UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Requirements engineering and use case modeling in UML

PROGETTAZIONE, ALGORITMI E COMPUTABILITÀ
(38090-MOD1)

Corso di laurea Magistrale in Ingegneria Informatica

RELATORE
Prof.ssa Patrizia Scandurra

SEDE
DIGIP

DATA
04-10-2021

Outline

- Requirements engineering
- User-centric design (UCD)
- Use case modeling in UML

Requirement Engineering

Consists of:

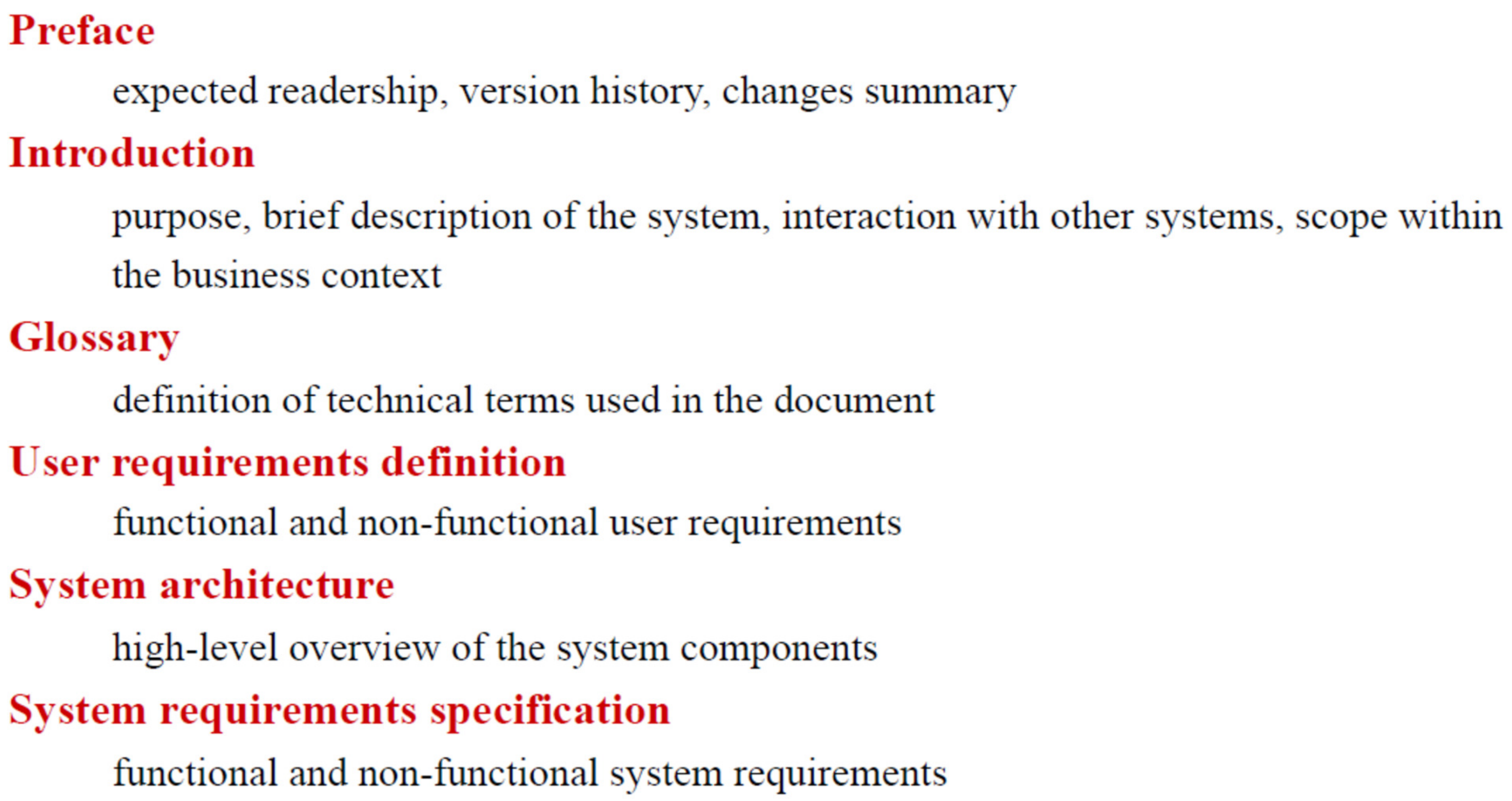
- **Requirement elicitation (or gathering)**
  - include interviews, questionnaires, user observation, workshops, brainstorming, use cases, role playing and prototyping
- **Requirement analysis**
  - determining whether the stated requirements are clear, **complete**, **consistent** and **unambiguous**, and resolving any apparent conflicts
- **Requirement documentation (or recording)**
  - Various forms of documents including a summary-list in natural-language, use cases, user stories, process specifications and a variety of models including data models
- **Requirement specification, validation&verification**
  - By using formal specification methods

The requirements analysis document (or specification document)

- The requirements document is the official statement of what is required of the product developers
  - Should include both a *definition* and a *specification* of requirements
  - Should set of **WHAT** the product should do (*problem domain*) rather than **HOW** it should do it (*solution domain*)
- Why document requirements?
  - Serves as a contract between the customer and the developer
  - Serves as a source of test plans
  - Serves to specify project goals and plan development cycles and increments

Template for the req analysis doc based on the IEEE 830-1998 standard

- *(IEEE Recommended Practice for Software Requirements Specifications)* 1/2



Preface

expected readership, version history, changes summary

Introduction

purpose, brief description of the system, interaction with other systems, scope within the business context

Glossary

definition of technical terms used in the document

User requirements definition

functional and non-functional user requirements

System architecture

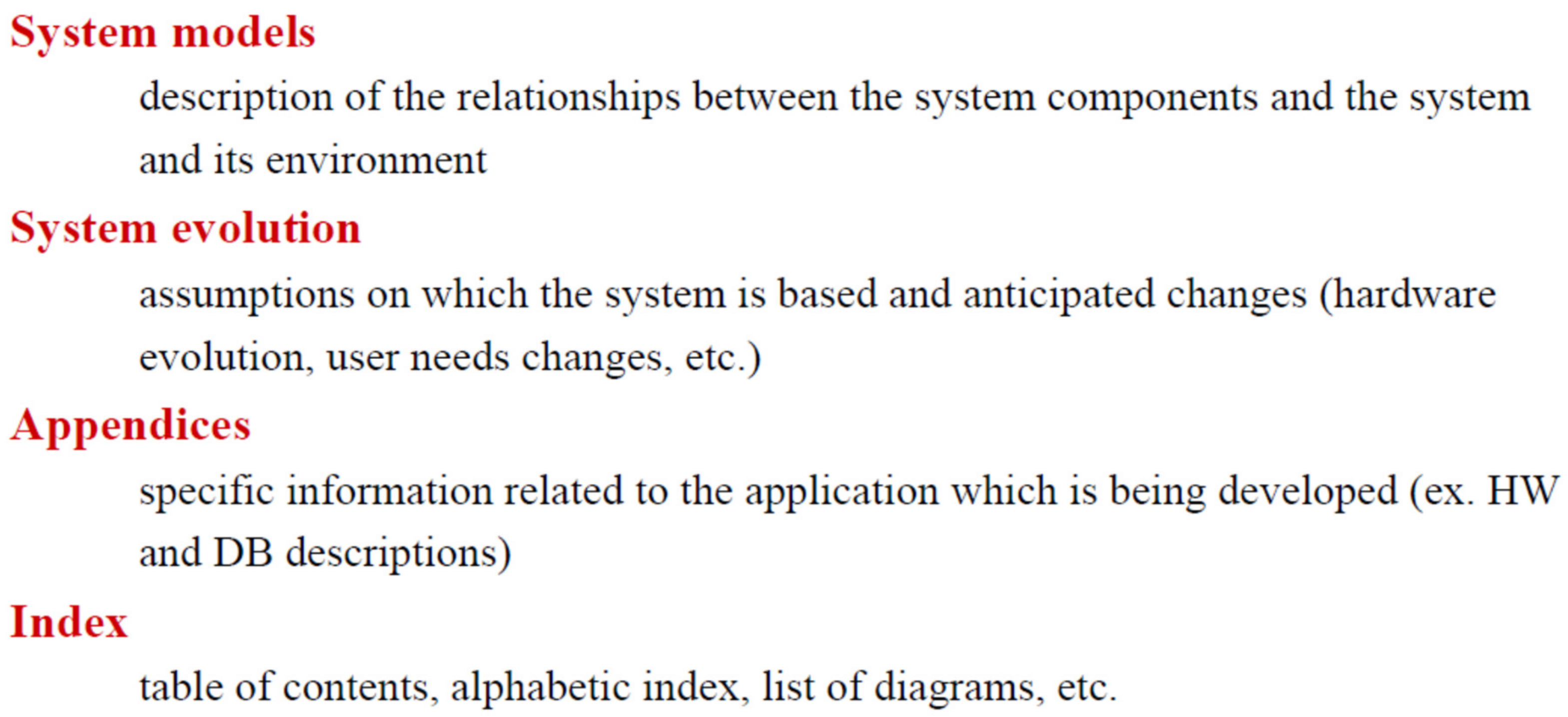
high-level overview of the system components

System requirements specification

functional and non-functional system requirements

Template for the req analysis doc based on the IEEE 830-1998 standard

- *(IEEE Recommended Practice for Software Requirements Specifications)* 2/2



System models

description of the relationships between the system components and the system and its environment

System evolution

assumptions on which the system is based and anticipated changes (hardware evolution, user needs changes, etc.)

Appendices

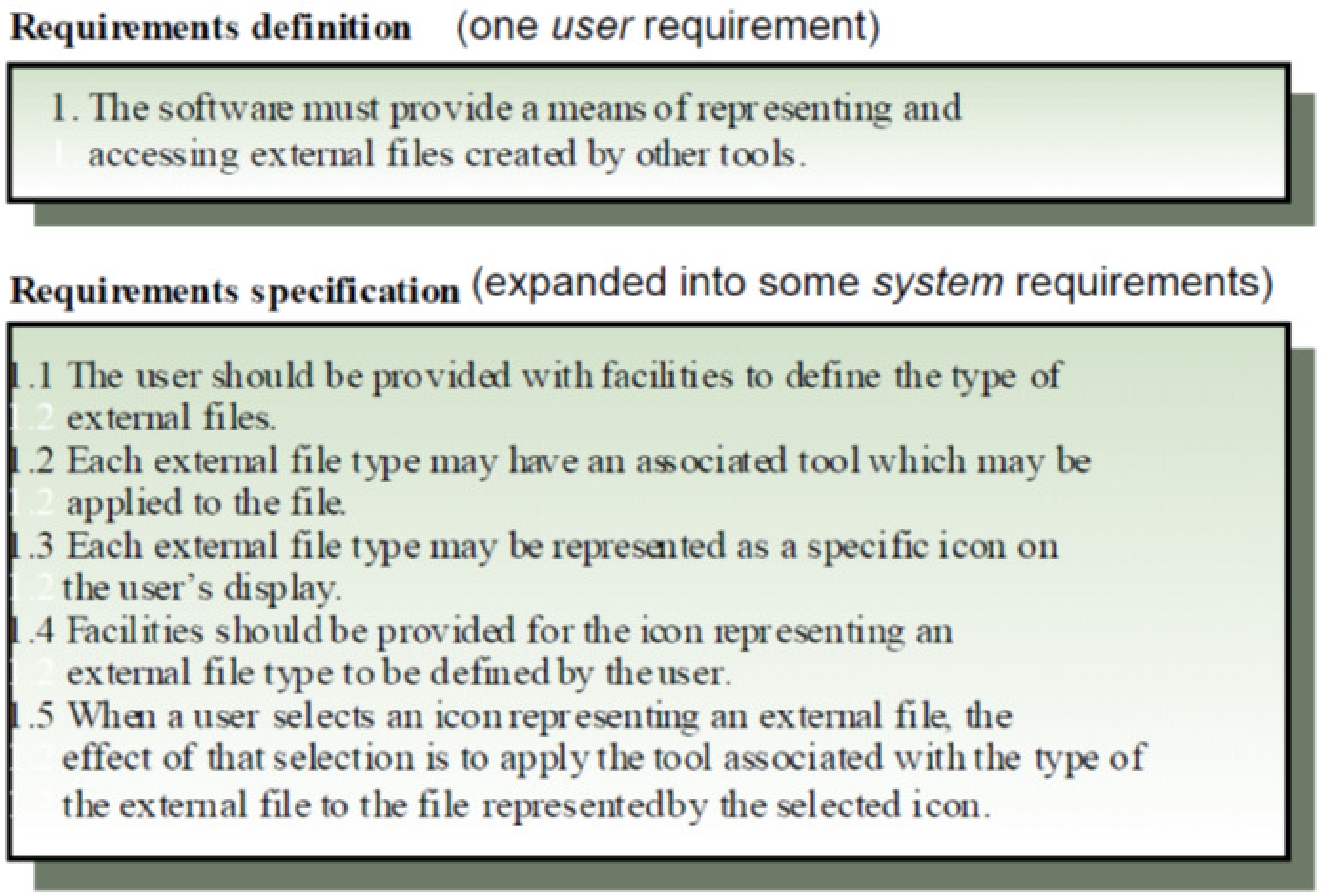
specific information related to the application which is being developed (ex. HW and DB descriptions)

Index

table of contents, alphabetic index, list of diagrams, etc.

User requirements and system requirements

- One user requirement implies many system requirements



Requirements definition (one *user* requirement)

1. The software must provide a means of representing and accessing external files created by other tools.

Requirements specification (expanded into some *system* requirements)

1.1 The user should be provided with facilities to define the type of external files.
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specific icon on the user's display.
1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
1.5 When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file representedby the selected icon.

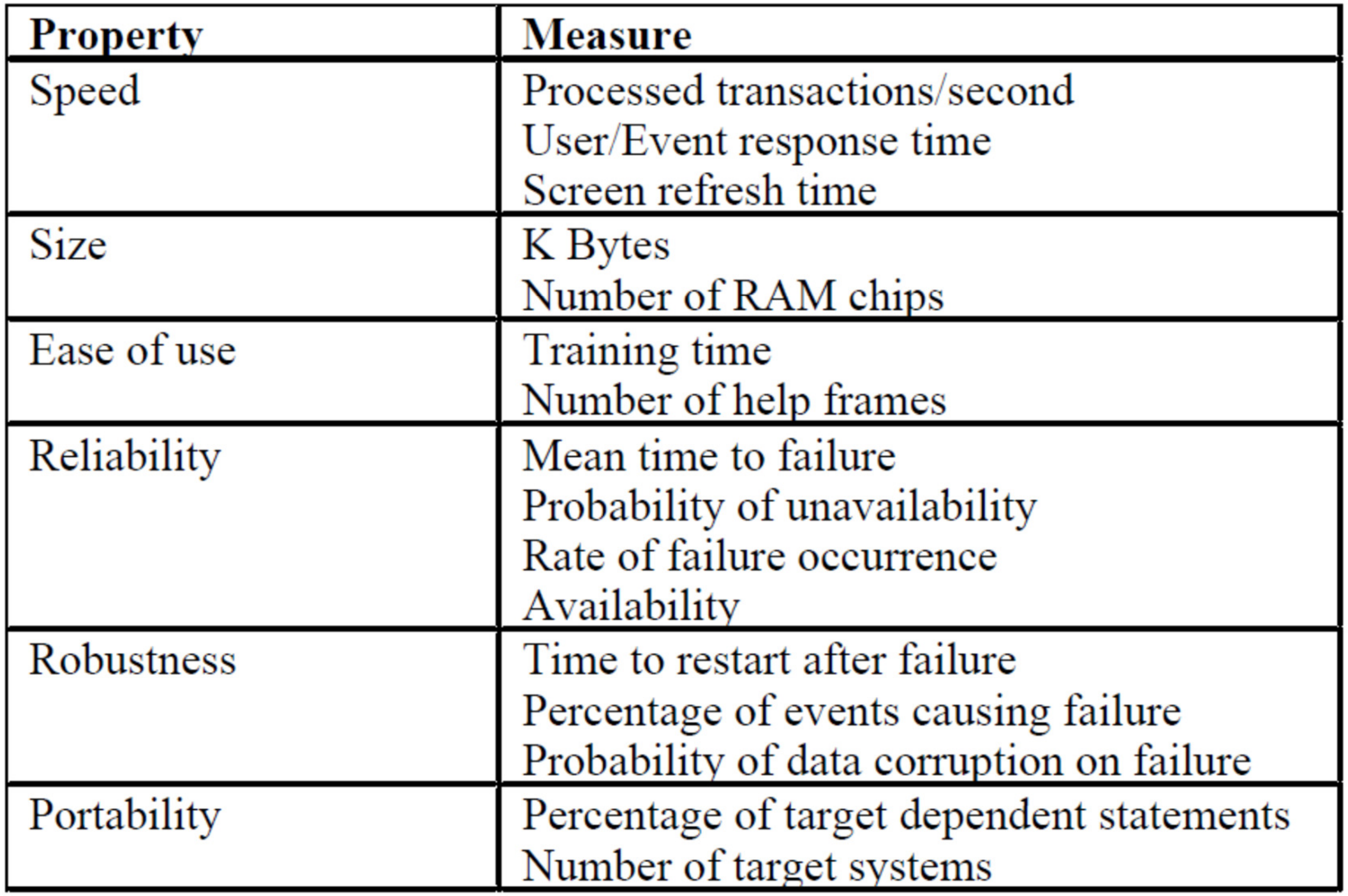
Requirements types

- ***Functional* requirements**
  - statements of *services* the product should provide, how the product should *react* to particular inputs and how the product should *behave* in particular situations
    - *The blood pressure monitor will measure the blood pressure and display it on the in-built screen.*
- ***Non-functional* requirements**
  - describe *properties* and/or *constraints* on the services or functions offered by the product (e.g., execution speed, reliability, etc.)
    - Performance: *The blood pressure monitor will complete a reading within 10 seconds.*
    - Reliability: *The blood pressure monitor must have a failure probability of less than 0.01 during the first 500 readings.*
    - Constraints: timing, accuracy. *The blood pressure monitor will take readings with an error less than 2%.*
- ***Domain requirements***
  - derived from the application domain and describe product characteristics and features that reflect the domain
    - There shall be a standard user interface to all databases which shall be based on the Z39.50 standard

Quantifying non-functional requirements

- Non functional requirements generically **provided by the user** (e.g., the product has to be *easy-touse*) may turn to be **not quantifiable** and thus hard to verify
- It is mandatory to specify non functional requirements by use of a ***measure*** that eventually allows to **quantitatively verify** if the product meets or not those requirements

Examples of misures for non-functional requirements



Property	**Measure**
Speed	Processed transactions/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 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

User Centred Design (UCD)

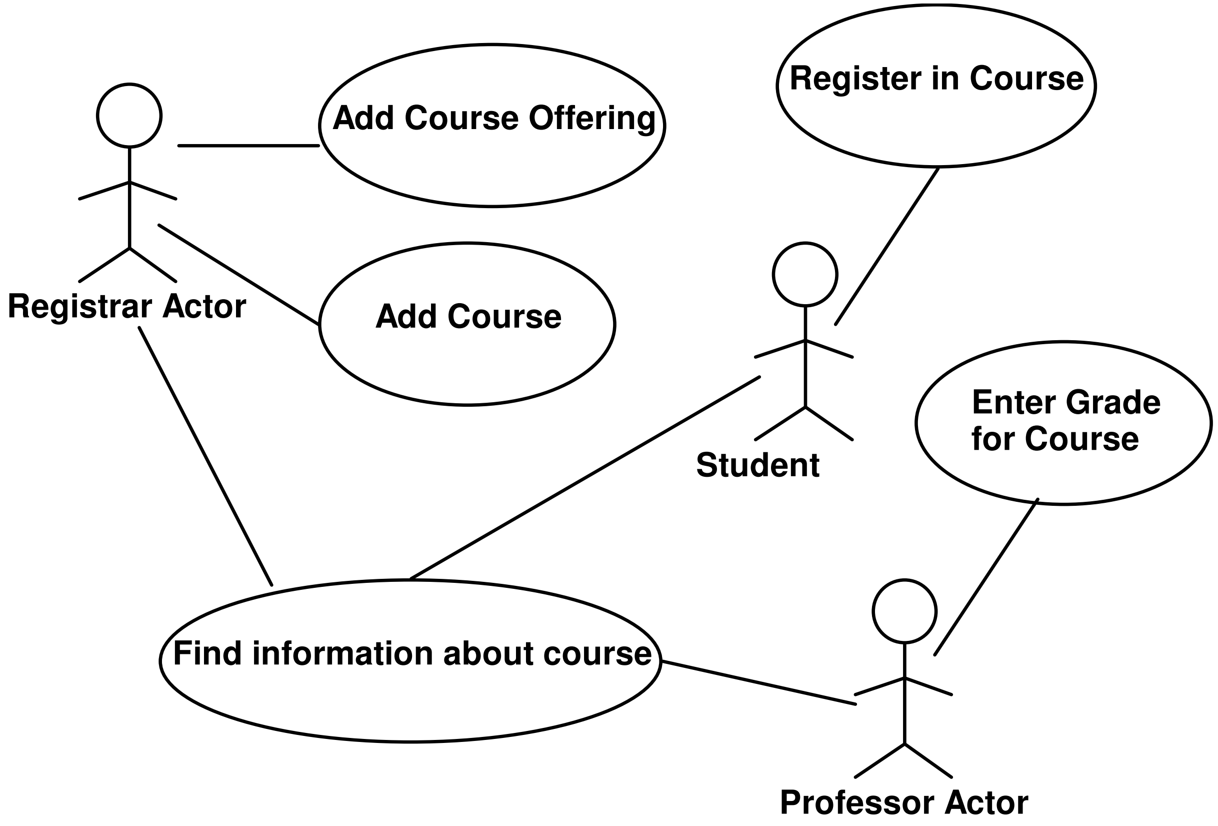
- Software development should focus on the needs of users
  - Understand your users
  - Design software based on an understanding of the users’ tasks
    - **Use case analysis** is the recommended way
    - To be done after the collect requirements process
  - Ensure users are involved in *decision making processes*
    - All decisions that relate to requirements and UI
  - Design the user interface following guidelines for good usability
  - Have users work with and give their feedback about *prototypes*, *on-line help* and *draft user manuals*

Developing a use case model of a system

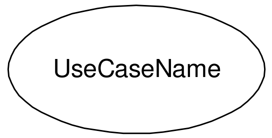
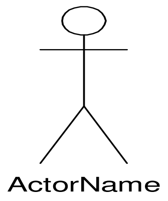
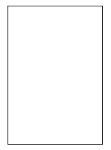
- A view of a system that emphasizes the behavior as it appears to outside users to explore how users will work with your system
- A use case model partitions system functionality into transactions (**use cases**) that are meaningful to users (**actors**)
  - An *actor* is a *role* that a user or some other system plays when interacting with your system
  - A *use case* is a typical *sequence of actions* that a system performs in order to complete a given task

Use-case models

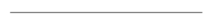
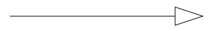
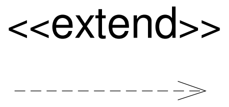
- A ***use case model*** consists of:
  - a UML ***use case diagram*** indicating a set of *use cases* and *how they are related*
  - ***use case descriptions***
  - (Optional) ***Information flow*** can be modeled using UML activity diagrams
  - (Optional) ***Use case scenarios*** can be modeled using sequence diagrams



UML Use Case Modeling: Core Elements

Construct	Description	Syntax
use case	*A sequence of actions*, including variants, that a system (or other entity) can perform, interacting with actors of the system	
actor	A coherent set of *role*s that users of use cases play when interacting with these use cases	
system boundary	Represents the *boundary* between the physical system and the actors who interact with the physical system	

UML Use Case Modeling: Core Relationships

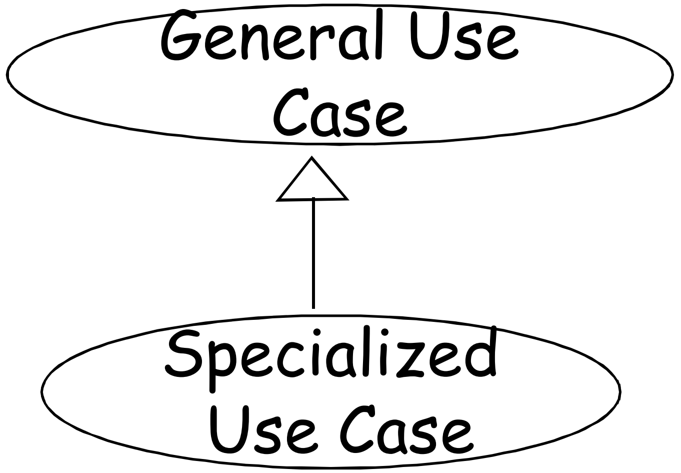
Construct	Description	Syntax
association	The participation of an actor in a use case. i.e., instance of an actor and instances of a use case communicate with each other.	
generalization	A taxonomic relationship between a more general use case (actor) and a more specific use case (actor).	
extend	A relationship from an *extension* use case to a *base* use case, specifying how the behavior for the extension use case can be inserted into the behavior defined for the base use case.	

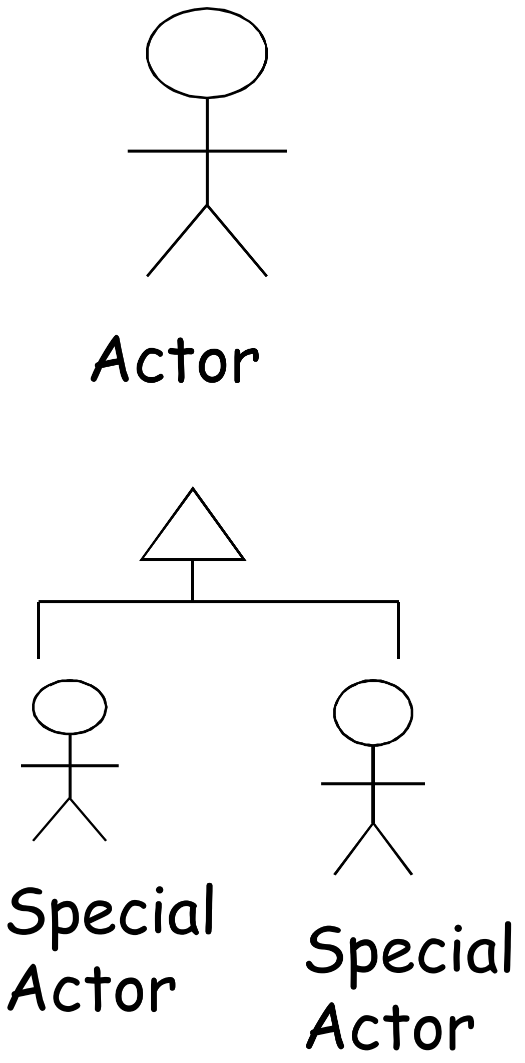
UML Use Case Modeling: Core Relationships

Construct	**Description**	**Syntax**
include	A relationship from a *base* use case to an *inclusion* use case, specifying how the behavior for the inclusion use case is inserted into the behavior defined for the base use case.	<<include>> ---------->

Generalizations

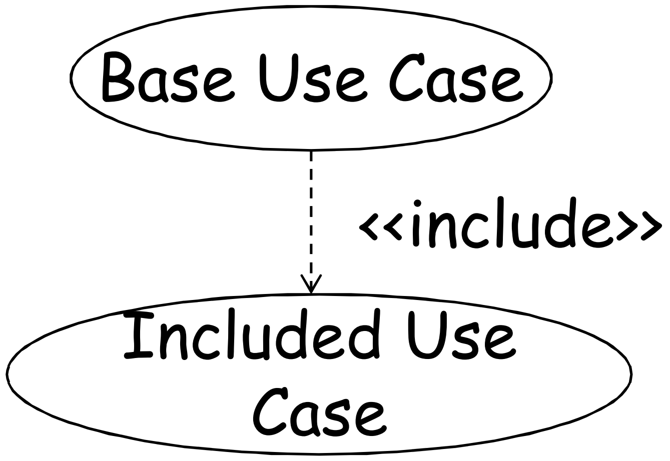
- Much like superclasses in a class diagram
- A generalized use case represents *several similar* use cases
- One or more specializations provides details of the similar use cases





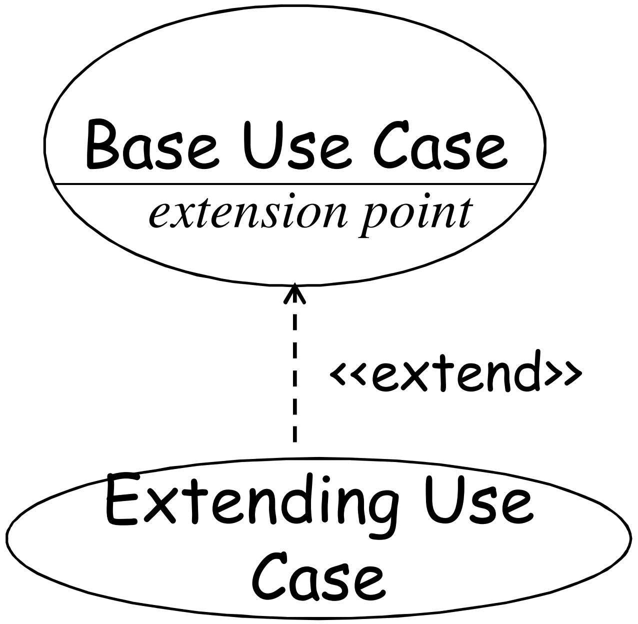
Inclusions

- Allow one to express *commonality* between several different use cases
- Are included in other use cases
  - Even very different use cases can share sequence of actions
  - Enable you to avoid repeating details in multiple use cases
- Represent the performing of a *lower-level task* with a lower-level goal

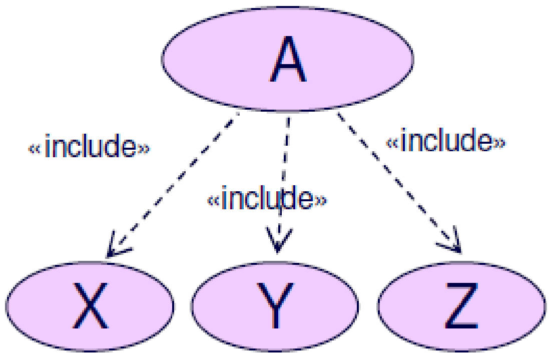


Extensions

- Used to make *optional* interactions explicit or to handle *exceptional* cases
- By creating separate use case extensions, the description of the basic use case remains simple
- A use case extension must list all the steps from the beginning of the use case to the end
    - Including the handling of the unusual situation

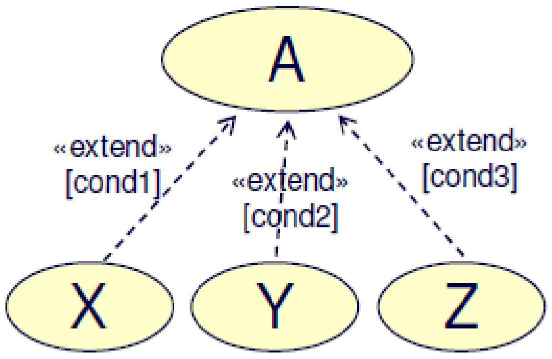


Use case relationships: summary



«include»

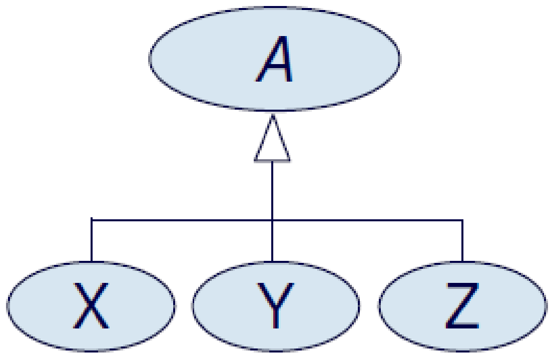
A *always* contains
X, Y and Z



«extend»

A
or A+iX
or A+jY
or A+kZ
or A+iX+jY
or A+iX+kZ
or A+jY+kZ
or A+iX+jY+kZ

i, j, k = no of times cond 1, 2 or 3 is true



generalisation

X, Y *or* Z
(*A* is abstract)

Use case descriptions

- Textual description that should cover the ***full sequence of steps*** from the beginning of a task until the end
- A use case should describe the ***user's interaction*** with the system
  - not the computations the system performs
- A use case should be written so as to be as *independent* as possible from any particular user interface design
  - Use abstract commands like "Choose the Open command …" instead of "Push the Open button …"

How to describe a single use case

A. Name: Give a short, descriptive name to the use case.

B. Actors: List the actors who can perform this use case.

C. Goals: Explain what the actor or actors are trying to achieve.

D. Preconditions: State of the system before the use case.

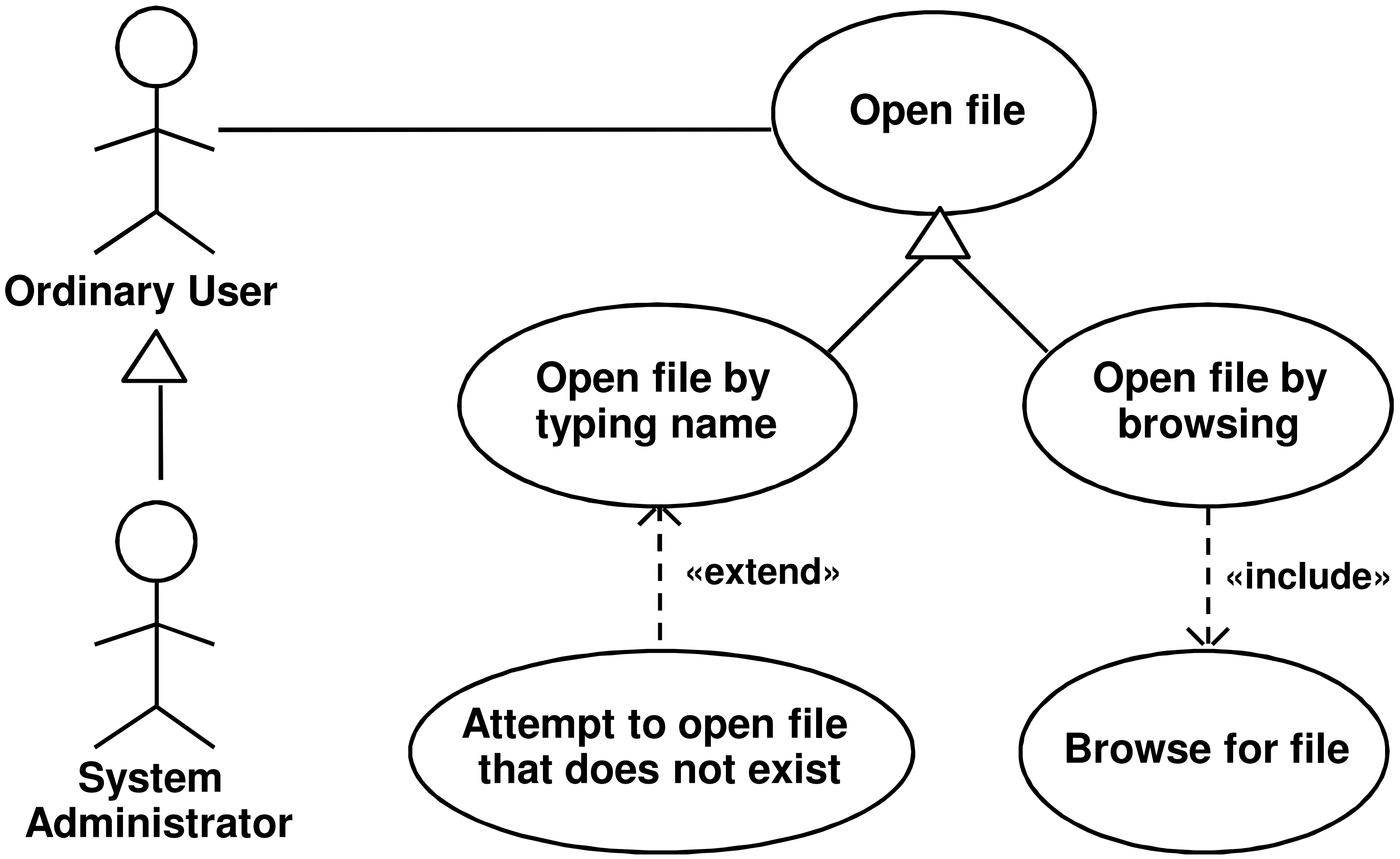
E. Description: Give a short informal description.

F. Steps: Describe each step using a 2-column format, which the left column showing the actions taken by the actor, and the right column showing the system's responses.

G. Related use cases and alternative flows: generalizations, specializations, inclusions, extensions of this use case.

H. Postconditions: State of the system in following completion.

A (small) complete example



Example description of a use case

Use case: **Open file**

Related use cases:
Generalization of:
• Open file by typing name
• Open file by browsing

Steps:

Actor actions	**System responses**
1. Choose ‘Open…’ command	2. File open dialog appears
3. Specify filename	
4. Confirm selection	5. Dialog disappears

Example (continued)

Use case: Open file by browsing

Related use cases:
Specialization of: Open file
Includes: Browse for file

Steps:

Actor actions	**System responses**
1. Choose ‘Open…’ command	2. File open dialog appears
3. **Include** (Browse for file)	
4. Confirm selection	5. Dialog disappears

Example (continued)

Use case: **Browse for file (inclusion)**

Steps:

Actor actions	**System responses**
1. If the desired file is not displayed, select a directory	2. Contents of directory is displayed
3. Repeat step 1 until the desired file is displayed	
4. Select a file	

Example (continued)

Use case: **Open file by typing name**

Related use cases:
Specialization of: Open file

Steps:

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	5. Dialog disappears

Example (continued)

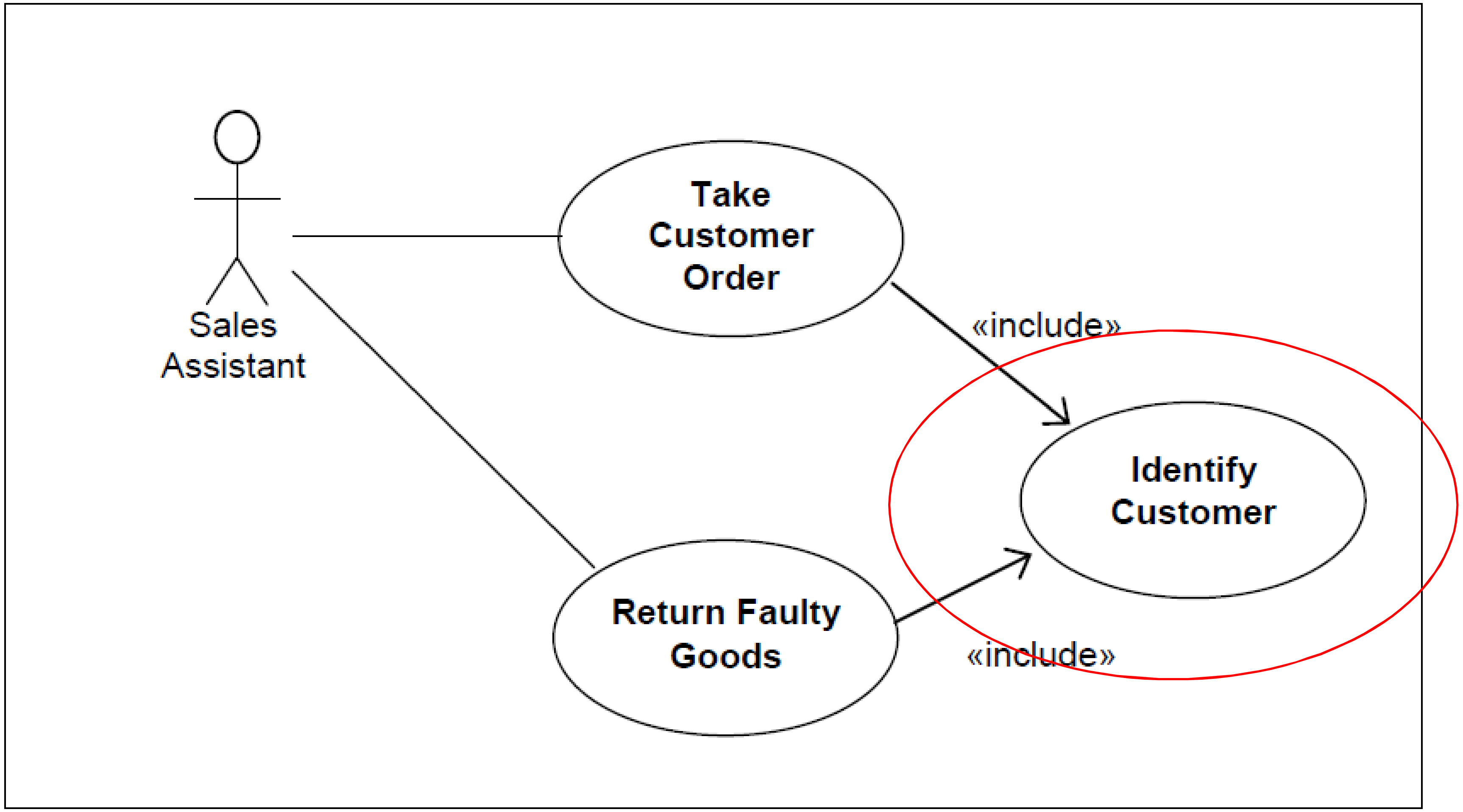
Use case: **Attempt to open file that does not exist**

Related use cases:
Extension of: Open file by typing name

Actor actions	**System responses**
1. Choose 'Open…' command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	5. System indicates that file does not exist
6. Correct the file name	
7. Click 'Open'	8 Dialog disappears

Use case description with alternative flows

- Consider this example



Use case description with alternative flows

Use Case: “Identify Customer”

Basic Flow:

1. Actor enters search criteria, surname and postcode
2. System displays matching Customers
3. Actor selects Customer
4. System displays Customer details
5. Actor confirms Customer

Alternative Flows:

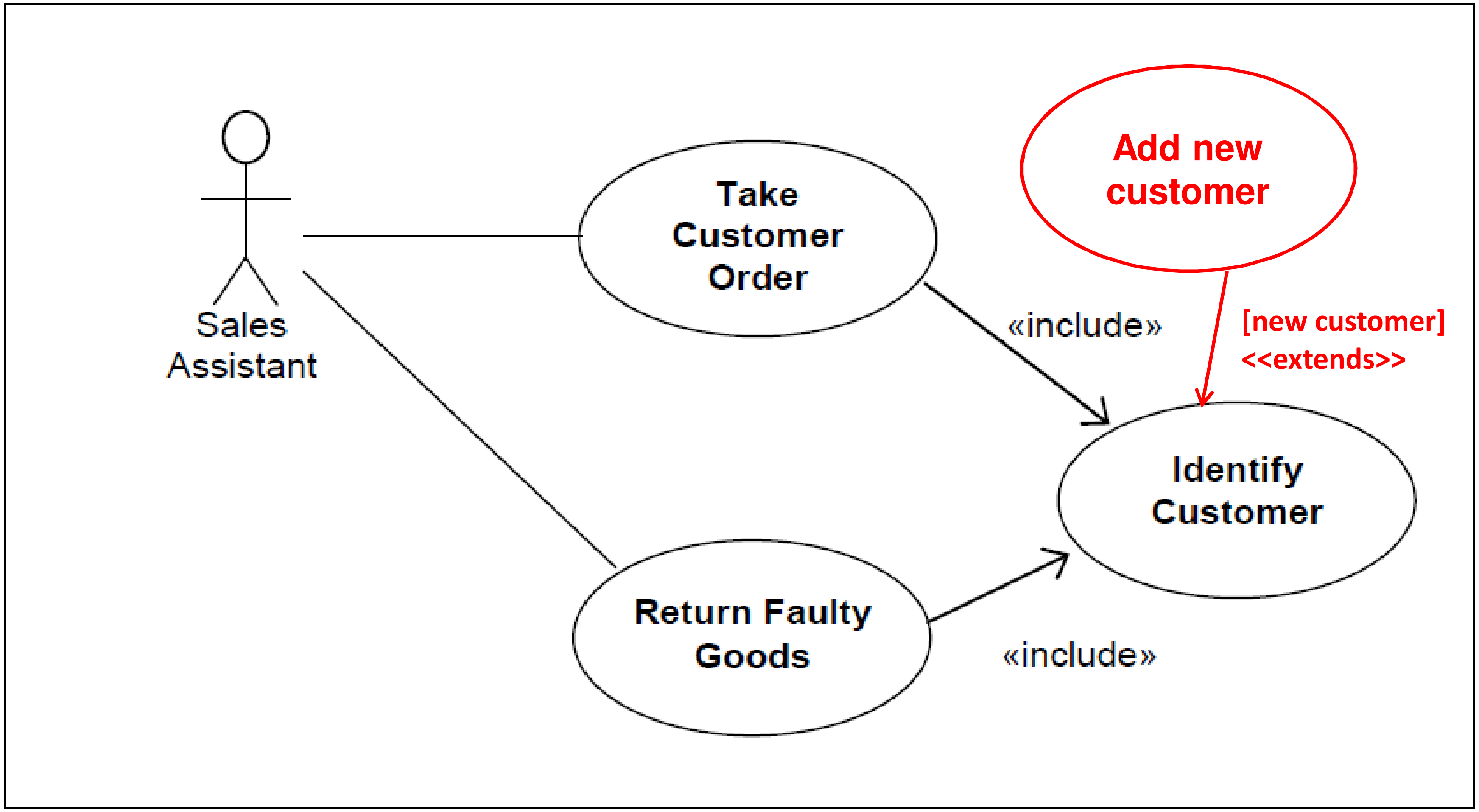
[new customer]

After step 2, when the System does not display the required Customer, Actor creates new Customer,

1. Actor selects to add new Customer
2. Actor enters Customer details

Resume at step 5, to confirm Customer

Use case description: alternative flows versus «extend» use cases



Use case description: alternative flows versus «extend» use cases

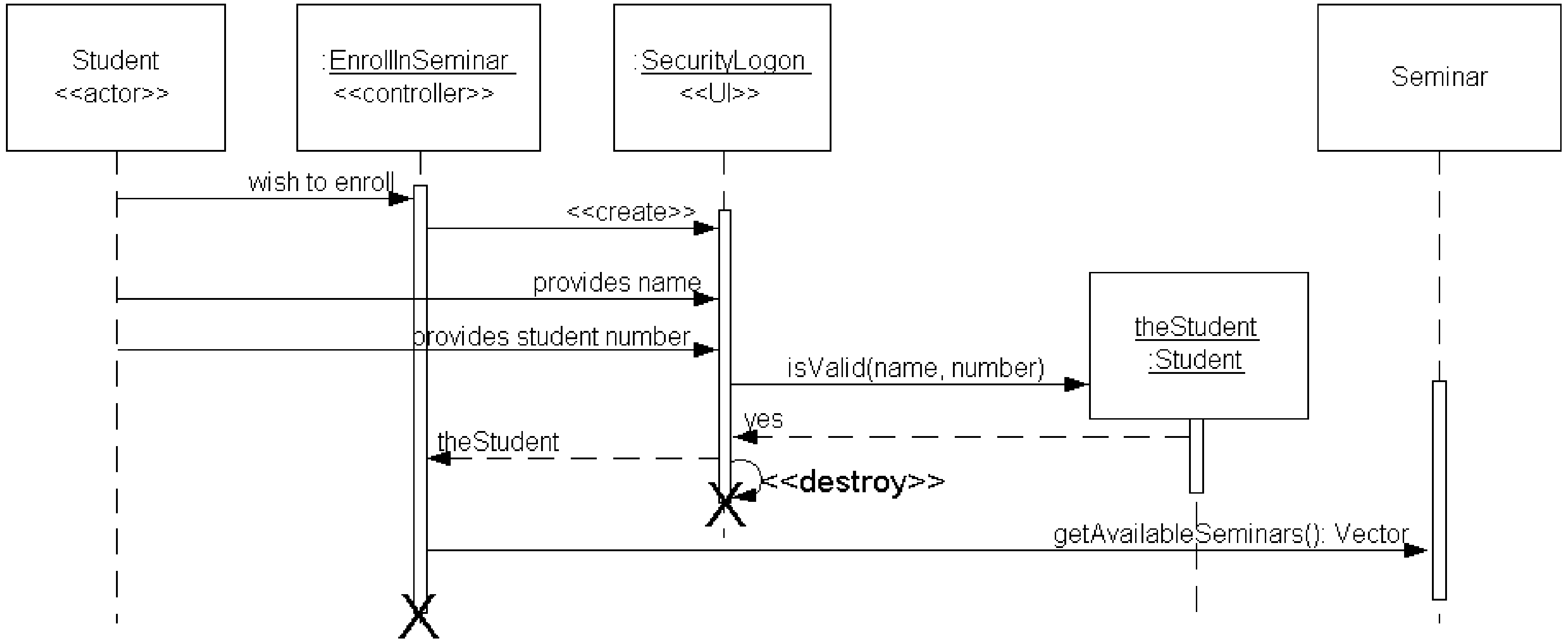
- The **«extend» relationship** allows us to modify the behavior of the base use case **to add something extra to the base flow**
  - as **a conditional «include»**
- We could do this **also through an alternative flow**, **however**, **the use case may become difficult to manage**
  - new functionality may open up a whole raft of possibilities and with further sub flows

Scenarios

- A scenario is an *instance* of a use case
  - It expresses a *specific occurrence* of the use case
    - a specific actor …
    - at a specific time …
    - with specific data
- Represented, e.g., by UML sequence diagrams
- Useful for **testing purposes**

Scenario – example of sequence diagram

Use case: Enrolling a student



Student
<<actor>>
:EnrollInSeminar
<<controller>>
:SecurityLogon
<<UI>>
Seminar
wish to enroll
<<create>>
provides name
provides student number
theStudent
:Student
isValid(name, number)
yes
theStudent
<<destroy>>
getAvailableSeminars(): Vector