Software Construction and User Interfaces (SE/ComS 319)

Ali Jannesari

Jinu Susan Kabala

Department of Computer Science

Iowa State University, Fall 2021

SYSTEM MODELING

Outline

- Introduction to requirement engineering
- Use case model
    - Actors, relationships between use cases
- Use case text
- Activity diagrams
- Objectives of system modeling
- Review of object orientation

Introduction to

REQUIREMENT ENGINEERING

What are requirements?

- The services the software should provide
- Functional requirements: specify the functions of the software
  - Describe the interactions between the system and the system environment, regardless of implementation (input, output, ...).
    - "A police officer must be able to request resources."
- Non-functional requirements (quality requirements): specify how well the software performs its functions
  - Aspects that are not directly related to the functional behavior of the system.
    - "The response time must be less than a second."
- The constraints the software should follow
  - Constraints are specified by the customer or the environment.
    - "The implementation must be done in Java.”

Functional vs. non-functional requirements

Functional

- Describe user tasks that the system must support
- Formulated as actions
  - "Notify interested parties”
  - "Create a new table"

Non-functional

- Describe properties of the system or the domain
- Formulated as **constraints** or **assertions**
  - "Every user input needs to be recognized in less than a second"
  - "A system crash should not lead to data loss"

Types of non-functional requirements

Quality requirements

- Usability
- Reliability
- Robustness
- Safety (Security)
- Performance
- Response time
- Scalability
- Throughput
- Availability
- Maintainability
- Customization
- Extensibility

Constraints

- Implementation
- Interfaces
- Operating environment
- Delivery
- Legal issues
  - Licenses
  - Certificates
  - Data protection and privacy

Non-functional requirements – Examples

Try to use quantitatively measurable metrics to describe them

- **Usability**: How easy the actors can perform a function with a system
  - Usability is one of the most commonly used terms (a very important non-functional requirement of **UI applications**)
  - Usability must be measurable (otherwise it is just marketing)
    - "Number of steps to place an internet order in the browser.”
- **Robustness**: The ability of a system to continue the function when:
  - The user makes an incorrect entry (incorrect operation)
  - Operating conditions are not met
    - "Operating temperature -10 ° F to + 50 ° F.”
    - "Maximum number of requests is 2000 / s."

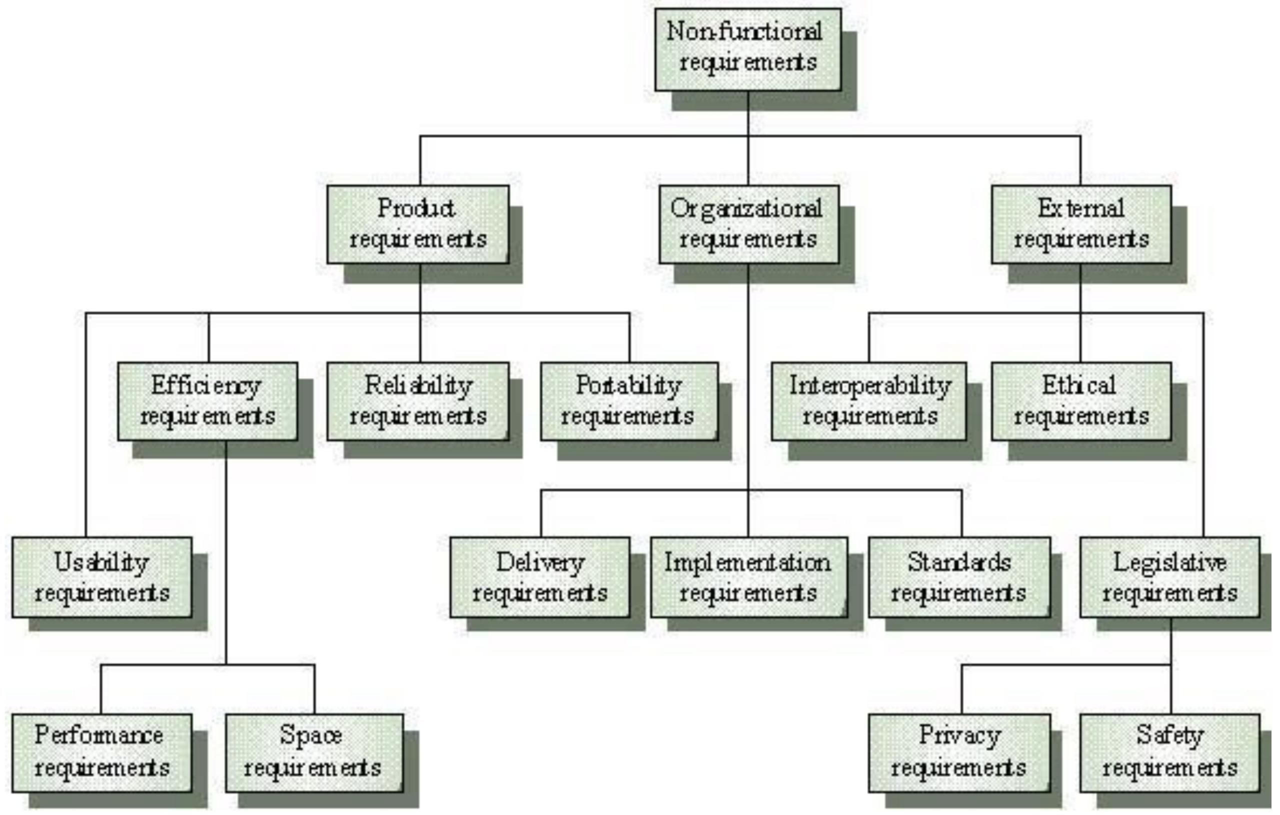
Non-functional requirements – Examples

- **Availability**
  - The ratio of trouble-free operating time to total time
  - Available time / (service available time + service down time)
    - "The system is not available for less than 5 minutes per week"

- **Performance**
  - Time complexity, space complexity, scalability, throughput, latency, space

- **Reliability**
  - How likely the service will go down at time T

Non-functional requirements



What is requirement engineering?

- Requirement Engineering (RE) is the process of
  - Finding out,
  - Analyzing,
  - Documenting, and
  - Validation of the requirements.

Requirements = Stories in TDD process!

How to find out and represent the requirements?

- Requirements elicitation
  - Questionnaires
  - Interviews
  - Task analysis
  - **Scenarios**
    - A scenario is s the description of an event or sequence of actions and events
  - **Use case diagrams** (system modeling)

Story Cards for representing requirements in TDD process!

Scenarios

- A scenario
  - Is the description of an event or sequence of actions and events
  - Is the description of how to use a textual system from **a user's perspective (a story in XP/TDD)**.
  - Can contain text, images, videos, and schedules, as well as details about the workplace, the social environment, and resource constraints.

Called as a Story in TDD process!

Scenarios – Example ”Burning warehouse”

- While **Bob** drives his main car along the main road, he notices smoke rising from a warehouse. His colleague, Alice, **reports the emergency** from the vehicle.
- **Alice** enters the address of the warehouse into her mobile computer, a brief description of the location (e.g., north-west corner) and a priority.
- She confirms her entry and waits for a confirmation.
- **John**, the **dispatcher** at the control room, is alerted to the emergency by a beep on his computer. He analyzes the information Alice sent him and confirms the message. He alerts the fire department and passes the expected time of arrival to Alice.
- Alice receives the confirmation and expected arrival time.

Scenarios – Example ”Burning warehouse”

Remarks on the given scenario:

- It is a special scenario
    - It describes a single instance of reporting a fire.
    - It does not describe all possible situations in which a fire can be reported.
- Participating actors:
    - Policeman, Dispatcher (Bob, Alice, John)

What is not a requirement?

- System structure, implementation details
- Development methods
- Development environment
- Mostly not required: programming language, reusability
- It is clear that none of the above points are customer constraints

Validation of requirements

- The validation of requirements is a **quality assurance** step that is normally performed during requirement engineering
- **Correctness**
  - The requirements correctly represent the customer's point of view.
- **Completeness**
  - All situations in which the system can be used are described, including errors and operating errors.
- **Consistency**
  - No functional or non-functional requirements contradict each other.
- **Uniqueness**
  - Requirements can only be interpreted in one way

Validation of requirements

- **Feasibility**
  - Requirements can be met and delivered (in XP/TDD as an iterative agile process: stories will be break down for weekly iterations).
- **Traceability**
  - It will be possible to associate each system function with one or a set of requirements that need the function.
- Challenges with the validation:
  - **Requirements change** during the planning phase (for instance after each iteration in XP/TDD)
  - Inconsistencies can occur with every change
  - **Tool support is required!**

Who will need the requirement document?

- Users
- Design team
- Developers
- Testing team

Story Cards as requirement documents in TDD/XP for the whole team!

Summary

- Introduction to requirements engineering
  - Functional vs. nonfunctional requirements
  - Requirements elicitation
  - Scenarios (stories)
  - Validation of requirements, etc.

SYSTEM MODELING

Use case diagrams – UML

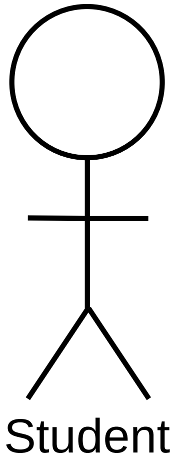
- Use case diagrams are used during the requirement engineering to represent the externally visible behavior of the system.
- An actor specifies a role of a user or other system that interacts with the system we are analyzing.
- A use case represents a class of functions offered by the system.
- A use case model is the set of all use cases that describe the entire functionality of the system.
- A use case diagram includes
    - Actors, use cases, associations, system boundary

Actors

- An actor is a model for an external entity that interacts with the system:
  - Administrator, end user, environment, external systems, ...
- An actor has a unique name and optionally a description.
- Example:
  - **Student**: A person who is studying or training at a university or college
  - **Random number generator**

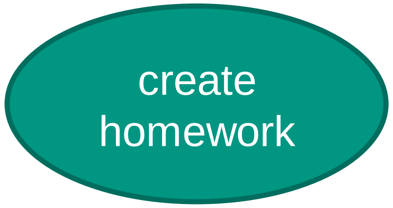
optional description

name

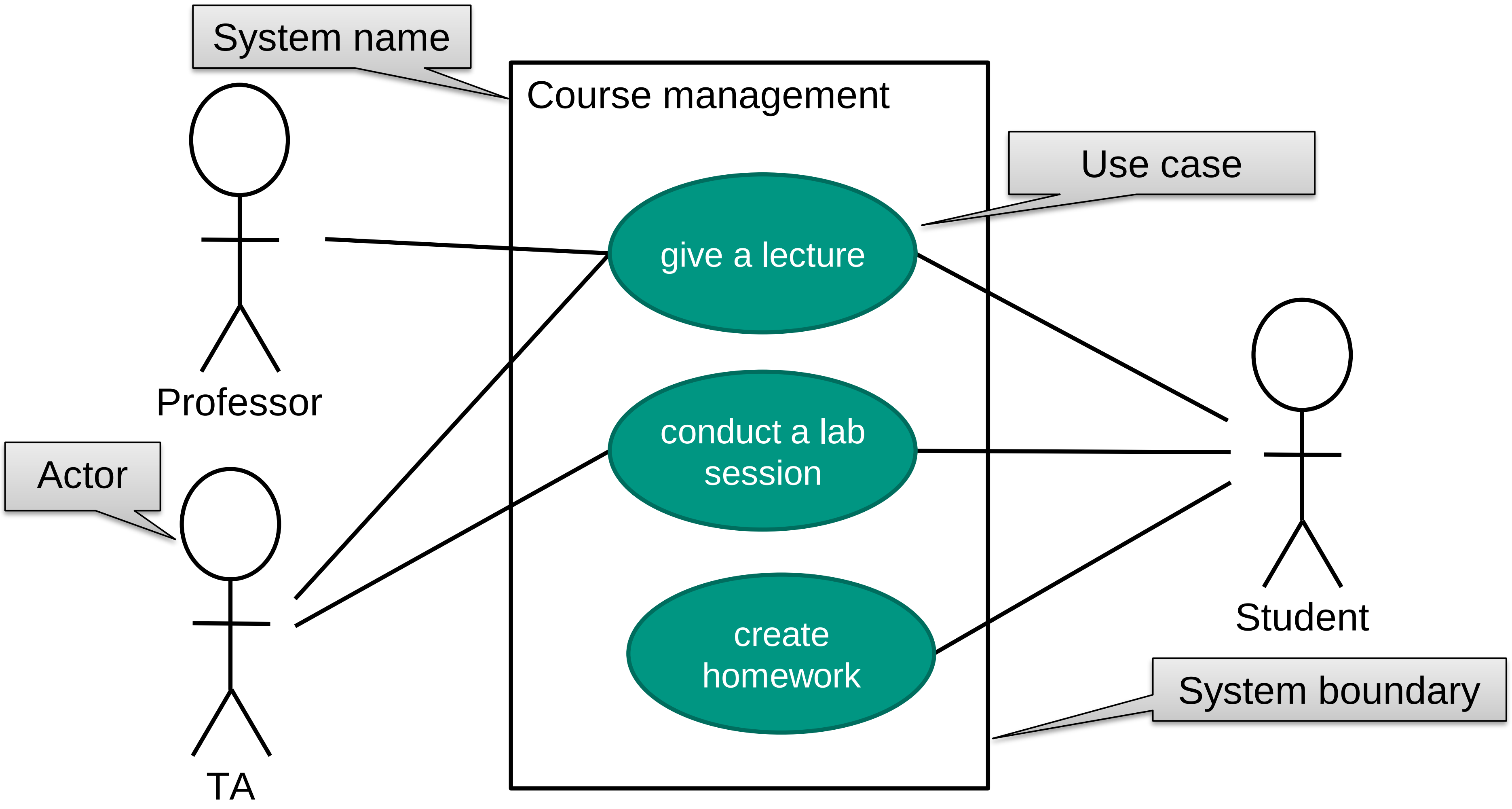


Use cases – How to describe them?

- Use cases can be described with text, with a focus on the interaction between actor and system.
- The description of a use case with text consists of 6 parts:
  - Unique name
  - Participating actors
  - Input actions
  - Output actions
  - Event flow
  - Special requirements
- They (use cases) also can be described with activity diagram



Use case diagram – Example



How do you find use cases?

- Choose a limited, vertical section of the system (for example a scenario)
  - Discuss this in detail with the user to determine the preferred interaction style of the user
- Choose a horizontal section (for example, many scenarios) to define a larger area of the system
  - Discuss the area with the user
- Use meaningful prototypes (mock-ups) for visual support
- Figure out what the user is doing
  - Observation (good)
  - Questioning (often inaccurate answers)

Use case from scenario

- Find all use cases in the example scenario that all instances specify how to report a fire.
  - Example: "**Report Emergency**" in the first paragraph of the scenarios is a candidate for **a use case**
- Describe each of these use cases **as accurately as possible**:
  - Participating actors
  - Describe their input actions
  - Describe their event flow
  - Describe their output actions
  - Describe exceptions
  - Describe non-functional requirements

Use case: ”Report emergency” (1)

- Name of use case: Report emergency
- Participating actors:
  - Policeman (Bob and Alice in this scenario)
  - Dispatcher (John in this scenario)
- Exceptions:
  - The police officer will be notified immediately if the connection between the terminal and the headquarters breaks.
  - The dispatcher is notified immediately when the connection between a police officer and the headquarters breaks.

Use case: ”Report emergency” (2)

- Event flow:
  - The policeman activates the "Report Emergency" feature on his terminal. **FRIEND system** [an external system] responds by displaying a form.
  - The police officer completes the form by entering the emergency level, the type of assignment, the address, and a brief description of the situation. The policeman also describes a reaction to the emergency situation.
  - The dispatcher creates an incident in the database by calling the "Open Incident" use case. He chooses a reaction and confirms the message.
  - The policeman receives the confirmation and chooses the reaction.
- Nonfunctional requirements:
  - The police report will be confirmed within 30 seconds. The answer arrives at the police no later than 30 seconds after being sent by the dispatcher.

Use case: ”Request resources” (1)

- Actors:
  - Operation Manager: The person responsible for the deployment
  - Resource Requester: Responsible for requesting and releasing resources managed by the FRIEND system.
  - Dispatcher: enters incidents, updates and deletes incidents in the system. He is also responsible for closing incidents.
  - Policeman: Reports incidents

Use case: ”Request resources” (2)

- Name of Use Case: Request resources
- Participating actors:
  - Policeman (Bob and Alice in this scenario)
  - Dispatcher (John in this scenario)
  - Resource requester
  - Operation manager
- Input actions:
  - The resource requester has selected an available resource
- Flow of events:
  - The resource requester chooses an incident
  - The resource is assigned to the incident

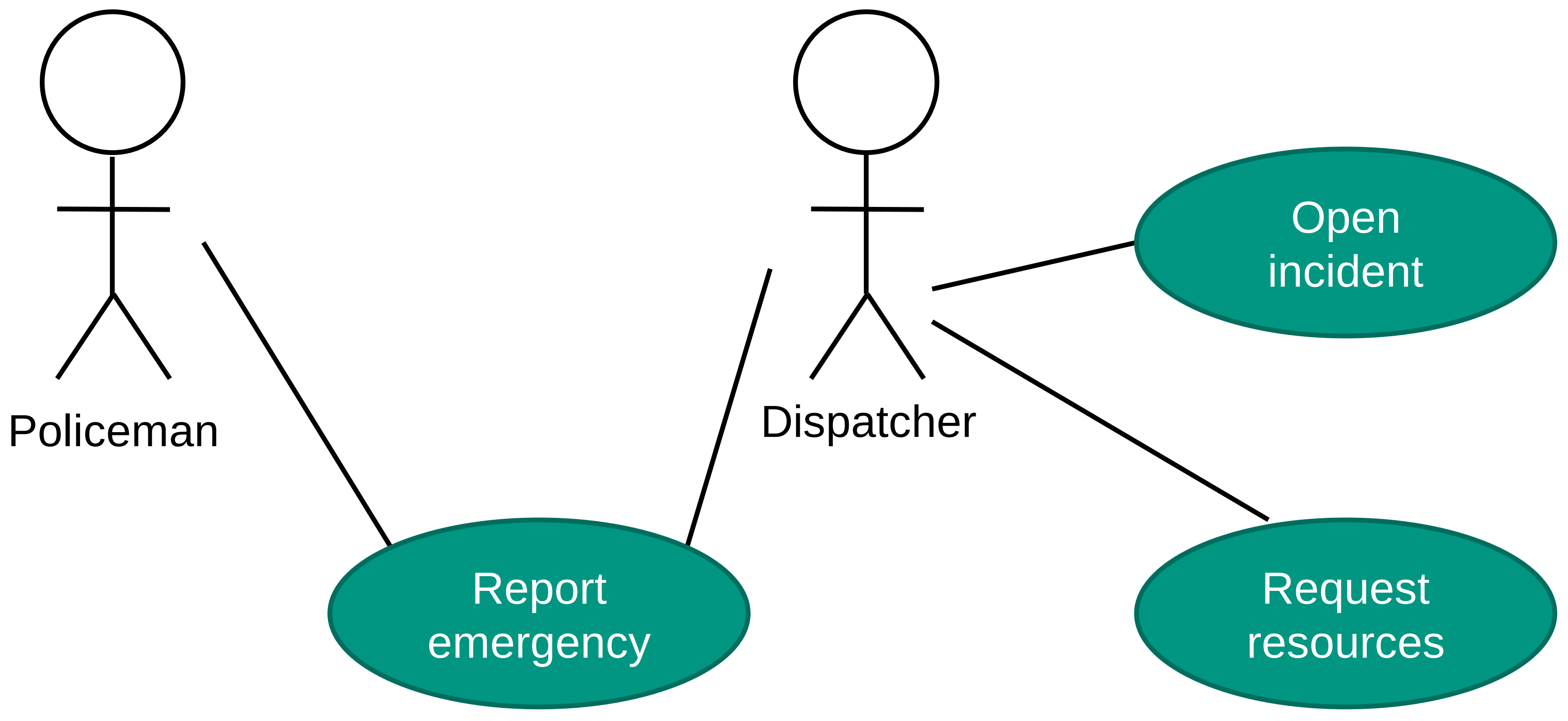
Use case: ”Request resources” (3)

- Output actions:
  - The use case is ready when the resource has been assigned.
  - The selected resource is not available for other requests.
- Special requirements:
  - The Operations Manager is responsible for the use of resources

How to formulate use cases?

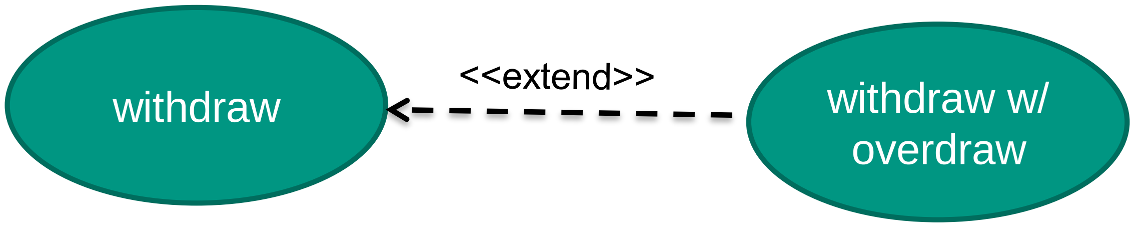
- Name of the use case:
    - For example: Report emergency
- Find the actors:
    - Generalize the concrete names ("Bob") to participating actors ("Policeman")
    - Participating actors:
        - Policeman (Bob and Alice in the example scenario)
        - Dispatcher (John in the example scenario)
- Find the event flow:
    - Described in natural language

Use case model – Report emergency



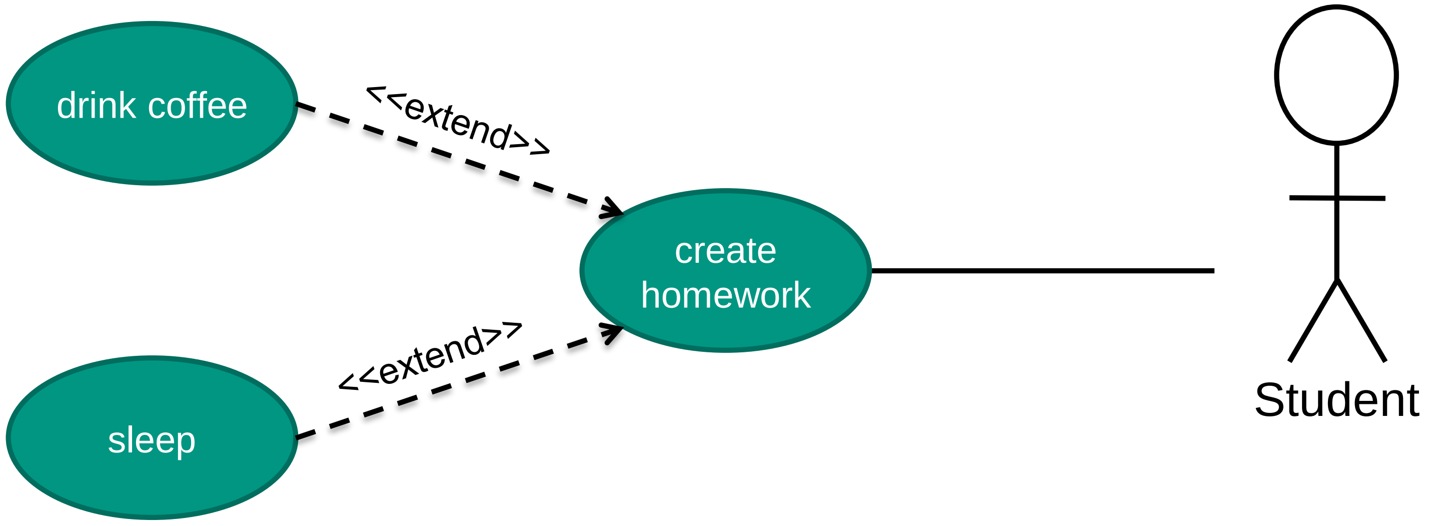
<<extend>> Relationship

- Use cases can be related (associated) to each other.
- **Extend** relationship represents rarely-called use cases or exceptional functionality.
  - A relationship between one use case which is extended by some optional use case (added features).
  - For example, use withdraw money can be extended by use case process overdraw



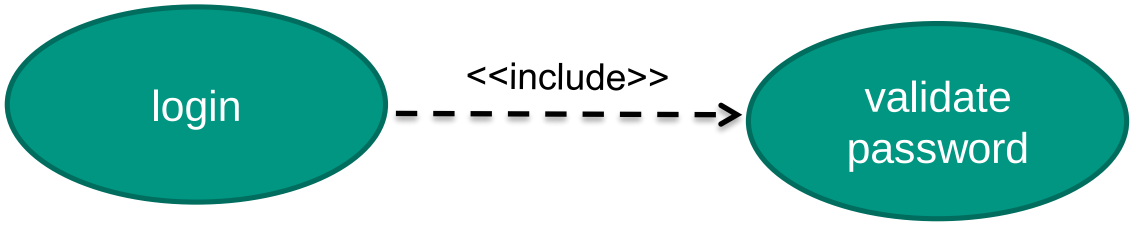
<<extend>> Relationship

- Extraordinary event flows are pulled out of the main event for the sake of clarity.
  - The direction of an <<extend>> relationship is to the extended use case (a kind of subclass relation).
  - Use cases that are exceptional flows can extend more than one use case.



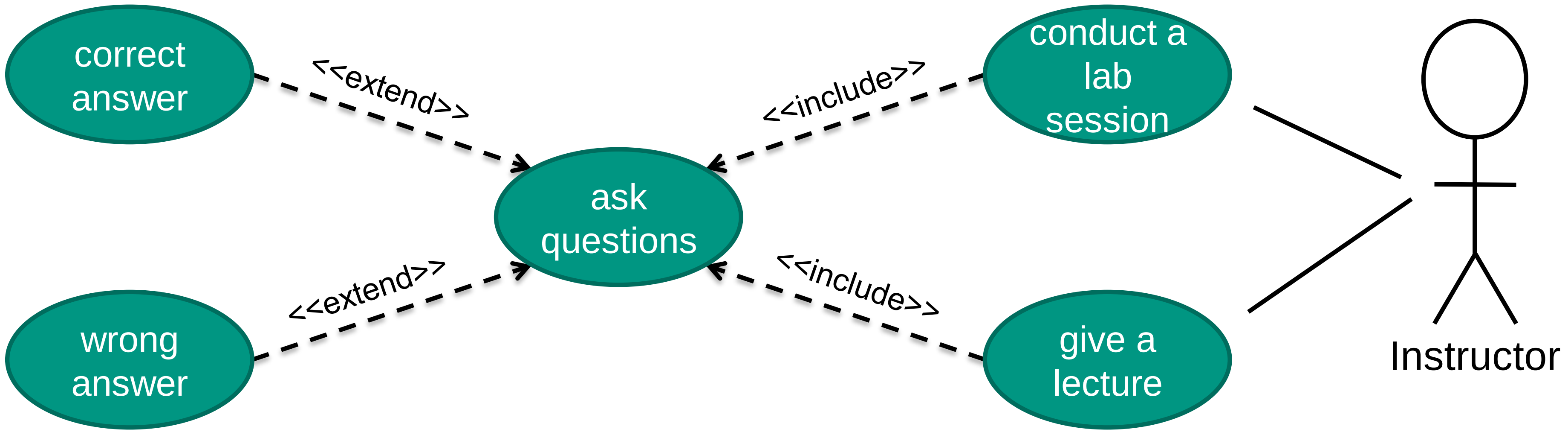
<<include>> Relationship

- **Include** relationship represents functionality that is used by more than one use case.
  - A relationship between one use case which requires the existence of another use case, and the latter, in isolation, is not meaningful to the user.
  - For example, validate password use case is included in login use case



<<include>> Relationship

- <<include>> relationships are general functions that are used in more than one use case.
    - <<include>> behavior is factored out for reusability reasons.
    - The direction of an <<include>> relationship goes to the included use case.



Use case text – Describing use case in text format

- Use case name
- Main **scenario**
  - **Steps**
- Extensions
  - Extension condition; steps

- Specify **what** to do, not **how** to do
- Do not specify user interface
- Optional: priority, trigger, pre-condition, post-condition (guarantees), sub-use case

Use case text – Example

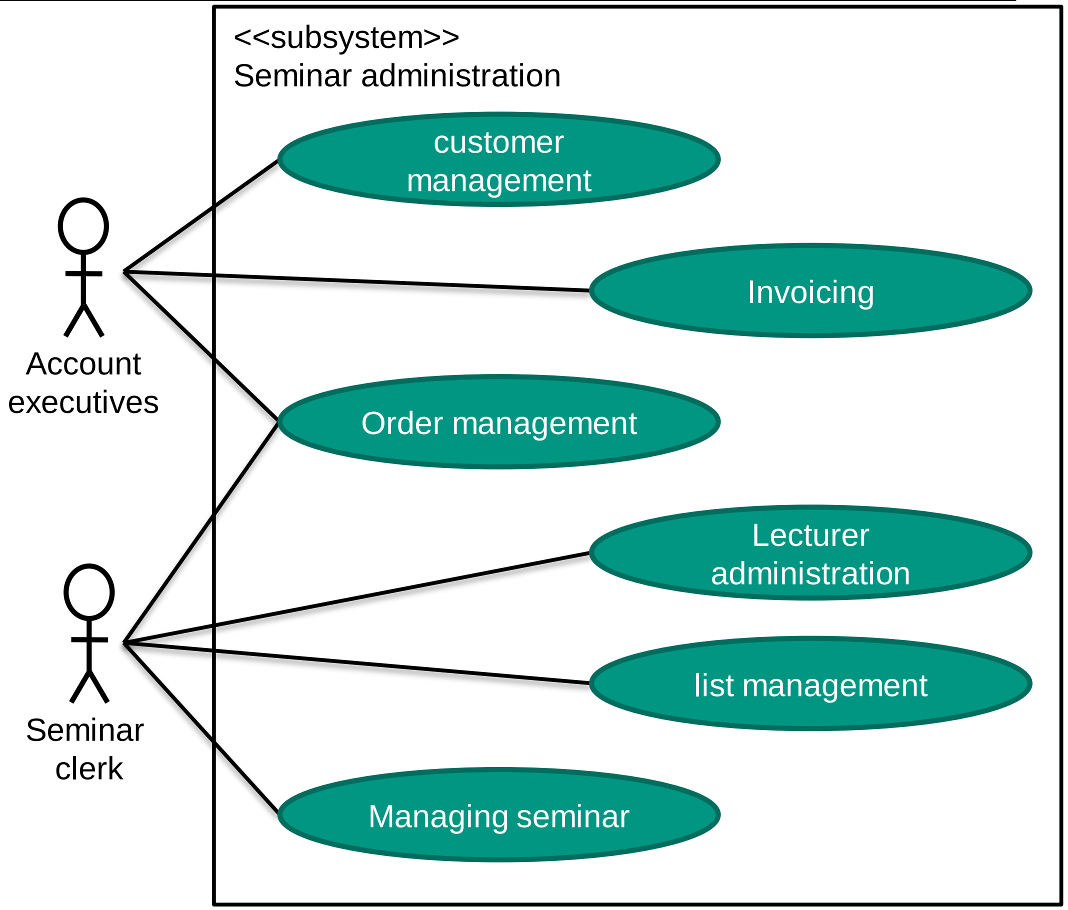
- **Name**:
  - Create homework
- **Participating actor:**
  - College student
- **Input condition:**
  - Student receives exercise sheet
  - Student is healthy
- **Output condition:**
  - Student makes solution
- **Flow of events**:
  - Student brings current exercise sheet
  - Student reads through the tasks
  - Student solves the task and enters it into the computer
  - Student prints the solution
  - Student submit the solution
- **Special requirements:**
  - No

Requirements specification vs. analysis models

- Both have the requirements for the system from the user's perspective as a goal
  - The requirement specification (e.g. scenarios) uses natural language (derived from the nature of the problem)
  - The analysis model uses formal or semi-formal notation
    - This course uses UML

Example: Seminar administration

- System Models – Use case: "Seminar administration”
- Actors:
  - Account Executive
  - Seminar clerk
- Use cases:
  - customer management
  - Invoicing
  - ...
- Textual description of the use case is missing in this example



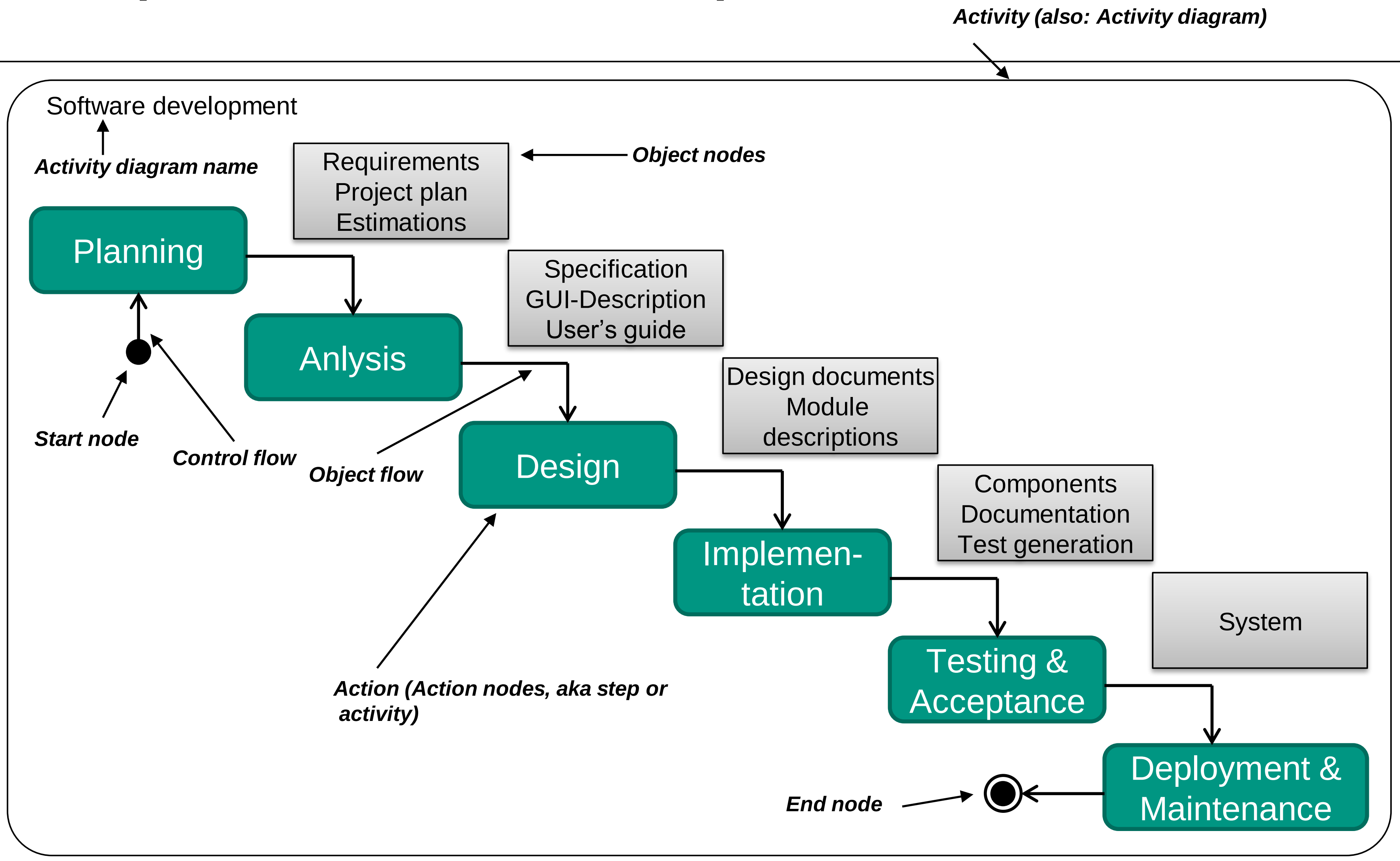
Activity diagrams – UML

- An activity – multiple actions
  - Can be used to describe a use case
  - Can represent parallel relationship
- An activity diagram describes a procedure
  - Operational or business processes
  - Technical processes of workflows and use cases
  - Concrete algorithmic processes in programs
- Activity diagrams consist of
  - Action, object nodes and control nodes, as well
  - Object flows and control flows.

Activity diagram – Main components

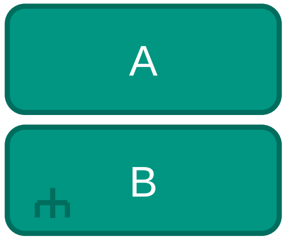
- Main components
  - Start
  - Actions
  - Fork/Join
  - Decision/Merge
  - Flow
  - Final

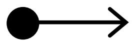
Example – Waterfall software process model

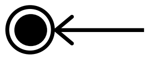


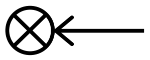
Activity diagram symbols and elements (1)

- Actions
  - Elementary action
  - Nested action
- Nodes
  - Starting node
    - Starting point of a process
  - End nodes
    - Ends all actions and control flows
  - Flow final
    - Ends a single object flow and control flow



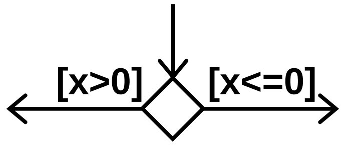




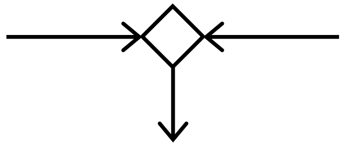


Activity diagram symbols and elements (2)

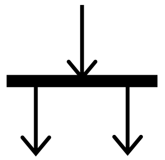
- Decision
  - Conditional branching



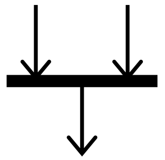
- Merging
  - "or” connecting



- Forking
  - Dividing a control flow



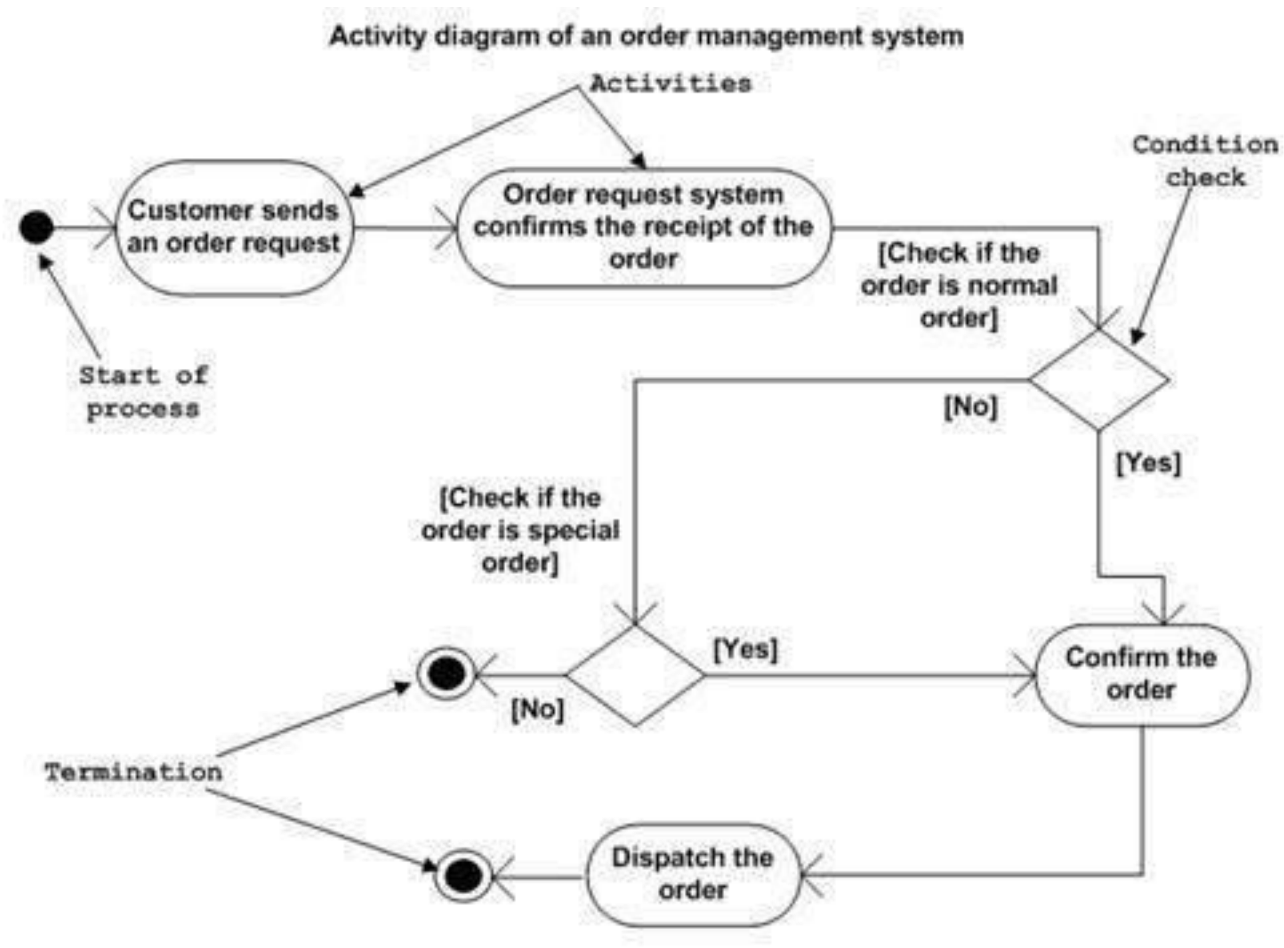
- Synchronization
  - ”and" joining



Activity diagram – Example: Order management

- An activity diagram for order processing

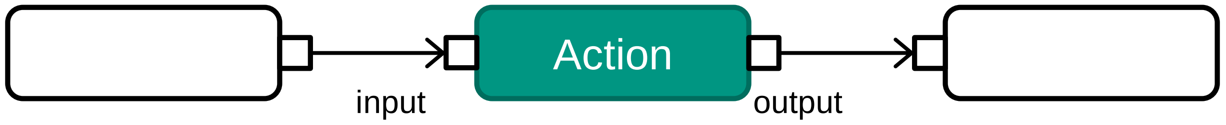
Source: https://www.tutorialspoint.com/uml/



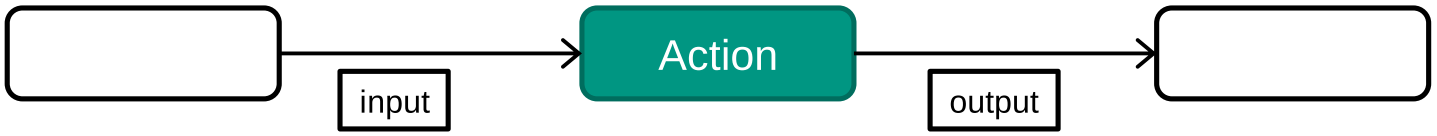
Activity diagram of an order management system
Activities
Condition check
Customer sends an order request
Order request system confirms the receipt of the order
[Check if the order is normal order]
Start of process
[No]
[Yes]
[Check if the order is special order]
[Yes]
[No]
Confirm the order
Termination
Dispatch the order

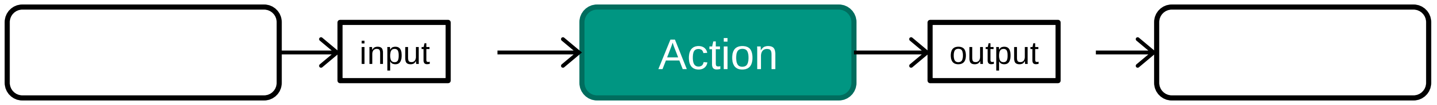
Activity diagram symbols and elements (3)

- Object node
  - Input and output data of an action
  - Representation by pin



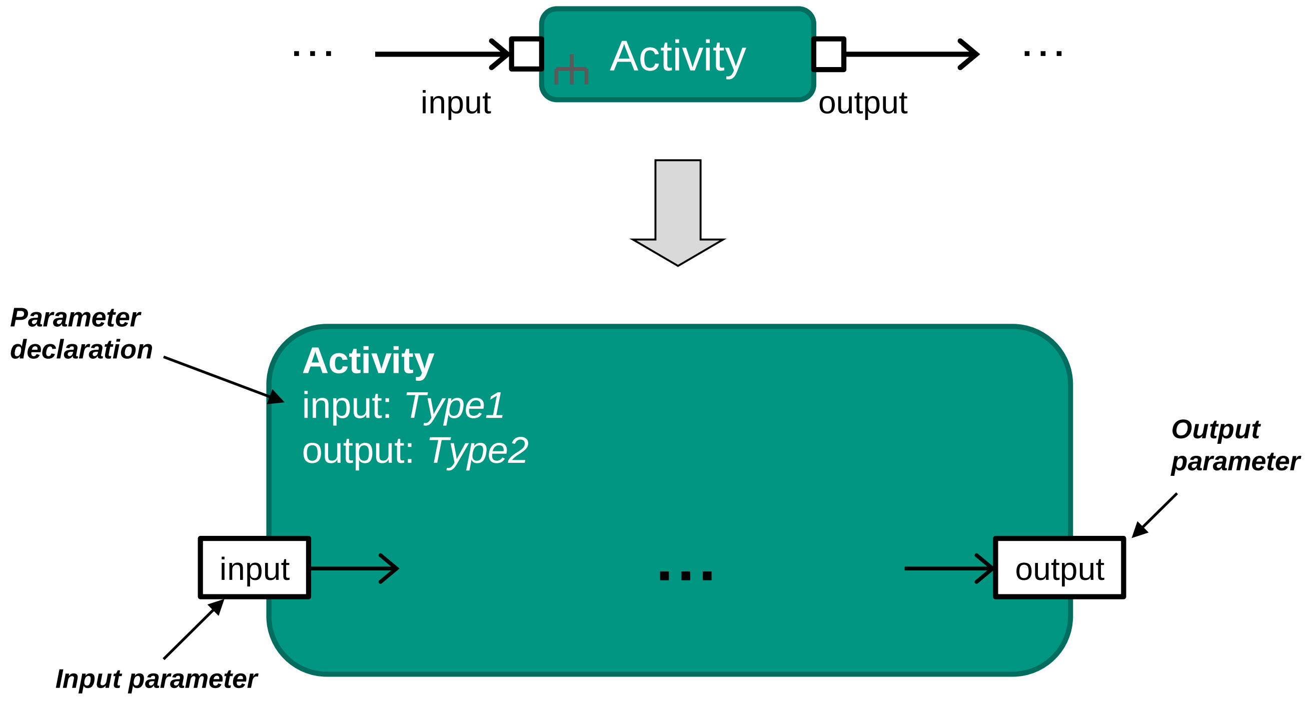
  - Alternative representation





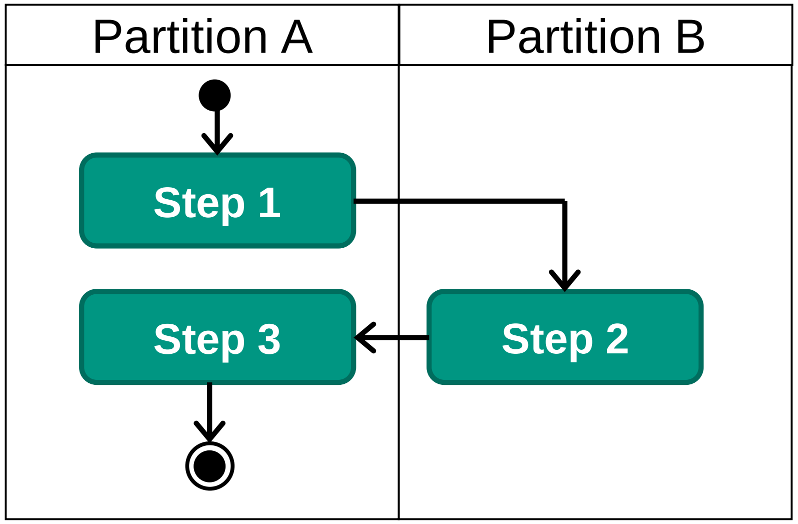
Activity diagram symbols and elements (4)

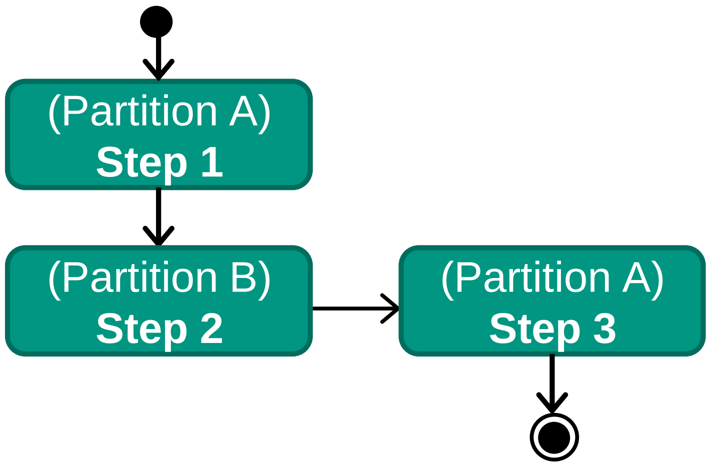
- Parameters of activities



Activity diagram symbols and elements (5)

- Partitions (areas of responsibility)
  - Partitions describe who or what is responsible for a node or what common feature characterizes it.
    - For example, partitions could be different computers working together (e.g. server and client)

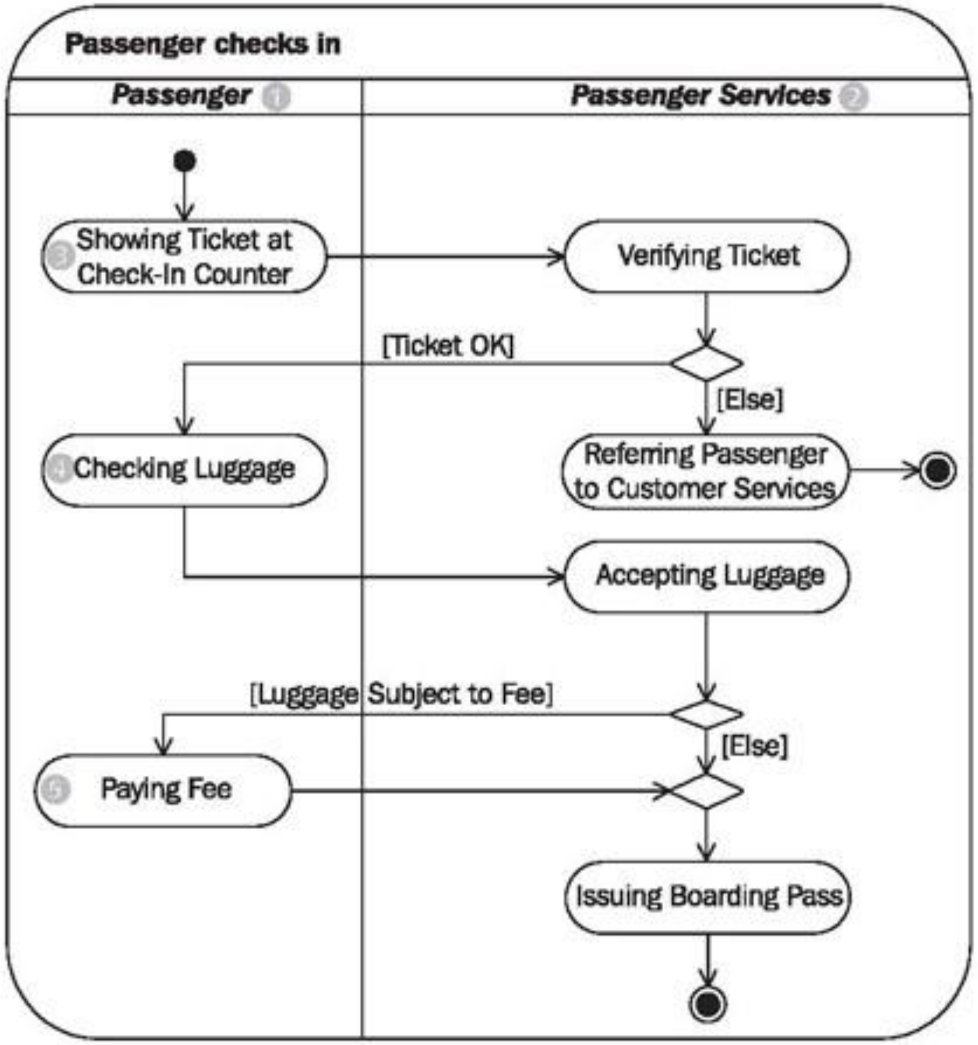




Activity diagram – Example with partitions

- An activity diagram with partitions

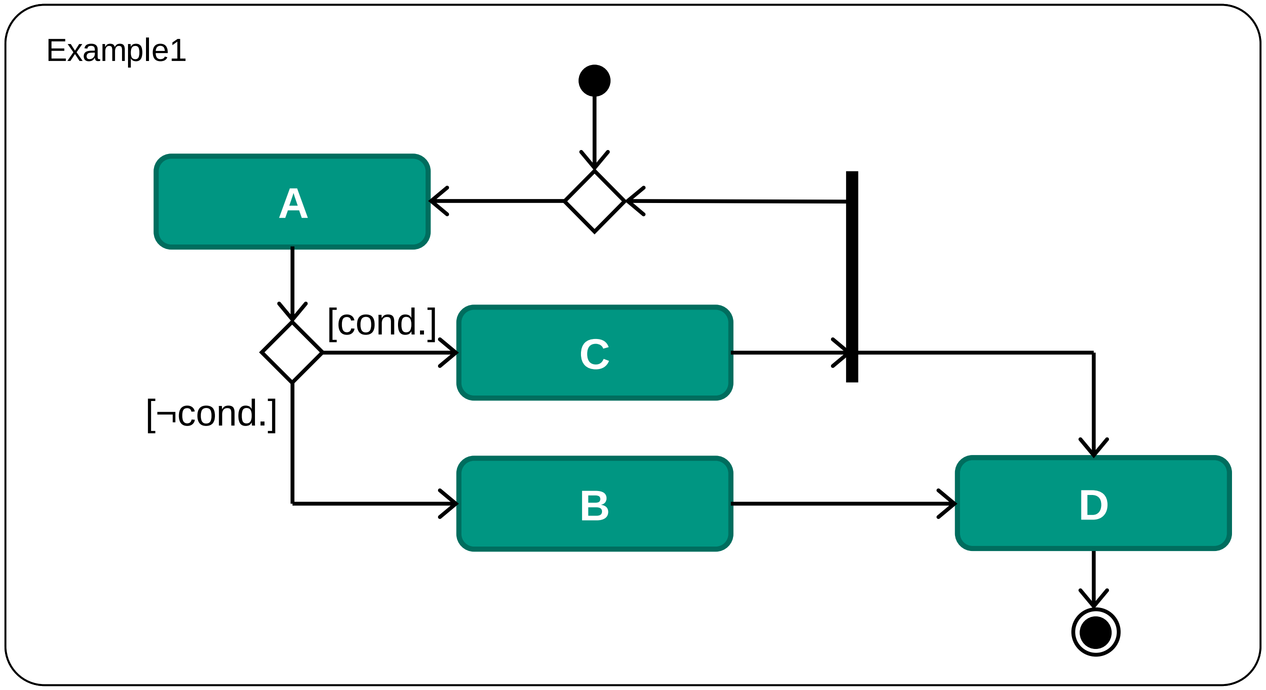
Source: https://sourcemaking.com/uml/



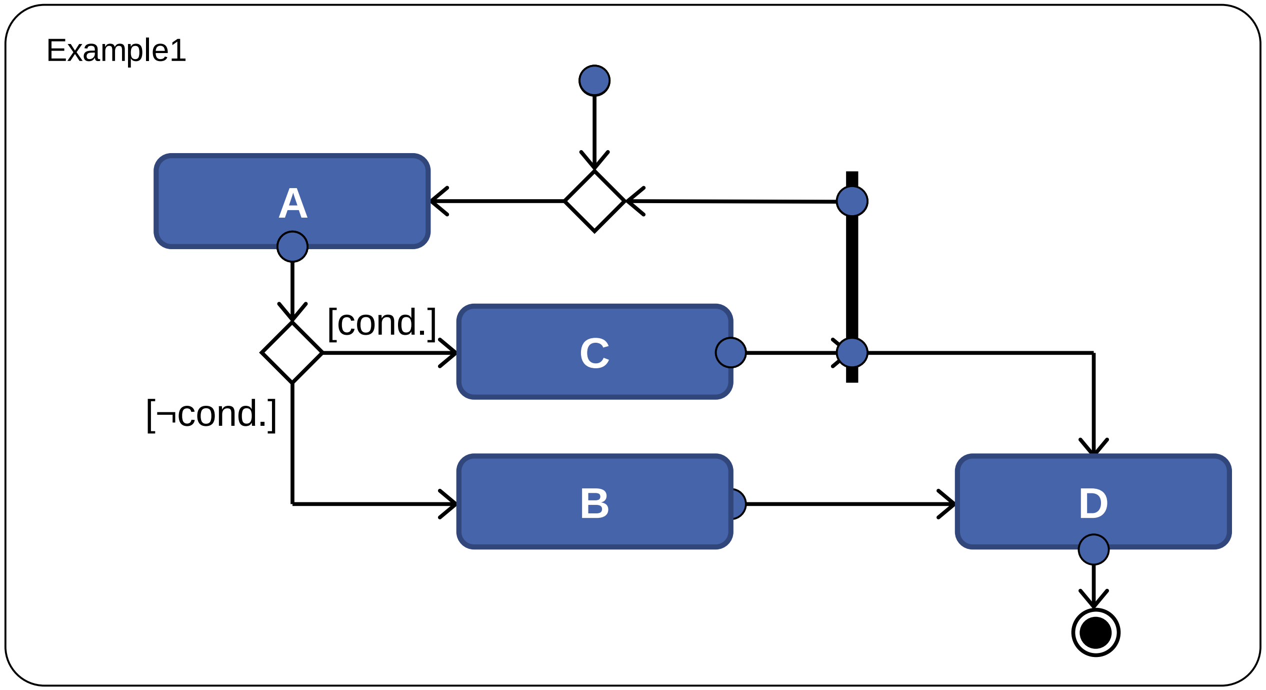
Execution semantics – Actions

- The sequence of actions is controlled by control flow and object flow edges, (Abbr. CFE or OFE)
  - Control marks move on control flow edges, object marks move on object flow edges
- An action can only be executed if all incoming CFE and OFE edges carry marks
- When the action begins, marks are taken from the incoming edges:
  - An object mark of each OFE edges
  - All control marks from the CFE edges
- After ending the action, marks will be offered on **all** outgoing CFE and OFE edges, which in turn will be available to other actions.

Execution semantics – Example



Execution semantics – Example (animation)



OBJECTIVES OF SYSTEM MODELING

Objectives of system modeling

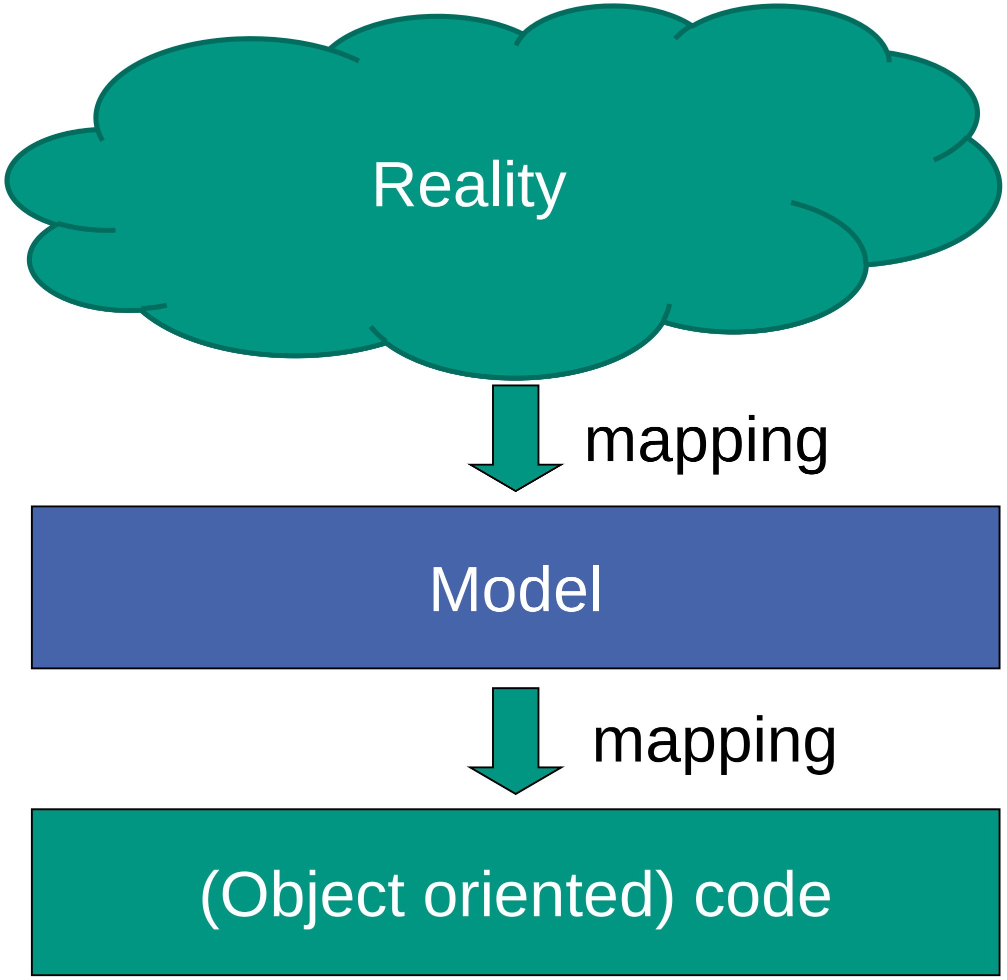
- In system modeling, the specifications are created (story cards)
- The specification defines ("models") the system to be created (or changes to an existing system)
  - **Completely and accurately** that developers can implement the system without having to ask or guess what to implement.
- The requirement specification does **not** describe **how** to be implemented, but only **what** is to be implemented.
  - e.g. algorithms and data structures are **not** described!
- The requirement specification is a refinement of the specifications.
- The requirement specification provides a model of the system to be implemented.

Model types

- **Functional model** (knowledge about **functionalities** – from the specifications)
  - Scenarios (use case text), use case diagrams
- **Object model** (knowledge about **relationships**)
  - Class diagrams and object diagrams
- **Dynamic model** (knowledge about **causality**)
  - Sequence diagrams
  - State chart diagrams
  - Activity diagrams

Model and reality

- Reality **R**
  - Real things, people, concepts, etc.,
  - Procedures that take a certain amount of time
  - Relationships between things, persons, concepts
- Model **M**: abstraction from existing or imaginary
  - Things, people, concepts, etc.
  - Processes
  - Relationships in between



Why models?

- We use models ...
  - To **abstract** from details of reality, we can conclude on the complex reality by making simple, well-defined deduction steps on the model.
  - To get **knowledge** about past or present.
  - To make **predictions** about the future.
- Especially for software models
  - We use models (of the software) to develop an understanding of the possible future reality (the software).
  - In other words, we use **models to see what we would get if we built it**.

What is a "good" model?

- Relations that are valid in reality R are also valid in model M.
  - π: mapping of reality R onto a model M (abstraction)
  - f_M: relationships between abstractions in M
  - f_R: equivalent relations between real things in R
- In a good model, the following diagram is commutative:

$$\begin{tikzcd}
\mathbf{M} \arrow[r, "f_M"] & \mathbf{M} \\
\mathbf{R} \arrow[u, "\pi"] \arrow[r, "f_R"'] & \mathbf{R} \arrow[u, "\pi"']
\end{tikzcd}$$

Meta-modelling: Models of models of models ...

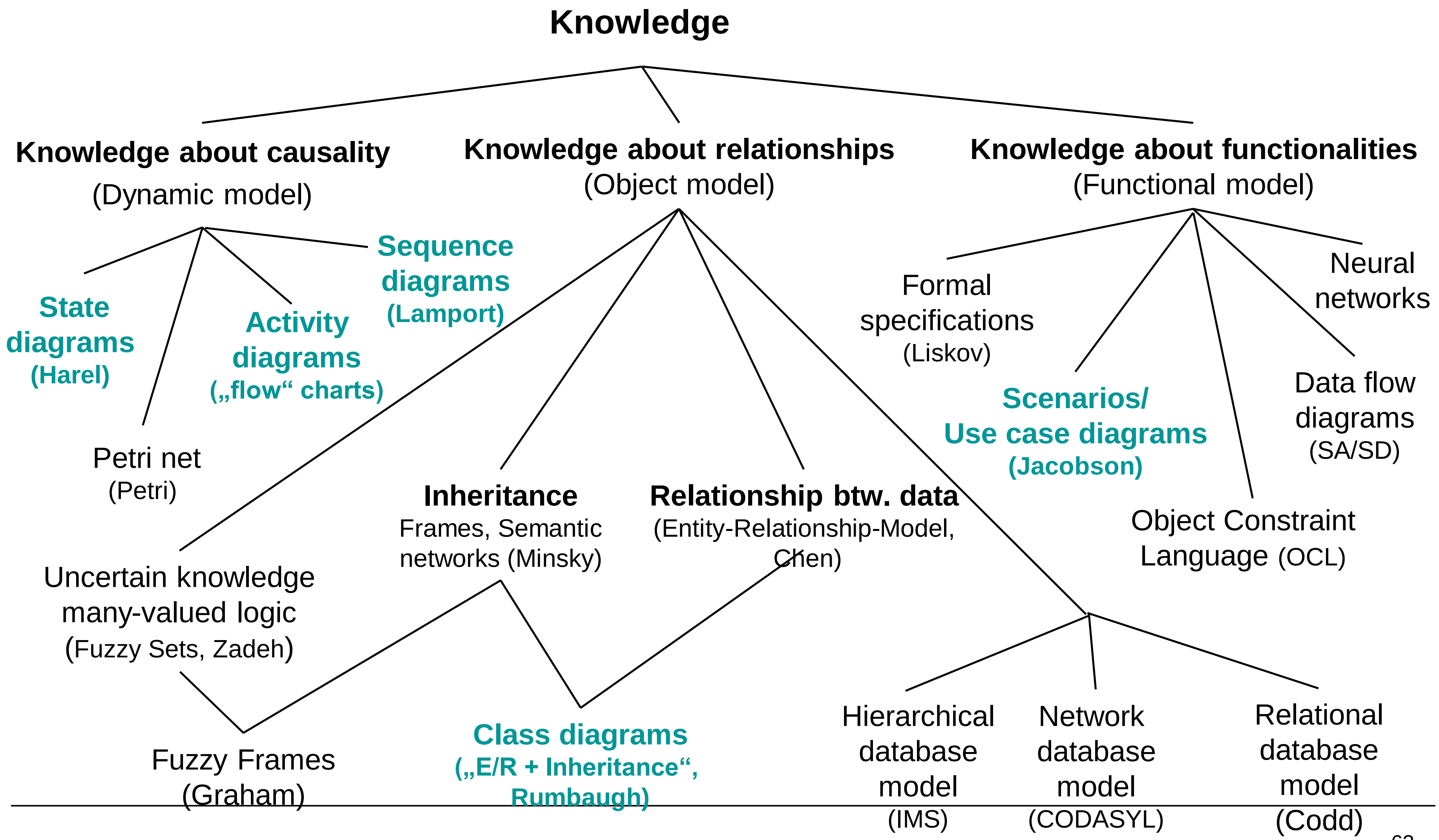
- Modeling is relative
- We can think of a model as reality and define a new model (with more abstractions)
- Software development is called **model transformation**

$$\begin{tikzcd}
 & \cdots & \\
M_2 \arrow[rr, "f_{M2}"] & & M_2 \\
M_1 \arrow[u, "\text{Analysis}"] \arrow[rr, "f_{M1}"] & & M_1 \arrow[u, "\pi_2"'] \\
R \arrow[u, "\text{Requirement Engineering}"] \arrow[rr, "f_R"'] & & R \arrow[u, "\pi_1"']
\end{tikzcd}$$

Realities for software engineers!

- Software engineers can model and implement different "realities":
  - Model an **existing system** (physical, technical, social or software system) and build an implementation (realization)
    - The existing "software system" is an important special case: we speak of "legacy system”
  - Model and realize (implement) an **idea** without a corresponding counterpart in reality
    - A visionary scenario or a customer requirement
    - In such cases, often only a part of the original model is built, because the rest is e.g. too complicated, too expensive or useless

How do we model complex systems?


Knowledge
Knowledge about causality
(Dynamic model)
Knowledge about relationships
(Object model)
Knowledge about functionalities
(Functional model)
State diagrams (Harel)
Petri net (Petri)
Activity diagrams („flow“ charts)
Sequence diagrams (Lamport)
Uncertain knowledge many-valued logic (Fuzzy Sets, Zadeh)
Fuzzy Frames (Graham)
Inheritance Frames, Semantic networks (Minsky)
Relationship btw. data (Entity-Relationship-Model, Chen)
Class diagrams („E/R + Inheritance“, Rumbaugh)
Hierarchical database model (IMS)
Network database model (CODASYL)
Relational database model (Codd)
Formal specifications (Liskov)
Scenarios/ Use case diagrams (Jacobson)
Object Constraint Language (OCL)
Data flow diagrams (SA/SD)
Neural networks

REVIEW OF OBJECT ORIENTATION

What is the first programming language you learned?

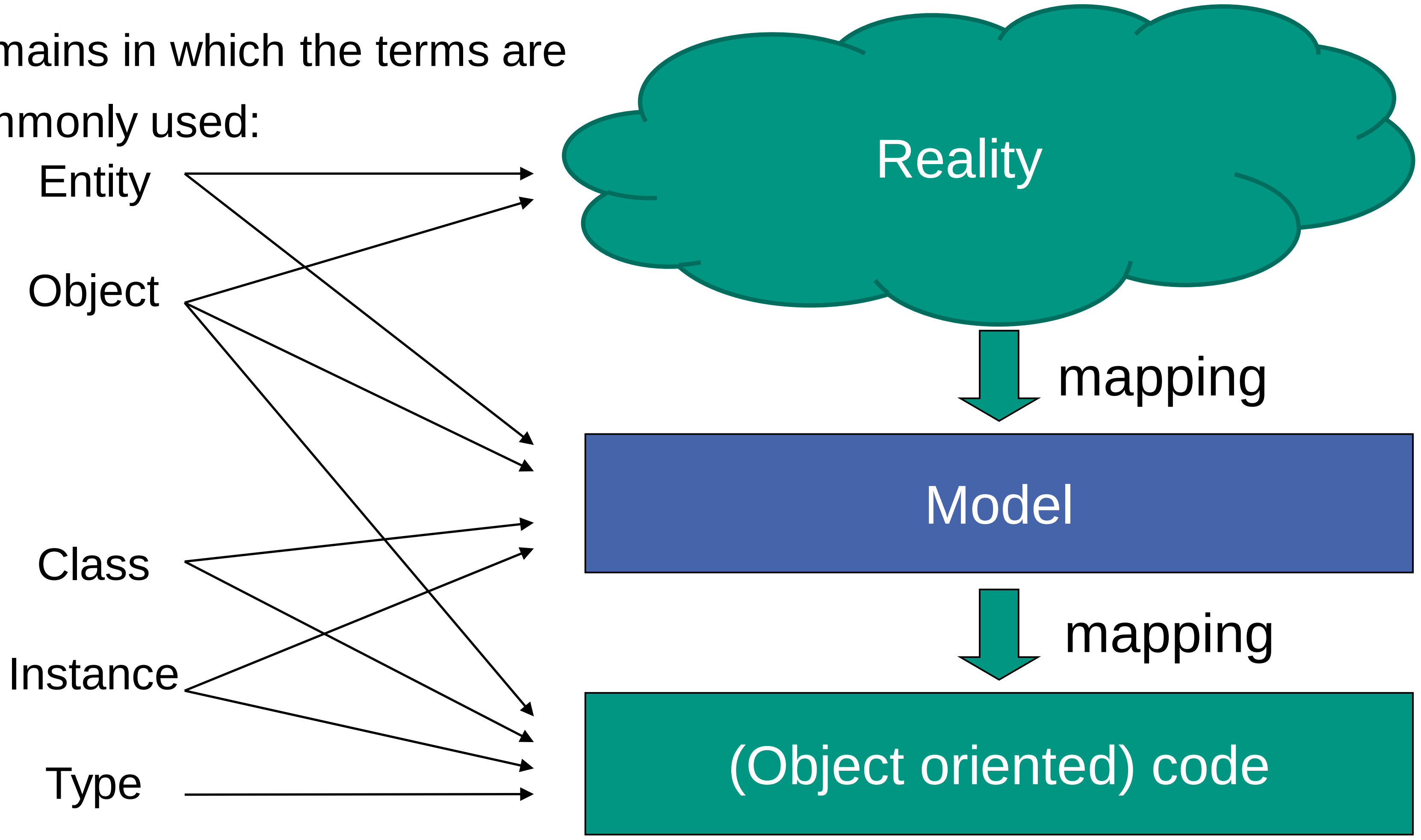
- Java
- C#
- C/C++
- Python
- Javascript

Object, class and instance (1)

- **Object** is individual recognizable, clearly identifiable from other objects
  - Everything you can call a noun or name.
  - Charles S. Peirce: “By an object, I mean anything that we can think, i.e. anything we can talk about.”
- **Class** is a category over the set of all objects – a template
  - As a rule, the categorization will be based on some sort of "similarity" of the objects.
  - The category may also be empty: a class designates the basic idea / concept of things and exists regardless of whether or not there is an instance.
- **Instance** is a concrete occurrence from a certain class.
  - A unique copy of a Class representing an Object

Object, class and instance (2)

- Domains in which the terms are commonly used:



Attribute and object identity

- **Attribute** is a defined and existing property for all instances of a class, that …
    - can be given for each individual copy independently of the others
    - has a clearly defined value
- Attributes could contain constraints
    - Requires additional code (→ Object Constraint Language)
- **Object identity**: The existence of an object is independent of its attribute values
    - Two objects are distinguishable even if they have the same attribute values.

Object state

- **State**: A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

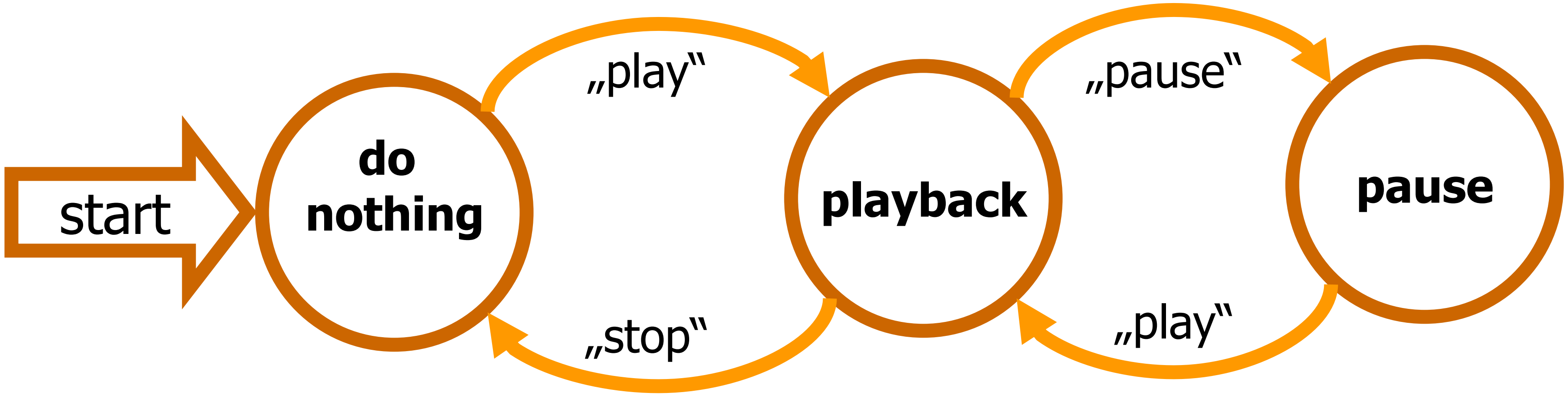
- **State of an object**
  - As long as an object is in a state, it always reacts to its environment in the same (call / use) context.
  - As the state changes, the object reacts differently than before in at least one context (external view).

Encapsulation (1)

- The features of encapsulation are supported using classes
  - Class: data + operation
- **Encapsulation**: Although the state of the object is visible to the outside, it is managed inside the object (and thus only changed in a controlled manner).
  - The packing of data and methods (functions) into a single component. It allows selective hiding of properties and methods in a class by building an impenetrable wall to protect the code from accidental corruption.
  - Note: Modifying an attribute value could cause the state to change.
    - **Example**: If the value of the “Remaining Time” attribute of an “Hourglass” object changes to 0, the “Hourglass” state will change to “Ring”.

Encapsulation (2)

- **Example** for encapsulation principle: No one would think of fiddling around inside a video recorder to put it in the "play" state



Message to an object via method calls

- **Message exchange:** a certain object (= message receiver) is asked to make a state transition
- Therefore: **message exchange** = **method call** for a specific object
  - **Example**: By using the Play button, you can send the message "Change to Playback"' to the device which the button belongs
- Methods can change the state of an object
  - The available methods define the acceptable messages that can be sent to an object (external view)
  - **Problem**: When should I send which message to an object?
    - Answer: This is specified with a state diagram (see later)

Method signature

- Method signature consisting of:
  - method name
  - return type
  - parameter list
    - The parameters are the ”reference data" of the message
    - The recipient object can also be considered as the "zeroth parameter", as the "address" of the message
- Notation:

```
MethodName (ParameterList): ReturnType;
```

- Notation (parameter list):

```
AttributeName: Type [, AttributeName: Type] *
```

Summary

- System modeling
  - Use cases (UML)
- Use case text
- Activity diagram
- Objectives of system modeling
- Review of Object orientation

Back-up

Outline

- Introduction to requirement engineering
  - Definitions
  - Requirement, requirement engineering, why?
  - How to write the requirement document (Story card in XP agile)?
  - How to find out and model the requirements?
- Use case model
  - Actors, relationships between use cases
- Use case text
- Activity diagrams
- Objectives of system modeling
- Review of object orientation

What are requirements?

- IEEE Standard 610.12-1990 definition:
  "A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
  The set of all requirements forms the basis for subsequent development of the system or system component".