Software Engineering I (02161)

Week 2

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2016

Contents

Basic Activities in Software Development

- Understand and document what kind of the software the customer wants
  - → Requirements Analysis
  - → Requirements Engineering
- Determine how the software is to be built
  - → Design
- Build the software
  - → Implementation
- Validate that the software solves the customers problem
  - → Testing

Requirements Analysis

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

- Describe mainly the *external behaviour* of the system and *not* how it is realised
  - *what* not *how*
- Techniques for discovering, understanding, and documentation
  - Glossary: Understand the problem domain
  - Use Cases: Understand the functionality of the system
  - User Stories: Understand the functionality of the system

Types of Requirements

- Functional Requirements
  - E.g. the user should be able to plan and book a trip
- Non-functional Requirements
  - All requirements that are not functional
  - E.g.
    - Where should the software run
    - What kind of UI the user prefers

Travel Agency Example: User Requirements

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

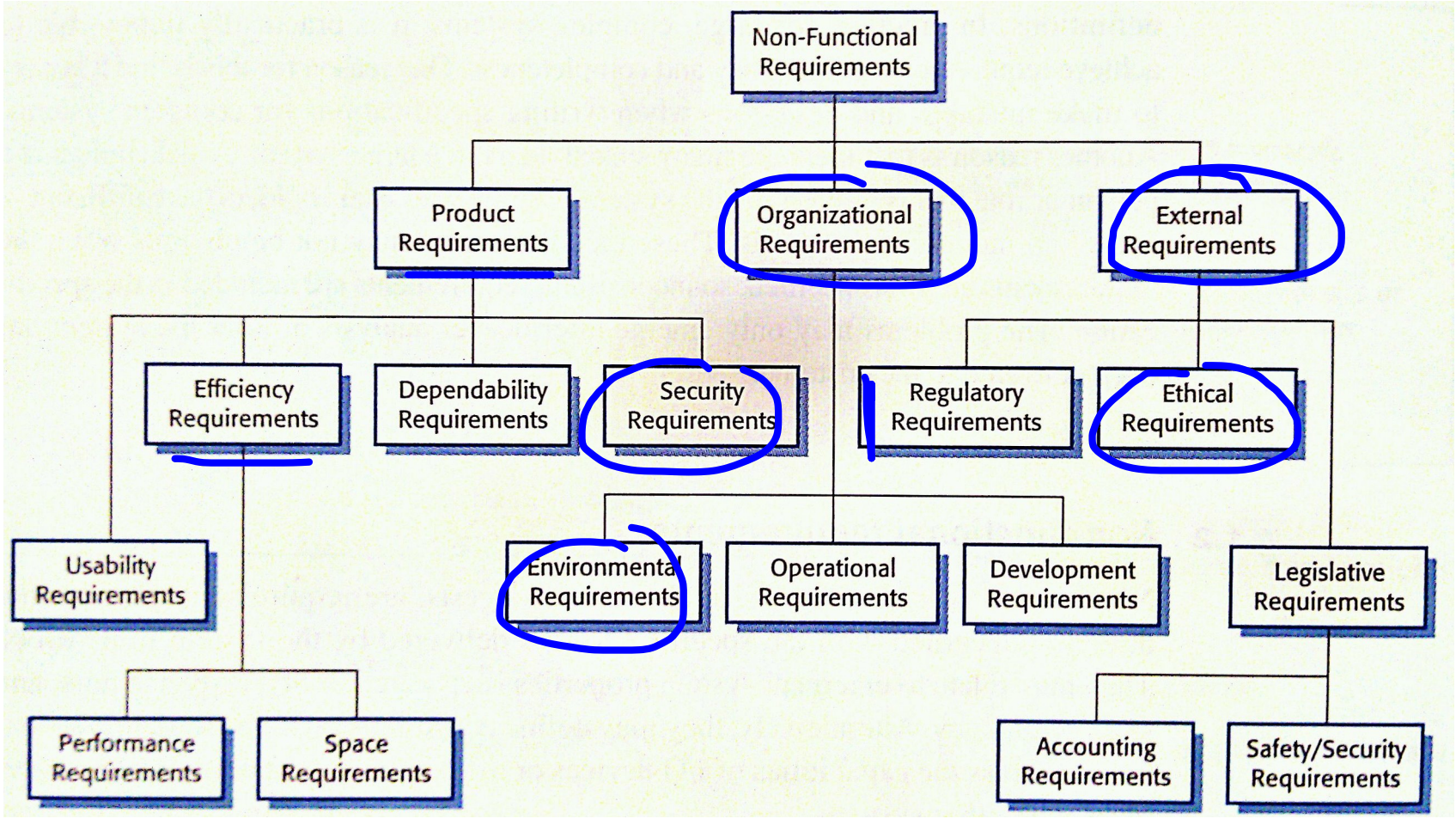
- Problem description / user requirements
  - TravelGood wants to offer a trip-planning and booking application to its customers. The application should allow the customer to plan trips consisting of flights and hotels. First the customer should be able to assemble the trip, before he then books all 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.

$\rightarrow$ Not enough: Text needs to be analysed and system requirements extracted

Travel Agency

- Functional Requirements
  - ”plan a trip, book a trip, save a planned trip for later booking, . . . ”
- 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 (it has to a pass a usability test)”

Non exclusive checklist of non-functional requirements



Characteristics of good requirements

- Testability
  - $\rightarrow$ manual/automatic acceptance tests
- Measurable
  - Not measurable: *The system should be easy to use by medical staff and should be organised in such a way that user errors are minimised*

Characteristics of good requirements

- Testability
  - → manual/automatic acceptance tests
- Measurable
  - Not measurable: *The system should be easy to use by medical staff and should be organised in such a way that user errors are minimised*
  - Measurable: *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.*

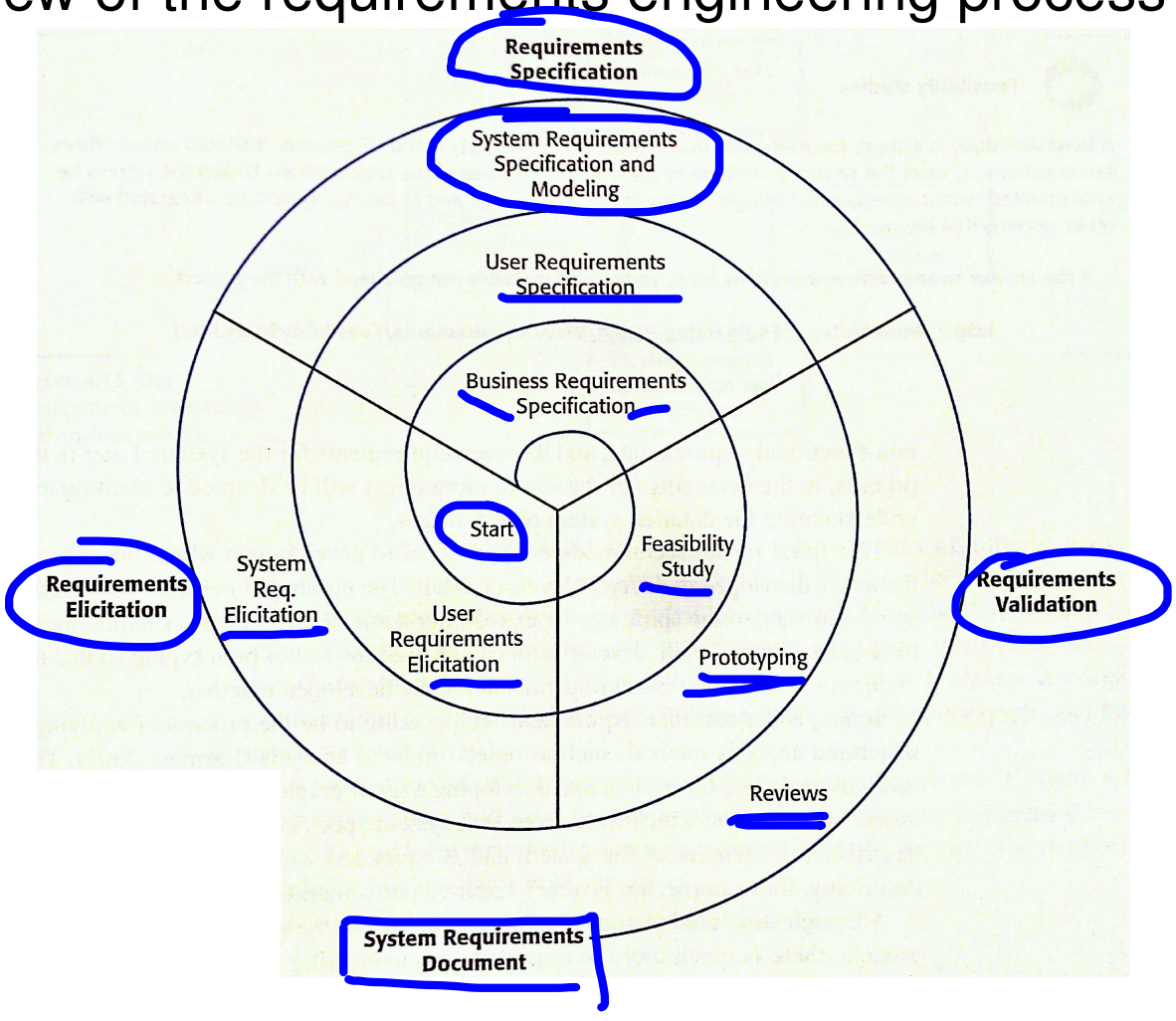
Possible measures

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

Contents

Requirements engineering process

A spiral view of the requirements engineering process



Requirements Engineering Process: Techniques

- Elicitation
  - Problem description
  - Interviews
  - Glossary
  - Use Cases / User Stories
- Specification
  - Glossary
  - Use Cases / User Stories
- Validation
  - Inspection
    - Validity, Consistent, Complete, Realistic, ...
  - Creation of tests

Contents

Glossary

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

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.

Example

Part of a glossary for the travel agency

User: The person who is using the travel agency

Trip: A trip is a collection of hotel and flight informations. A trip can be booked and, if booked, cancelled.

Booking a trip: A trip is booked by making a hotel reservation for the hotels on the trip and a flight booking for the flights of the trip

Flight booking: The flight reservation is booked with the flight agency and is payed.

Reserving a hotel: A hotel is reserved if the hotel informed that a guest will be arriving for a certain amount of time. It is possible that the hotel reservation requires a credit card guarantee.

...

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

Contents

Use Case

Use cases capture functional requirements

→ Naming convention: ”Do something” (= functionality): ”verb + noun”

Use Case

A Use Case is a set of interaction scenarios of one or several actors with the system serving a common goal.

Use Case Diagram

A use case diagram provides and overview over the use cases of a *system* and *who* is using the functionality.

Detailed Use Case description

A detailed use case description describes the interaction between the user and the system as a set of *scenarios*

(Detailed) Use Case Example: *search available flights*

name: search 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)

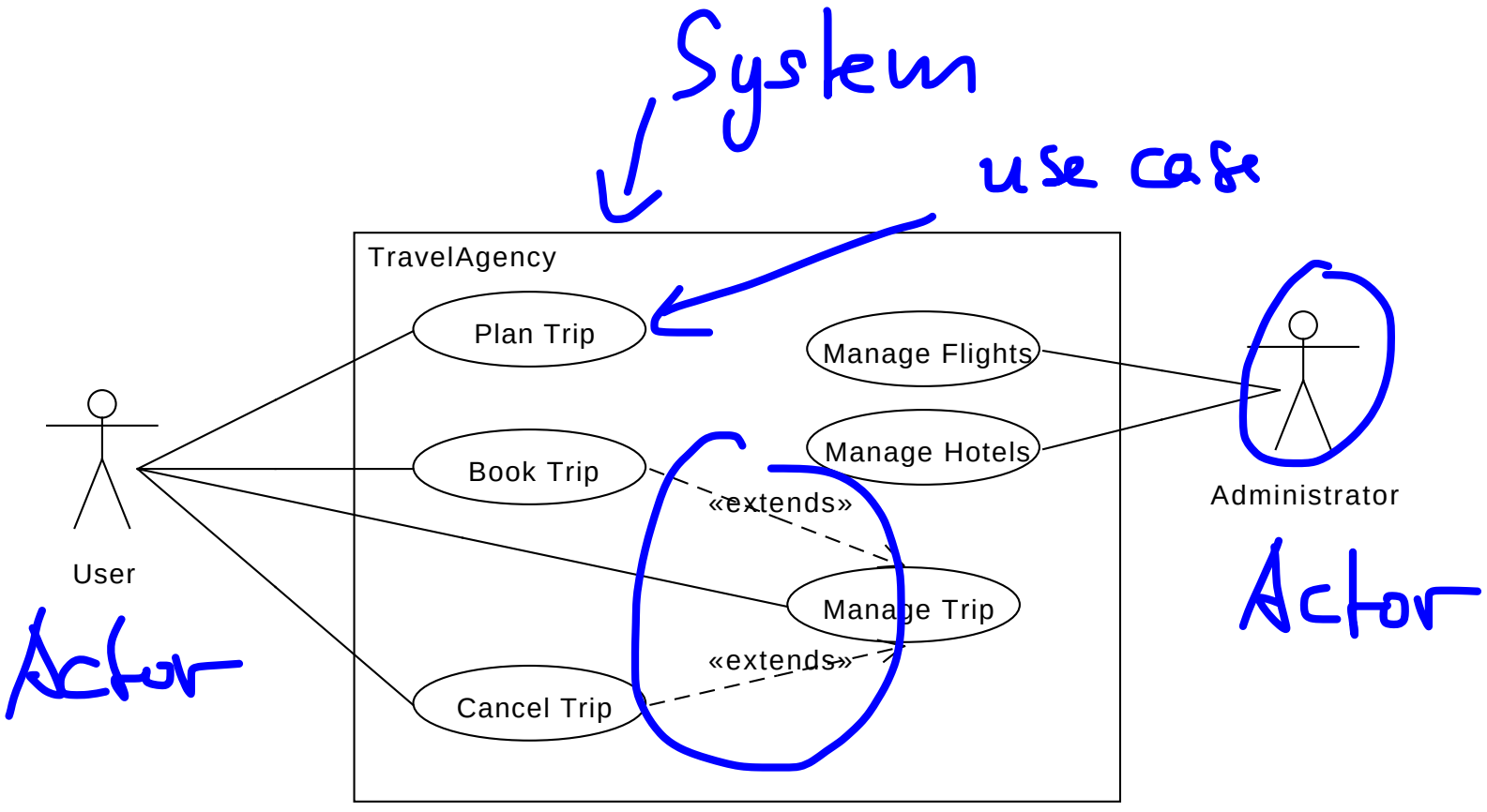
2. The system notifies the user of that fact and terminates and starts the use case from the beginning

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

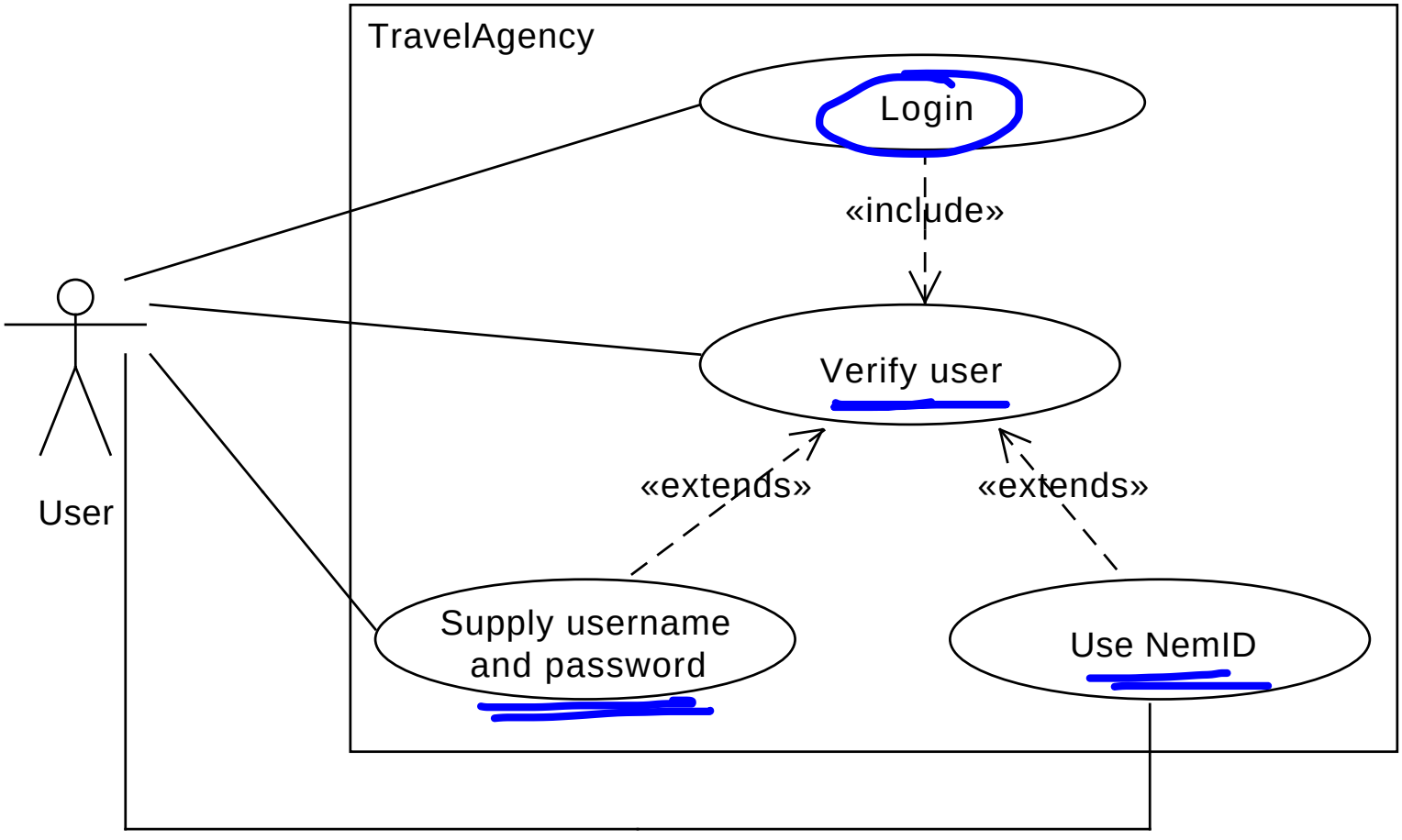
Use Case Diagram



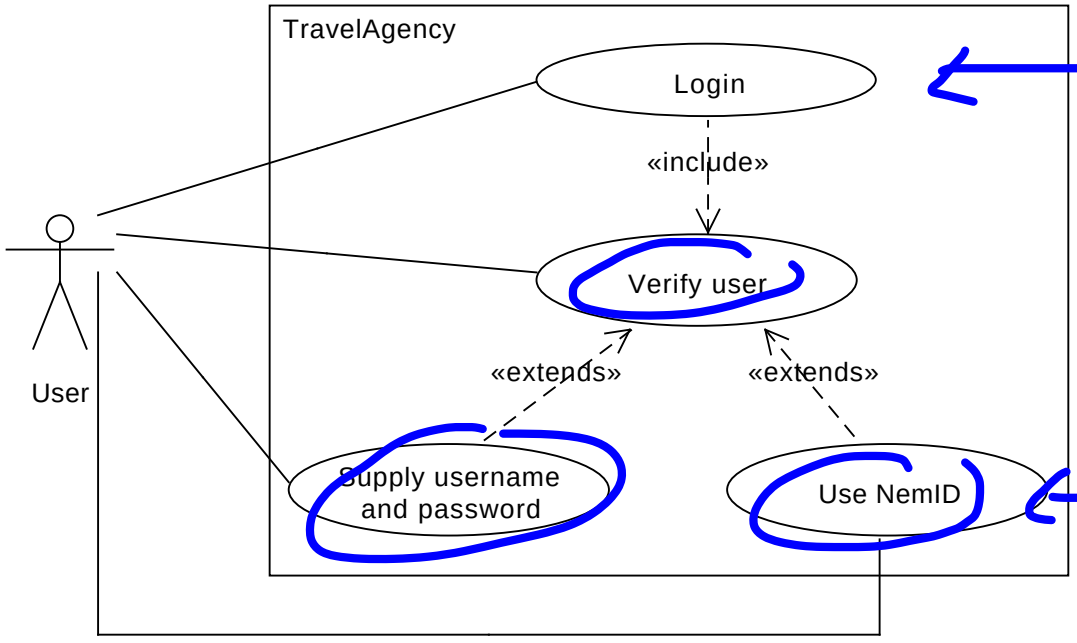
Relations between use cases

extends: optional part as a use case

includes: mandatory part of a use case



Use extends/include sparingly



- Use extends/include only when:
  - Interactions are reused by other use cases, e.g. login?
  - Relationship between abstract and concrete (cf. next slide)
  - A use case contains optional interactions and it makes sense to describe these as a use case themselves
- Extends/include don't show the order of interactions in a use case

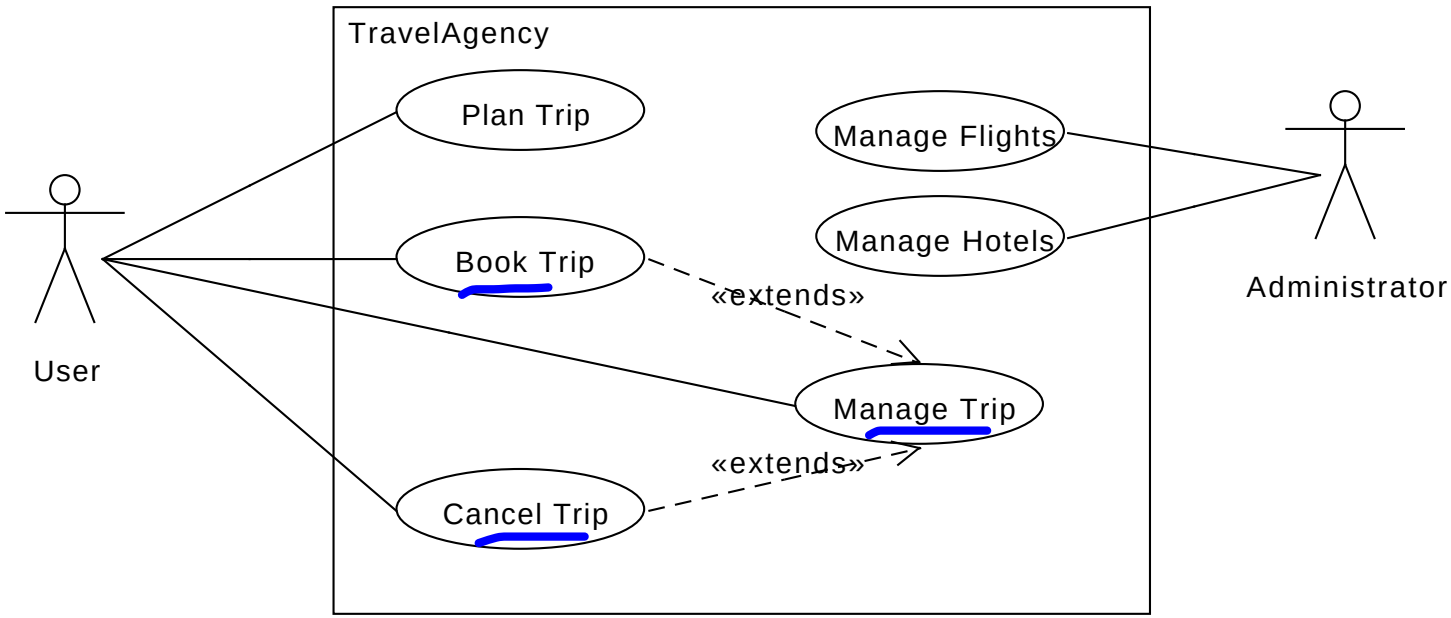
→ When in doubt, don't use extends/include

Types of use case diagrams

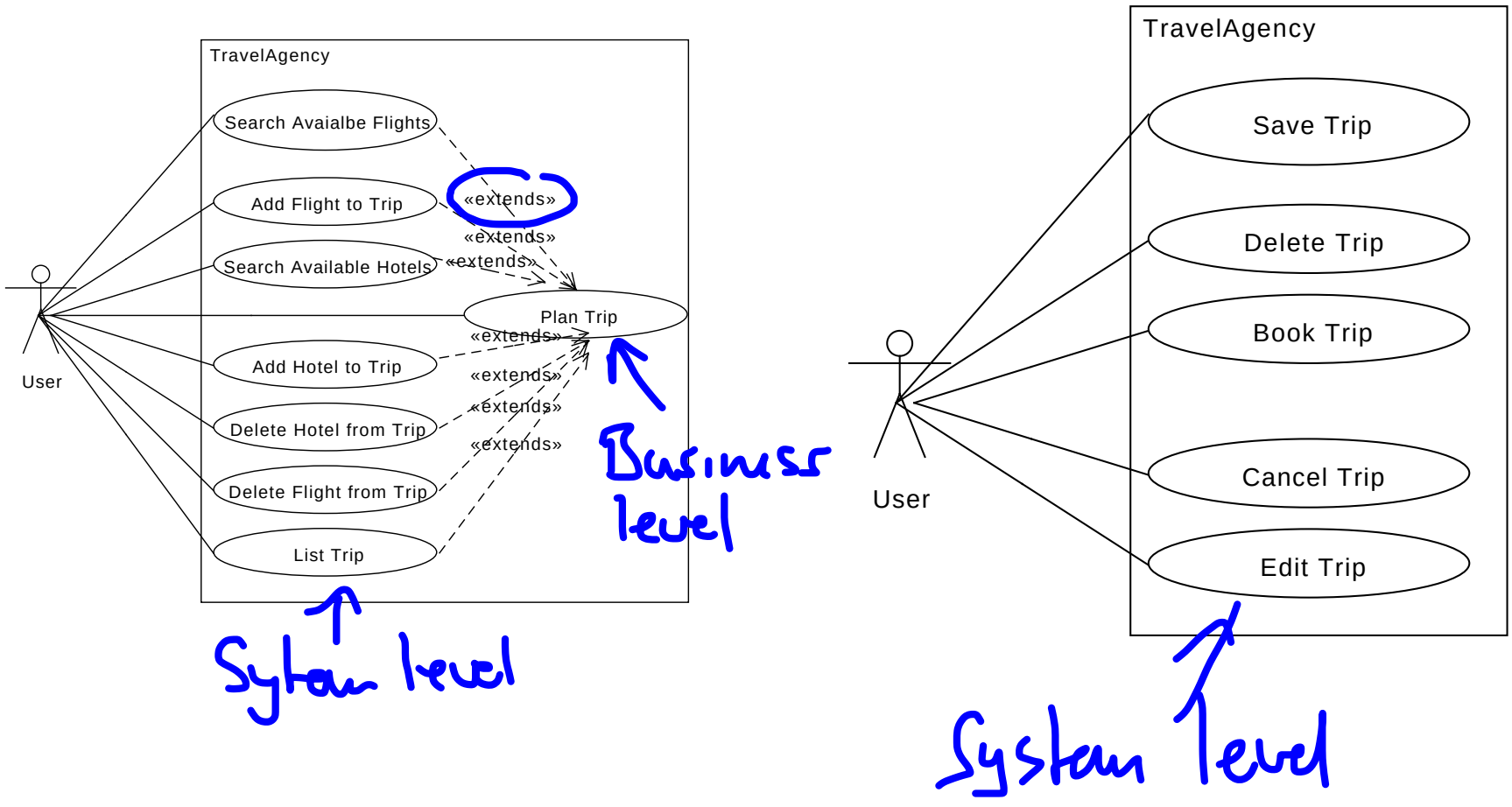
a) *Business use cases* (*kite level* use case (from Alistair Cockburn))
b) *System use cases* / *sea level* use case
c) Use cases included in *sea level* use cases are called *fish* level use cases by Alistair Cockburn

UML Destilled, Martin Fowler

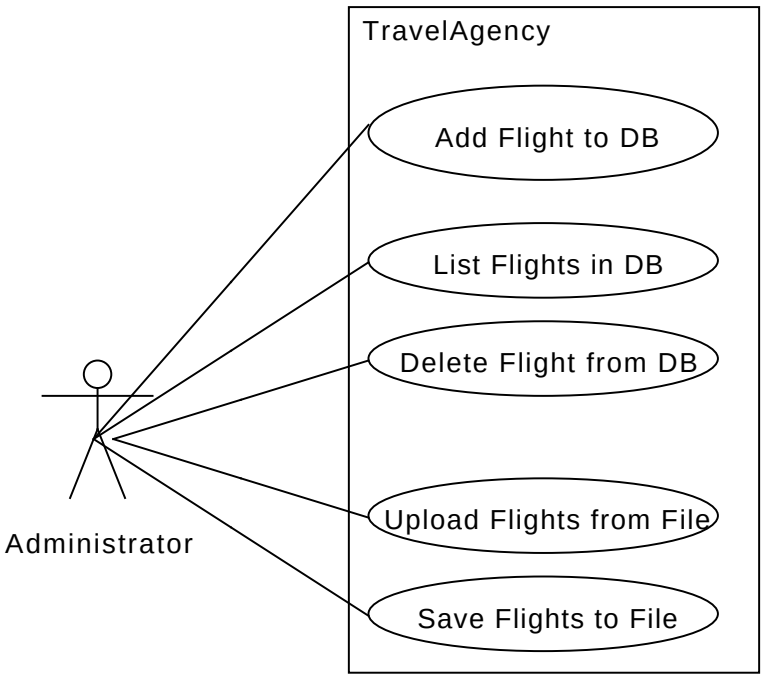
Business Use Cases

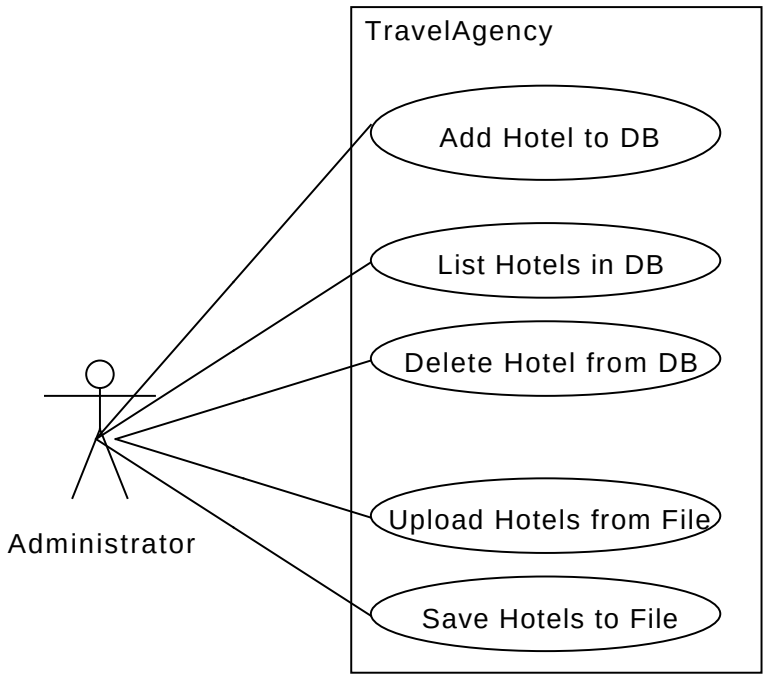


System Use Cases Part I



System Use Cases Part II





Detailed use cases: Template

Template *to be used in this course* 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 (optional)

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 scenarios:

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

(postcondition)

→ To be used in the examination report

Detailed use case *search available flights*

name: search 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)

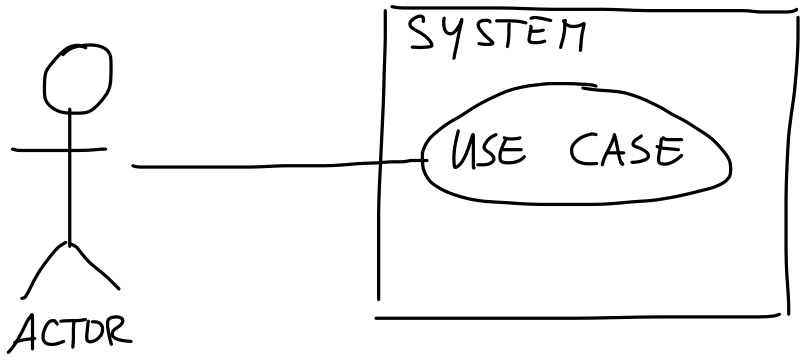
   2. The system notifies the user of that fact and terminates and starts the use case from the beginning

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

Use case scenarios



- Use case scenarios = interaction between an actor and the system
  - Anything the user does with the system
  - System responses
  - Effects visible/important to the customer
- Not part of the interaction: What the system internally does

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

interaction of actor with the system

Detailed use case *plan trip*

This use case *extends* other use cases

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 order until finished

1. search available flights (use case)
2. add flight to trip (use case)
3. search available 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

Detailed use case *save trip*

name: save trip

description: provides the current trip with a name and saves it for later retrieval

actor: user

precondition: the current trip is not empty

main scenario:

1. user provides a name for the trip
2. System confirms trip saved

alternative scenarios:

1: the name is not valid

2: notify the user of the fact and end the use case

1: a trip with the name already exists

2: ask the user if the trip should overwrite the stored trip

3a: If yes, overwrite the stored trip

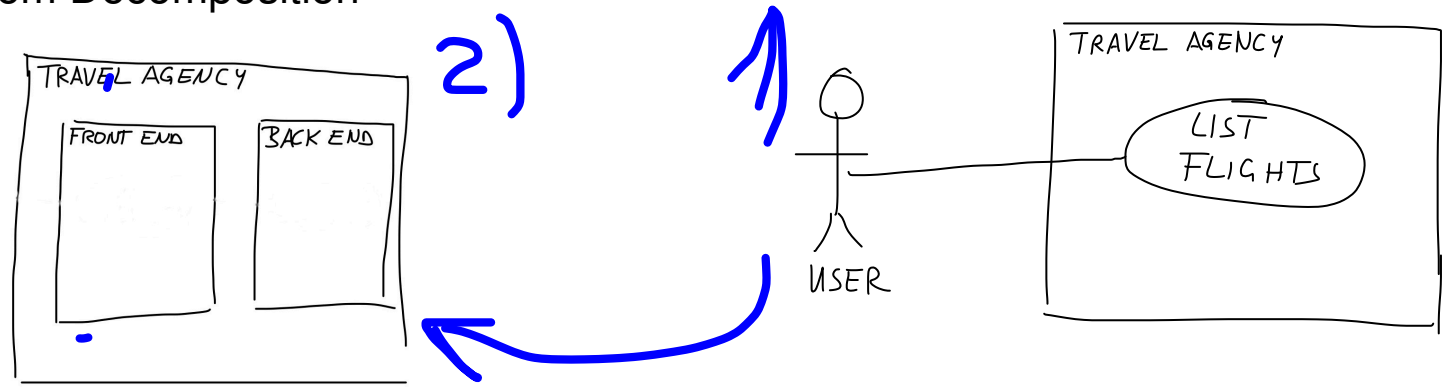
3b: If no, end the use case

Use cases and system boundary

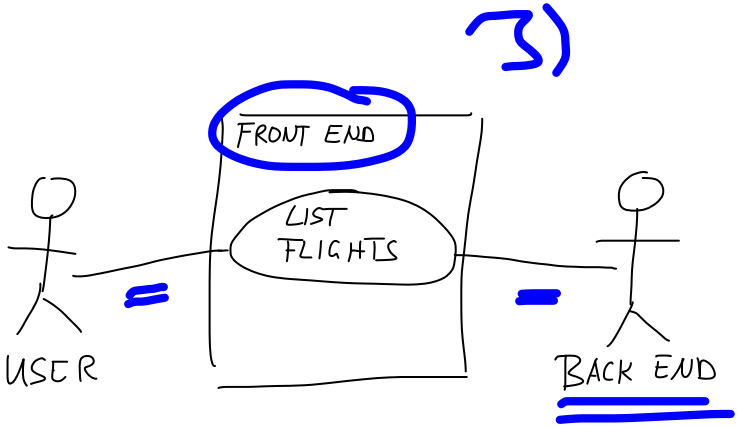
Actors and use cases depend on the system boundary:

Design by decomposition

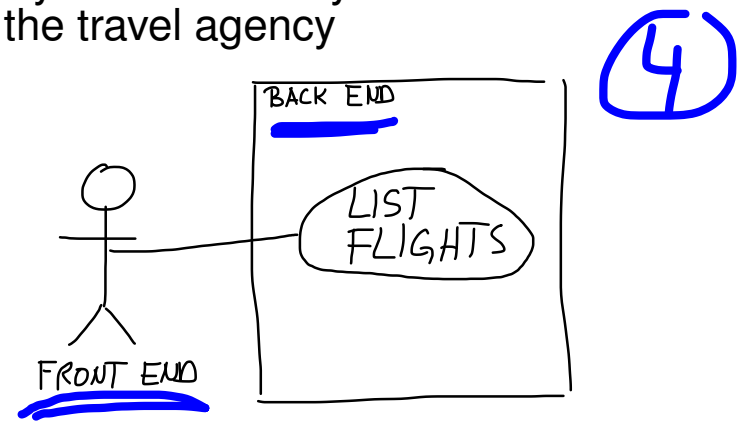
- System Boundary: Travel Agency
- System Decomposition



- System Boundary: Front end of the travel agency



- System Boundary: Back end end of the travel agency



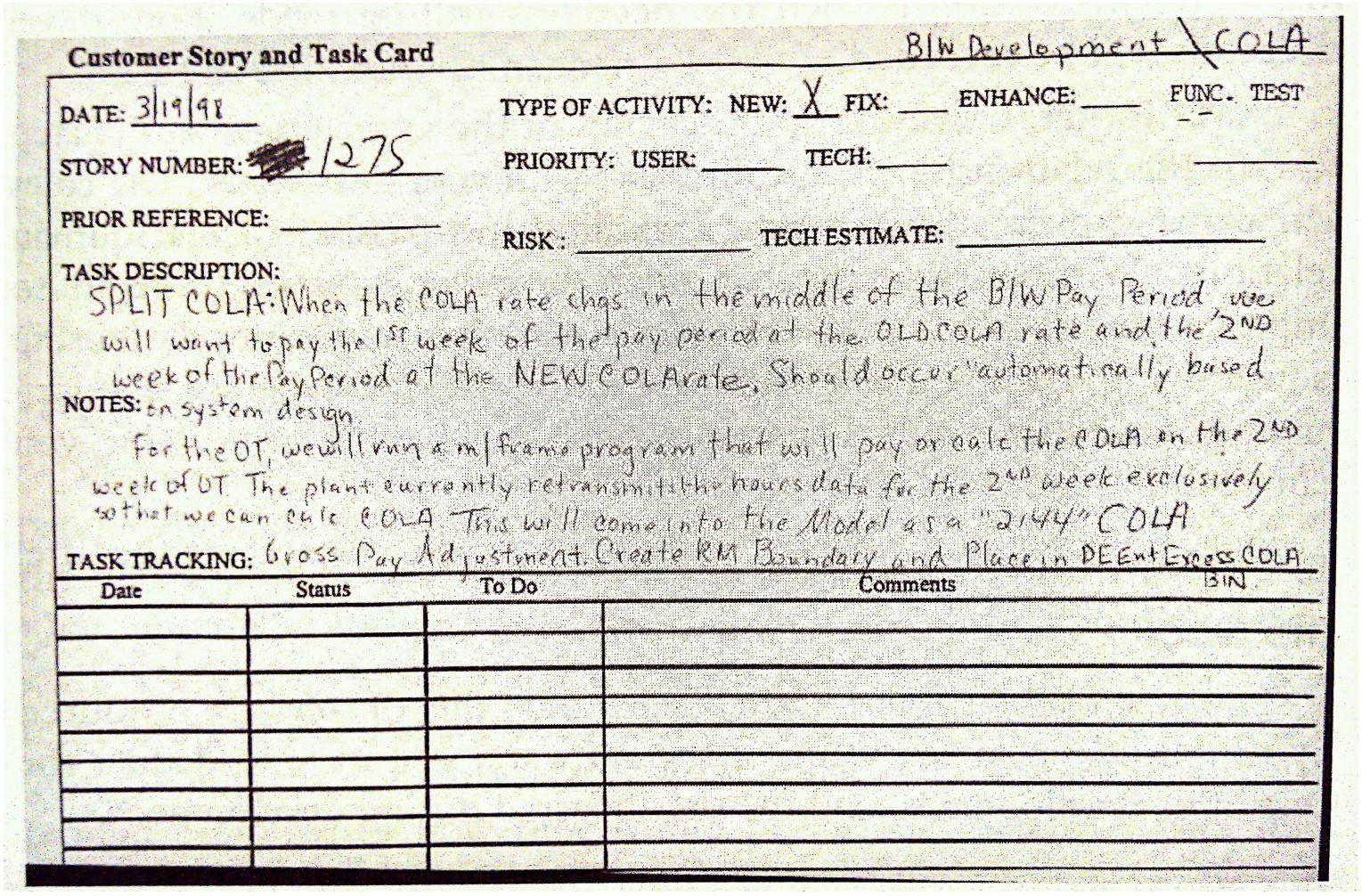
Contents

User stories

- Introduced with Extreme Programming
- 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:
  - Contain one main scenario
  - Are concrete (i.e. use concrete data)
  - User stories can be defined for non-functional requirements

    ”The search for a flight from Copenhagen to Paris shall take less than 5 seconds”
- Documented by user story cards, i.e. index cards

Example of a User story 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 DEEntExpress COLA BIN.

Date	Status	To Do	Comments

Kent Beck, Extreme Programming, 1st ed.

- User story card: A contract between the customer and the devloper to talk about the user story

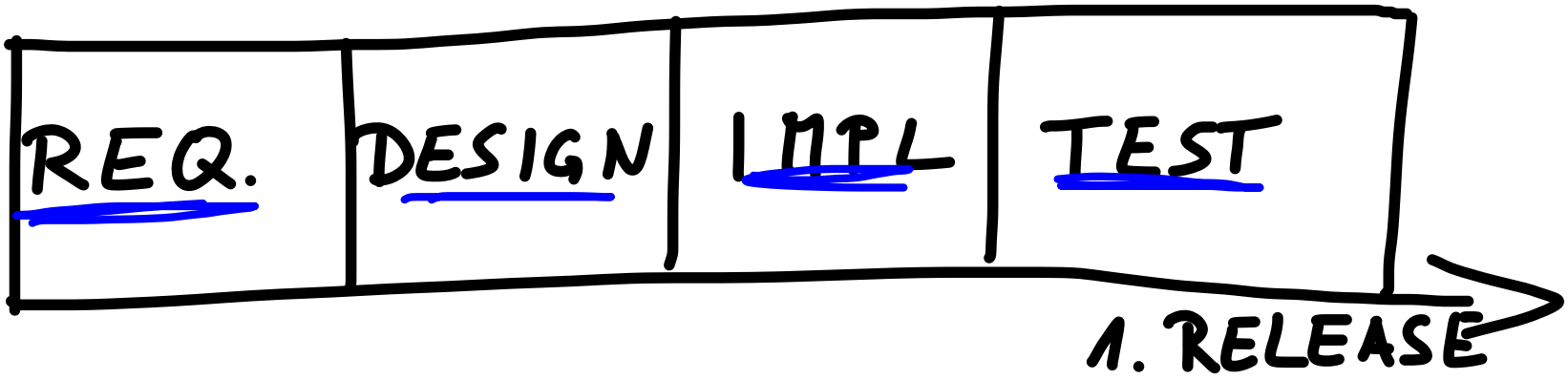
User stories and requirements engineering

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

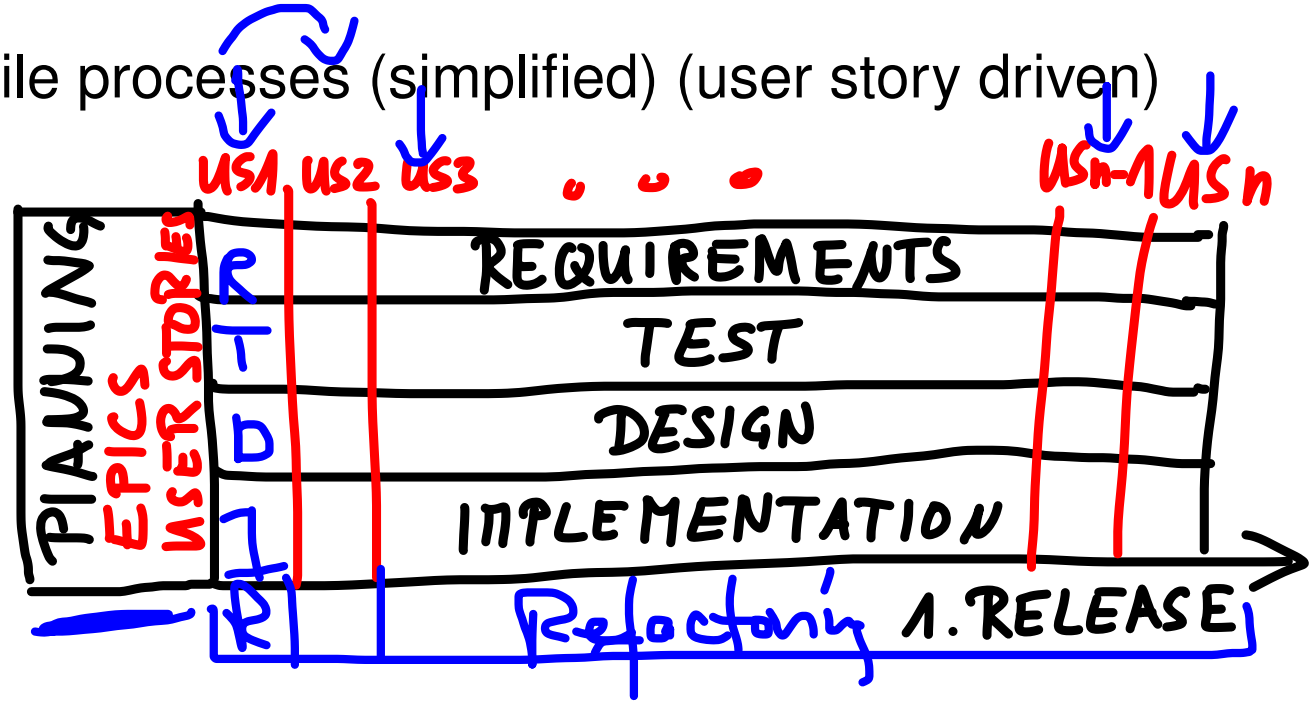
→ Compare with waterfall: Already in the requirements phase make all the requirements as precise and detailed as possible

Software Development processes

- Traditional (waterfall process)



- Agile processes (simplified) (user story driven)



Comparision: User Stories / Use Cases

Use Story

- one concrete scenario/feature
- functional + non-functional requirements

- several *abstract* scenarios with one goal
- only functional requirements

Combining Use Cases and User Stories

1. Use cases:
   - Gives an overview over the possible interactions

   → use case diagram
2. Derive user stories from use case scenarios (i.e. main- and alternative)
3. Implement the system driven by user stories
   - Note that different scenarios in use cases may have different priorities

     → Not necessary to implement all scenarios of a use case immediately

Contents

Summary

- Requirements analysis is about finding out *what* the software should be able to do, not *how*
- Types: *functional* and *non-functional* requirements
- Qualities: *testable* and *measurable*
- Process: *Discover* (Elicitation), *Document* (Specification), *Validate* (Validation)
- Glossary: Defines a *common language* between customer and software developer
- Use cases
  - Used for both *finding* and *documenting* the requirements
  - What are the *functions* the user can perform with the software?
- User stories
  - Focus on user relevant scenarios
  - Can be used for functional and non-functional requirements
  - Can be derived from use case scenarios

Exercises

- For this week
  - `http://www2.imm.dtu.dk/courses/02161/2016/slides/exercise02.pdf`
- Still ongoing: programming exercises
  - `http://www2.imm.dtu.dk/courses/02161/2016/index2.html`

Contents

Booleans

```
if (string.equals("adminadmin")) {
   adminLoggedIn = true;
} else {
   adminLoggedIn = false;
}
```

true

Booleans

```
if (string.equals("adminadmin")) {
   adminLoggedIn = true;
} else {
   adminLoggedIn = false;
}
```

Don’t use conditionals to set a boolean variable

- Better

```
adminLoggedIn = string.equals("adminadmin");
```

Booleans

```
if (!adminLoggedIn == false) {
   throw new OperationNotAllowedException();
} else {
   if ( adminLoggedIn == true ) books.add(book);
}
```

Booleans

```
if ( adminLoggedIn == false) {
   throw new OperationNotAllowedException();
} else {
   if ( adminLoggedIn == true ) books.add(book);
}
```

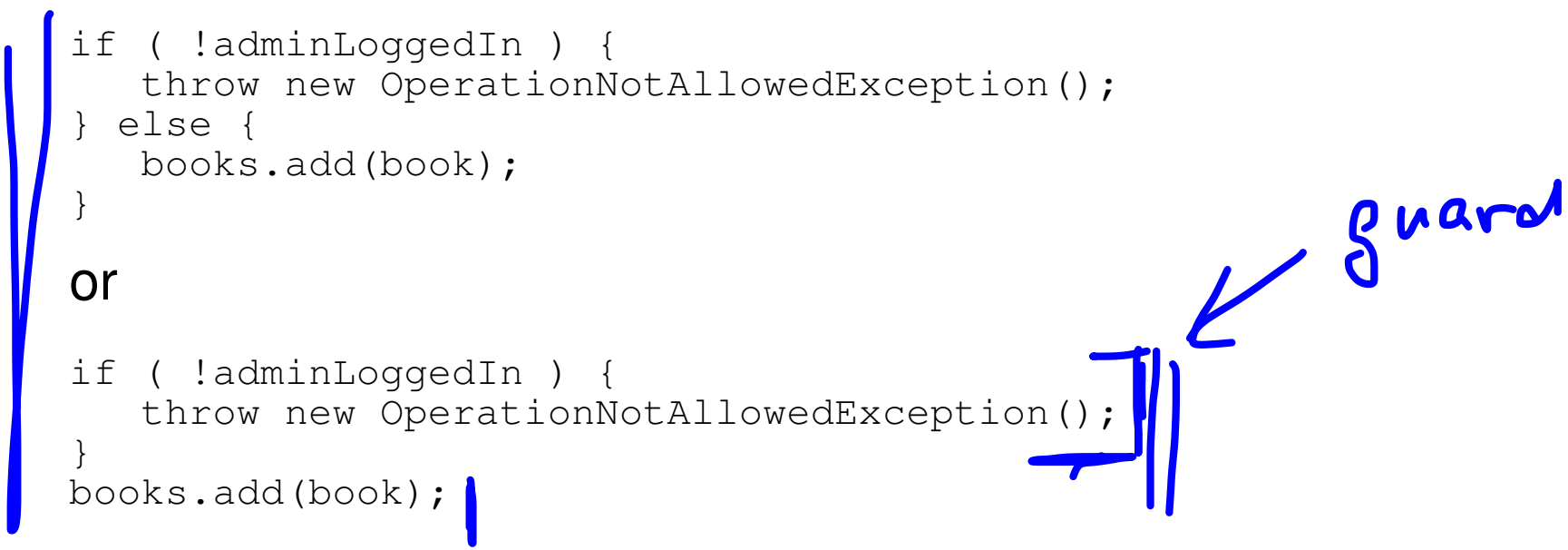
Use boolean variables directly; don't compare boolean variables with true or false

- Better

```
if ( !adminLoggedIn ) {
   throw new OperationNotAllowedException();
} else {
   books.add(book);
}
```

or

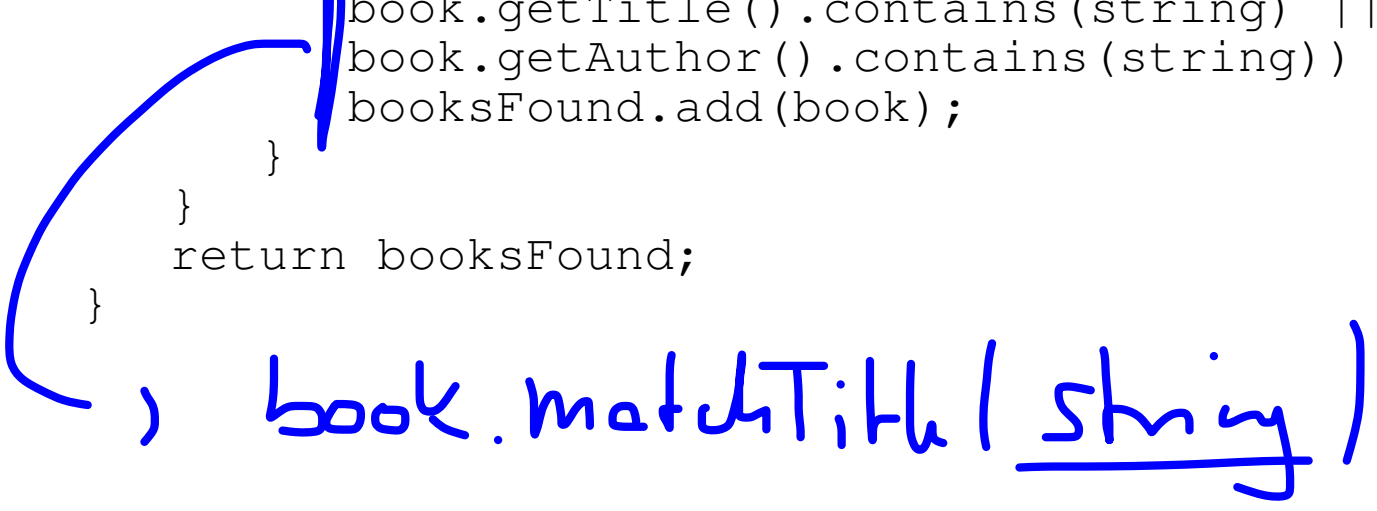
```
if ( !adminLoggedIn ) {
   throw new OperationNotAllowedException();
}
books.add(book);
```



Delegate Responsibility

- Original

```
public List<Book> search(String string) {
    List<Book> booksFound = new ArrayList<Book>();
    for (Book book : books) {
        if (book.getSignature().contains(string) ||
            book.getTitle().contains(string) ||
            book.getAuthor().contains(string)) {
            booksFound.add(book);
        }
    }
    return booksFound;
}
```



Delegate Responsibility

- LibraryApp delegates *contains* functionality to class book

```
public List<Book> search(String string) {
   List<Book> booksFound = new ArrayList<Book>();
   for (Book book : books) {
      if (book.contains(string)) {
         booksFound.add(book);
      }
   }
   return booksFound;
}
```

finding someth.

much more general

could provide matching of a pattern

- In class Book

```
public boolean contains(String string) {
   return signature.contains(string) ||
          title.contains(string) ||
          author.contains(string)
}
```

matching

Java 8 Streams

Advantages:

- Separation of concerns: LibraryApp is searching, Book is providing matching criteria
- Matching criteria can be changed without affecting the search logic