02291: System Integration

Hubert Baumeister
hub@imm.dtu.dk

Spring 2012

Contents

1 Requirements Model

Activities in Software Development

- *Understand* and *document* what kind of the software the customer wants

  → Requirements Analysis

- Determine *how* the software is to be built

  → Design

- *Build* the software

  → Implementation

- *Validate* that the software solves the customers problem

  → Testing

  → Verification

  → Evaluation: e.g. User friendlieness

Travel Agency Example

The travel agency TravelGood comes to you as software developers with the following proposal for a software project:

Problem Description

TravelGood wants to offer a trip-planning and booking application to its customers. The application should allow the customer to plan trips consisting of flights and hotels. First the customer should be able to assemble the trip, before he then books all the flights and hotels in on step. The user should be able to plan several trips. Furthermore it should be possible to cancel already booked trips.

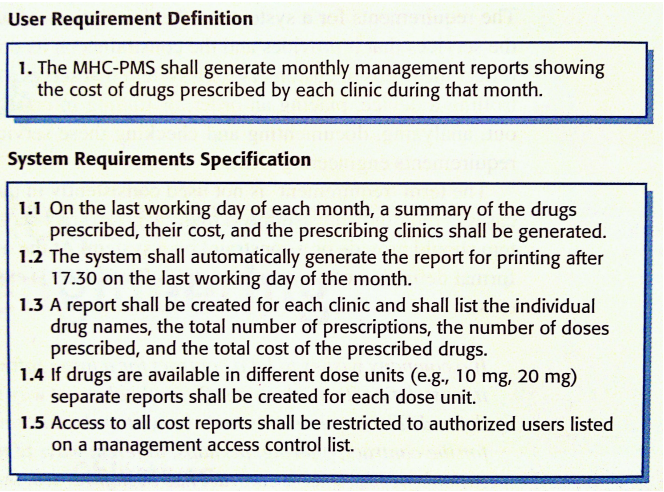
- What do you do?
- What are the requirements?

Requirements

- Requirements of a system define what the system should be doing
- It is the bases for the designer and the programmer to build the system
  - Defines the system to build such that the customer is satisfied
  - But also allows to plan how the system is built
- Customer and system builder need to agree on the requirements
- The requirements are usually defined in strong cooperation between customer and system builder

Usages of the term requirement

1. User Requirements
   - Used to describe the requirements in informal language and in broad terms.
   - Intended, e.g., to solicit bids from software companies
2. System Requirements
   - More precise than user requirements
   - Used in the contract phase to define how the system should work



User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Types of Requirements

Functional Requirements
Describe the users expectation *which functionalities* the system should have: E.g.

- the user should be able to plan and book a trip

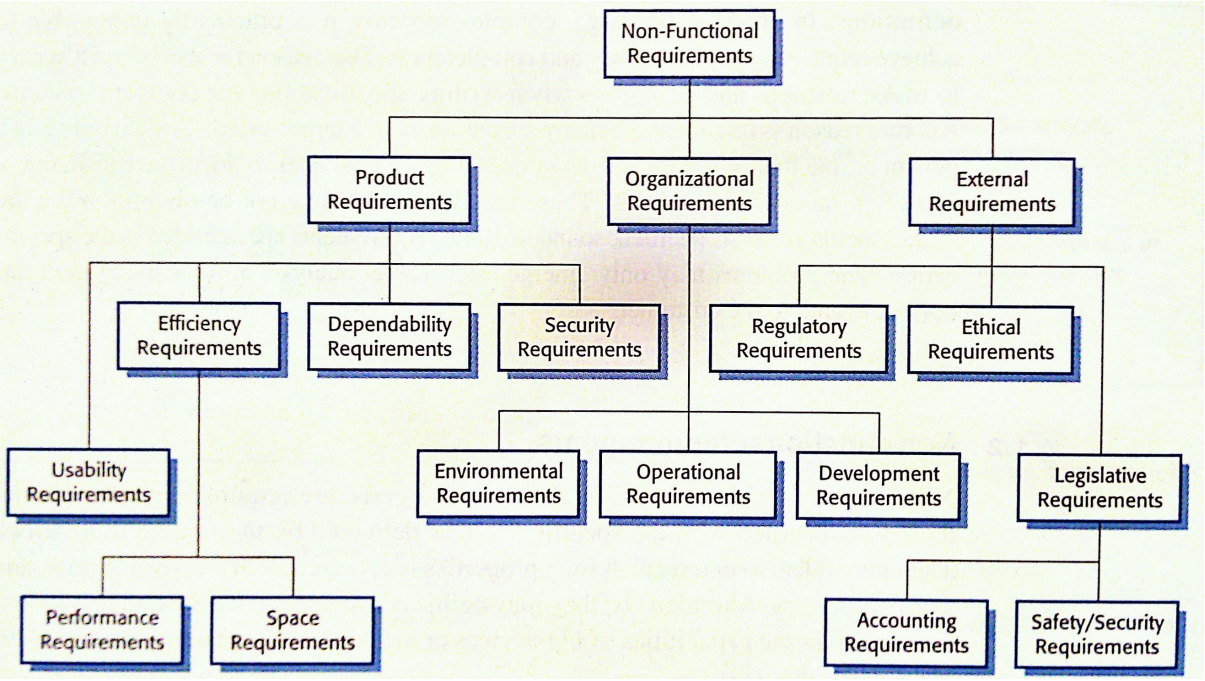
Non-functional Requirements
Everything which the user requires from the system, but which is not functionality

Examples of non-functional requirements

- Where should the software run (e.g. operating system, software environment, . . .)
- What kind of UI the user prefers (e.g. stand alone application, Web application, command line interface, graphical interface, . . .)

- Travel Agency Example of non-functional requirements
  - System should be a Web application accessible from all operating systems and most of the Web browsers
  - It must be possible to deploy the Web application in a standard Java application servers like GlassFish or Tomcat
  - The system should be easy to handle

Categories of non-functional requirements



Ian Sommerville, Software Engineering - 9

Characteristics of good requirements

- testable
  - One should be able to devise a test that can decide whether the system satisfies the requirements or not.
  - Tests can be manual or automatic: Nowadays the tests are preferrably automatic
- mesurable
  - To make non-functional requirements testable, they should be measurable

Example of measurable requirements

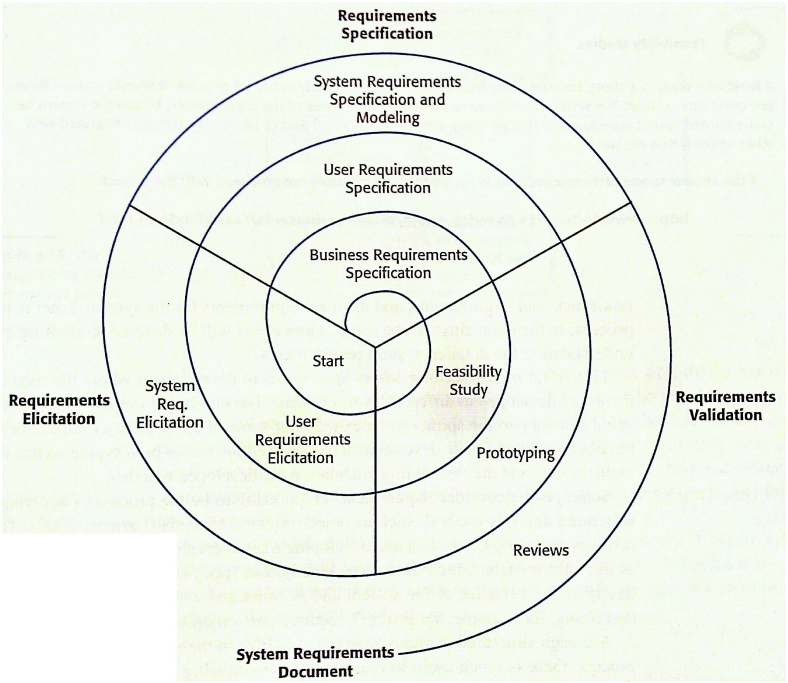
- The system should be easy to use by medical staff and should be organised in such a way that user errors are minimised
  - Can't be measured; when does the system satisfy the requirement?
- Better: Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

Possible measures

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure 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

Ian Sommerville, Software Engineering - 9

Requirements engineering process



Ian Sommerville, Software Engineering - 9

The process iterates to the three steps of requirements engineering

- Requirements elicitation
- Requirements documentation
- Requirements validation

More information on the details of this process can be found in course 02264: Requirements Engieering

Requirements Engineering

- Requirements elicitation and analysis

  - How to discover the requirements?
    * Several techniques
      · Interviews
      · *Use cases*
      · Scenarios → *User stories* in XP
      · ...

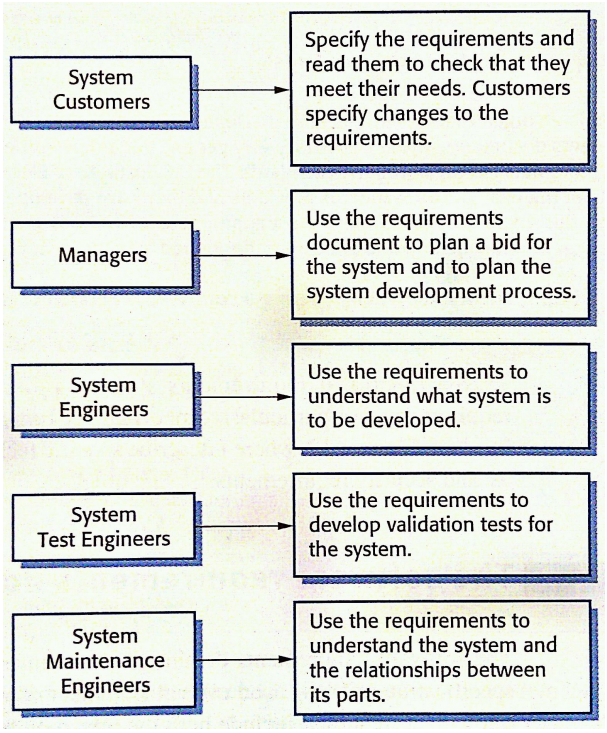
- Requirements documentation
  - Requirements documentation are important
    * to record the requirements (it is easy to forget requirements if they are not written down)
    * traceability: Important to traceback the design and implementation to the requirements
    * to agree upon requirements with the customer
      → requirements creep
        · Question: how to deal with changing and unclear requirements?
      → use an agile process
      → freeze the specification of the requirements as late as possible

- Requirements validation
  - Checks
    * Validity checks
      · Does the system fit the needs of the user?
    * Consistency checks
      · Are the requirements consistent with each other?
    * Completeness checks
      · Are they complete?
    * Realism checks
      · Can they be realised?
    * Verifiability
      · Can one decide if the system fulfils a requirement or not?
  - Validation techniques
    * Requirements reviews
    * Prototyping
    * Test-case generation

Contents of the software requirements document

- Generic document structure (IEEE standard)
  - Preface
  - Introduction
  - *Glossary*
  - User requirements definition
  - System architecture
  - System requirements specification
    * e.g. *Use Case Diagram* and *detailed use cases*
  - System models
    * *Domain model* (using a class diagram)
    * *Business Processes* (using activity diagrams) → next week

  - (System evolution) (added by Ian Summerville)
  - Appendices
  - Index
- Users of the software requirements document



Requirements issues

- *Refrain from inventing requirements*
  - Ask the customer what he wants
    * If you are in doubt how to interpret some requirements
    * If you have new ideas for requirements
    * If you think that requirements are missing
  - You waste time and resources if you build something which does not add value to the customer
- Problem descriptions can be very vague

  $\rightarrow$ it is important to *discuss* with the *customer* what his/her requirements are
- Requirements can change
  - e.g. after the *customer* has seen a first version of the software
  - the *business situation* has changed (cf. finance crises)

2 Domain model

Domain Model

- Purpose: capture the customer's knowledge of the domain so that the system builders have the *same knowledge*

- Helps customer and system builders to speak the same language

→ Necessary to define the terminology used

→ *Glossary*

→ Relationships between terms are shown in a *class diagram*

→ Related to the concept of an ontology

→ If necessary, make business processes visible

→ Represented by UML Activity Diagrams

Glossary

glossary (plural glossaries)
"1. (lexicography) A list of *terms* in a particular domain of knowledge with the *definitions* for those terms." (Wikitionary)

- List of terms with explanations
- Terms can be nouns (e.g. those mentioned in a problem description) but also verbs or adjectives e.t.c.
- A glossary is prerequisit for defining an ontology

Example

Part of a glossary for a library application
Book

- A book is a is a conceptual entity in a library. A book is defined by its title, the name of his authors, the publisher and the edition. A library can have several copies of the same book.

Copy

- A copy is a physical copy of a particular book. For example, the library has three copies of the book "Using UML" by Perdiate Stevens. . . .

. . .

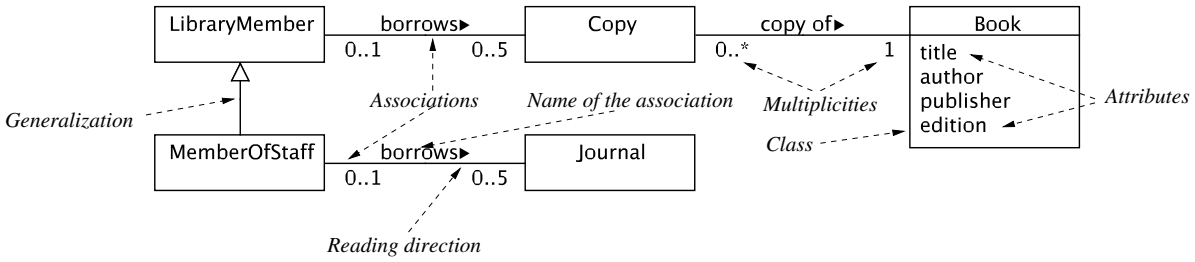
- *Warning*
  - Capture only knowledge relevant for the application
  - Don't try to capture all possible knowledge

Terms and their relations

- Class diagrams can be used for showing terms and their relations
  - The UML diagram type used most
  - Describes the type of objects in a system and their static relationships
- Usually
  - Associations
  - Classes (for nouns)
  - Generalizations
    - when terms are related by the "is-a" relationship
  - use of attributes depends on what one wants to show
  - commonly no operations, but can have operations if this contributes to the understanding of the domain (e.g. verbs → operations)
- *Warning*
  - The class diagram shows the customer knowledge and should not be biased by the implementation

Domain model (terms and their relations)

First class diagram concepts



3 Use Cases and Use Case Diagrams

Purpose of Use Cases

- Capture mainly functional requirements
- Use Cases for planning
  - Use Case Driven Design
  - Planning Game (from Extreme Programming)
- System Validation
  - Show that the sceanarios of the use cases can be realized by the system, e.g. by drawing sequence diagrams

  → Walking the use case
- Use Cases describe what is to be achieved and *not* how

Use Case

Introduced by Ivar Jacobson in the early 1990's

Use Case

- "A use case is a set of scenarios [that describe the interaction between an actor and the system] tied together by a common user goal" (modified from Martin Fowler, UML Destilled)

Use Case Example: Travel Agency use case *list available flights*

name: list available flights

description: the user checks for available flights

actor: user

main scenario:

1. The user provides information about the city to travel to and the arrival and departure dates
2. The system provides a list of available flights with prices and booking number

alternative scenario:

1a. The input data is not correct (see below)
   2. The system notifies the user of that fact and terminates and starts the use case from the beginning

2a. There are no flights matching the users data
   3. The use case starts from the beginning

note: The input data is correct, if the city exists (e.g. is correctly spelled), the arrival date and the departure date are both dates, the arrival date is before the departure date, arrival date is 2 days in the future, and the departure date is not more then one year in the future

Interactions

- Interactions between an actor and the system.
- An actor can be a user, but also another software system.
- The system itself can be the whole system, or subsystems, or even single classes
- Interactions
  - What the user does with the system: press a button, input some text, approach a barrier, ...
  - The reaction of the system, that is visible to the user
- Not part of interactions:
  - What the system internally does

Detailed use cases: Template

Template to be used for detailed use case descriptions

name: The name of the use case

description: A short description of the use case

actor: One or more actors who interact with the system

precondition: Possible assumptions on the system state to enable the use case

main scenario: A description of the main interaction between user and system

→ Note: should *only* explain what the system does from the *user's* perspective

alternative scnearios: Secondary scenarios; *fail* scenarios

postcondition: What has been achieved after the use case has been executed?

note: Used for everything that does not fit in the above categories

One can find many different types of templates in the literature. However, all have in common to state the main scenario and the alternative scnearios "A use case is a set of scenarios tied together by a common user goal" (From Martin Fowler, UML Destilled)

Precondition / Postcondition

- Precondition: A description of the state of the system, that is required before the interaction of the use case starts
  - E.g. the user is logged in for an add flight use case
- Postcondition: A description of the state of the system, after the scenarios have been performed
  - E.g. The system stores the trip under the name of the client for a "save trip" use case

Travel Agency: detailed use case *cancel trip*

name: cancel trip

description: cancels a trip that was booked

actor: user

precondition:

- the trip must have been booked
- the first date for a hotel or flight booking must be one day in the future

main scenario:

1. user selects trip for cancellation
2. the system shows how much it will cost to cancel the trip
3. selected trip will be cancelled after a confirmation

Travel Agency: detailed use case *plan trip*

name: plan trip

description: The user plans a trip consisting of hotels and flights

actor: user

main scenario:

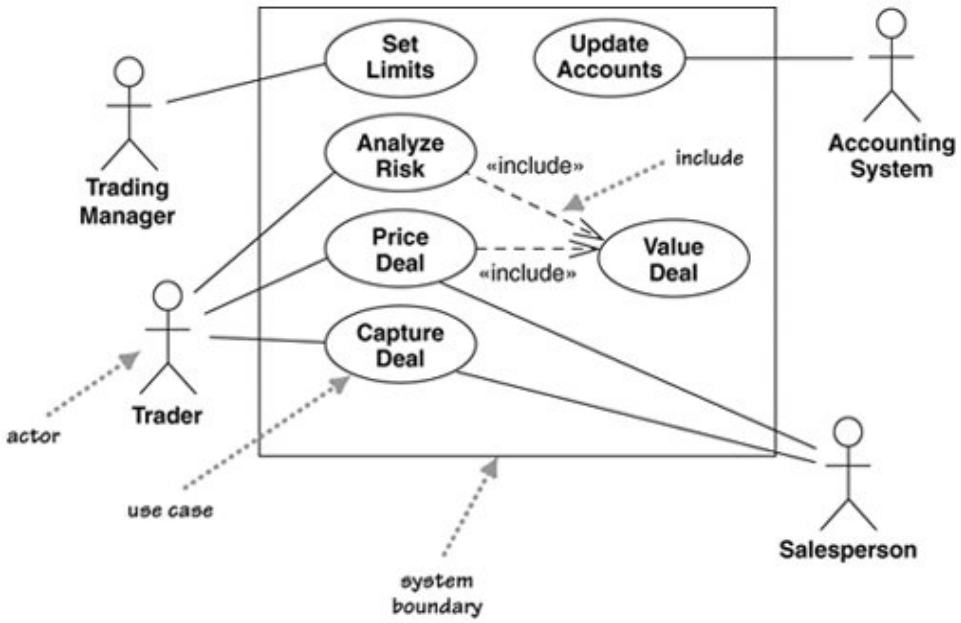
repeat any of the following operations in any oder until finished

1. list avaialbe flights (use case)
2. add flight to trip (use case)
3. list availabe hotels (use case)
4. add hotel to trip (use case)
5. list trip (use case)
6. delete hotel from trip (use case)
7. delete flight from tip (use case)

Note: the trip being planned is referred to as the current trip

This use cases uses other use cases to acomplish its goal. This corresponds to the ”includes” dependency in use case diagrams.

Use Case Diagrams Concepts



- Remarks
  - UML realizes Use Case diagrams as *class diagrams*:
    - Classes: actor and use case
    - Associations: Lines between actor and use case (*no arrow*)
    - Dependencies: Broken arrows between use cases (a broken line)
    - Inheritance: Lines with a closed arrow (example later)

Note: Actors Actors

- Who should be actor

  - Beneficionary of the use case
  - Participant in the use case
- What role does the actor play
  - not specific persons like action John Doe
  - ”A person wearing a particular hat”
- Actors can be
  - Human: e.g the user of the system
  - Non Human: an external system or device

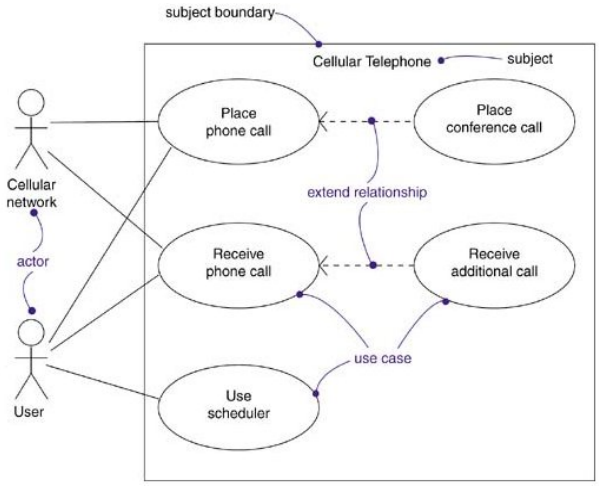
Subject of a use case

Subject of a use case

- The class described by a set of use cases
- Usually these are systems or subsystems

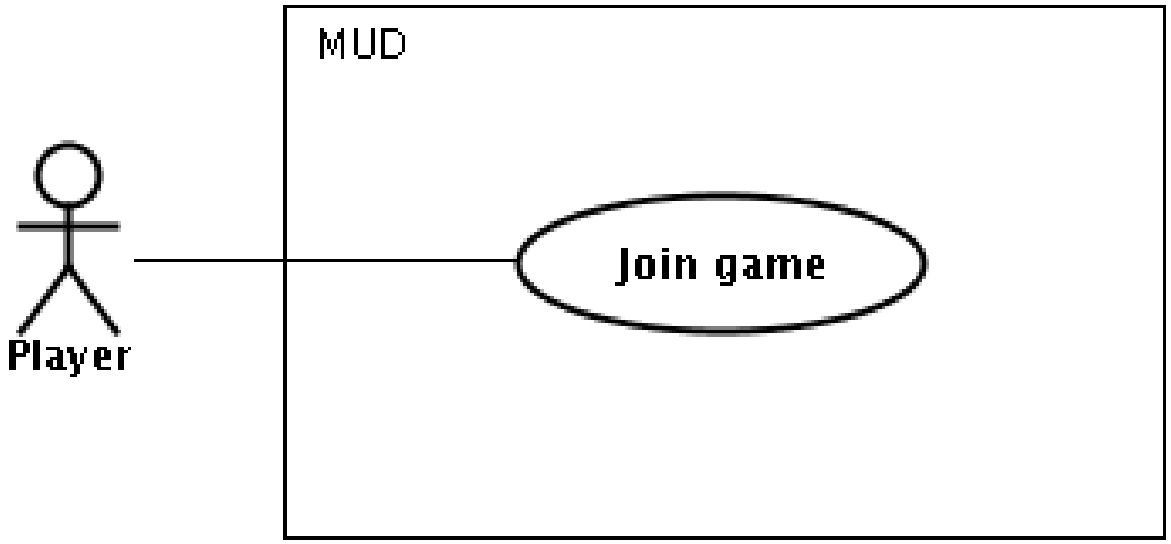
Subject of a use case / system boundary

- Shown as a subject/system boundary

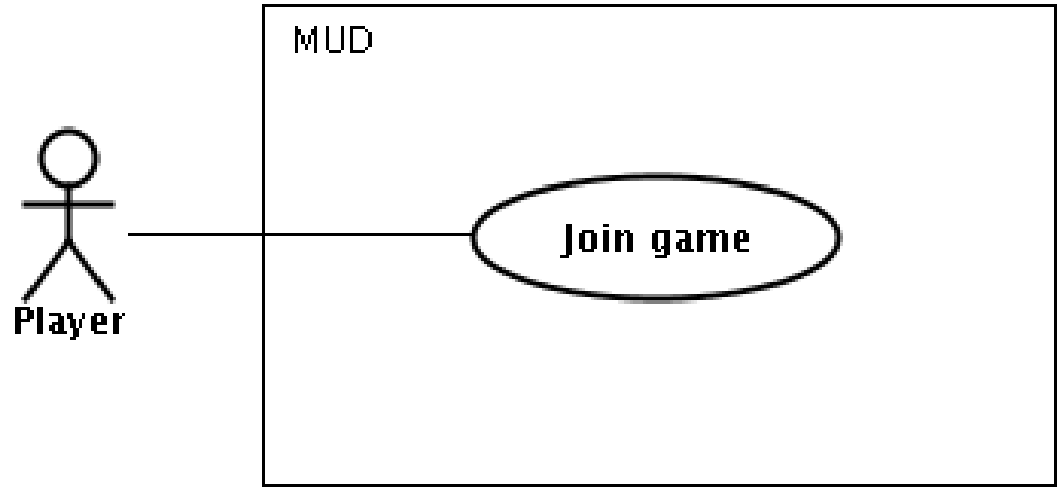


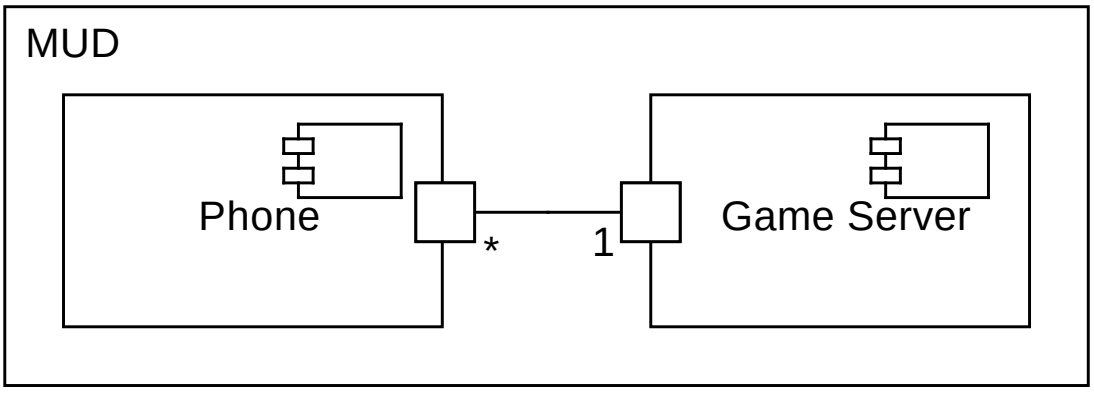
Use cases and System Boundaries

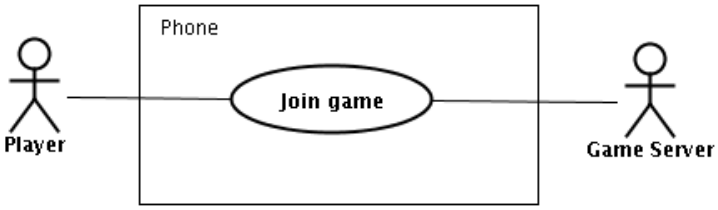
- Subsystems of a system don’t appear as actors



Use cases and subsystems





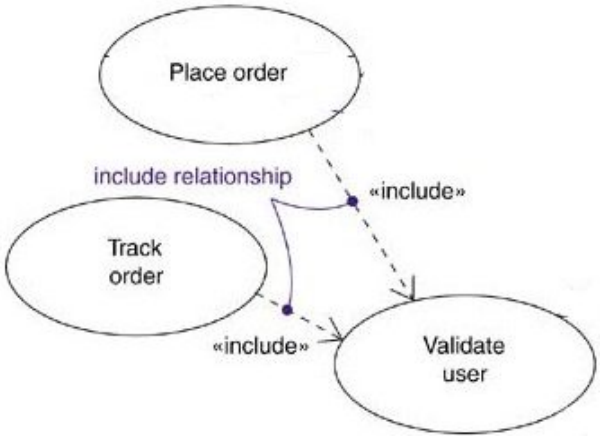


- Use cases can be used at different abstraction levels
  - High level business use cases
  - Low level system level use cases
  - → Low level system use cases can be used to specify the requirements for *subsystems*

Relationships among use cases and actors

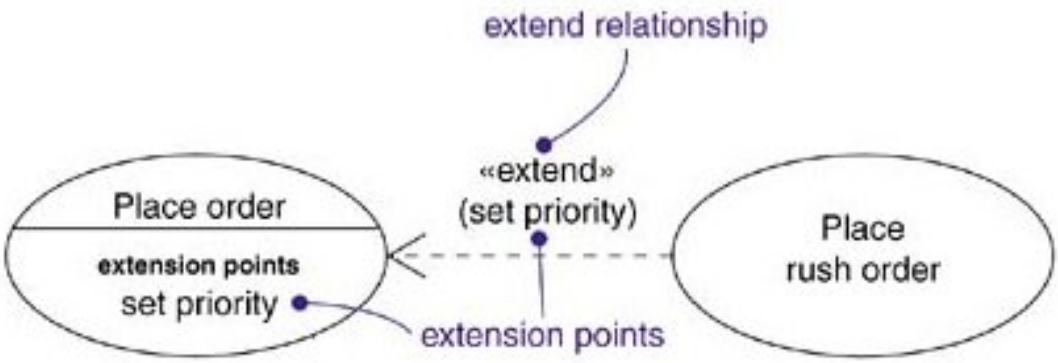
- Includes Relationship
  - One use case can include another use case
- Extends Relationship
  - One use case can extend another use case at some extension point
- Generalizations
  - Actors can inherit from each other
  - Use cases can inherit from each other
  - → Note: Almost all model elements in the UML can be used to inherit from

Includes Relationship



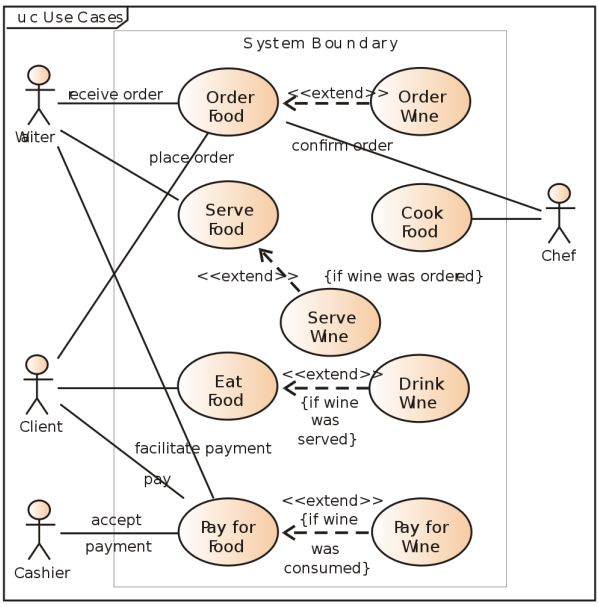
- One use case can include another one
- To execute the place order and track order use cases, both execute also the validate user use case
- One uses include to extract behaviour that is common to several use cases

Extends Relationship



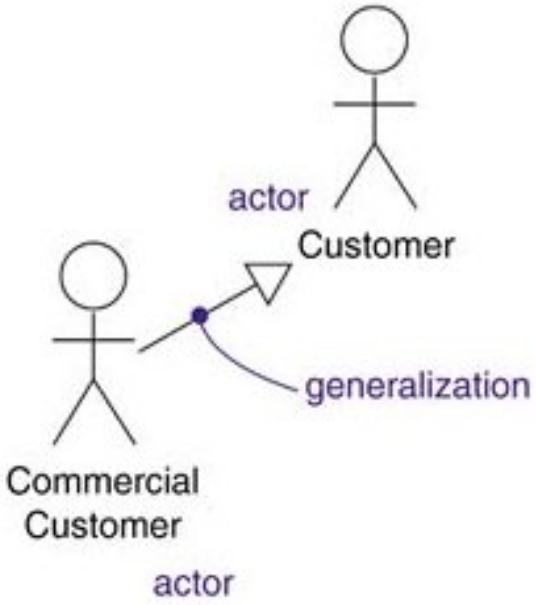
- One use case can extend another use case at a given extension point
- place rush order is executed when the execution of place order comes to the extension point set priority
- extends is used if one wants to indicate *variation* of the original use case
- extends denotes an *optional* path, in contrast to includes, which denotes a *mandatory* path. extends can also be used for conditional paths or for a choice of paths depending on the choice of the actor

Extends Relationshiop



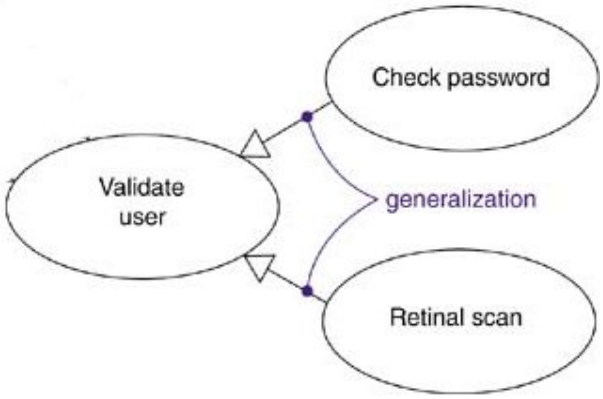
Here the ordering of the wine is optional and therefore extends is used and not includes.

Generalisation between actors



A commercial customer is a *special kind of* Customer

Generalisation between use cases

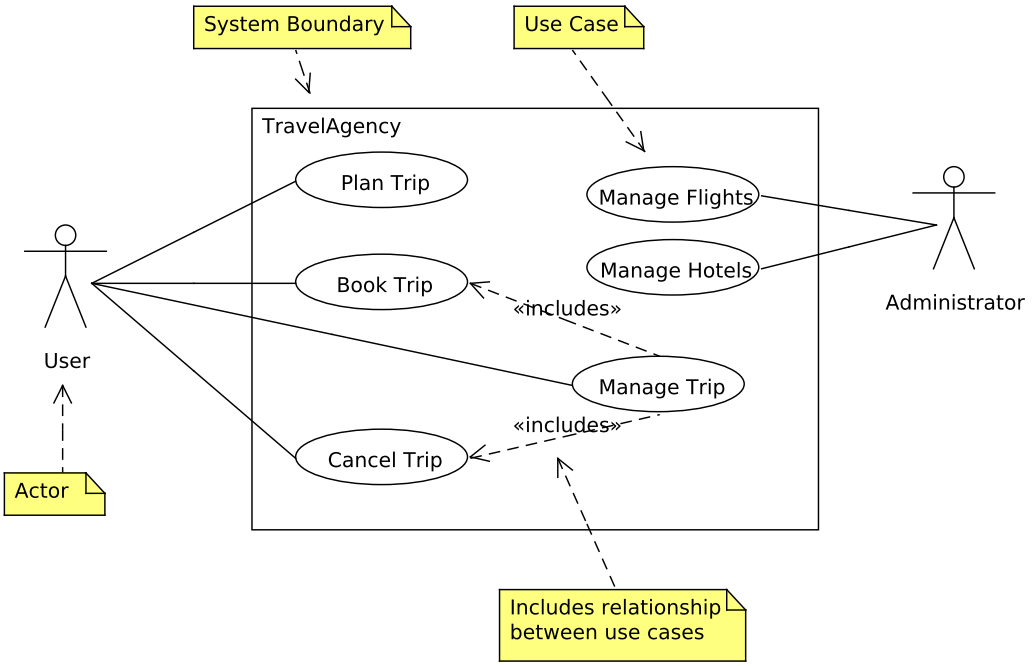


- The more special use case has the same *goal* (or a more specialised goal) than the more general use case.
- The check password and retinal scan use cases are *specializations* of the validate user use case.

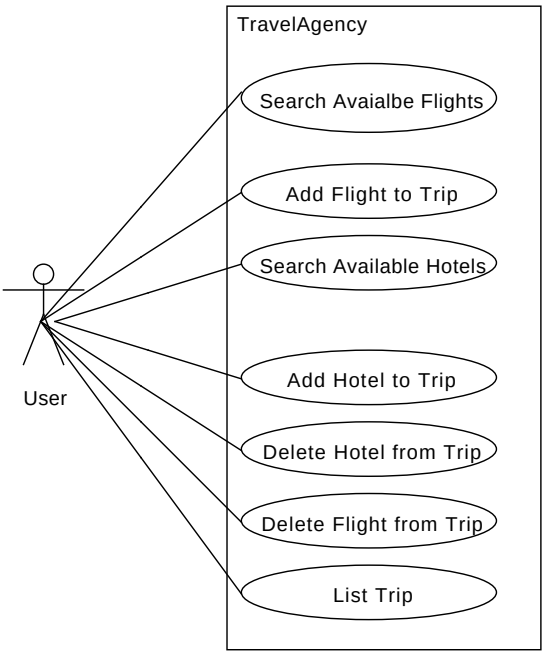
Abstraction levels of use cases

- Business use case or Kite level use case (Alistair Cockburn)
- System level use case or Sea level use case
  - More detailed: Fish level use case

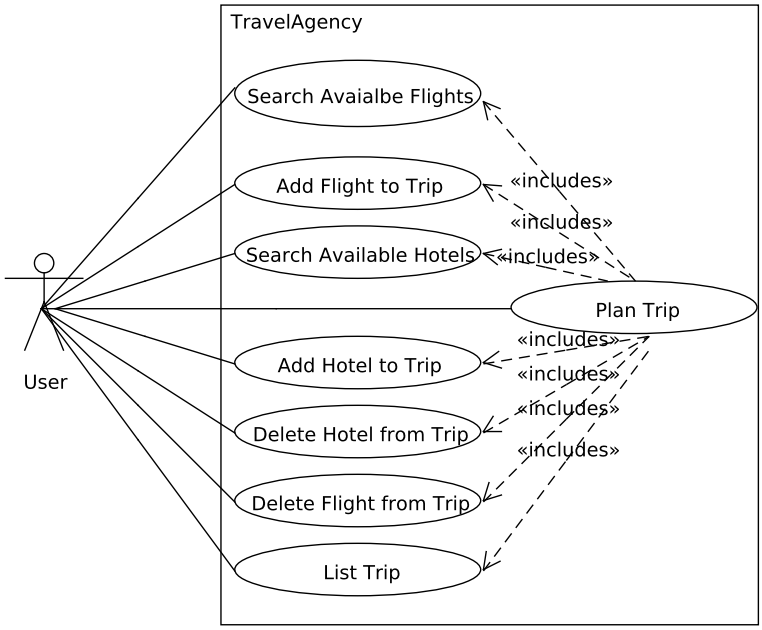
Travel Agency functional requirements: Business use cases



Travel Agency functional requirements: System level use cases (only part of the system0



Travel Agency functional requirements: System level use cases realting to busines use cases



Use Case Benefits

- Technique for capturing the functional requirements of a system
- Use cases are easily understandable by business users

  → Use cases allow to tell stories
- Use case alternative paths capture additional behaviour that can improve system robustness
- Use cases in planning
  - Basis for the estimating, scheduling, and validating effort
  - Use cases can be relatively easily added and removed from a software project as priorities change.
- Test Cases (System, User Acceptance and Functional) can be directly derived from the use cases

Use Case Limitations

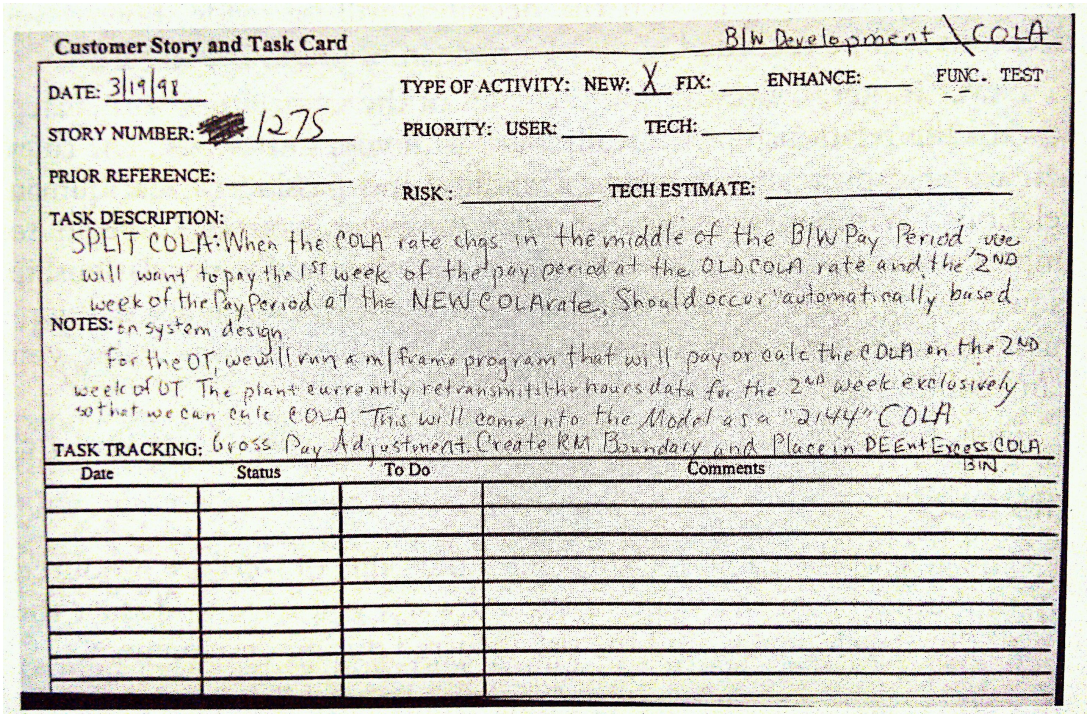
- Not good for capturing non-interaction based requirements e.g.
  - algorithm or mathematical requirements
  - non-functional requirements (such as platform, performance, timing, or safety-critical aspects)
- Abstracts away from the GUI
  - Use case theory suggests that UI not be reflected in use cases
  - but GUI mock ups (paper based, powerpoint based, etc.), prototypes may be more useful than abstract functionality

4 User Stories

User Stories

- Introduced with Extreme Programming
- Similar to Use Cases
- Focus on features
  - ”As a customer, I want to book and plan a single flight from Copenhagen to Paris”.
  - Recommended, but not exclusive: ”As a <role>, I want <goal/desire> so that <benefit>”
- Difference to Use Cases: User stories can be defined for non-functional requirements
  - ”The search for a flight from Copenhagen to Paris shall take less than 5 seconds”
- User stories have been introduced with Extreme Programming. They are very close to the concept of use cases, but also are different. User stories fullfill the same purpose as use cases, i.e. keeping track of the requirements of the system.
- However, user stories differ from use cases, as they focus on one *feature* of the software. A feature can be a functional requirement of the system, but also a non-functional requirement. Note that with use cases, the focus is on the functionality of the system and not on the non-functional aspects. Thus a use case mainly represents functional requirements, while a user story can represent both, a functional and a non-functional requirement.
- How the user story works, is providing a *story* of how a user uses the system. E.g. ”As a customer, I would like to book and plan a single flight from Copenhagen to Paris”.
- One can also incorporate non-functional requirements in a user story; e.g. ”As a customer, wihtin five seconds I would like have a list of all flights from Copenhagen to Paris that start on a given date.”
- Another example for a user story for a non-functional requirement: ”The communication of the travel agency with the bank shall be encrypted”
- A user story describes a scenario of interaction with the system relevant for the user of the system
- Can be, e.g., a main scenario of a use case; but also one of the alternative or exceptional scenarios
  - e.g. borrow book scenario
  - focus on: books (not general media), number of books borrowed (no overdue books), e.t.c.
  - On contrast: a use case for borrow books need to describe all these aspects
- Can define also non-functional requirements
- Are documented informally as *index cards* and formally using *acceptance tests*

Example of a User story index card



Customer Story and Task Card

B/W Development \ COLA

DATE: 3/19/98

TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ FUNC. TEST ___

STORY NUMBER: 1275

PRIORITY: USER: ___ TECH: ___

PRIOR REFERENCE: ___

RISK: ___ TECH ESTIMATE: ___

TASK DESCRIPTION:

SPLIT COLA: When the COLA rate chgs in the middle of the B/W Pay Period we will want to pay the 1st week of the pay period at the OLD COLA rate and the 2ND week of the Pay Period at the NEW COLA rate. Should occur automatically based on system design.

NOTES: For the OT, we will run a m/frame program that will pay or calc the COLA on the 2ND week of OT. The plant currently retransmits the hours data for the 2ND week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA

TASK TRACKING: Gross Pay Adjustment. Create RM Boundary and Place in DEEntExcess COLA BIN.

Date	Status	To Do	Comments

Kent Beck, Extreme Programming, 1st ed.

This is one of the orignal user story cards used by Kent Beck in the project, where the Extreme Programming methodology has been defined. More recently, a more stylistic form of user stories have evolved, i.e. the form is "As a ..., I would like to ...". Note also, that the detailed scenario of the interaction is usually not part of the description of a user story (as found on an index card). Instead, the precise way of how the interaction happens is to be discussed with the customer while the user story is estimated and later implemented. The basic idea here is, that with Extreme Programming a representative of the customer is part of the developer team, and that he can be asked about the exact scenario behind a user story. This makes the process more flexible and helps to reduce the overhead of completely fixing the scenarios for each user story. The cost for this is, that a representative of the customer (i.e. the owner of the requirements) has to be present in the development of the system. Alternatively, the development team can itself provide a customer substitute, e.g. someone that has control over the requiments and decides on how the system should look like. This is, e.g. useful, in mass product development.

- Important: Requirements engineering is done *in parallel* with the development of the system
  - User story index cards are created by the customer and discussed with the developer
  - User story index cards are assigned to iterations based on *importance* to the customer
  - Only within each iteration the user stories are refined and tests are implemented
- Two level approach
  1) Make *coarse* user stories for planning
  2) *Detail* user stories when they are about to be implemented

  → Compare with waterfall: Already in the requirements phase make all the requirements as precise and detailed as possible

Difference User Story and Use Case

User Story

- One *concrete* scenario
- functional *and* non-functional requirements

- *implicit* scenarios, that means that the detailed description of the scenario is not put on the card, but is discussed with the customer on estamating the scnerio and on implementing the scenario. This has the benefit of not putting too much work in detailed description of scenarios before the user story is being implemented. When implementing a user story, the implicit scenario is made explicit by writing a test for that scenario
- several user stories for main an alternative scenarios. This allows one to schedule more important scenarios of different use cases first, before implementing less important alternative scenarios of the same use case.

Use case

- Several "abstract" scenarios having the same goal (main and alternative scenarios)
- Only functional requirements
- Explicit scenarios
- several scenarios tied together by one goal. This means one has less use cases, so it is easier to get an overview over the system and check for completeness of the requirements

Combining use cases and user stories

- Use cases are good to give an overview of the functionality of the system

  $\rightarrow$ in particular use case diagrams
- Use cases group scenarios with a similar goal

  $\rightarrow$ makes it easier to check scenarios for completeness
- Transform the use case scenarios to user story and use them for the software development

5 Summary

Summary

- Requirements Engineering
  - What the customer expects from the system
- Requirements Process: Elicitation, Documentation, Validation
- Requirements Elicitation: Interviews, Scenarios, *Use Cases*
- Requirements Documentation
  - Domain Model: Class diagram + possibilty activity diagram for business processes
  - Use Cases:
    - 1) Use Case Diagram $\rightarrow$ provides an overview over the functionality of the system
    - 2) Detailed use cases $\rightarrow$ provide the details of the
  - User Stories
- Use case driven developement