Introduction to Software Engineering

03 Requirements Analysis

Stephan Krusche

10 May 2022
Technical University of Munich
https://ase.in.tum.de

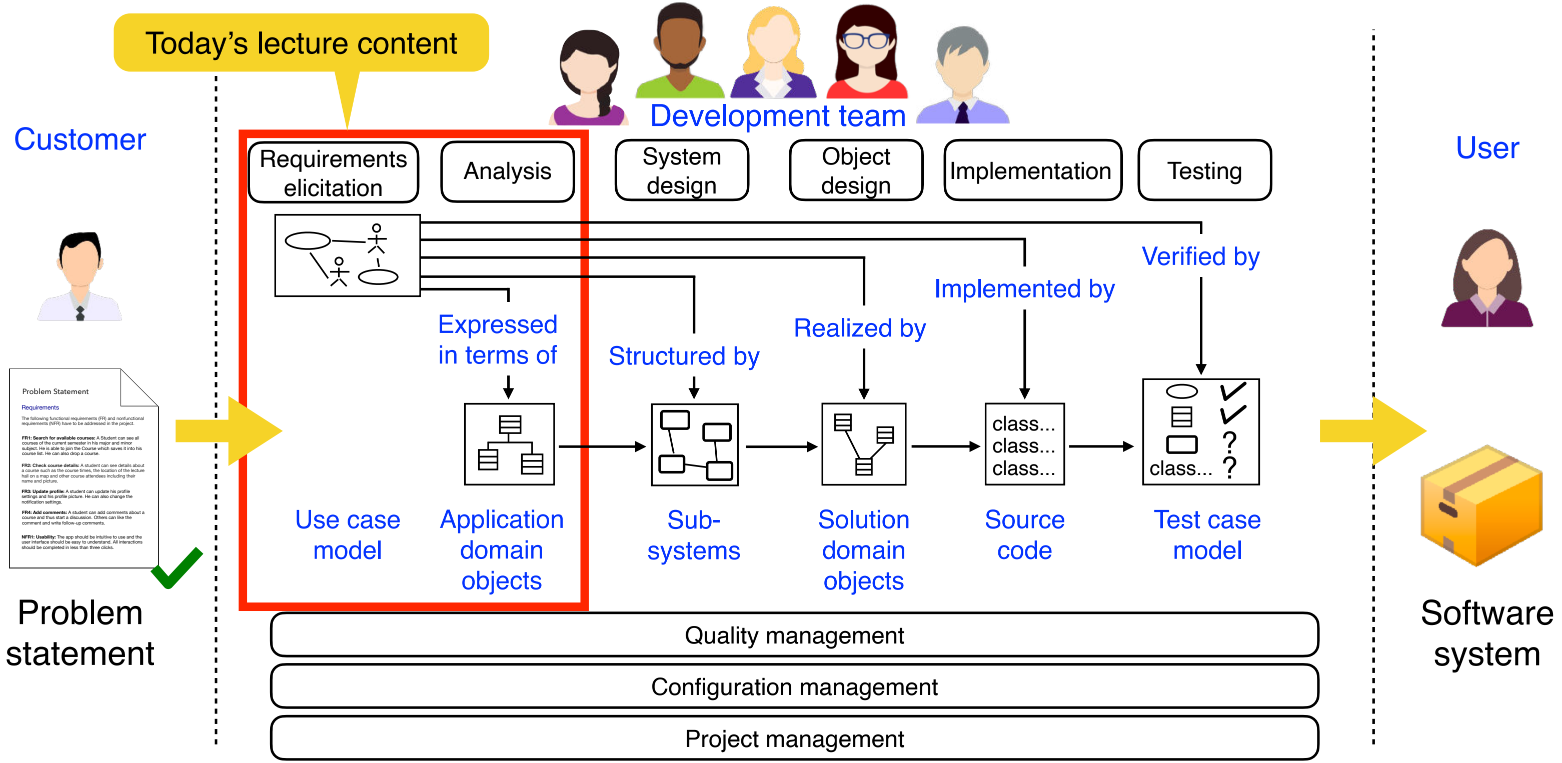
Roadmap of the lecture

- **Context and assumptions**
  - We have completed an initial explanation of use cases and class diagrams
  - You know the most important activities of model-based software engineering
  - You understand Scrum, UML diagrams, JavaFX, and Gradle
- **Learning goals: at the end of this lecture you are able to**
  - Explain the differences between requirements elicitation and analysis
  - Apply scenario-based design
  - Define functional and nonfunctional requirements
  - Apply object modeling using stereotypes
  - Model the dynamic behavior using communication diagrams

Course schedule (Garching)

#	Date	Subject
1	26.04.22	Introduction
2	03.05.22	Model-based Software Engineering
3	10.05.22	Requirements Analysis
4	17.05.22	System Design I
5	24.05.22	System Design II
6	31.05.22	Object Design I
	07.06.22	**Holiday (no lecture, no tutor groups)**
7	14.06.22	Object Design II
8	21.06.22	Testing
	28.06.22	**no lecture, no tutor groups**
9	05.07.22	Software Lifecycle Modeling
10	12.07.22	Software Configuration Management
11	19.07.22	Software Quality Management
12	26.07.22	Project Management

Overview of model based software engineering



Today's lecture content

Customer

Development team

User

Requirements elicitation

Analysis

System design

Object design

Implementation

Testing

Expressed in terms of

Structured by

Realized by

Implemented by

Verified by

Use case model

Application domain objects

Sub-systems

Solution domain objects

Source code

Test case model

Problem statement

Software system

Quality management

Configuration management

Project management

Problem Statement

Requirements

The following functional requirements (FR) and nonfunctional requirements (NFR) have to be addressed in the project.

FR1: Search for available courses: A Student can see all courses of the current semester in his major and minor subject. He is able to join the Course which saves it into his course list. He can also drop a course.

FR2: Check course details: A student can see details about a course such as the course times, the location of the lecture hall on a map and other course attendees including their name and picture.

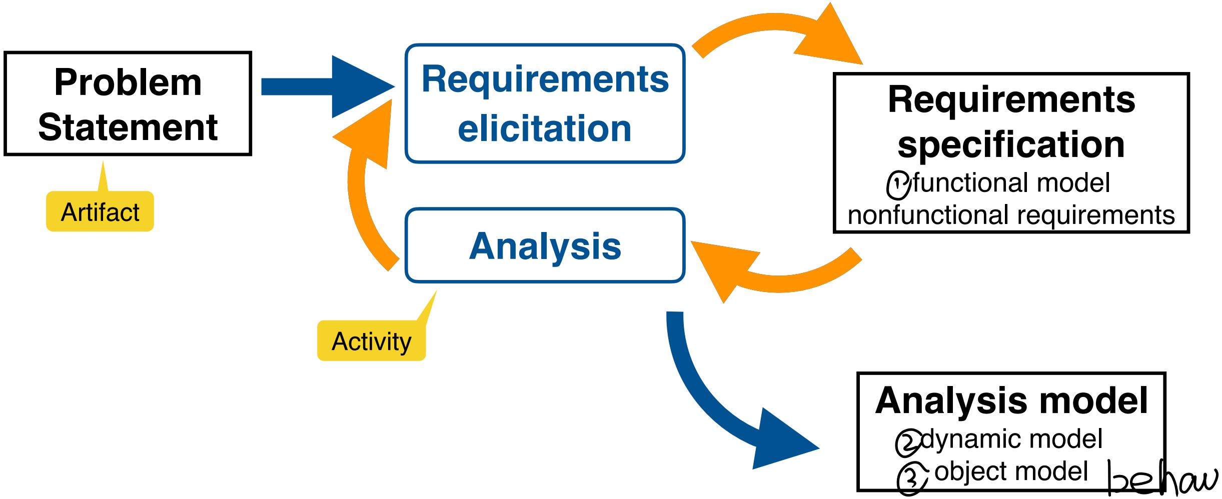
FR3: Update profile: A student can update his profile settings and his profile picture. He can also change the notification settings.

FR4: Add comments: A student can add comments about a course and thus start a discussion. Others can like the comment and write follow-up comments.

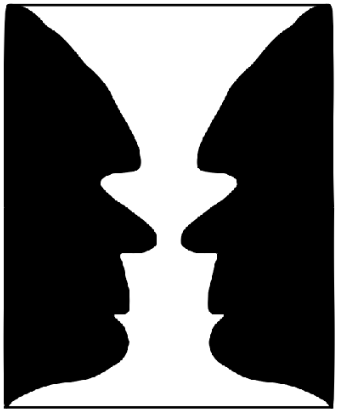
NFR1: Usability: The app should be intuitive to use and the user interface should be easy to understand. All interactions should be completed in less than three clicks.

Overview: requirements engineering

- **Requirements elicitation:** describe the purpose of the system
- **Analysis:** create a model of the system, which is correct, complete, consistent, and verifiable



behaviour



An optical illusion

Ambiguity in drawings → we need to be **consistent** in our models

①+②+③ System Ob

Requirements engineering

Combination of the two activities: **requirements elicitation** and **analysis**

- An activity that defines the requirements of the system under construction
- Also called requirements analysis
- **Requirements elicitation**
  - Definition of the system in terms understood by a customer or user
  - **Result**: requirements specification
- **Analysis**
  - Definition of the system in terms understood by a developer
  - **Result**: analysis model (also called technical specification, in German: “Lastenheft”)

Outline

- **Requirements elicitation**
  - Types of requirements
  - Scenarios
  - Use cases
- Analysis
  - Decomposition
  - Object modeling
  - Stereotypes
  - Dynamic modeling

Activities during requirements elicitations

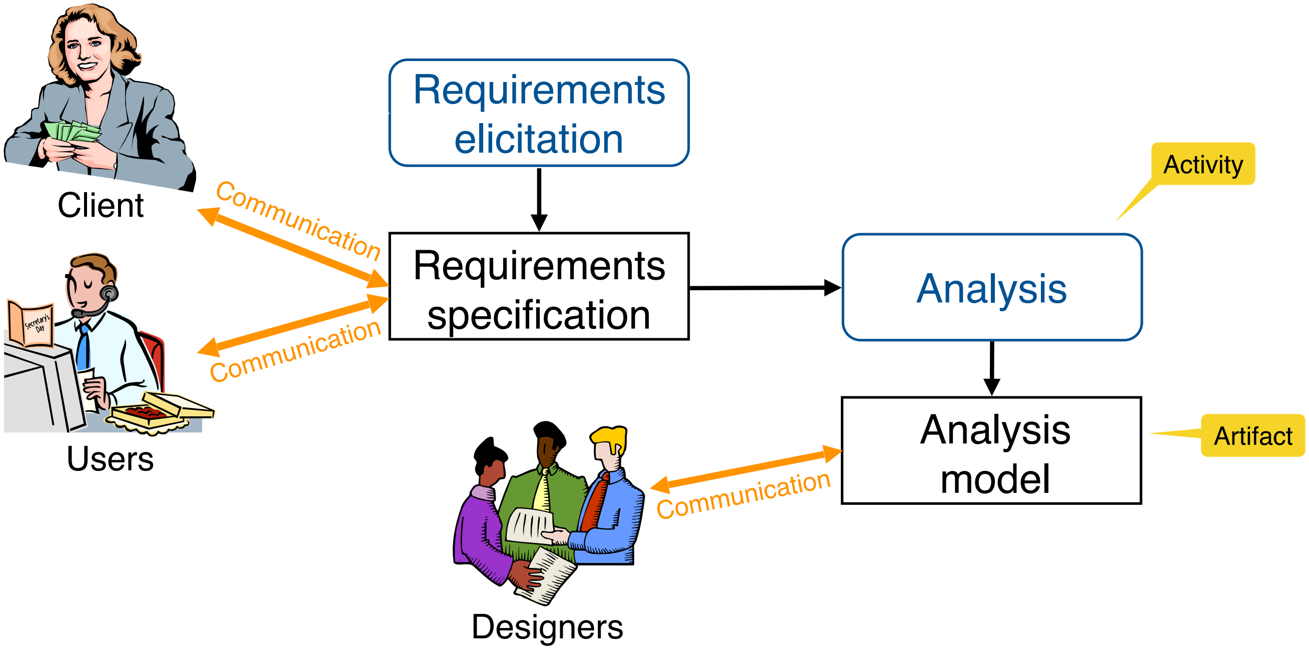
- **Identify actors:** different types of users of the future system
- **Identify scenarios:** a set of detailed descriptions for typical functionality provided for the future system written in natural language 具体
- **Derive use cases:** generalize and abstract the scenarios to represent the functionality of the future system think abstract
- **Refine use cases:** detail each use case and describe the behavior of the system in the presence of errors and exceptional conditions
- **Identify relationships among use cases:** dependencies among use cases reduce duplication (e.g. «extend», «include»)
- **Identify nonfunctional requirements:** agree on measurable quality aspects of the system that are not directly related to functionality

Requirements elicitation is a development activity

- Determine the **requirements** of the system specified by the customer and / or the user
- Another formulation: “from the problem statement to the requirements specification”
- Still a very **informal process** with many problems
- Many software projects fail because of a bad / wrong requirements elicitation
- In **agile projects**: requirements elicitation is carried out continuously

(detail)

Requirements engineering: informal model



Client

Users

Designers

Communication

Communication

Communication

Requirements elicitation

Requirements specification

Analysis

Analysis model

Activity

Artifact

Requirements specification vs. analysis model

- Both are models focusing on the requirements from the user's view of the system
- The **requirements specification** uses natural language (derived from the problem statement)
  - Verb noun analysis (e.g. Abbott's technique)
  - Example: scenario
- The **analysis model** uses a (semi-) formal language
  - Example: UML

Requirements

- **Features** that the system must **have** in order to be **accepted** by the client
- **Constraints** that the system must **satisfy** in order to be **accepted** by the client
- Requirements describe the user's view of the system
- Identify the **what** of the system, not the **how**

Part of requirements

- Functionality
- User interaction
- Error handling
- Environmental conditions (interfaces)

Not part of requirements

- System design
- Implementation technology
- Development methodology

Review: typical software development activities

Requirements elicitation	What is the problem?	Application domain
Analysis		
System design	What is the solution?	
Object design	What are the best data structures and algorithms for the solution?	Solution domain
Implementation	How is the solution constructed?	
Testing	Is the problem solved?	
Delivery	Can the customer use the solution?	Application domain
Maintenance	Are enhancements needed?	

Requirements elicitation: **difficulties**

Questions that need to be answered

1. How can we identify the **purpose** of a system?
   - What are the requirements, what are the constraints?限制
2. How can we identify the system boundary?
   - What is **inside**, what is **outside** the system?
   - Defining the system **boundary** is often difficult

Outline

- Requirements elicitation
  - ➡ **Types of requirements**
  - Scenarios
  - Use cases
- Analysis
  - Decomposition
  - Object modeling
  - Stereotypes
  - Dynamic modeling

Types of requirements elicitation

1. Greenfield engineering

- Development from scratch, no prior system exists
- Requirements extracted from client and user
- Triggered by user needs

2. Re-engineering

(from other company)

- Re-design or re-implementation of an existing system
- Requirements triggered by new technology or new user needs

3. Interface Engineering

- Provide services of an existing system in a new environment
- Requirements triggered by technology or new market needs

→ Each of these requirements elicitation types should start with a problem statement

Types of requirements

- Functionality: what is the software supposed to do?
  - External interfaces (➡ actors): interaction with people, hardware, other software

Functional requirements

Functionality

- Includes
  - Relationship of **outputs to inputs**
  - Response to **abnormal situations**
  - **Exact sequence** of operations
  - **Validity checks** on the inputs (birthday)
- Functional requirements (FRs) should be phrased as an **action** (imperative mood)
- Examples
  - Withdraw money
  - Transfer money

Do **not** write **withdrawing money**
Do **not** write **money withdrawal**

- Natural text follows the action to explain the functionality in more detail

Types of requirements

- Functionality: what is the software supposed to do?
  - External interfaces (➡ actors): interaction with people, hardware, other software

Functional requirements

- Quality requirements
  - Usability
  - Reliability
  - Performance
  - Supportability
- Constraints (pseudo requirements)
  - Required standards, operating environment, etc.

Nonfunctional requirements

➡ **FURPS** is an acronym representing a model for classifying software attributes (functional and nonfunctional requirements)

Nonfunctional requirements (NFRs)

Criteria for defining NFRs

- **Bounded:** when they lack bounded context, NFRs may be irrelevant and lead to significant additional work
  - **Example:** an airplane’s flight controls should be more rigid in terms of reliability than the infotainment system
- **Independent:** NFRs should be independent of each other so that they can be evaluated and tested without impacting other system qualities
- **Measurable:** NFRs that cannot be measured are too vague and can easily be misunderstood

不暧昧

- **Testable:** NFRs must be stated with objective, measurable, and testable criteria

Nonfunctional requirements

- Usability
- **R**eliability
  - Robustness
  - Safety
  - Security
- **P**erformance
  - Response time
  - Throughput
  - Availability
  - Accuracy
- **S**upportability
  - Adaptability
  - Maintainability
  - Portability

Nonfunctional requirements definitions (1)

- **Usability:** the ease with which actors can use system functions
  - Usability is one of the most frequently misused requirement terms
  - **Bad example:** “The system is easy to use”
  - **Important: usability must be measurable; otherwise it is marketing!**
  - **Good example:** “passengers need at most 5 clicks to purchase a ticket”

如需協助或遇緊急事故
請按「HELP」鍵
Press the" HELP "button
for assistance and
emergencies
請先按「Air」鍵，再輸入下列數字鍵，直接與有關之航空公司聯絡。
Press the" Air " button, then the two digit number listed to access airline enquiry hot line

Usability categories

NON
的要求

1. **Learnability:** is the user interface easy to learn?
2. **Efficiency:** once it is learned, is it fast to use?
3. **Memorability:** is it easy to remember what the user has learned?
4. **Error handling and robustness:** can the system recover from errors?
5. **Satisfaction / user experience:** is the user interface enjoyable to use?

UI

https://www.nngroup.com/articles/usability-101-introduction-to-usability

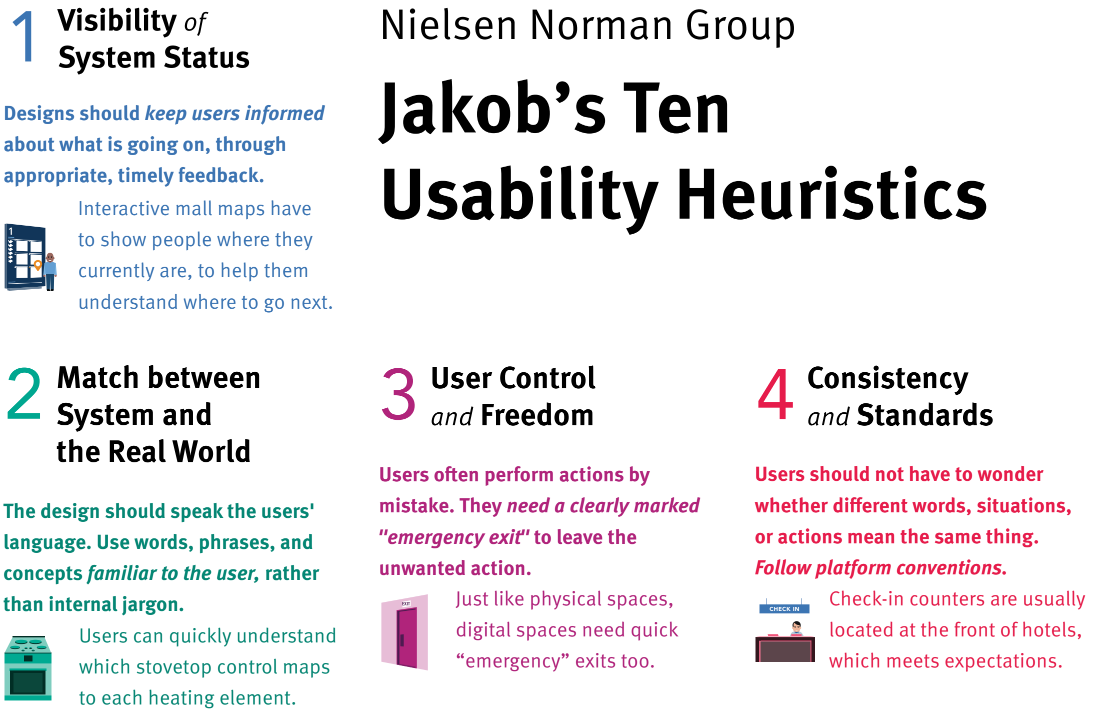
Nielsen’s 10 heuristics (overview)

1. Visibility of system status
2. Match between system and the real world
3. User control and freedom
4. Consistency and standards
5. Error prevention
6. Recognition rather than recall
7. Flexibility and efficiency of use
8. Aesthetic and minimalist design
9. Help users recognize, diagnose, and recover from errors
10. Help and documentation

Hint: download and print the free usability heuristic posters
Hang them at home, in your office, or gift them to a fellow student

https://www.nngroup.com/articles/ten-usability-heuristics

Nielsen’s 10 heuristics (details)



Nielsen Norman Group

Jakob’s Ten Usability Heuristics

1 Visibility *of* System Status

Designs should *keep users informed* about what is going on, through appropriate, timely feedback.

Interactive mall maps have to show people where they currently are, to help them understand where to go next.

2 Match between System and the Real World

The design should speak the users' language. Use words, phrases, and concepts *familiar to the user,* rather than internal jargon.

Users can quickly understand which stovetop control maps to each heating element.

3 User Control *and* Freedom

Users often perform actions by mistake. They *need a clearly marked "emergency exit"* to leave the unwanted action.

Just like physical spaces, digital spaces need quick “emergency” exits too.

4 Consistency *and* Standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. *Follow platform conventions.*

Check-in counters are usually located at the front of hotels, which meets expectations.

https://media.nngroup.com/media/articles/attachments/Heuristic_Summary1-compressed.pdf

Nielsen’s 10 heuristics (details, continued)

5 Error Prevention

Good error messages are important, but the best designs carefully *prevent problems* from occurring in the first place.

Guard rails on curvy mountain roads prevent drivers from falling off cliffs.

6 Recognition Rather Than Recall

***Minimize the user's memory load* by making elements, actions, and options visible. Avoid making users remember information.**

People are likely to correctly answer “Is Lisbon the capital of Portugal?”.

7 Flexibility *and* Efficiency of Use

Shortcuts — hidden from novice users — may *speed up the interaction* for the expert user.

Regular routes are listed on maps, but locals with more knowledge of the area can take shortcuts.

8 Aesthetic *and* Minimalist Design

Interfaces should not contain information which is irrelevant. Every extra unit of information in an interface *competes* with the relevant units of information.

A minimalist three-legged stool is still a place to sit.

9 Recognize, Diagnose, *and* Recover from Errors

Error messages should be expressed in plain language (no error codes), precisely indicate the problem, and constructively suggest a solution.

Wrong-way signs on the road remind drivers that they are heading in the wrong direction.

10 Help *and* Documentation

It’s best if the design *doesn’t need* any additional explanation. However, it may be necessary to provide documentation to help users complete their tasks.

Information kiosks at airports are easily recognizable and solve customers' problems in context and immediately.

https://media.nngroup.com/media/articles/attachments/Heuristic_Summary1-compressed.pdf

Nonfunctional requirements

- **U**sability
- **R**eliability
  - ➡ Robustness
  - ➡ Safety
  - ➡ Security
- **P**erformance
  - Response time
  - Throughput
  - Availability
  - Accuracy
- **S**upportability
  - Adaptability
  - Maintainability
  - Portability

Nonfunctional requirements definitions (2)

- **Robustness:** the ability of a system to maintain a function…
  - …when the user enters a wrong input
  - …when there are changes in the environment
  - **Example:** the system can tolerate temperatures up to 90°C

- **Safety:** protection against unwanted incidents (risk reduction)
  - Random incidents are usually unwanted, happening as a result of one or more coincidences
  - **Example:** car tires need at least 4mm profile, otherwise you are not allowed to drive

- **Security:** protection against intended incidents
  - Security is a means to achieve safety
  - **Examples:** prevention of attacks on the target, intrusion, deliberate and planned inclusion of viruses and trojan horses

Nonfunctional requirements

- **U**sability
- **R**eliability
  - Robustness
  - Safety
  - Security
- **P**erformance
  - Response time
  - Throughput
  - Availability
  - Accuracy
- **S**upportability
  - Adaptability
  - Maintainability
  - Portability

Nonfunctional requirements definitions (2)

- **Performance**
  - Number of simultaneous users supported
  - Amount of information handled
  - Number of transactions processed within certain time periods (average and peak workload)
    - **Example:** 95% of the transactions shall be processed in less than 1 second

- **Availability**
  - The ratio of the expected uptime of a system to the sum of the expected uptime and downtime
  - **Example:** the availability of the system is at least 99.7%
  - That means the downtime is at most 30 minutes per week (10050/10080)

Nonfunctional requirements

- **U**sability
- **R**eliability
  - Robustness
  - Safety
  - Security
- **P**erformance
  - Response time
  - Throughput
  - Availability
  - Accuracy
- **S**upportability
  - ➡ Adaptability
  - ➡ Maintainability
  - Portability

Nonfunctional requirements definitions (4)

- **Adaptability**
  - The ability of a system to adapt to changed circumstances
  - An adaptive system is an open system that is able to change its behavior when the environment changes
  - **Example**: the rain sensor for the wiper works correctly during different weather conditions (sun, fog, rain, snow, thunderstorm)

- **Maintainability**
  - The ease with which a system can be modified by a developer to
    - Correct defects (bugfixes)
    - Deal with new requirements
    - Cope with a changed environment
  - **Example**: car tires can be changed within 15min

Example: nonfunctional requirements for an **ATM machine**

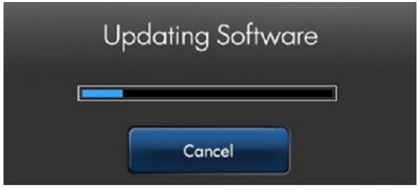
- **Usability**
  - Users interact with a touch screen
  - The touch screen shall be non-reflective
  - Text shall appear in letters at least 1cm high
- **Security:** the system shall under no circumstances leak PIN numbers or account information to unauthorized users
- **Performance:** each individual transaction shall take less than 10s

Constraints

- Also called pseudo requirements (regulation)
- Compliance with standards: report format, audit tracing, laws

- **Implementation requirements**
  - Usage of specific tools, programming languages, and frameworks
  - **However:** the development technology and methodology should not be constrained by the client. **Fight for it!**
  - **Example:** the system has to be developed in Java

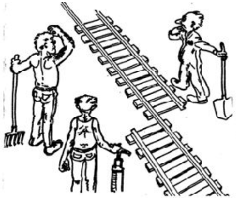
- **Operations requirements**
  - Administration and management of the system
  - **Example** for the ATM machine: remote system updates must be possible



Constraints

- **Packaging requirements**
  - Constraints on the actual delivery of the system
  - **Example:** “the software must be delivered as a mobile app into the iOS app store”

- **Interface requirements**
  - Constraints imposed by external systems
  - **Example:** “the system needs to read files in the Microsoft Word format”



- **Legal requirements**
  - The software system must comply with federal law regulations
  - **Example:** “government software must comply with Section 508 of the Rehabilitation Act of 1973, to make it accessible for people with disabilities”

Techniques to describe requirements

- **Goal:** bridging the conceptual gap between end users and developers

1. **Scenario:** describes the use of the system as a series of interactions between a specific end user and the system
   - A scenario is very specific and includes names, numbers and instances (**concrete**)
   - A scenario describes a single **instance** of a use case
   - “Object level”
2. **Use case:** describes a set of scenarios of a generic end user, called actor, interacting with the system
   - A use case is an **abstraction** and describes all possible instances (**generic**)
   - “Class level"
3. **User story:** describes a functional requirement from the perspective of an end user (use in agile projects, e.g., Scrum)

Outline

- Requirements elicitation
  - Types of requirements
  - ➡ **Scenarios**
  - Use cases
- Analysis
  - Decomposition
  - Object modeling
  - Stereotypes
  - Dynamic modeling

Scenarios

- **Scenario:** a concrete, focused, informal description of a feature of the system used by an actor
  - Central is the textual description (natural language) of the usage of a system: written from an end user's point of view
  - A scenario can also include video and pictures (storyboards)
  - It may also contain details about the workplace, social situations and resource constraints
- **Scenario-based design:** scenarios are used as the basis for the design of hypothetical interactions of the users with a new system

Scenario example (natural language)

Joe wants to take the subway from Munich Marienplatz to Garching Forschungszentrum and selects a single day ticket for Munich Zone M-2. The ticket machine displays a price of 10,10€. Joe inserts a 20€ bill. The ticket machine returns 9,90€ and prints the single day ticket. Joe takes the change of 9,90€ and the ticket and goes to the U6.

Use of scenarios in development activities

- Scenarios can be used in many activities during the software lifecycle
  - **Requirements elicitation:** as-is scenario, visionary scenario (未来)
  - **Client acceptance test:** evaluation scenario
  - **System deployment:** training scenario

Types of scenarios (1)

- **As-is scenario:** describes a **current situation** or the usage of an **existing system**
  - **Example:** Stephan and Bernd play chess via postcards
  - Commonly used in re-engineering projects

- **Visionary scenario:** describes a **future system**
  - **Example:** Jan and Evgeny play chess using a brainwave recognition machine
  - Used in all types of projects: greenfield, interface engineering and re-engineering
  - Usually not formulated by the user or developer alone

Types of scenarios (2)

- **Evaluation scenario:** description of a user task against which the system is to be **evaluated**
  - **Example:** The system must be demonstrated with two users (one novice, one expert) playing in a wrestling tournament

- **Training scenario:** A description of the step by step instructions that guide a novice user through a system
  - **Example:** How to play Tic Tac Toe with augmented reality glasses

Heuristics for finding scenarios (1)

Communication

- Don’t expect the client to be verbose if the system does not exist
  - Clients understand the application domain (problem domain), not the solution domain
- Don’t wait for information if the system exists
  - Don’t think: “What is obvious does not need to be said”
- Engage in a dialectic approach
  - Help the client to formulate the requirements
  - The client then helps you to understand the requirements
  - The requirements often evolve while these scenarios are being formulated
  - Usually the problem statement is a good start

Heuristics for finding scenarios (2)

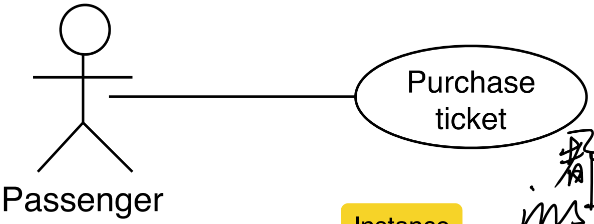
- Ask yourself or the client the following questions
  - What are the primary tasks of the system?
  - What data will the actor create, store, change, remove or add to the system?
  - What external changes does the system need to know about?
  - What changes or events will the actor need to know?
- However, don’t rely on questions and questionnaires alone
- Insist on task observation if the system already exists (interface engineering or re-engineering)
  - Speak to the end user, not just to the client
  - Expect resistance and try to overcome it

Scenario example (natural language)

Joe wants to take the subway from Munich Marienplatz to Garching Forschungszentrum and selects a single day ticket for Munich Zone M-2. The ticket machine displays a price of 10,10€. Joe inserts a 20€ bill. The ticket machine returns 9,90€ and prints the single day ticket. Joe takes the change of 9,90€ and the ticket and goes to the U6.

Scenario example (formalized)

1) **Name**
2) **Participating actors**
3) **Flow of events**



都是 instance

Instance

1) Name: Purchase ticket

2) Participating actors:

Joe: Passenger

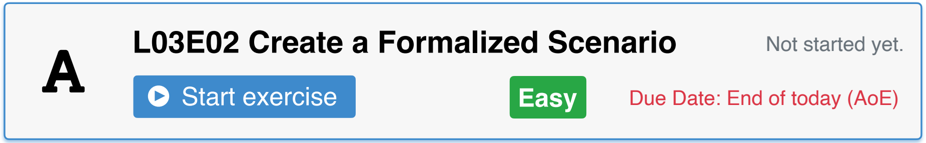
Instance

Actor step

System step (indented)

3) Flow of events

1. Joe wants to take the subway from Munich Marienplatz to Garching Forschungszentrum and selects a single day ticket for Munich Zone M-2
    2. The ticket machine displays a price of 10,10€
3. Joe inserts a 20€ bill
    4. The ticket machine returns 9,90€
    5. The ticket machine prints the single day ticket
6. Joe takes the change of 9,90€ and the ticket and goes to the U6



A

L03E02 Create a Formalized Scenario

Not started yet.

Due Date: End of today (AoE)

6 min

4 pts

- **Problem statement**
  - Read the natural language scenario on Artemis (based on the university app)
  - Create a formalized scenario (as shown on the previous slide)
- **Hints**
  - Define a meaningful name, the participating actors and the flow of events
  - Make sure to distinguish between actor and system steps
  - Make sure the scenario uses concrete names (**no abstractions**!)
  - Make sure to follow all conventions of a formalized scenario

Outline

- Requirements elicitation
  - Types of requirements
  - Scenarios
  - ➡ **Use cases**
- Analysis
  - Decomposition
  - Object modeling
  - Stereotypes
  - Dynamic modeling

After the scenario is formulated

- Find functions in the scenario where an actor interacts with the system
- Example in the **Purchase ticket** scenario
  Joe inserts money into the ticket machine

Here **insert money** would be a candidate for a use case

Another candidate would be **take change**

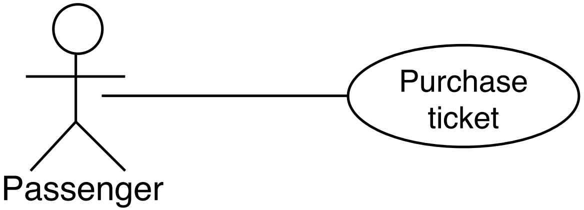
- Describe each of these use cases in more detail
  1. Name
  2. Participating actors
  3. Flow of events
  4. Entry condition
  5. Exit condition
  6. Special requirements

Review: scenario vs. use case

1. **Scenario:** describes the use of the system as a series of interactions between a specific end user and the system
    - A scenario is very specific and includes names, numbers and instances (**concrete**)
    - A scenario describes a single **instance** of a use case
    - “Object level”
2. **Use case:** describes a set of scenarios of a generic end user, called actor, interacting with the system
    - A use case is an **abstraction** and describes all possible instances (**generic**)
    - “Class level"

Textual use case description: example

1) Name
2) Participating actors
3) Flow of events
4) Entry conditions
5) Exit conditions
6) Special requirements



1) Name: Purchase ticket

2) Participating actors: Passenger Joe

Abstract version of the previous scenario

3) Flow of events

1. The passenger selects the number of zones to be traveled
    2. The ticket machine displays the amount due
3. The passenger inserts at least the amount due
    4. The ticket machine returns change
    5. The ticket machine issues the ticket

4) Entry conditions

- The passenger stands in front of the ticket machine & has a hand
- The passenger has sufficient money to purchase a ticket

5) Exit conditions

- The passenger has the ticket

6) Special requirements

- The ticket machine is connected to a power source

Review: what should NOT be in the requirements?

实现层面

- A description of the system structure
- The development methodology
- A description of the ~~development environment~~
- A specific implementation ~~language~~
- A specific implementation ~~technology~~

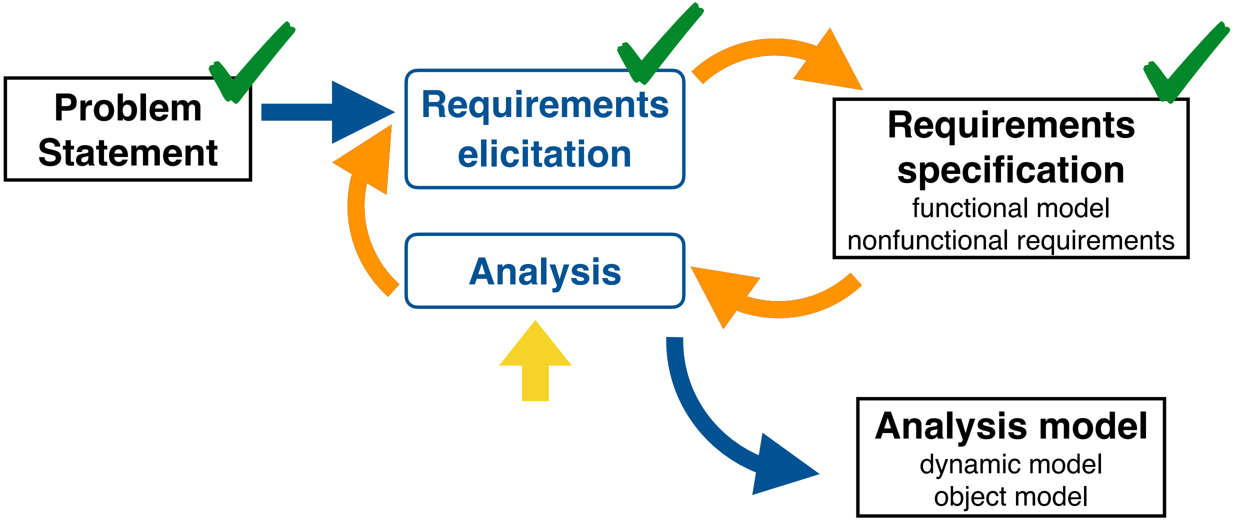
➡ It is desirable that none of the above are constrained by the client

Outline

- Requirements elicitation
  - Types of requirements
  - Scenarios
  - Use cases
- **Analysis**
  - **Decomposition**
  - Object modeling
  - Stereotypes
  - Dynamic modeling

Overview: requirements elicitation and analysis

- **Requirements elicitation:** describe the purpose of the system
- **Analysis:** create a model of the system, which is correct, complete, consistent, and verifiable



Analysis concepts

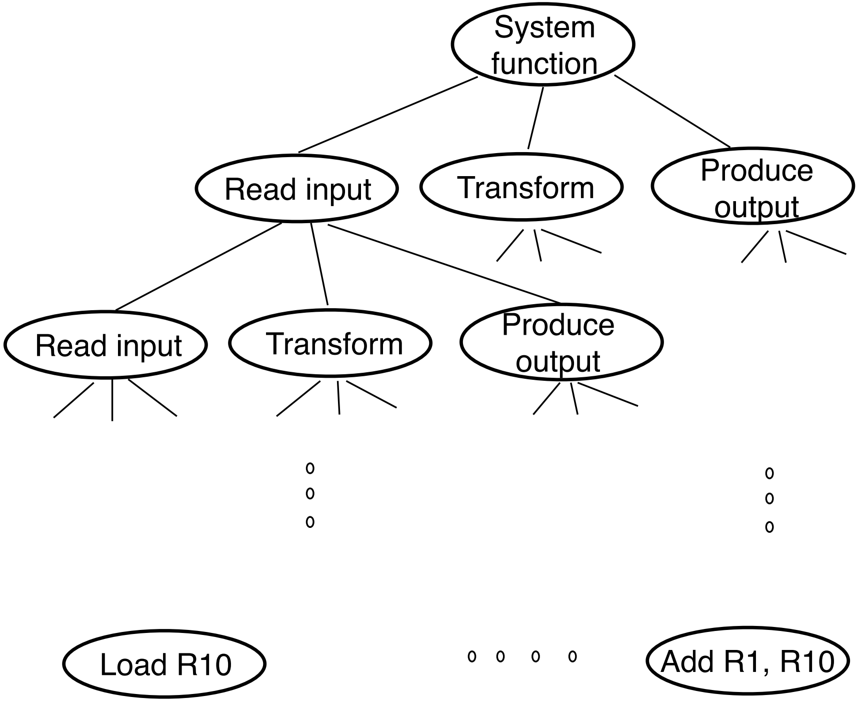
- ➡ **Decomposition:** a technique used to master complexity (divide and conquer)
- **Analysis model:** the object model and the dynamic model of a system to be developed
- **Generalization and specialization:** hierarchies can be detected in two different ways to adopt object oriented programming principles like inheritance/polymorphism and abstraction
- **Entity, boundary and control objects:** objects can be divided into three major categories describing their use inside the system

Functional vs. object oriented decomposition

- **Functional decomposition**
  - The system is decomposed into functions
  - Functions can be decomposed into smaller functions (C)
- **Object oriented decomposition**
  - The system is decomposed into classes (“objects”)
  - Classes can be decomposed into smaller classes (Java)

Which decomposition is the right one?

Functional decomposition



Top Level functions

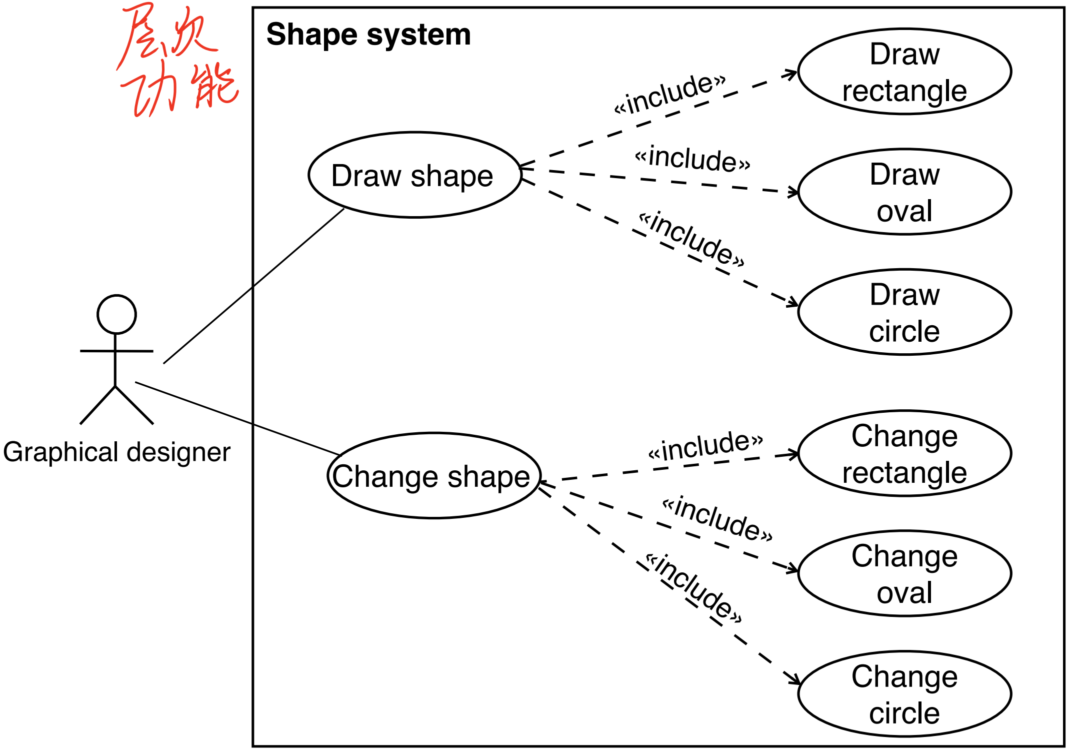
Level 1 functions

Level 2 functions

Machine instructions

Functional decomposition example: shape system

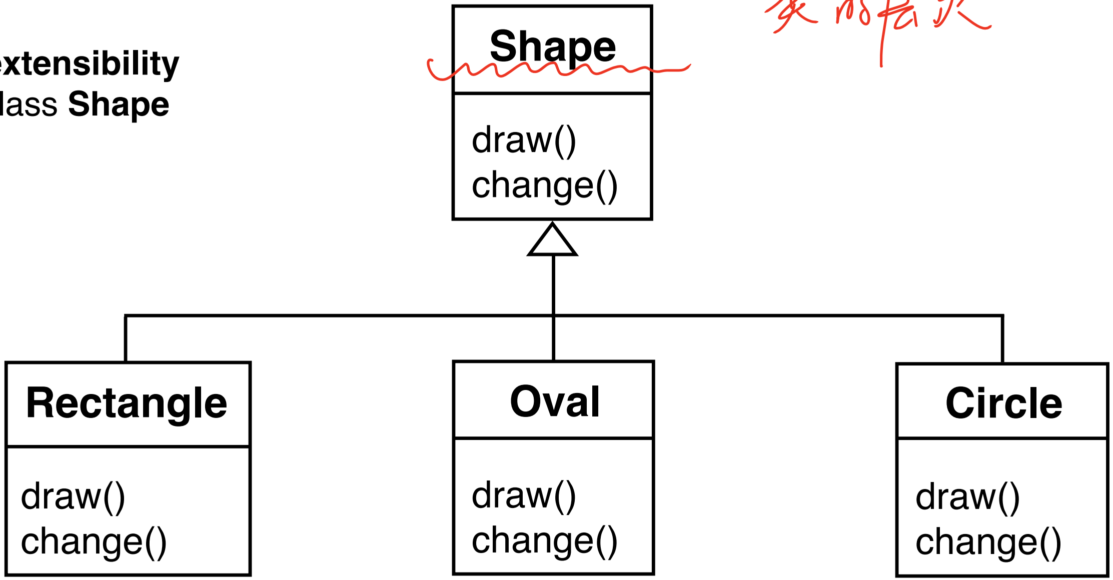
层次
功能



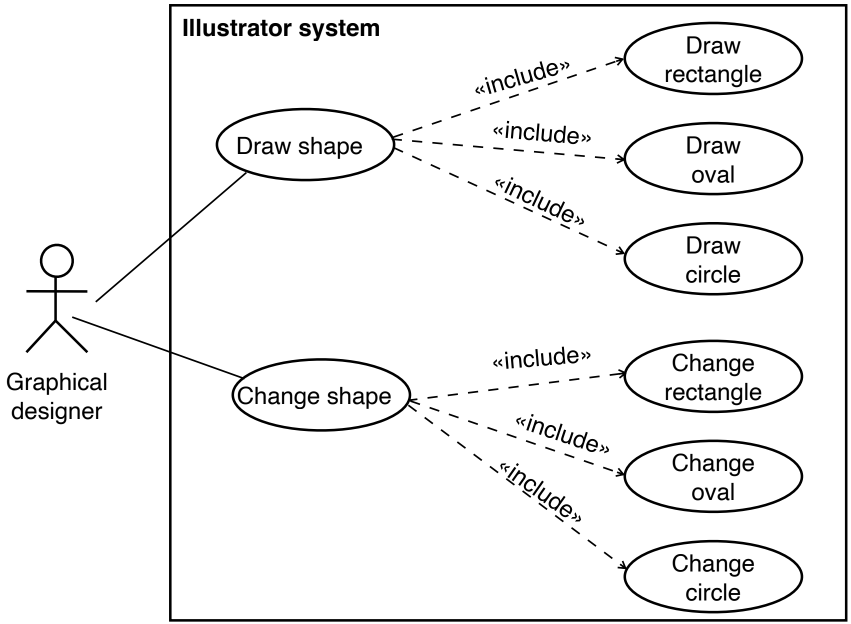
Object-oriented decomposition example: shape system

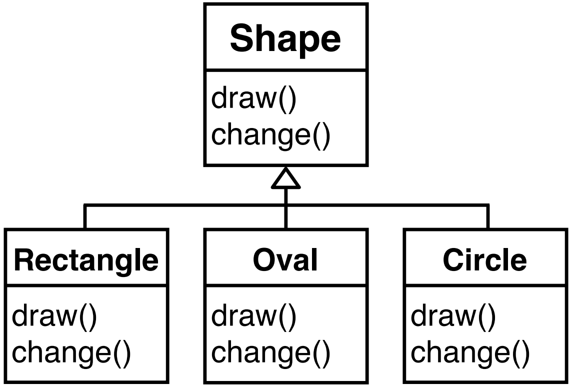
类的层次

Allows **extensibility** of the class **Shape**



Functional decomposition vs. object oriented decomposition





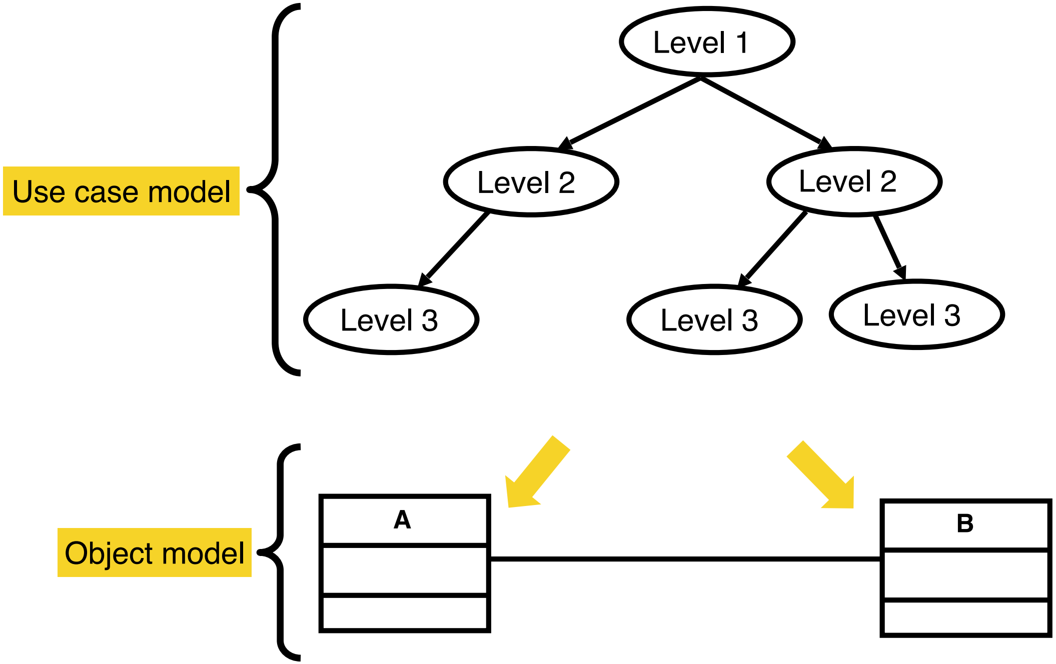
Functional decomposition

- **Problem:** the functionality is spread all over the system
  - Source code is hard to understand 不易读
  - Source code is complex and impossible to maintain 不易改
  - User interface is often awkward and non-intuitive
- **Consequence:** a maintainer must often understand the whole system before making a single change to the system

Model-based software engineering approach

1. Focus on the **functional requirements**
2. Find the corresponding **use cases** (events)
3. Identify the **participating objects**
4. Use these participating objects to create the first iteration of the **analysis object model**

Model-based software engineering approach



Top level use cases
("business processes")

Level 2 use cases

Level 3 use cases

A and B are called
participating objects

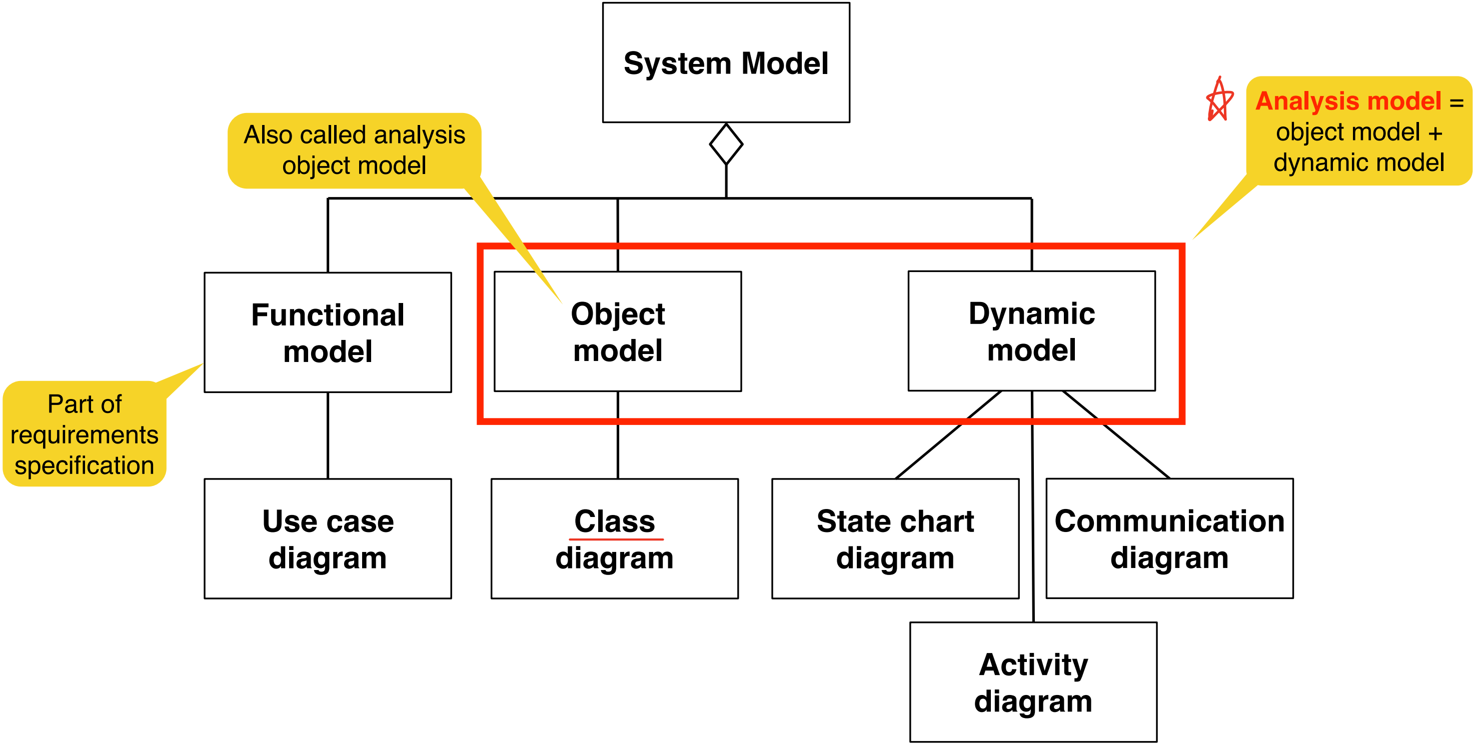
Outline

- Requirements elicitation
  - Types of requirements
  - Scenarios
  - Use cases
- Analysis
  - Decomposition
  - ➡ **Object modeling**
  - Stereotypes
  - Dynamic modeling

Analysis concepts

- ✓ **Decomposition:** a technique used to master complexity (divide and conquer)
- ➡ **Analysis model:** the object model and the dynamic model (behaviour) of a system to be developed
- **Generalization and specialization:** hierarchies can be detected in two different ways to adopt object oriented programming principles like inheritance/polymorphism and abstraction
- **Entity, boundary and control objects:** objects can be divided into three major categories describing their use inside the system

The **analysis model** is part of the system model



Also called analysis object model

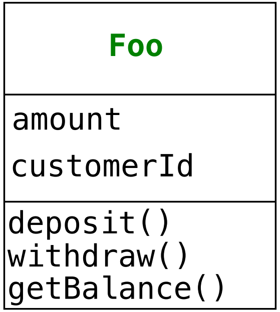
Analysis model = object model + dynamic model

Part of requirements specification

Object model

- **Purpose:** define the structure of the system by identifying objects, attributes, operations (methods) and associations

Let’s practice object modeling with class diagrams



Foo

amount

customerId

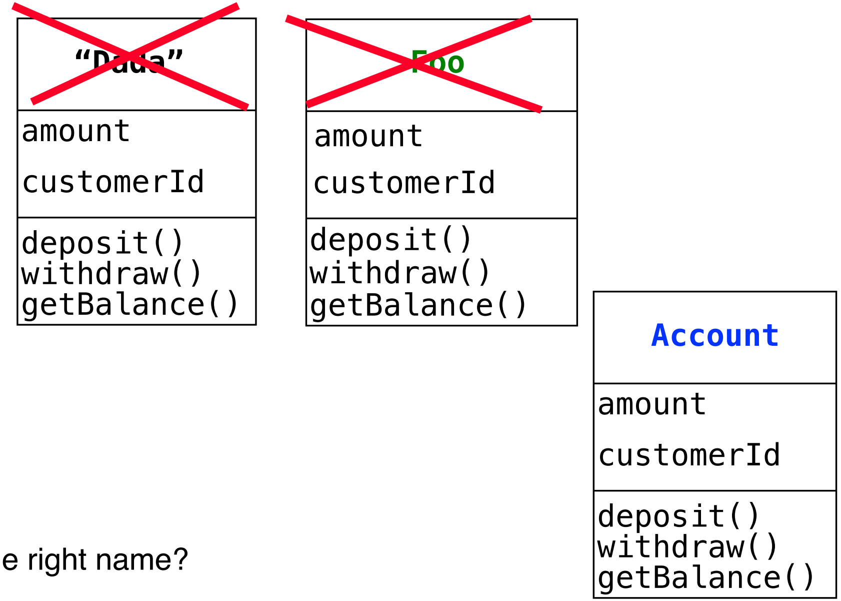
deposit()
withdraw()
getBalance()

First step: class identification

Identify the name of the class, find attributes and operations

Is Foo the right name?

Object modeling in practice: brainstorming



~~"Dada"~~

amount

customerId

deposit()
withdraw()
getBalance()

~~Foo~~

amount

customerId

deposit()
withdraw()
getBalance()

Account

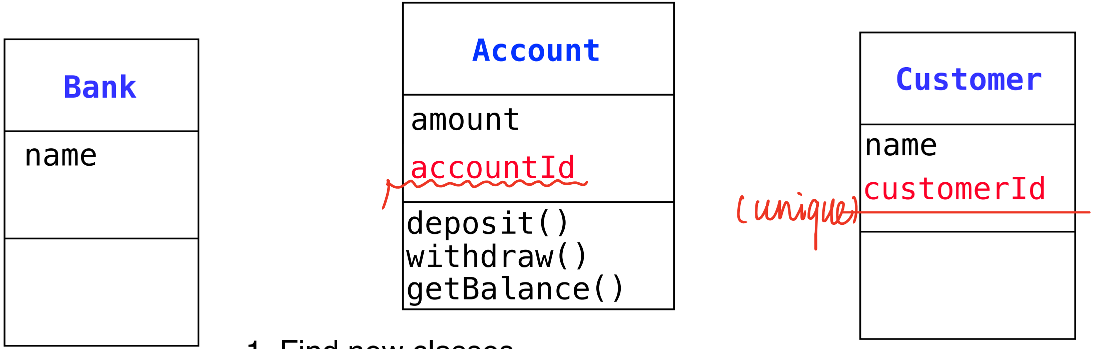
amount

customerId

deposit()
withdraw()
getBalance()

Is Foo the right name?

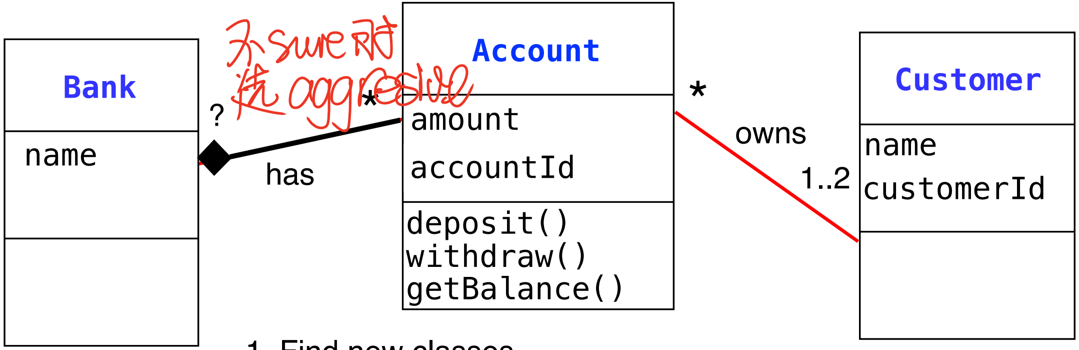
Object modeling in practice: more classes



1. Find new classes

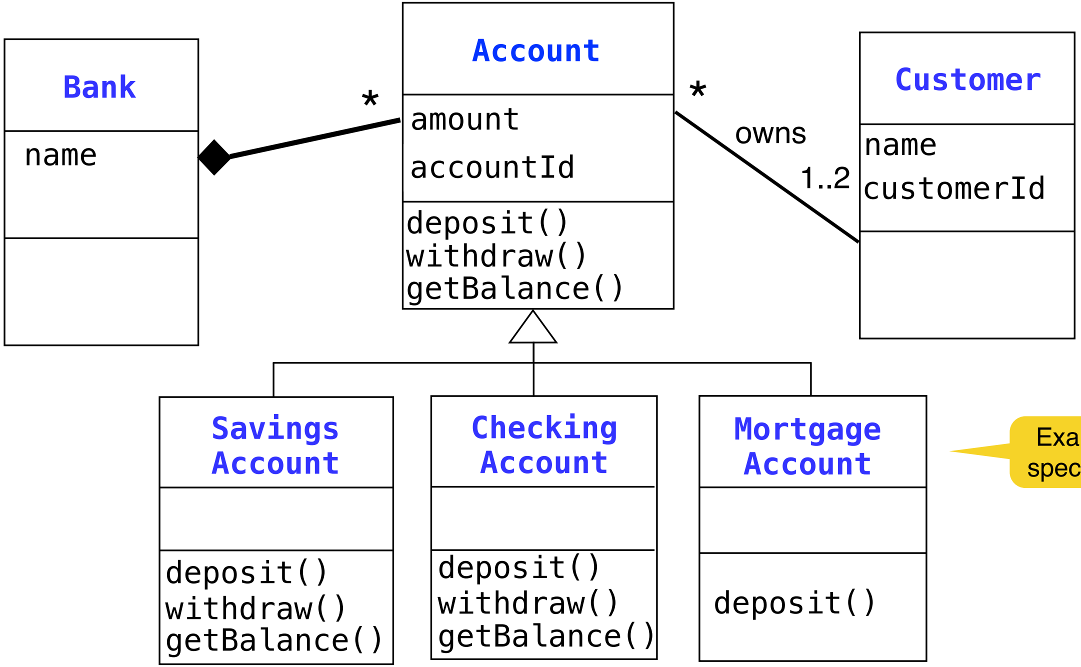
2. Review names, attributes and methods

Object modeling in practice: associations



1. Find new classes
2. Review names, attributes and methods
3. Find associations between classes
4. Label the generic associations
5. Determine the multiplicity of the associations
6. Review associations

Object modeling in practice: find taxonomies



Example for specialization

Generalization and specialization

1. Generalization

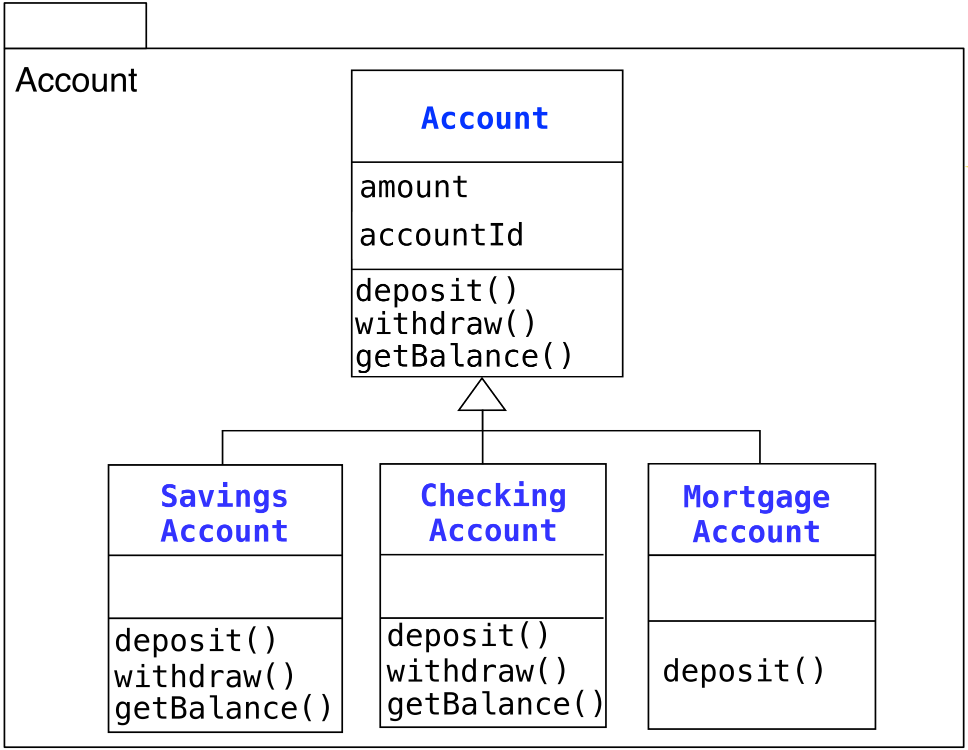
- Identifies abstract concepts from lower-level ones
- Identify common features among different concepts and create an abstract concept
- Common speech: “from low-level to high-level”, “from subclass to superclass”

2. Specialization

- Identifies specialized concepts from higher-level ones
- Identify special features and create more concrete concepts
- Common speech: “from high-level to low-level”, “from superclass to subclass”

➡ Both lead to taxonomies (**inheritance**) in the object model

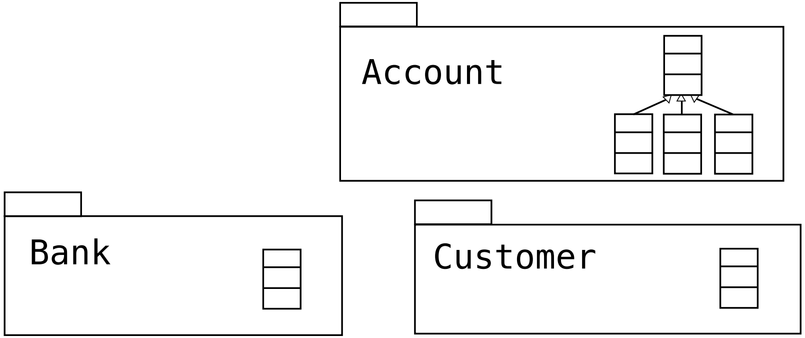
Object modeling in practice: simplify, organize



Organize taxonomies in a UML package

UML package notation

- Packages help you to organize UML models to increase their readability
- We can use the UML package mechanism to organize classes into subsystems



- Any complex system can be decomposed into subsystems, where each subsystem is modeled as a package

More details in **Lecture 04** on **System Design**

Outline

- Requirements elicitation
  - Types of requirements
  - Scenarios
  - Use cases
- Analysis
  - Decomposition
  - Object modeling
  - ➡ **Stereotypes**
  - Dynamic modeling

UML supports different types of objects (stereotypes)

memories [一直存在]

- **Entity objects**: represent the persistent information tracked by the system (application domain objects, also called “business objects”)
- **Boundary objects**: represent the interaction between the user and the system

  GUI
- **Control objects**: represent the control tasks to be performed by the system

  Algorithm

订阅数等

Analysis concepts

✓ **Decomposition:** a technique used to master complexity (divide and conquer)

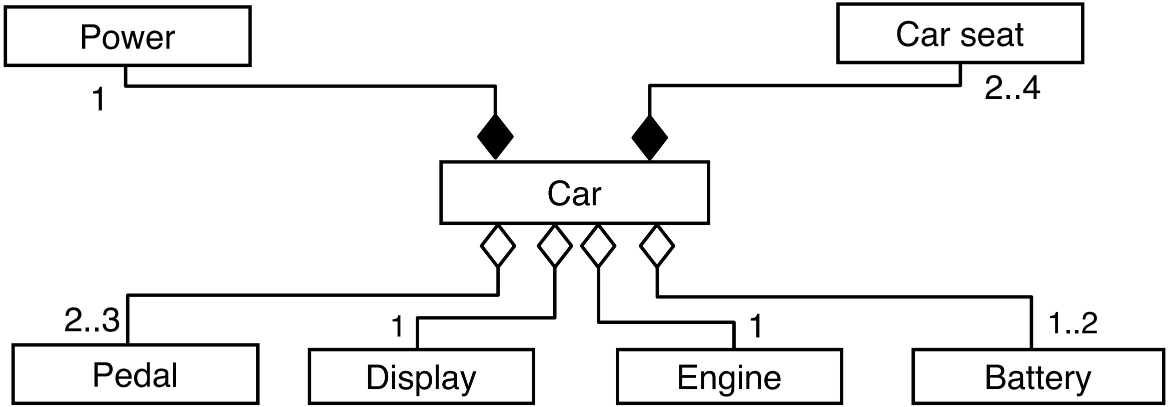
✓ **Analysis model:** the object model and the dynamic model of a system to be developed

✓ **Generalization and specialization:** hierarchies can be detected in two different ways to adopt object oriented programming principles like inheritance/polymorphism and abstraction

中间人 GUI

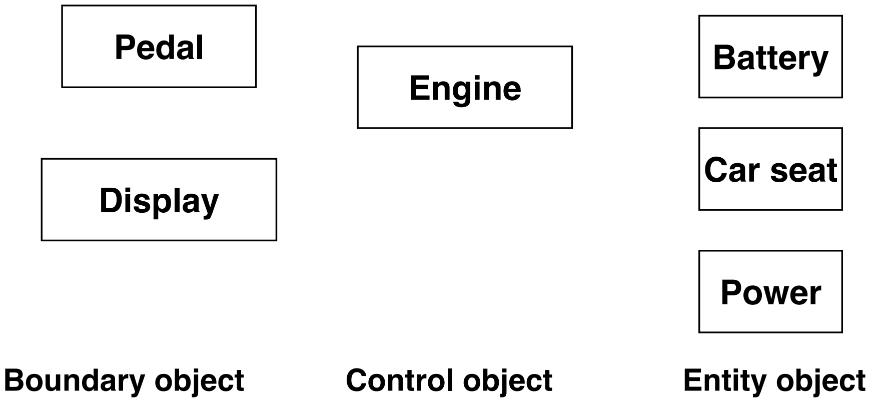
→ **Entity, boundary and control objects:** objects can be divided into three major categories describing their use inside the system (map)

Modeling a car



Example: objects of a car

To distinguish different object types in a class diagram we use stereotypes



Pedal

Display

Engine

Battery

Car seat

Power

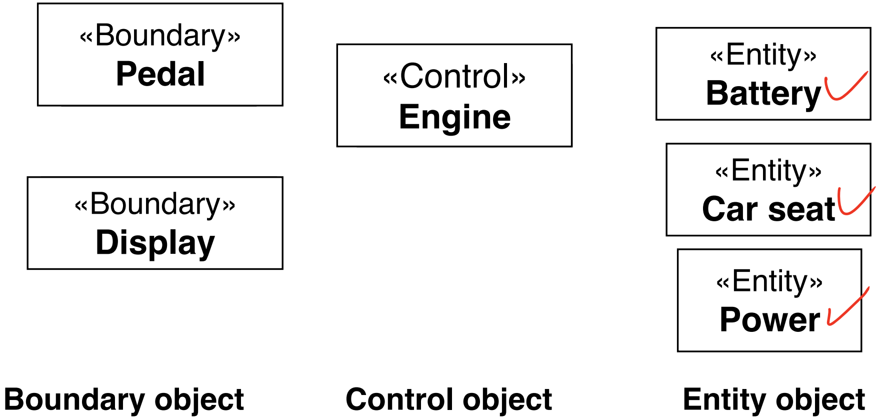
Boundary object

Control object

Entity object

Example: objects of a car

To distinguish different object types in a class diagram we use stereotypes



«Boundary»
Pedal

«Boundary»
Display

«Control»
Engine

«Entity»
Battery ✓

«Entity»
Car seat ✓

«Entity»
Power ✓

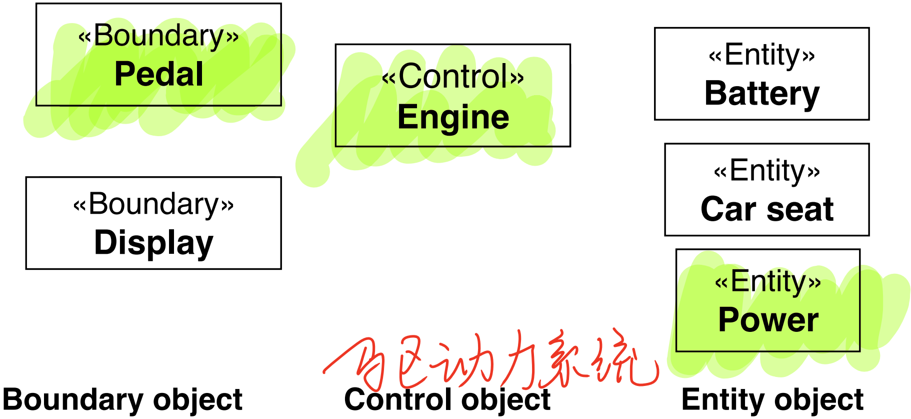
Boundary object

Control object

Entity object

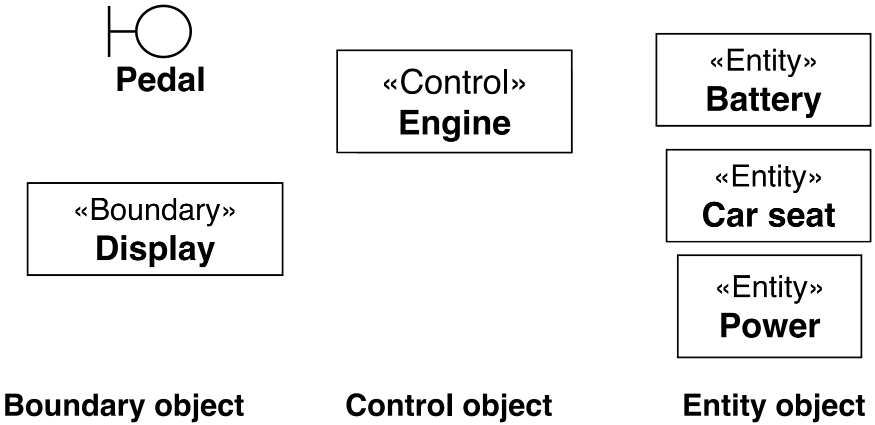
Graphical icons for object types

- We can also use **graphical icons** to identify a stereotype
- When the stereotype is applied to a UML model element, the icon is displayed beside or above the name



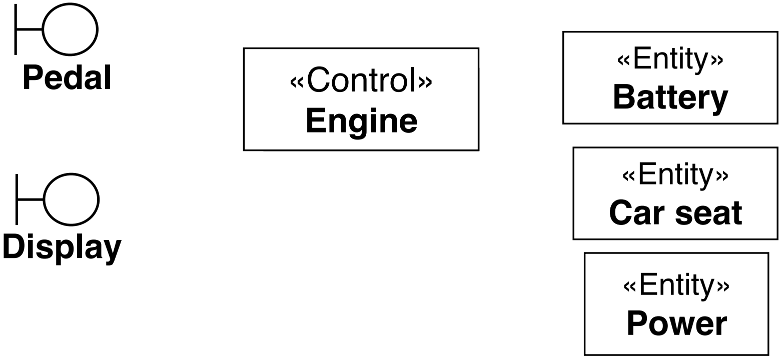
Graphical icons for object types

- We can also use **graphical icons** to identify a stereotype
- When the stereotype is applied to a UML model element, the icon is displayed beside or above the name



Graphical icons for object types

- We can also use **graphical icons** to identify a stereotype
- When the stereotype is applied to a UML model element, the icon is displayed beside or above the name



Pedal

Display

«Control»
Engine

«Entity»
Battery

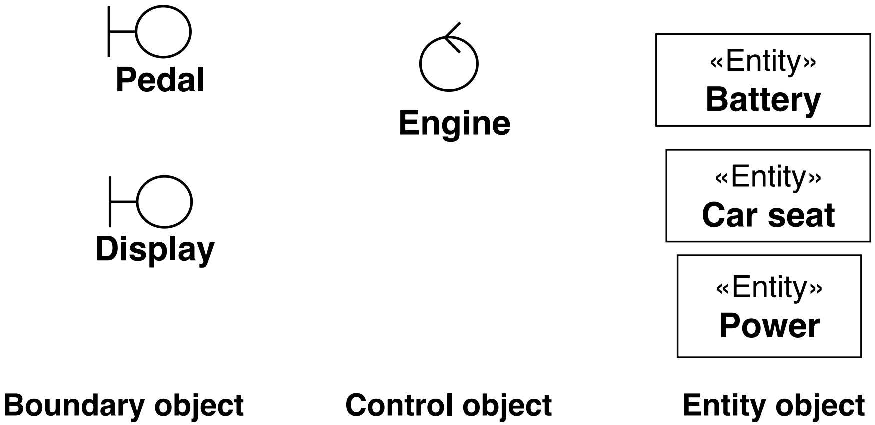
«Entity»
Car seat

«Entity»
Power

Boundary object **Control object** **Entity object**

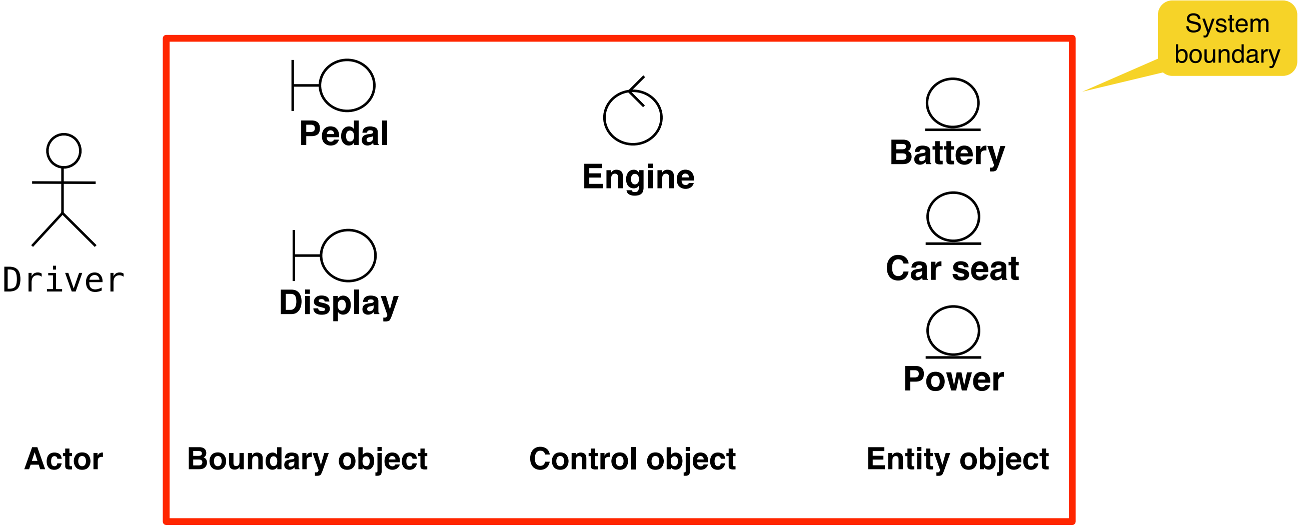
Graphical icons for object types

- We can also use **graphical icons** to identify a stereotype
- When the stereotype is applied to a UML model element, the icon is displayed beside or above the name



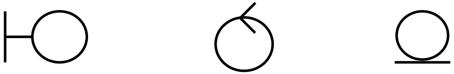
Graphical icons for object types

- We can also use **graphical icons** to identify a stereotype
- When the stereotype is applied to a UML model element, the icon is displayed beside or above the name



Stereotypes

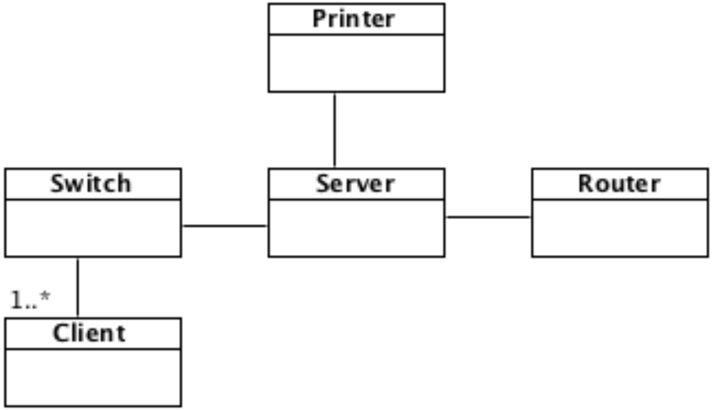
- UML is an **extensible** language
  - **Stereotypes** allow you to extend the vocabulary of the UML so that you can create new model elements derived from existing ones
- Stereotypes can be represented textually as well as with graphical icons
  - Class diagrams: «boundary», «control», «entity»



- Other stereotypes
  - Use case relationships: «extend», «include»
  - Subsystem interface: «interface»
  - Stereotypes for classifying method behavior: «constructor», «getter», «setter»

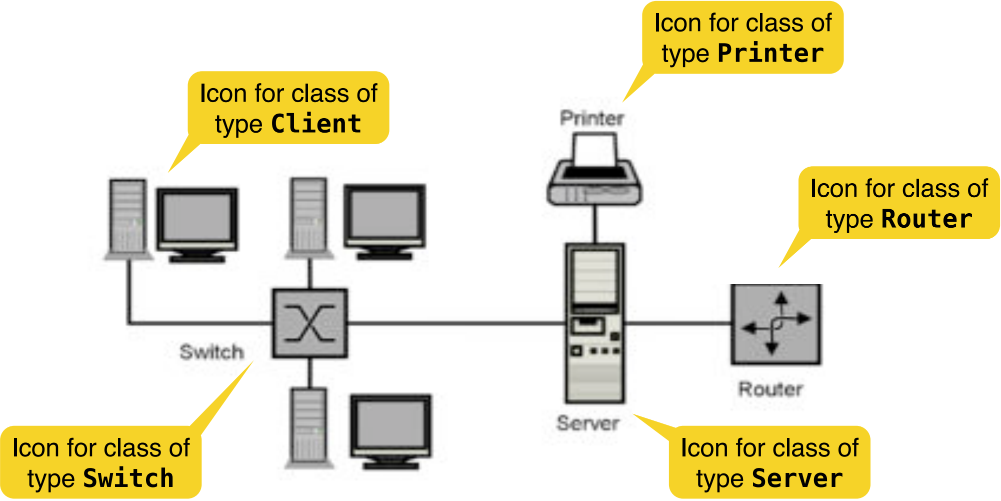
Another example for graphical icons: model of a network

- When modeling a network, we use a class diagram for the components of the network: Client, Switch, Server, Printer and Router



Another example for graphical icons: model of a network

- When modeling a network, we use a class diagram for the components of the network: **`Client`**, **`Switch`**, **`Server`**, **`Printer`** and **`Router`**



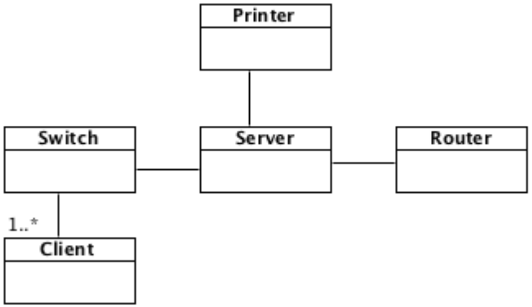
Icon for class of type **`Printer`**

Icon for class of type **`Client`**

Icon for class of type **`Router`**

Icon for class of type **`Switch`**

Icon for class of type **`Server`**

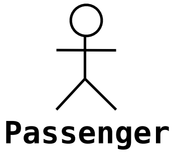


Advantages and disadvantages of stereotype graphics

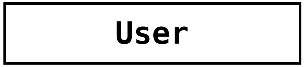
- **Advantages**
  - Easier to understand, especially if standardized in the application domain
  - Increase the readability of the diagram, especially if the client is not trained in UML
- **Disadvantages**
  - Harder to understand if developers are unfamiliar with the symbols
  - Additional symbols add to the burden of learning to read the diagrams

Important distinction: actor vs. class vs. object

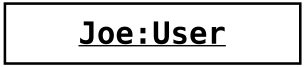
- **Actor**
  - Any entity outside the system which is interacting with the system (e.g. “Passenger”, “GPS satellite”)



- **Class**
  - A concept from the application domain or in the solution domain
  - Classes are part of the system model (e.g. “User”, “Ticket machine”, “Server”)



- **Object**
  - A specific instance of a class (e.g. “Joe, the passenger who is purchasing a ticket from the ticket machine”)



Purpose of modeling: why all these models?

- ✓ Functional model: describes the functionality of the system
  - Identification of functional requirements
  - Delivery of new operations (methods) for the object model

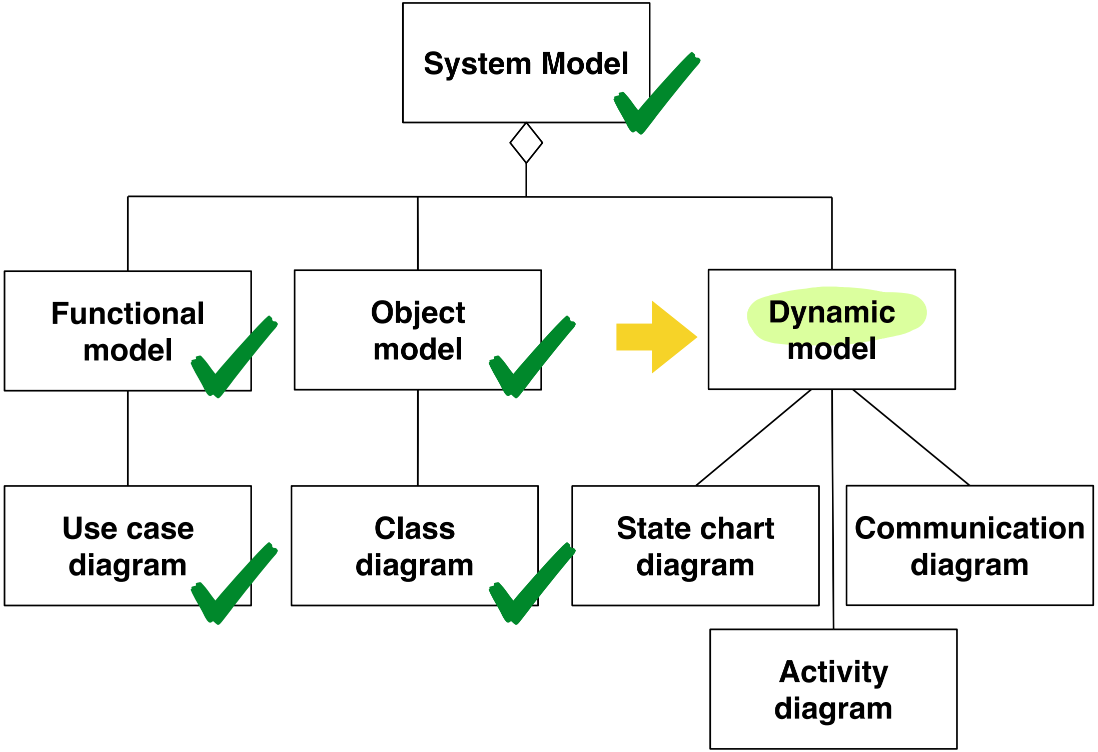
Using use cases and scenarios

- ✓ Object model: describes the structure of the system
  - Identify the structure of the system

Using classes, attributes, operations and associations

- Dynamic model: describes the dynamic behavior of the system

Modeling a system model with UML



Outline

- Requirements elicitation
  - Types of requirements
  - Scenarios
  - Use cases
- Analysis
  - Decomposition
  - Object modeling
  - Stereotypes
  - ➡ **Dynamic modeling**

Dynamic modeling

- Model the components of the system that have interesting dynamic behavior
- **State chart diagrams:** model the states of one class with interesting dynamic behavior
- **Activity diagrams:** model workflows within use cases and complex workflows in operations of objects (e.g. algorithms)
- **Communication diagrams:** model the interaction between multiple objects
  - Identify new operations for the object model

More details in **Lecture 9** on **Software Lifecycle Modeling**

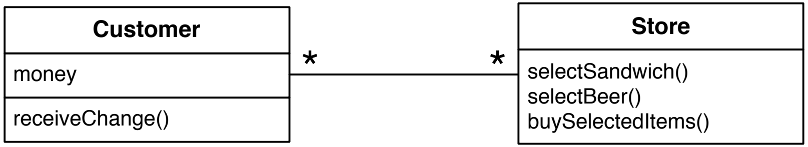
UML communication diagrams

- Visualize the **interactions** between objects as a flow of messages (i.e. events or calls of operations)
- Describe the **static structure** **and** the **dynamic behavior** of a system
  - Reuse the layout of classes and associations in the UML class diagram
  - The dynamic behavior is e.g. obtained from scenarios and use case descriptions
- **Messages** between objects are labeled with a number and placed next to the communication link
  - No distinction between different associations
  - Inheritance taxonomies are collapsed into single objects

UML communication diagrams

- Describe the interaction between **different objects** by using method invocation

- Class diagram **example**

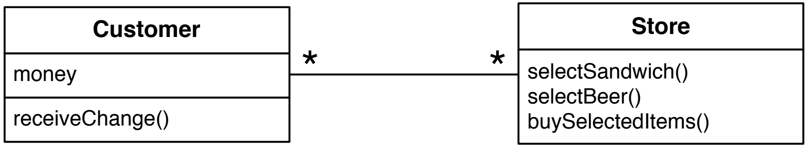


- Class diagrams contain associations, but they do not show the communication (message flow) between the objects

UML communication diagrams

- Describe the interaction between **different objects** by using method invocation

- Class diagram **example**

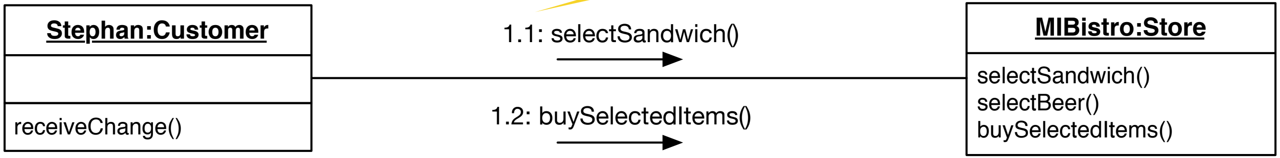


- Three different types of messages in **communication diagrams**
  1. **Sequential** messages
  2. **Conditional** messages
  3. **Concurrent** messages

Message examples

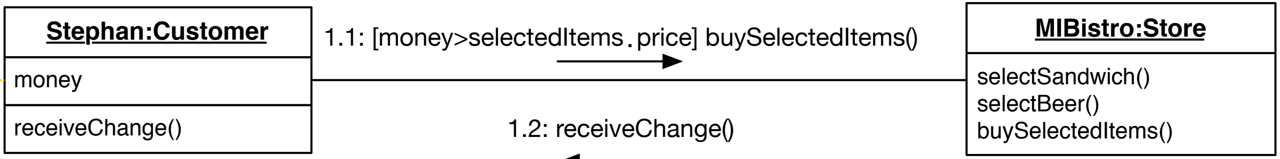
1. Sequential messages

1.1 and **1.2** express **sequential** messages (the order is important)



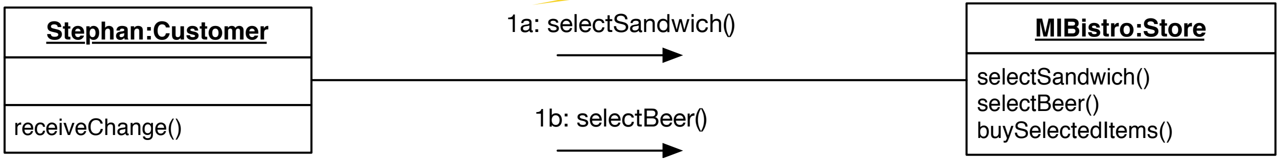
2. Conditional messages

You can add attributes, if they are used in the messages



3. Concurrent messages

1a and **1b** express **concurrent** messages (the order is **not** important)



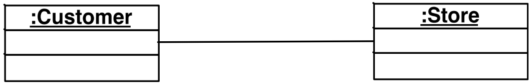
Recipe: from class diagrams to communication diagrams

Actions

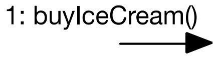
1. Take all the steps from the event flow of a use case

"Customer buys ice cream from the Store."

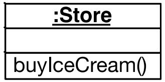
2. Instantiate the participating objects



3. Number the messages from each of the steps of the event flow

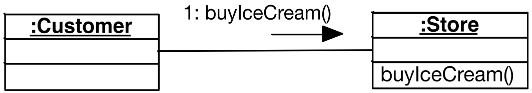


4. Is there a corresponding method in the receiver of the message?



No: refine your class diagram by adding a public method to the receiver

5. Draw the message from sender to receiver



Example of a visionary scenario

1) Name: Pass EIST exam

2) Participating actors
Peter: Student,
Stephan: Lecturer

3) Flow of events

1. Peter enrolls in the EIST course and starts the course
2. Stephan prepares the final exam
3. Peter takes the final exam
4. Stephan corrects the final exam
5. If Peter has passed the exam, he receives a certificate
6. Peter evaluates Stephan at the end

L03E03 Model a Communication Diagram

Not started yet.

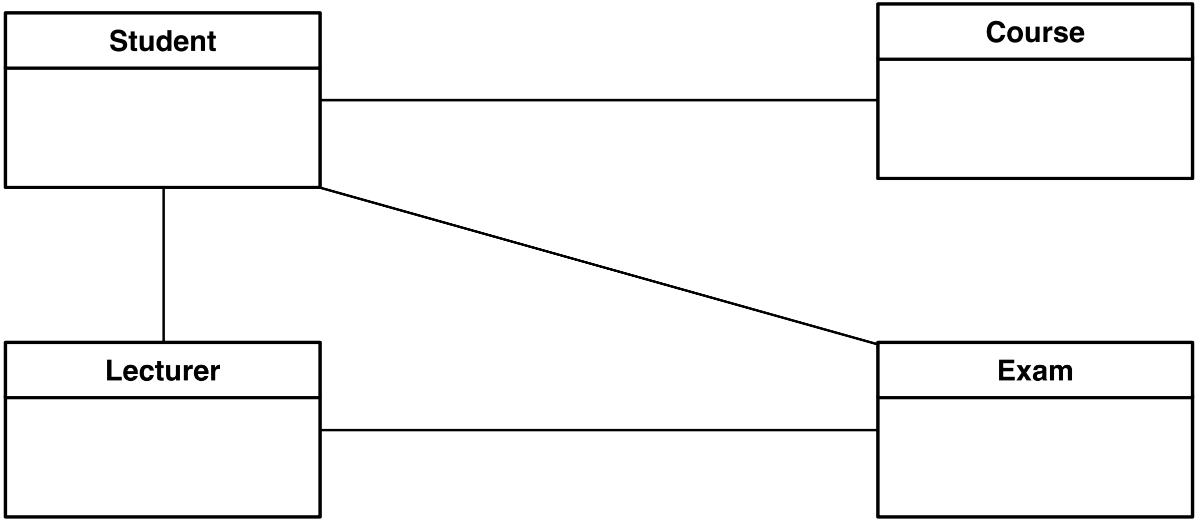
Due Date: End of today (AoE)

10 min

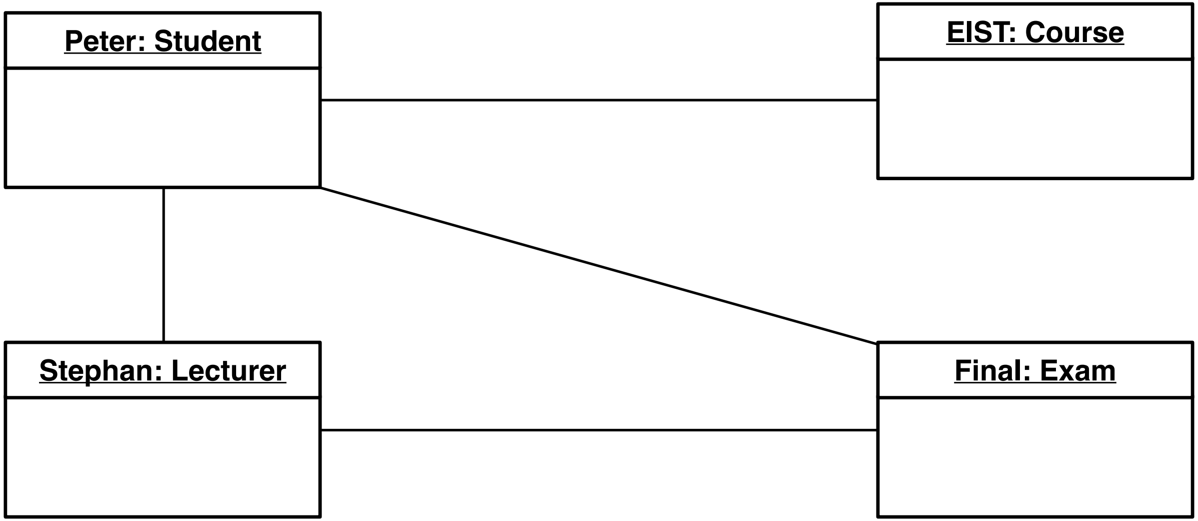
6 pts

- **Problem statement**
  - Model a communication diagram for the scenario **Pass EIST exam** (previous slide)
  - Add the objects and associations from the previous slide
  - Add the messages defined in the flow of events of the visionary scenario
  - For each message, add the corresponding method in the receiver object

Object model: identify objects



Object model: instantiate objects



Homework

- **H03E01** Bumpers Nonfunctional Requirements (text exercise)
- **H03E02** Visionary Scenario for Bumpers (text exercise)
- **H03E03** Communication Diagram (modeling exercise)
- Read more about **usability** and **Jakob Nielsen's 10 usability heuristics** (see readings)

→ Due until 1h before the **next lecture**

Summary

- **Requirements elicitation** and **analysis** as software development activities
- **Functional** requirements vs. **nonfunctional** requirements (URPS and constraints)
- Scenario-based design focuses on **scenarios** to describe the interaction between users and the system
- Deeper look into UML class diagrams, **use cases** and **communication diagrams**
- Practice **object modeling** and **dynamic modeling**
- UML is an extensible language: predefined types and **stereotypes**

Literature

- Barry Boehm: Software Engineering Economics. Englewood Cliffs, Prentice-Hall, 1981
- David Parnas: A rational design process: How and why to fake it, IEEE Trans. on Software Engineering, Vol 12(2), February 1986
- John M. Carrol, Scenario-Based Design: Envisioning Work and Technology in System Development, John Wiley, 1995
- Usability Engineering: Scenario-Based Development of Human Computer Interaction, Morgan Kaufman, 2001
- Jakob Nielsen: Usability 101: Introduction to Usability - https://www.nngroup.com/articles/usability-101-introduction-to-usability
- Jakob Nielsen: 10 Usability Heuristics for User Interface Design - https://www.nngroup.com/articles/ten-usability-heuristics
- Mike Cohn, User Stories Applied: For Agile Software Development, Addison-Wesley, 2004