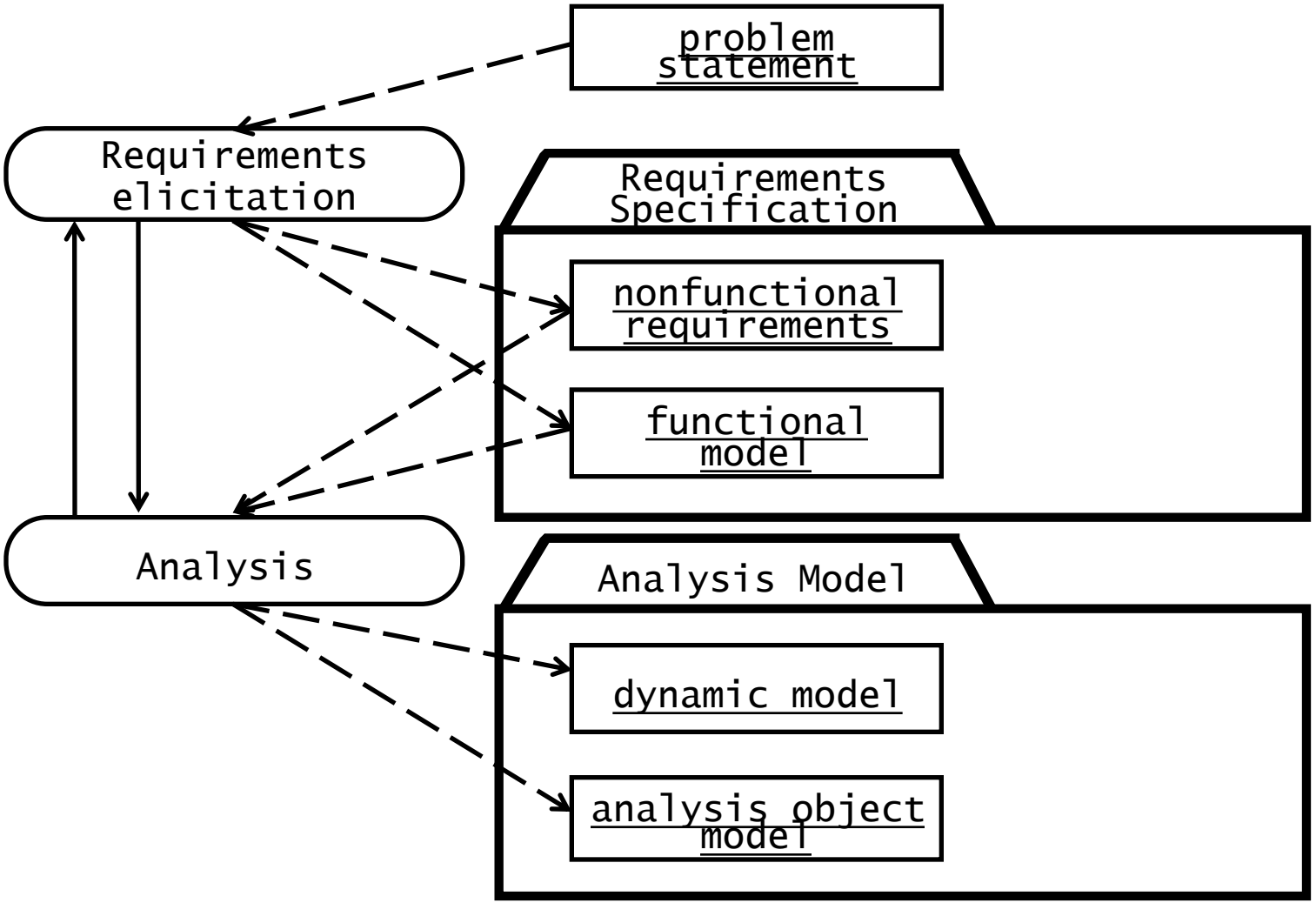
What do you see here?

Defining the Scope



Adopted from Lethbridge & Laganiere

Products of Requirements Process



Requirements Elicitation

- Very challenging activity
- Requires collaboration of people with different backgrounds
  - Users with application domain knowledge
  - Developer with solution domain knowledge (design knowledge, implementation knowledge)
- Bridging the gap between user and developer:
  - ***Scenarios:*** Example of the use of the system in terms of a series of interactions between the user and the system
  - ***Use cases:*** Abstraction that describes a class of scenarios
- The starting point is the problem statement

Problem statements

- A problem can be expressed as:
  - A *difficulty* the users or customers are facing,
  - Or as an *opportunity* that will result in some benefit such as improved productivity or sales.
- The solution to the problem normally will entail developing software
- A good problem statement is short and concise

Problem statement

- The problem statement is developed by the client as a description of the problem addressed by the system
- A good problem statement describes
  - The current situation
  - Objectives of the system
  - The functionality the new system should support
  - The environment in which the system will be deployed
  - Deliverables expected by the client
  - Delivery dates
  - A set of acceptance criteria

Current Situation: The Problem To Be Solved

- There is a problem in the current situation
  - Examples:
    - The response time when playing letter-chess is far too slow.
    - I want to play Go, but cannot find players on my level.
- What has changed? Why can we address the problem now?
  - There has been a change, either in the application domain or in the solution domain
  - ***Change in the application domain***
    - A new function (business process) is introduced into the business
    - Example: We can play highly interactive games with remote people
  - ***Change in the solution domain***
    - A new solution (technology enabler) has appeared
    - Example: The internet allows the creation of virtual communities.

ARENA: The Problem – Current Situation

- The Internet has enabled virtual communities
  - Groups of people sharing common of interests but who have never met each other in person. Such virtual communities can be short lived (e.g people in a chat room or playing a multi player game) or long lived (e.g., subscribers to a mailing list).
- Many multi-player computer games now include support for virtual communities.
  - Players can receive news about game upgrades, new game levels, announce and organize matches, and compare scores.
- Currently each game company develops such community support in each individual game.
  - Each company uses a different infrastructure, different concepts, and provides different levels of support.
- This redundancy and inconsistency leads to problems:
  - High learning curve for players joining a new community,
  - Game companies need to develop the support from scratch
  - Advertisers need to contact each individual community separately.

ARENA: The Objectives

- Provide a generic infrastructure for operating an arena to
  - Support virtual game communities.
  - Register new games
  - Register new players
  - Organize tournaments
  - Keeping track of the players scores.
- Provide a framework for tournament organizers
  - to customize the number and sequence of matchers and the accumulation of expert rating points.
- Provide a framework for game developers
  - for developing new games, or for adapting existing games into the ARENA framework.
- Provide an infrastructure for advertisers.

Types of requirements

- Functional requirements:
  - Describe the interactions between the system and its environment independent from implementation
  - Examples:
    - An ARENA operator should be able to define a new game.
- Nonfunctional requirements:
  - User visible aspects of the system not directly related to functional behavior.
  - Examples:
    - The response time must be less than 1 second
    - The ARENA server must be available 24 hours a day
- Constraints ("Pseudo requirements"):
  - Imposed by the client or the environment in which the system operates
    - The implementation language must be Java
    - ARENA must be able to dynamically interface to existing games provided by other game developers.

Functional requirements

- What *inputs* the system should accept
- What *outputs* the system should produce
- What data the system should *store* that other systems might use
- What *computations* the system should perform
- The *timing* and *synchronization* of the above

Non-Functional Requirements (main types)

1. Categories reflecting: usability, efficiency, reliability, maintainability

- **Usability** – ease of use
  - documentation and help, training, user interface, error handling
- **Reliability**
  - mean time between failures, graceful recovery, dependable (we can depend on it) = reliability +robustness+safety
- **Performance**
  - response time, transaction throughput, resource consumption, concurrency level, uninterrupted availability, accuracy of results
  - efficiency – in terms of time and cost
- **Supportability** = adaptability + maintainability + portability

Non-Functional Requirements (main types)

2. Categories constraining the *environment and technology* of the system.
   - Implementation, interface, operations, packaging, policy and legal issues

3. Categories constraining the *project plan and development methods*
   - Development process (methodology) to be used
   - Cost and delivery date
     - Often put in contract or project plan instead

Example of requirements from the text-book

- SatWatch is a 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 need 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 during blackout periods). In such 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, minutes, seconds, time zone) and on the bottom line, the date. 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.

Nonfunctional requirements for SatWatch.

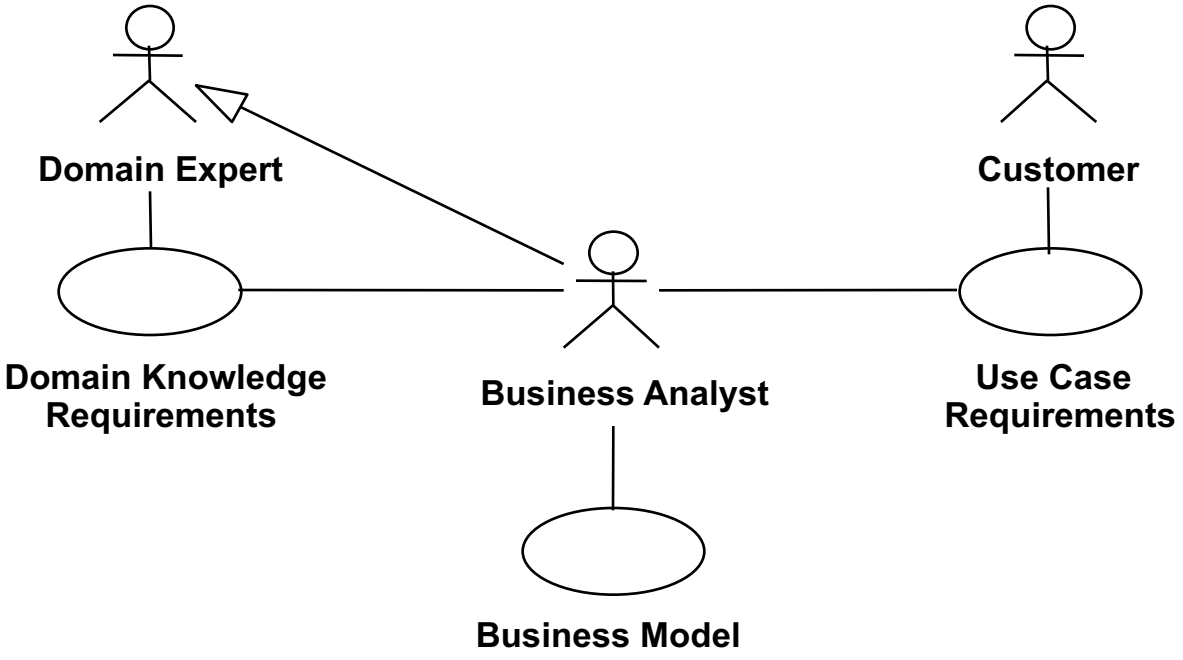
- Quality requirements for SatWatch
  - Any user who knows how to read a digital watch and understands international time zone abbreviations should be able to use SatWatch without the user manual.
    [**Usability requirement**]
  - As the watch doesn't have buttons, no software faults requiring the resetting of the watch should occur.
    [**Reliability requirement**]
  - SatWatch should display the correct time zone within 5 minutes of the end of a GPS blackout period.
    [**Performance requirement**]
  - SatWatch should measure time within 1/l00th second over 5 years.
    [Performance requirement]
  - SatWatch should display the correct time in all 24 time zones.
    [**Performance requirement**]
  - SatWatch should accept upgrades to its onboard via the Webify Watch serial interface.
    [**Supportability requirement**]
- Constraints for SatWatch
  - All related software associated with SatWatch, including the onboard software, will be written using Java to comply the company policy.
    [**Implementation requirement**]
  - SatWatch complies with the physical, electrical, and software interfaces defined by WebifyWatch API 2.0.
    [**Interface requirement**]

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 above are constrained by the client. Fight for it!

Requirements elicitation (actors)



Adopted from Maciazek

Methods of Requirements Elicitation

- Traditional methods
  - Interviewing customers and domain experts
  - Questionnaires
  - Observation
  - Study of documents and software systems
- Modern methods
  - Prototyping
  - Brainstorming
  - Joint Application Development (JAD)
  - Rapid Application Development (RAD)

Interviewing customers and domain experts

- **With customers** – mostly use case requirements
- **With domain experts** – frequently a straight knowledge transfer
- **Structured** (formal) interview
  - Open-ended questions (unanticipated responses)
  - Close-ended questions (a list of possible responses known)
- **Unstructured** (informal) interview
- **Questions to be avoided**
  - **Opinionated** questions (do we have to do things the way we do them?)
  - **Biased** questions (you are not going to do this, are you?)
  - **Imposing** questions (you do things this way, don't you?)
- **Summary report** of the interview should be sent to the interviewee within a day or two, along with a request for comments

Adopted from Maciazek

Interviewing customers and domain experts (possible questions)

- about specific details
  - 5 Ws: what, who, when, where, and why.
- about vision for the future
- about alternative ideas
  - these may be questions to an interviewee and suggestions from the interviewer
- about minimally acceptable solution to the problem
  - good usable systems are simple systems
- about other sources of information
  - can discover important documents and other knowledge sources unknown before to the interviewer
- soliciting diagrams
  - drawn by an interviewee to explain business processes may prove invaluable for understanding the requirements

Adopted from Maciazek

Questionnaires

- **In addition** to interviews
- A **passive** technique
  - advantage – time to consider the answers
  - disadvantage – no possibility to clarify questions and answers
  - who are the people which did not respond? how would they respond?
- **Close-ended questions**
  - **Multiple-choice** questions
    - additional comments may be allowed
  - **Rating** questions (e.g. strongly agree, agree, ...)
    - when seeking opinions
  - **Ranking** questions
    - ranked with numbers, percent values, etc.

Adopted from Maciazek

Observation

- When the user cannot convey sufficient information and/or has only fragmented knowledge
- **Three forms**
  - **Passive**
    - no interruption or direct involvement
    - video camera may be used
  - **Active**
    - Participating in activity
  - **Explanatory**
    - explaining what is done when observed
- Should be carried for a **prolonged time**, at different times and at different workloads
- People tend to **behave differently** when watched

Adopted from Maciazek

Study of documents and software systems

- **Always** used, but may target portions of the system
- **Use case** requirements studied through
  - Organizational documents
    - including procedures, policies, descriptions, plans, charts, internal and external correspondence, complains…
  - System forms and reports (if prior computer system exists)
    - record of change requests (defects and enhancements)
- **Domain knowledge** requirements study
  - domain journals and reference books
  - using Internet searches

Adopted from Maciazek

Methods of Requirements Elicitation

- Traditional methods
  - Interviewing customers and domain experts
  - Questionnaires
  - Observation
  - Study of documents and software systems
- Modern methods
  - Prototyping
  - Brainstorming
  - Joint Application Development (JAD)
  - Rapid Application Development (RAD)

Prototyping

- '**Quick and dirty**' solution to obtain feedback
- Necessary in **complex** and **innovative** projects
- Two kinds:
  - **Throw-away** prototype
    - targets requirement determination
  - **Evolutionary** prototype
    - targets the speed of product delivery

Adopted from Maciazek

Brainstorming

- It is a conference technique to form new ideas or to find a solution to specific problem **by putting aside** judgment, criticism, social inhibitions and rules to reach consensus among stakeholders
- The process:
  - prior to meeting, the **facilitator** provides **probortunity statement** (the problem/opportunity area)
    - generic trigger questions to challenge the participants (e.g. what features should the system support? what are the main 'business objects'?)
  - 12 to 20 participants **around a table**, feeling equal with the facilitator "one of the crowd"
  - answers to trigger questions **recorded and passed around** to stimulate more answers/ideas
  - answers/ideas get **anonymous**
  - answers/ideas are **discussed**
  - **voting** to prioritize answers/ideas

Adopted from Maciazek

Some typical trigger questions

- What features should be supported by the system?
- What are the input data and what are the outputs of the system?
- What questions should be raised in interviews or in questionnaires?
- What issues still need to be considered?
- What are the main risks in the project?
- What trigger questions should be asked during this or future brainstorming sessions?

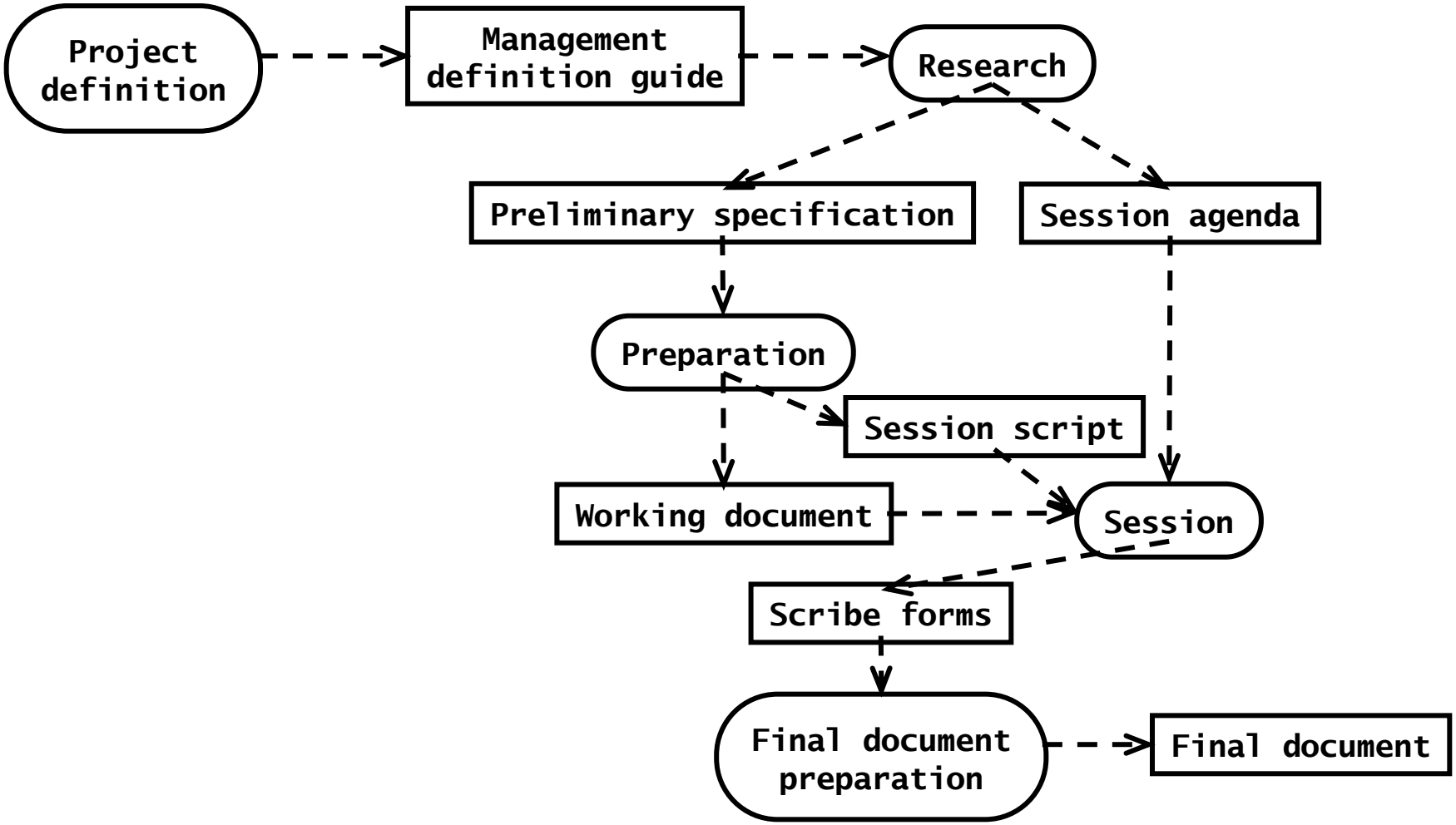
Adopted from Lethbridge and Laganiere

Joint Application Development (JAD)

- Brainstorming-like technique capitalizing on **group dynamics**
- Frequently part of **facilities management** - when the operation/implementation of an organization's information systems are contracted out to a third party
- The **membership**
  - Executive sponsor: System owner
  - Project leader
  - Session Leader/Facilitator (communication skills, not a stakeholder)
  - Scribe (touch typing, software development knowledge, CASE tool skills)
  - Participants/Customers: Users, Managers
  - Developers/Observers: development team

Adopted from Maciazek

JAD


Project definition
Management definition guide
Research
Preliminary specification
Session agenda
Preparation
Session script
Working document
Session
Scribe forms
Final document preparation
Final document

Rapid Application Development (RAD)

- Five techniques
  - **Evolutionary prototyping**
  - **CASE** tools
    - with code generation and round-trip engineering
  - Specialists with Advanced Tools (**SWAT**)
    - co-located with the users
  - **Interactive JAD**
    - scribe replaced by a SWAT team with collaborative CASE tools
  - **Timeboxing**
    - no 'scope creep', time-boxed project
- Problems
  - suitable for **smaller projects**; too risky for larger developments
    - inconsistent GUI designs
    - specialized solutions with little reuse potential
    - deficient docs
    - unsupportable solution likely

Requirements negotiation and validation

- Needed because requirements
  - **overlap** and **conflict** may be **ambiguous** or **unrealistic**
  - may remain **undiscovered**
  - may be **out of scope**
    - sometimes out of the "project" scope, but in the scope of the "information system"
- frequently done in parallel with requirements elicitation
- inseparable from the production of requirements document

Adopted from Maciazek

Requirements validation

- **Requirements validation criteria:**
  - **Correctness:**
    - The requirements represent the client's view.
  - **Completeness:**
    - All possible scenarios, in which the system can be used, are described, including exceptional behavior by the user or the system
  - **Consistency:**
    - There are no functional or nonfunctional requirements that contradict each other
  - **Unambiguity:**
    - A requirements cannot be interpreted in mutually exclusive ways

Requirements validation criteria:

- **Realism**
  - if the system can be implemented within constraints
- **Verifiability**
  - if repeatable tests can be designed to demonstrate its work
- **Traceability**
  - if each requirement can be traced throughout the software development

- Problem with requirements validation:
  - Requirements change very fast during requirements elicitation.

Examples of requirements

- The product shall have a good user interface
- The product shall be error free
- The product should respond in 1 second in most cases

Question

Specify which of these requirements are verifiable and which are not:

- “The ticket distributor must enable a traveler to buy weekly passes.”
- “The ticket distributor must be written in Java.”
- “The ticket distributor must be easy to use.”

Requirements risks and priorities

- **Risk** is a threat to the project plan
- Risks determine project's **feasibility**
- **Risk analysis** identifies requirements that are likely to cause development difficulties
- **Prioritization** is needed to allow easy re-scoping of the project when faced with delays

Adopted from Maciazek

Some risk categories

- **technical risk** - when a requirement is technically difficult to implement;
- **performance risk** - a requirement, when implemented, can adversely affect the response time of the system;
- **security risk** - when a requirement, when implemented, can expose the system to security breaches;
- **development process risk** - when a requirement calls for the use of unconventional development methods unfamiliar to developers (e.g. formal specification methods);
- **political risk** - when a requirement may prove difficult to fulfill for internal political reasons
- **legal risk** - when a requirement may fall foul of current laws or anticipated changes to the law;
- **volatility risk** - when a requirement is likely to keep changing or evolving during the development process.

Prioritizing requirements

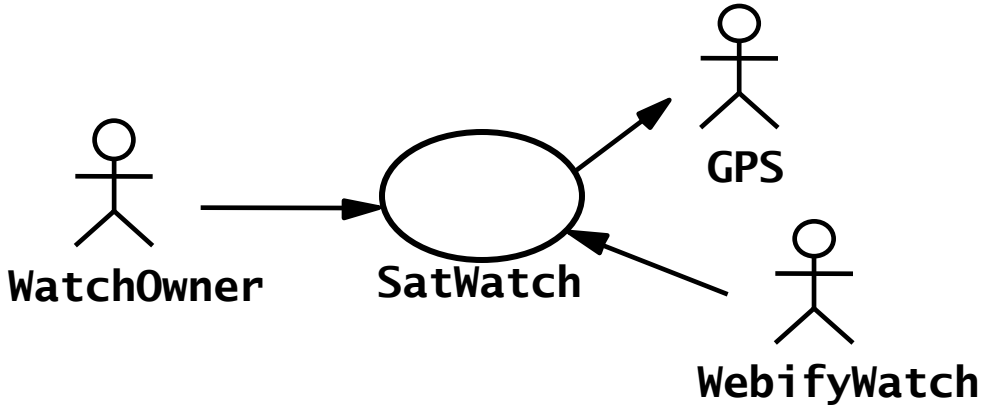
- High priority (“Core requirements”)
  - Must be addressed during *analysis, design, and implementation*.
  - A high-priority feature must be demonstrated successfully during client acceptance.
- Medium priority (“Optional requirements”)
  - Must be addressed during *analysis and design*.
  - Usually implemented and demonstrated in the second iteration of the system development.
- Low priority (“Fancy requirements”)
  - Must be addressed during *analysis* (“very visionary scenarios”).
  - Illustrates how the system is going to be used in the future if not yet available technology enablers are available

Requirements Elicitation Activities

- Identify Actors
- Identify Scenarios
- Identify Use Cases
- Refine Use Cases
- Identify Relationships Among Actors and Use Cases
- Identify Initial Analysis Objects
- Identify Nonfunctional Requirements

Actors identifying

- Actors are role abstractions



- Actors and objects
  - Actors are external
  - Objects are internal

Examples of questions for identifying actors

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

Identifying 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.

Identifying Scenarios

- Don't expect the client to be verbal if the system does not exist
- Don't wait for information even if the system exists
- Engage in a dialectic approach (evolutionary, incremental engineering)
  - You help the client to formulate the requirements
  - The client helps you to understand the requirements
  - The requirements evolve while the scenarios are being developed

Types of Scenarios

- As-is scenario:
  - Used to describe a current situation. Usually in reengineering. The user describes the system.
- Visionary scenario:
  - Used to describe a future system. Usually used in Greenfield engineering and Reengineering projects.
  - Can often not be done by the user or developer alone
- Evaluation scenario:
  - User tasks against which the system is to be evaluated.
- Training scenario:
  - Step by step instructions that guide a novice user through a system

Types of Scenarios

- Scenarios can have many different uses during the software lifecycle
  - ***Requirements Elicitation***: As-is scenario, visionary scenario
  - ***Client Acceptance Test:*** Evaluation scenario
  - ***System Deployment:*** Training scenario.

Finding Scenarios (heuristics)

- Ask 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 does the system need to know about?
  - What changes or events will the actor of the system need to be informed about?
- Don't rely on *questionnaires* alone.
- Insist on *task observation* if the system already exists (interface engineering or reengineering)
  - Ask to speak to the end user, not just to the software contractor
  - Expect resistance and try to overcome it

Example questions: Accident Management System

- What needs to be done to report a "Cat in a Tree" incident?
- What do you need to do if a person reports "Warehouse on Fire?"
- Who is involved in reporting an incident?
- What does the system do, if no police cars are available? If the police car has an accident on the way to the "cat in a tree" incident?
- What do you need to do if the "Cat in the Tree" turns into a "Grandma has fallen from the Ladder"?
- Can the system cope with a simultaneous incident report "Warehouse on Fire?"

Scenario Example: Warehouse on Fire

- 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 area appear 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.

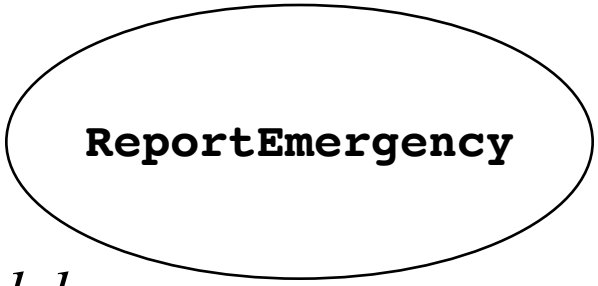
- Alice received the acknowledgment and the ETA.

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 and John

Identifying Use Cases

- A use case is a flow of events in the system, including interaction with actors
- It is initiated by an actor, after initation a use case may interact with other actors
- Each use case has a name
- A use case represents a complete flow of events
- Each use case has a termination condition



Use Case Model:

The set of all use cases specifying the complete functionality of the system

Simple Use Case Writing Guide

- Use cases should be named with verb phrases.
- Actors should be named with noun phrases
- The boundary of the system should be clear.
- Use case steps in the flow of events should be phrased in the active voice.
- The causal relationship between successive steps should be clear.
- A use case should describe a complete user transaction
- Exceptions should be described separately.
- A use case should not describe the user interface of the system.
- A use case should not exceed two or three pages in length.

Formulating use cases (first version)

1. Name the use case
   – Use case name: Accident

   Bad name – not a verb. ReportEmergency is much better
2. Find the actors

   - Generalize the concrete names ("Bob") to participating actors ("Field officer")
   - Participating Actors:
     - Field Officer (Bob and Alice in the Scenario)
     - Dispatcher (John in the Scenario)
3. Then concentrate on the flow of events

   - Use informal natural language
     - The FieldOfficer reports the accident
     - An ambulance is dispatched

       Causality: Which action caused the FieldOfficer to receive an acknowledgement? Passive voice. Who dispatches the ambulance?
     - The Dispatcher is notified when the ambulance arrives on site

       Incomplete transaction: What does the FieldOfficer do after the ambulance is dispatched?
4. Entry conditions: Officer is logged in into the system
5. Exit conditions: receiving acknowledgement or explanation of failure
6. Quality conditions: receiving acknowledgement within 30 sec

Refining use cases

- **Whereas the initial identification of use cases and actors focused on establishing the boundary of the system, the refinement of use cases yields increasingly more details about the features provided by the system and the constraints associated with them:**
  - The elements that are manipulated by the system are detailed.
  - The low-level sequence of interactions between the actor and the system is specified.
  - Access rights (which actors can invoke which use cases) are specified.
  - Common functionality among use cases are factored out.

Use Case Example: ReportEmergency (refined)

- Use case name: ReportEmergency
- Participating Actors:
  - Field Officer (Bob and Alice in the Scenario)
  - Dispatcher (John in the Scenario)
- 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.
- Flow of Events: **on next slide.**

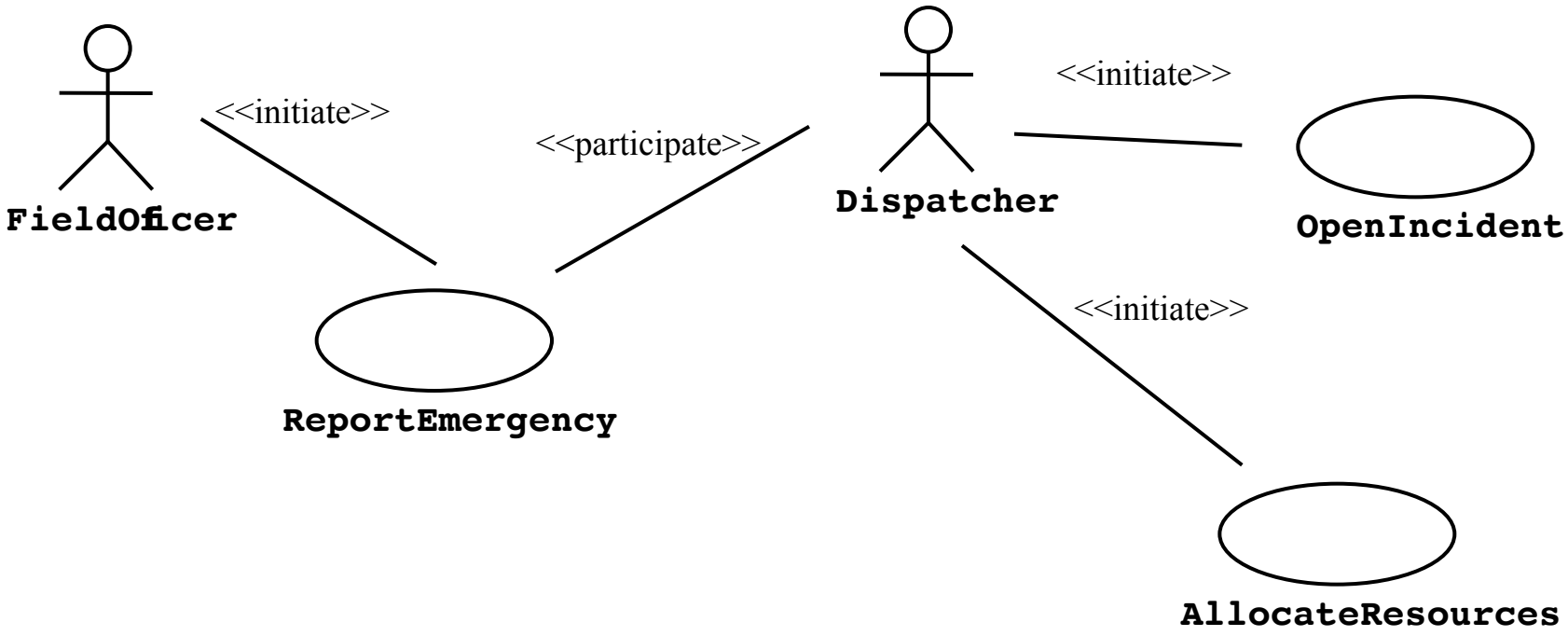
Use Case Example: ReportEmergency Flow of Events

- The **FieldOfficer** activates the "Report Emergency" function of her terminal.
- FRIEND (the system) responds by presenting a form to the officer.
- 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
- FRIEND receives the form and notifies the **Dispatcher**.
- 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.
- The FieldOfficer receives the acknowledgment and the selected response.

Identifying relationships among actors and use cases

- Even small and medium-sized systems may have many use cases
- Relationships between actors and use cases allow reduce complexity of the model and increase readability
- Communication relations between actors and use cases describe system in layers of functionality
- Extend relationships separate exceptional and common flows
- Include relationships reduce redundancy among the use cases

Communication relationships

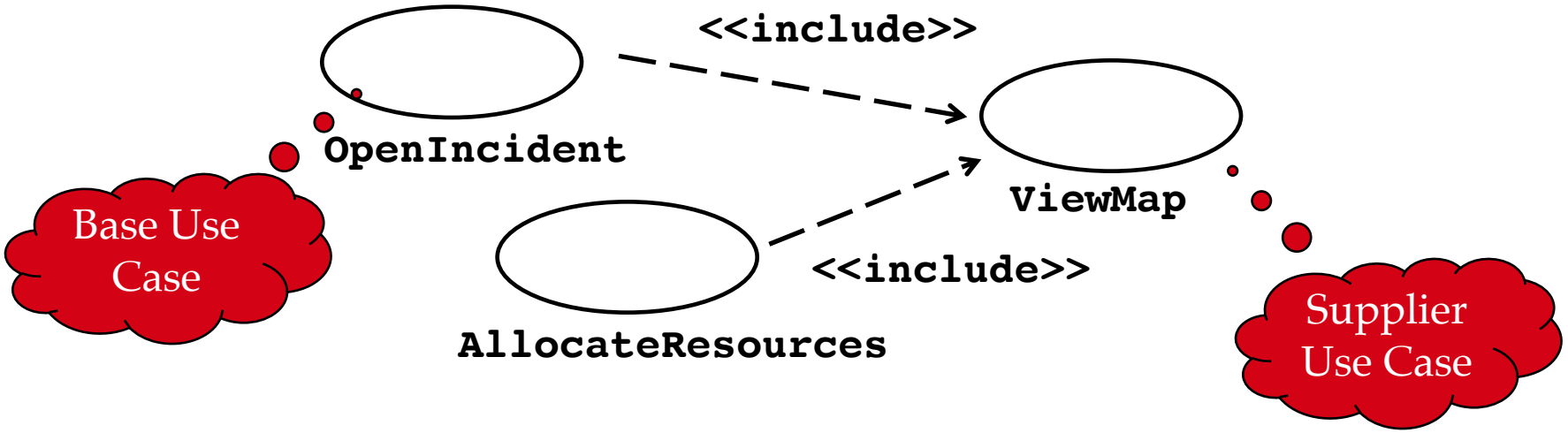


extend vs. include

- Use extend relationships for exceptional, optional or seldom-occurring behavior
- Use include relationships for behavior that is shared across two or more use cases
- Use discretion when applying the above heuristics and do not overstructure the use case model.

<<include>>: Reuse of Existing Functionality

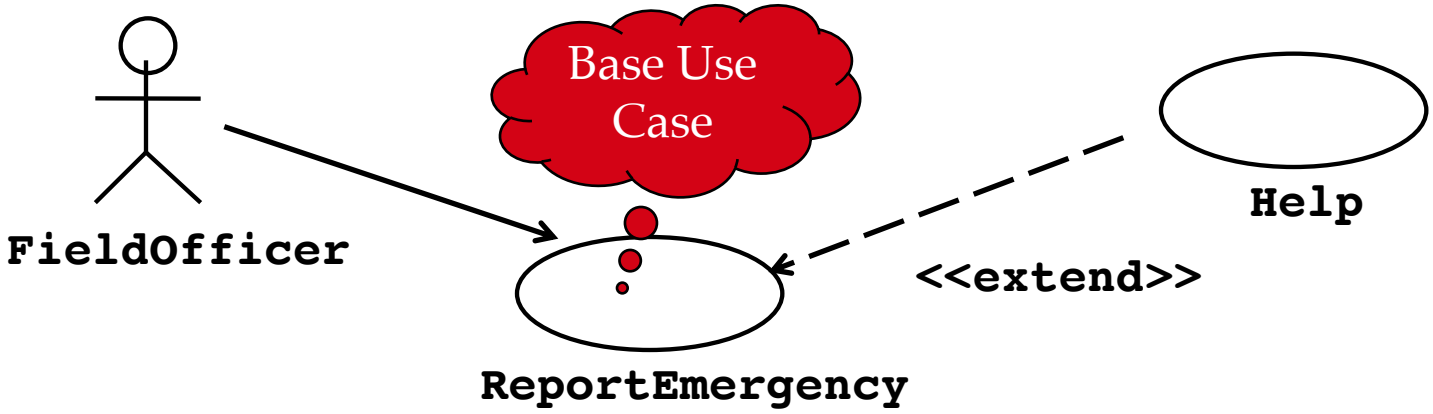
- Problem:
  - There are already existing functions. How can we *reuse* them?
- Solution:
  - The *include association* from a use case A to a use case B indicates that an instance of the use case A performs all the behavior described in the use case B ("A delegates to B")



Note: The base case cannot exist alone. It is always called with the supplier use case

<<extend>> Association for Use Cases

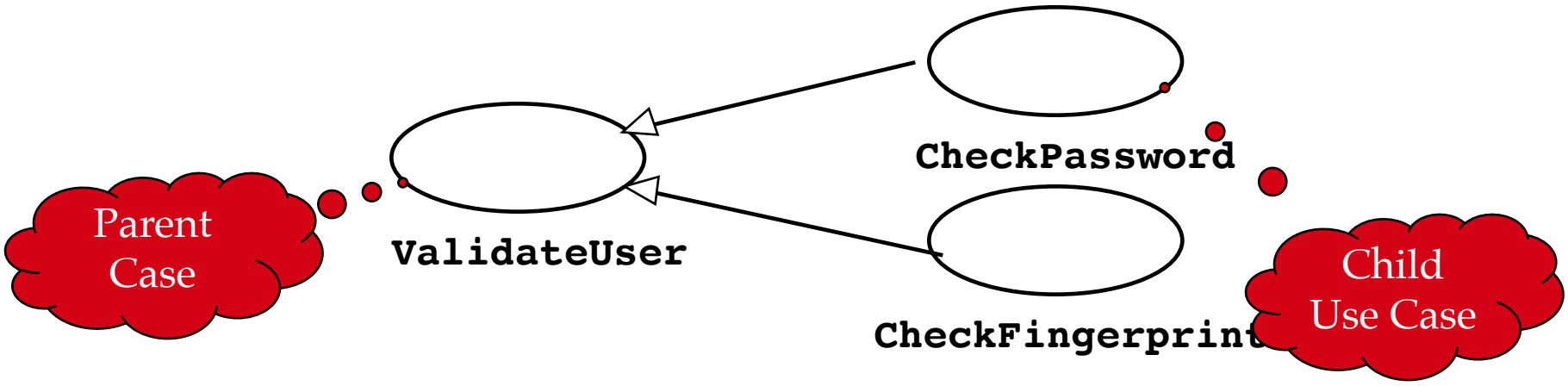
- Problem:
  - The functionality in the original problem statement needs to be extended.
- Solution:
  - An *extend association* from a use case A to a use case B indicates that use case B is an extension of use case A.



Note: The base use case can be executed without the use case extension in extend associations.

Generalization association in use cases

- Problem:
  - You have common behavior among use cases and want to factor this out.
- Solution:
  - The generalization association among use cases factors out common behavior. The child use cases inherit the behavior and meaning of the parent use case and add or override some behavior.



Identifying initial analysis objects

- Different terminology of developers and user is obstacle
- Building glossary constitutes the first step toward analysis. The glossary is included in the requirements specification and, later, in the user manuals.
- Identify common terms (“participating objects”) for each use case
- Identification of common terms (“participating objects”) results in the initial analysis object model (this is the first step toward complete analysis object model)
- For example: Common terms (“participating objects”) in ReportEmergency
  - Dispatcher, EmergencyReport, FieldOfficer, Incident…

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.

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)?

Summary

- The requirements process consists of requirements elicitation and analysis.
- Functional and non-functional requirements. Domain knowledge and use cases.
- Scenarios:
  - Great way to establish communication with client
  - Different types of scenarios: As-Is, visionary, evaluation and training
  - Use cases: Abstraction of scenarios
- Pure functional decomposition is bad:
  - Leads to unmaintainable code
- Pure object identification is bad:
  - May lead to wrong objects, wrong attributes, wrong methods
- The key to successful analysis:
  - Start with use cases and then find the participating objects

Next lecture

- Requirements analysis

  chapter 5 in the text-book