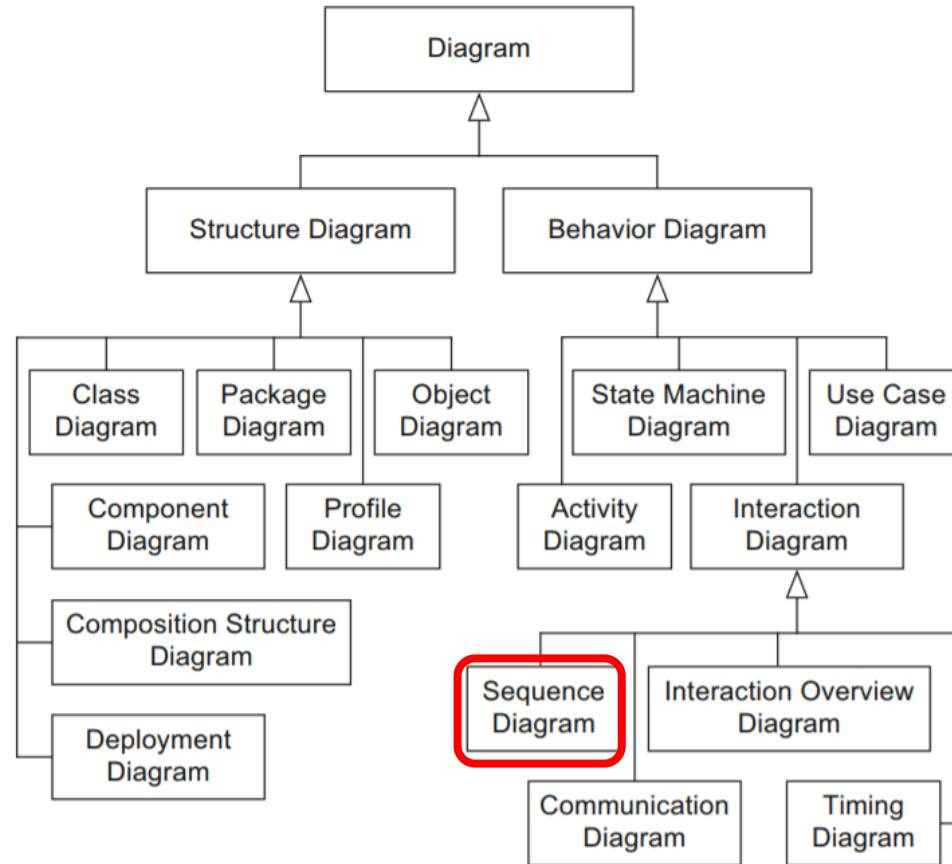# Lecture 10: Sequence Diagram & Activity Diagram

Sequence Diagram

UML Diagrams

Sequence Diagram

- Message among *interaction* partners.

- An interaction specifies how messages and data are exchanged between interaction partners.

- The interaction partners are either human, such as lecturers or students, or non-human, such as a server, a printer, or executable software.

- An interaction can also be a sequence of method calls in a program or signals such as a fire alarm and the resulting communication processes.
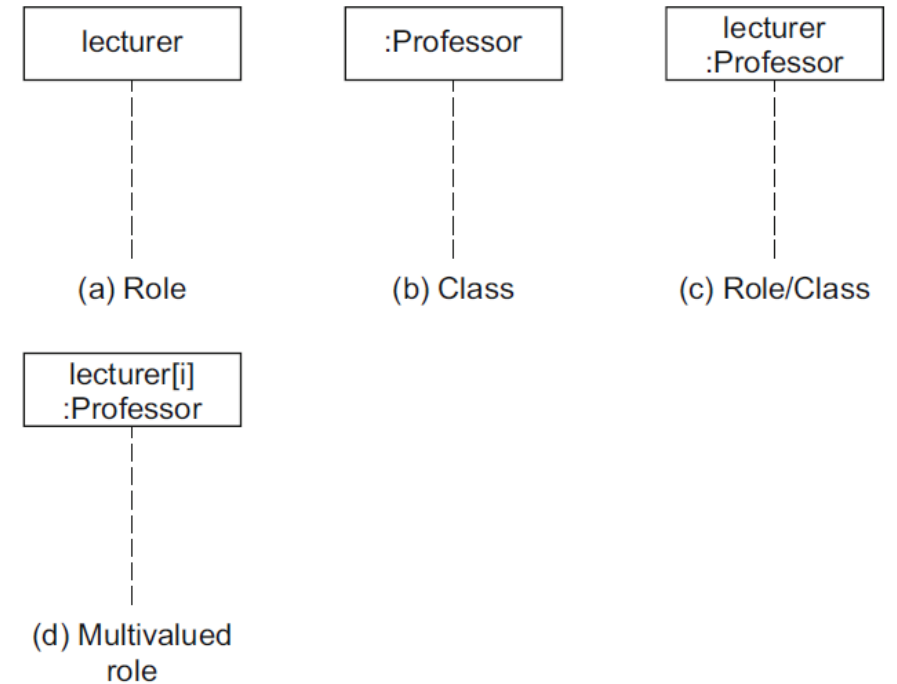
Interaction Partners

- In a sequence diagram, the interaction partners are depicted as *lifelines*.

- A lifeline is shown as a vertical, usually dashed line that represents the lifetime of the object associated with it.

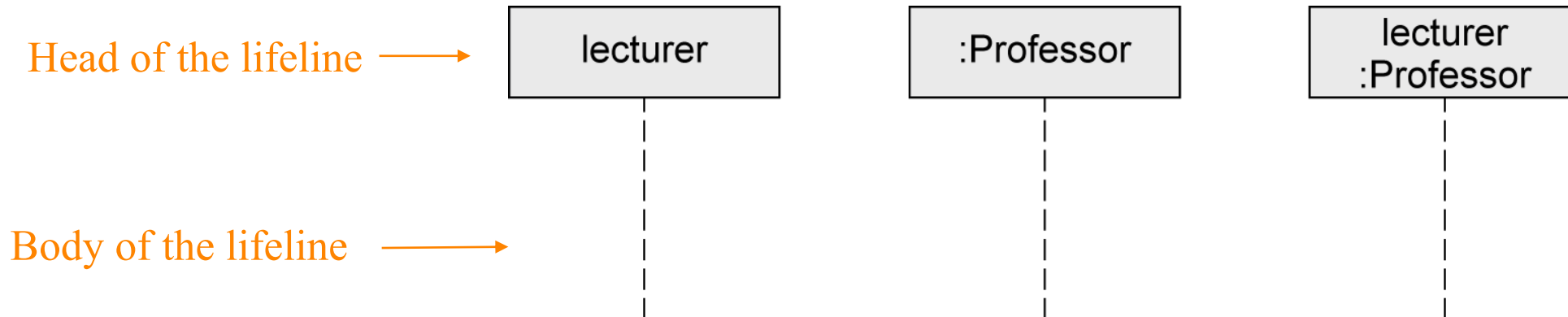# Interaction Partners (2)

- One of the two names may be omitted:
  - If you omit the class, you can omit the colon (Fig.(a));
 - If you specify only the class, the colon must precede the class name (Fig.(b));

lecturer

(a) Role

:Professor

(b) Class

lecturer :Professor

(c) Role/Class

lecturer[i] :Professor

(d) Multivalued role

Interaction Partners (3)

- Head of the lifeline
  - Rectangle that contains the expression **roleName:Class**

Head of the lifeline ⟶ | lecturer | | :Professor | | lecturer :Professor |
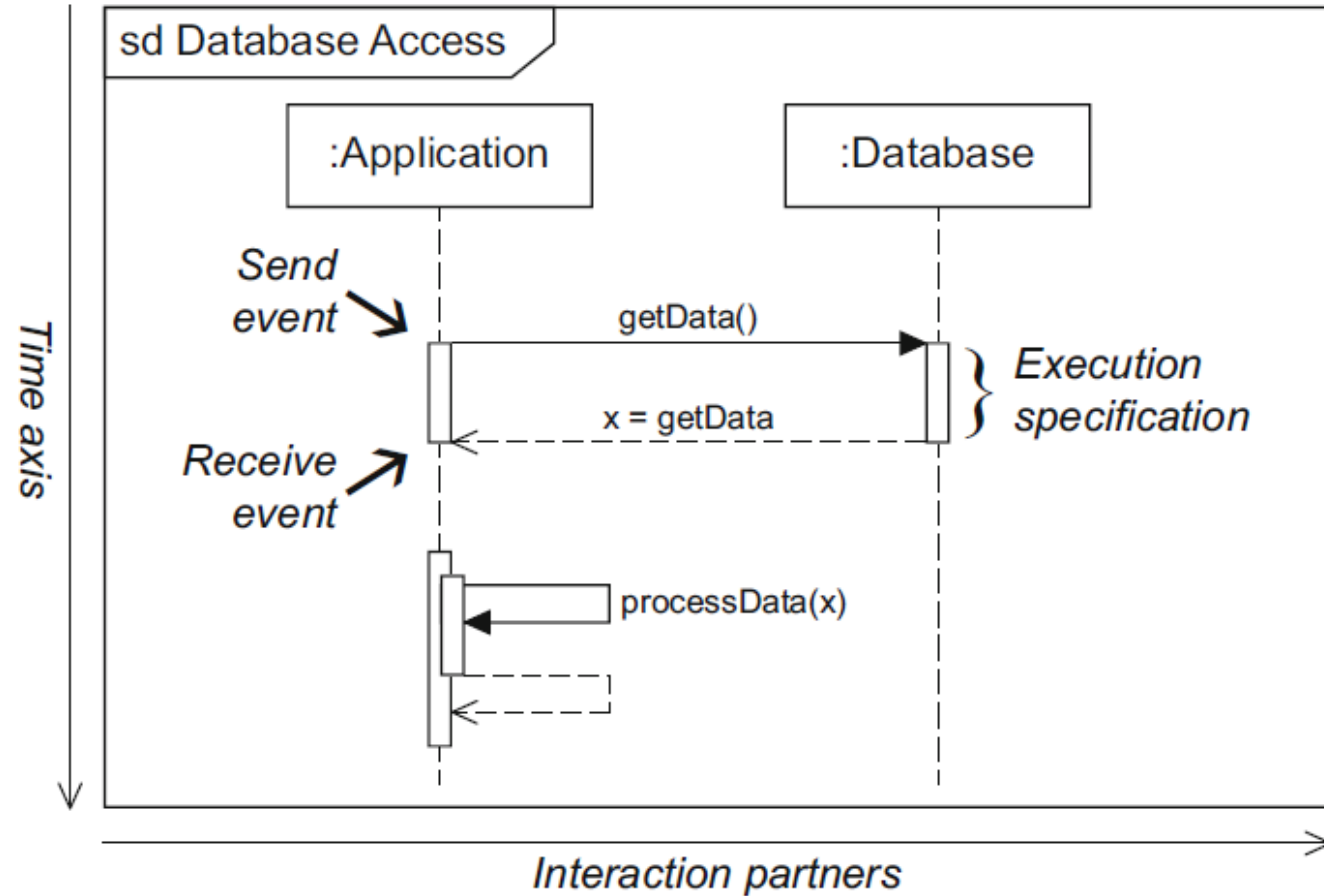
Body of the lifeline ⟶

Interaction Partners (4)

- Body of the lifeline
  - Vertical, usually dashed line
 - Represents the lifetime of the object associated with it

Head of the lifeline ⟶

Body of the lifeline ⟶

lecturer

:Professor

lecturer
:Professor
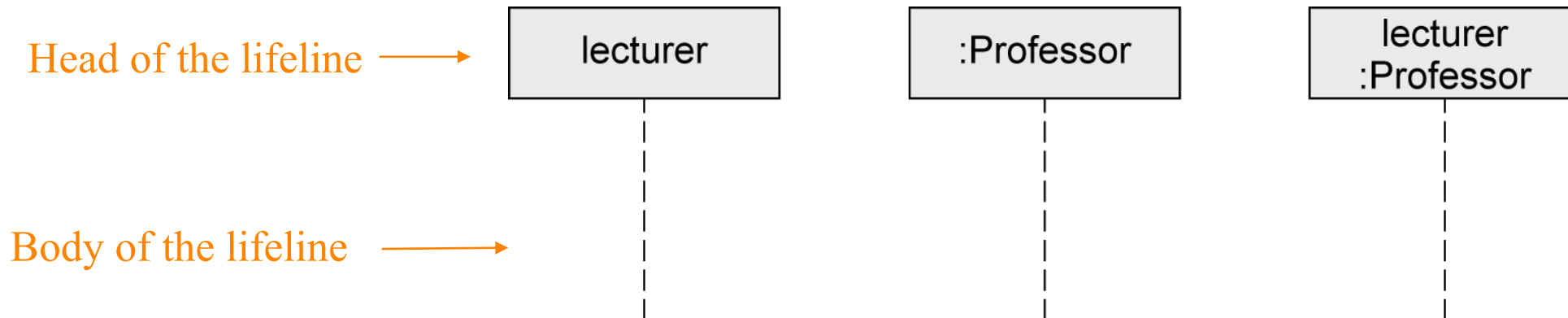
Exchanging Messages

- Two dimensions
  - Time
  - Interaction partners

- Execution specification
  - Self message

Exchanging Messages: Order

- Message order is chronical if messages on the same lifeline
  - It's a transitive relationship
- Unless specified otherwise, we *assume* that the message Send event and receive transmission does not require any time, meaning that the send event at event the sender and the receive event at the receiver take place at the same time.

(a) (b) (c)
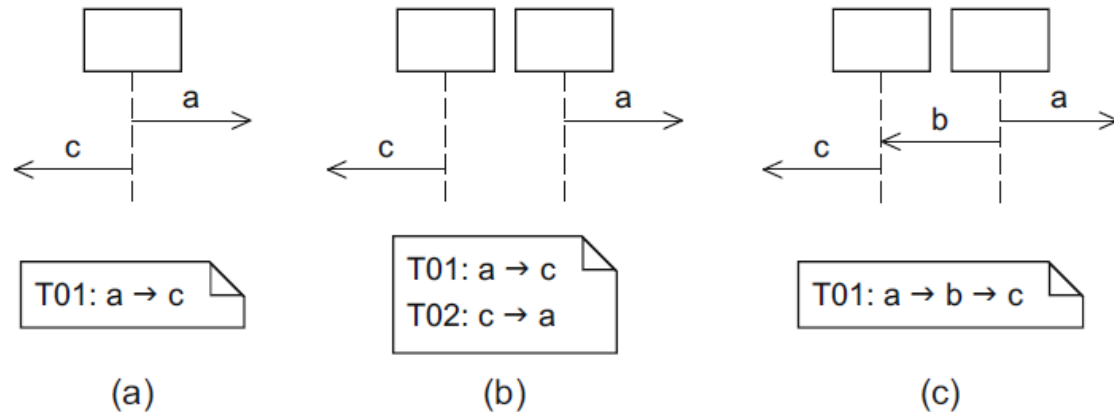
Exchanging Messages: Order (2)

- (a): a->c means that message a is sent before message c.
- (b): a->c / c->a
- (c): a->b->c (If a message b is inserted between a and c and this message forces a and c into a chronological order)

(a)   (b)   (c)
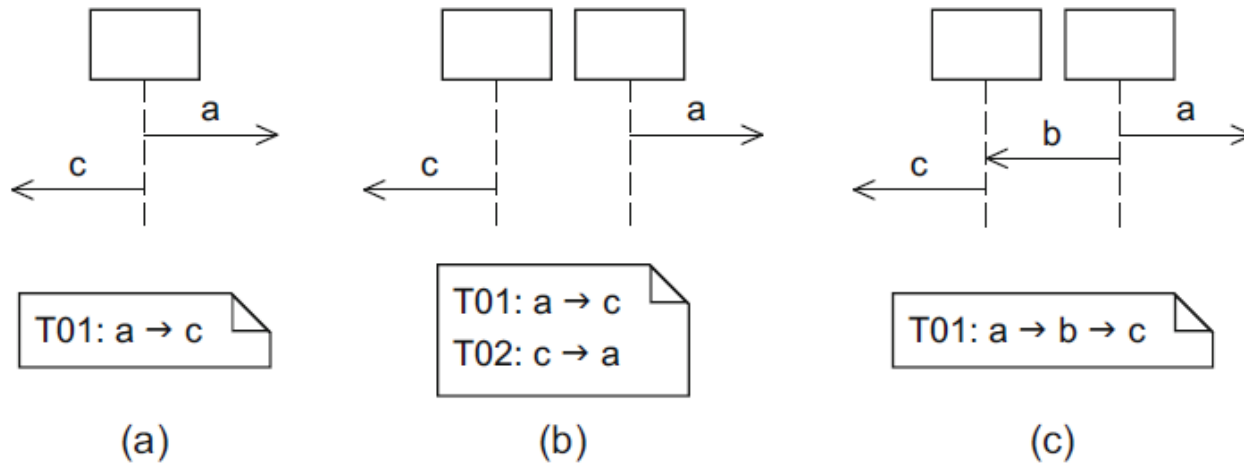
Exchanging Messages: Type

- In a sequence diagram, a *message* is depicted as an arrow from the sender to the receiver.

- Synchronous message
  - represented by an arrow with a continuous line and a filled triangular arrowhead.

- In the case of synchronous messages, the sender waits until it has received a response message before continuing.

Exchanging Messages: Type (2)

- The student registers with professor personally and the communication is therefore synchronous. The student waits until receiving a response message.

Synchronous

(b)
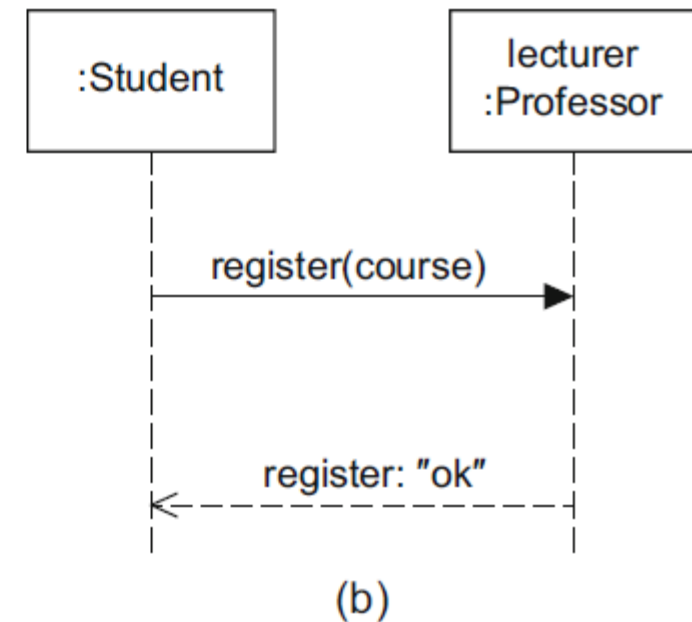
Exchanging Messages: Type (3)

- Asynchronous Message

- An asynchronous message is depicted by an arrow with a continuous line and an open arrowhead.

- In asynchronous communication, the sender continues after having sent the message.

Exchanging Messages: Type (4)

- In case (a), the registration is via e-mail, that is, asynchronous. The student does not explicitly wait for the receipt of the confirmation message.

Asynchronous

(a)
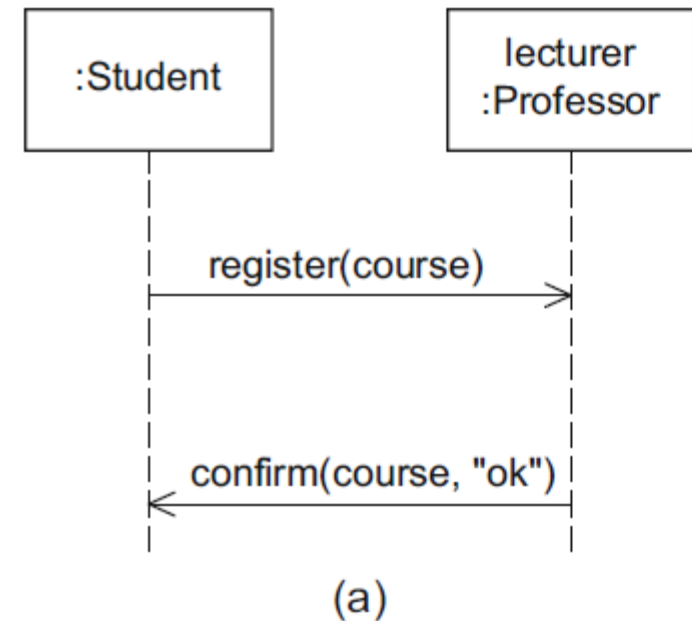
Special Messages

- ## Create message
  - Creating new object
  - The arrow is labeled with the keyword `new` and corresponds to calling a constructor in an object-oriented programming language.

- ## Destruction event
  - Destruction of an object
  - If an object is deleted during the course of an interaction, that is, a destruction event occurs.
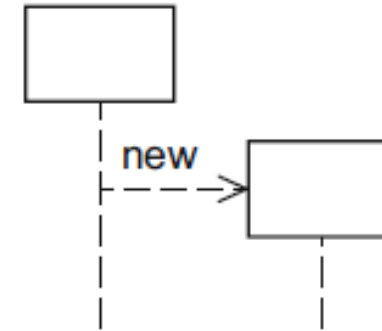
Special Messages (2)

- Found message
  - Unknown/irrelevant sender

- Lost message
  - Unknown/irrelevant receiver

- Time-consuming message
  - message with duration.
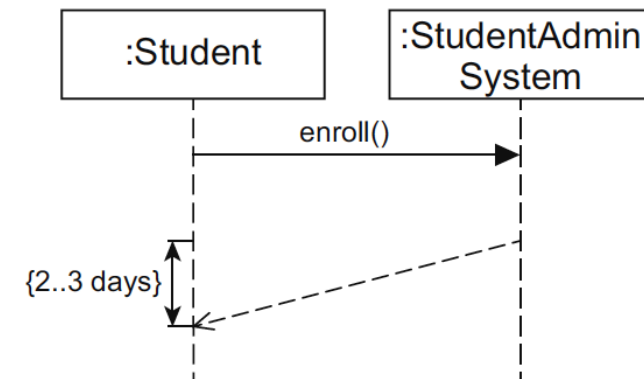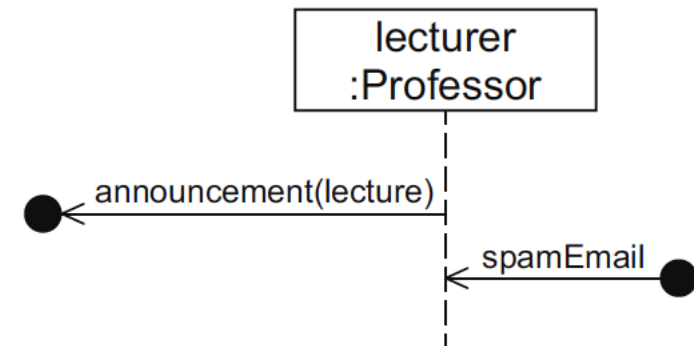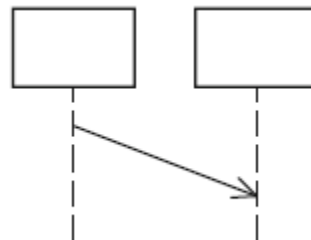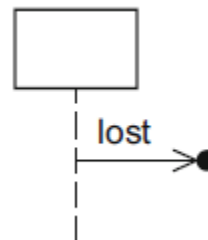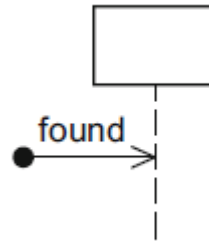
Combined Fragments

- Combined fragments (operators) model various control structures explicitly.

- Combined fragments describe a number of possible execution paths compactly and precisely.
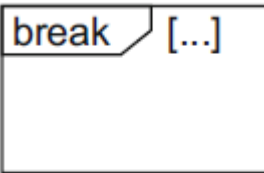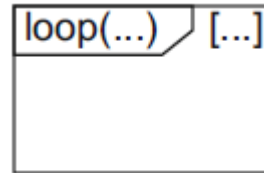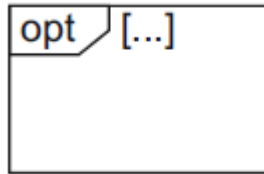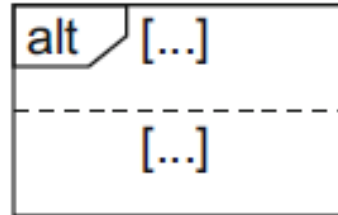
Operator
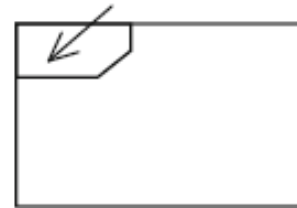
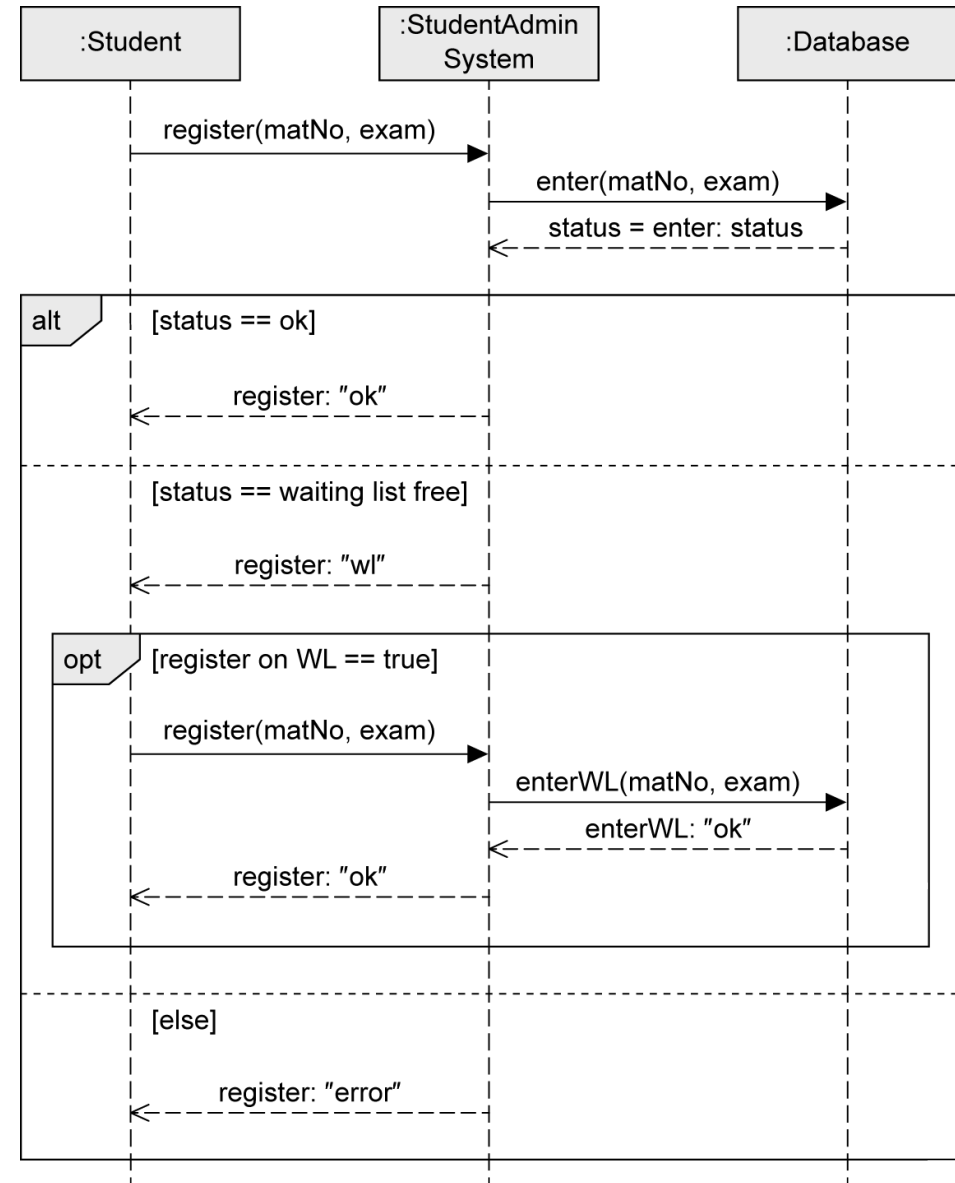*Operands*

Branches and loops

- **Alternative interactions**
  - Switch

- **Optional interactions**
  - "if" without an "else"

- **Iterative interactions**
  - For loop

- **Exception interactions**
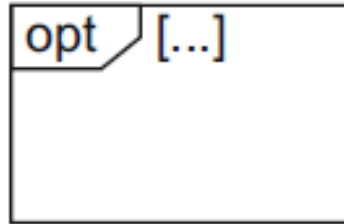 - Omit the remaining
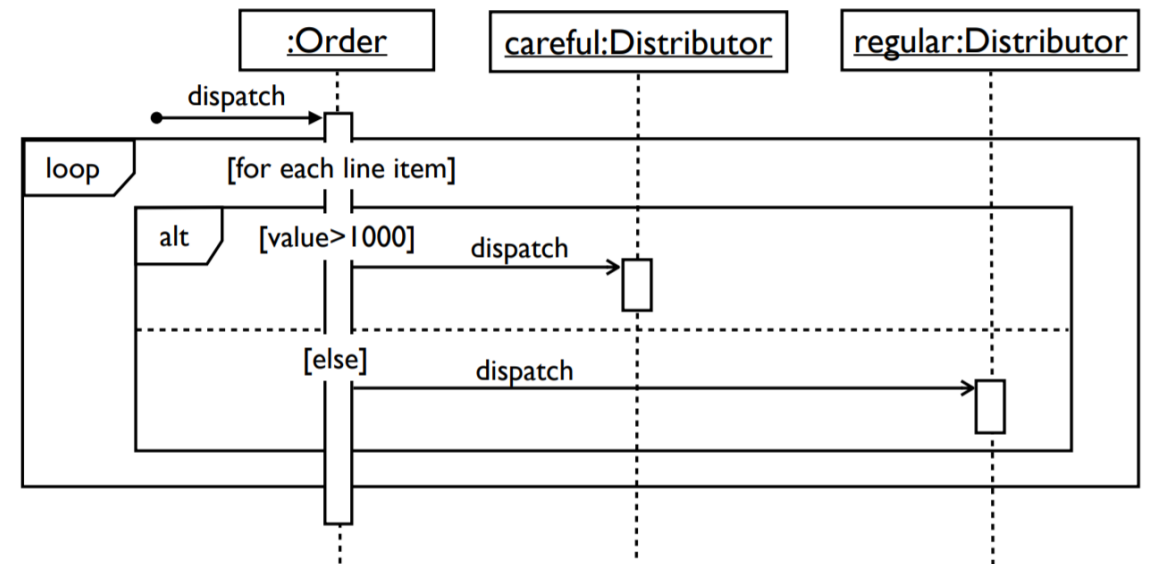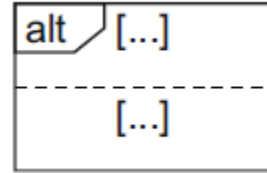
`alt` operator

- To model alternative sequences
- Similar to switch statement in Java
- Guards are used to select the one path to be executed
- Guards
  – Modeled in square brackets
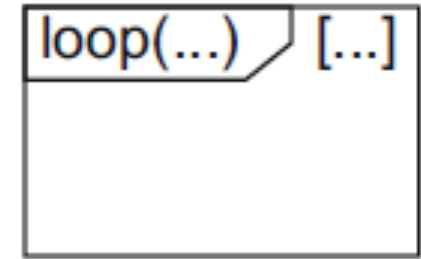 – default: `true`
  – predefined: `[else]`

opt operator

- To model an optional sequence
- Actual execution at runtime is dependent on the guard
- Exactly one operand
- Similar to **if** statement **without else** branch
- equivalent to **alt** fragment with two operands, one of which is empty
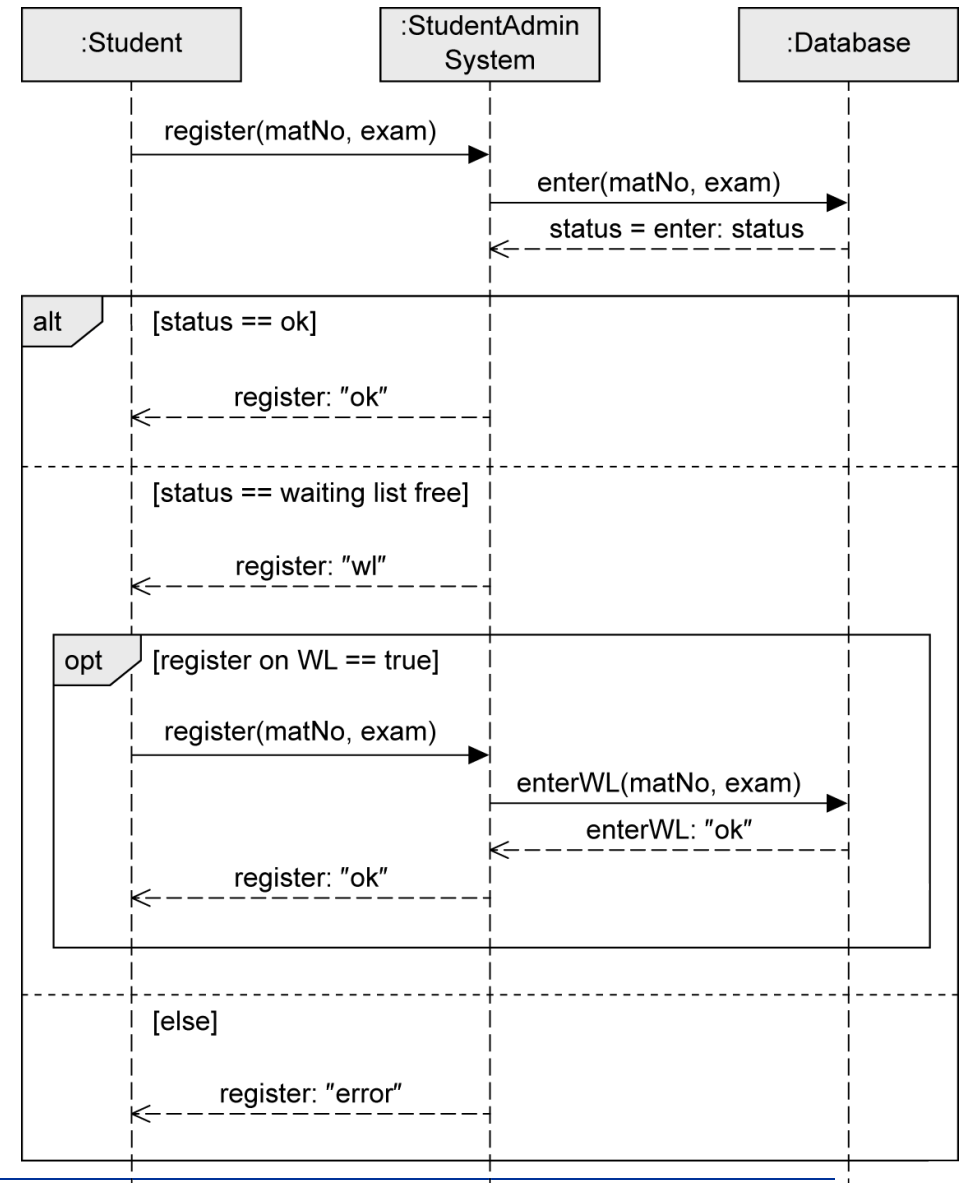
`loop` Operator

- To express that a sequence is to be executed repeatedly

- Exactly one operand

- Keyword loop followed by the minimal/maximal number of iterations **`(min..max`**) or (**`min,max`**)
  - default: **`(*)`** .. no upper limit

- Guard
  - Evaluated as soon as the minimum number of iterations has taken place
  - Checked for each iteration within the **`(min,max)`** limits
  - If the guard evaluates to false, the execution of the loop is terminated
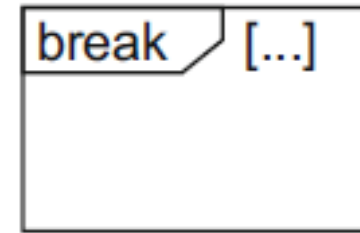
`loop` Operator (2)

loop(...) [...]

Max

Min
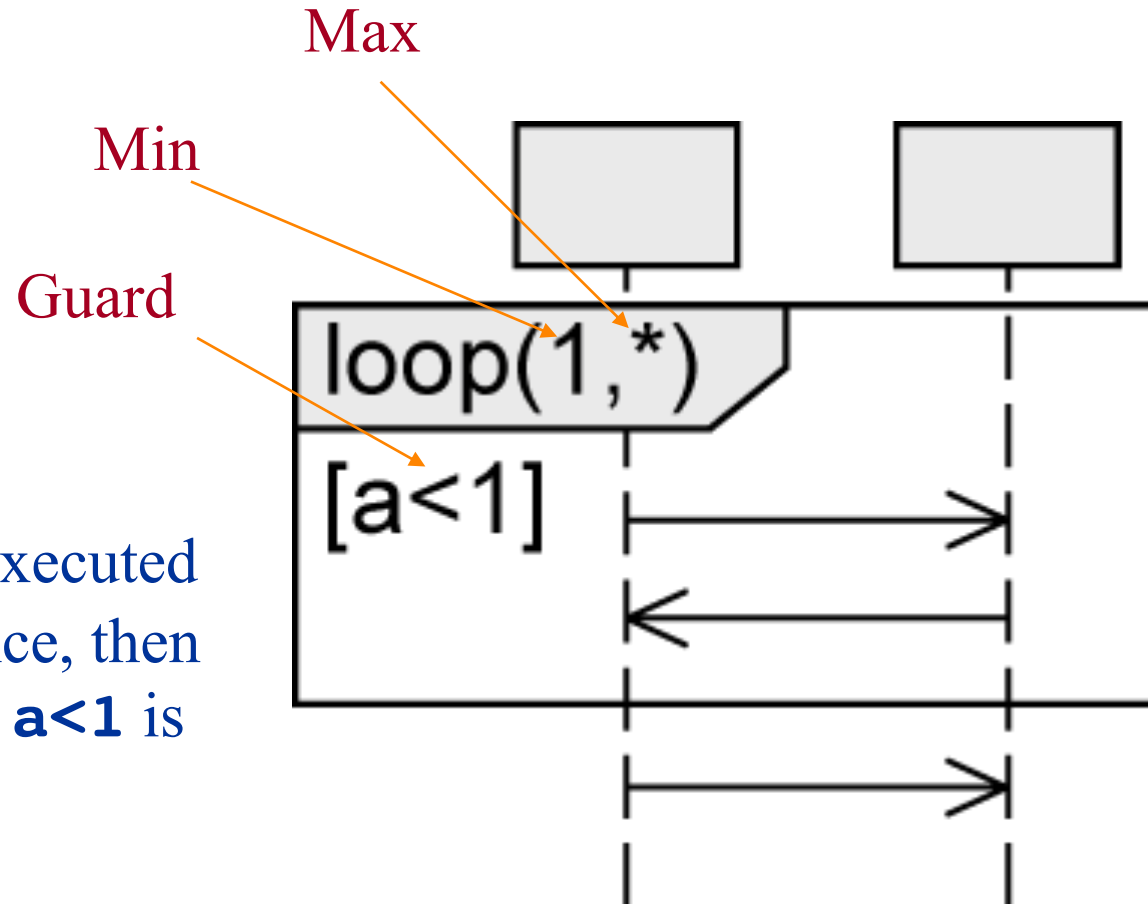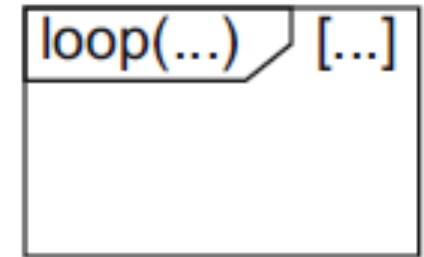
Guard

loop is executed at least once, then as long as **a<1** is true

loop(1,*)

[a<1]

Notation alternatives:

loop(3,8) = loop(3..8)
loop(8,8) = loop (8)
loop (*) = loop(0,*)

break Operator

- Simple form of exception handling
- Exactly one operand with a guard
- If the guard is true:
  - Interactions within this operand are executed
  - Remaining operations of the surrounding fragment are omitted
  - Interaction continues in the next higher level fragment
  - Different behavior than **opt** fragment
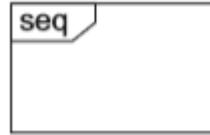
Not executed if break is executed
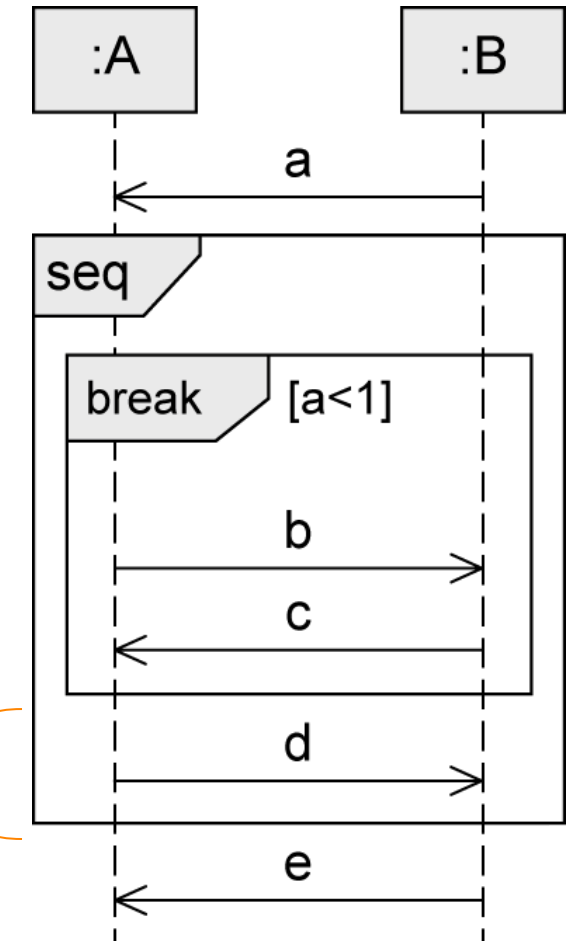
break Operator (2)
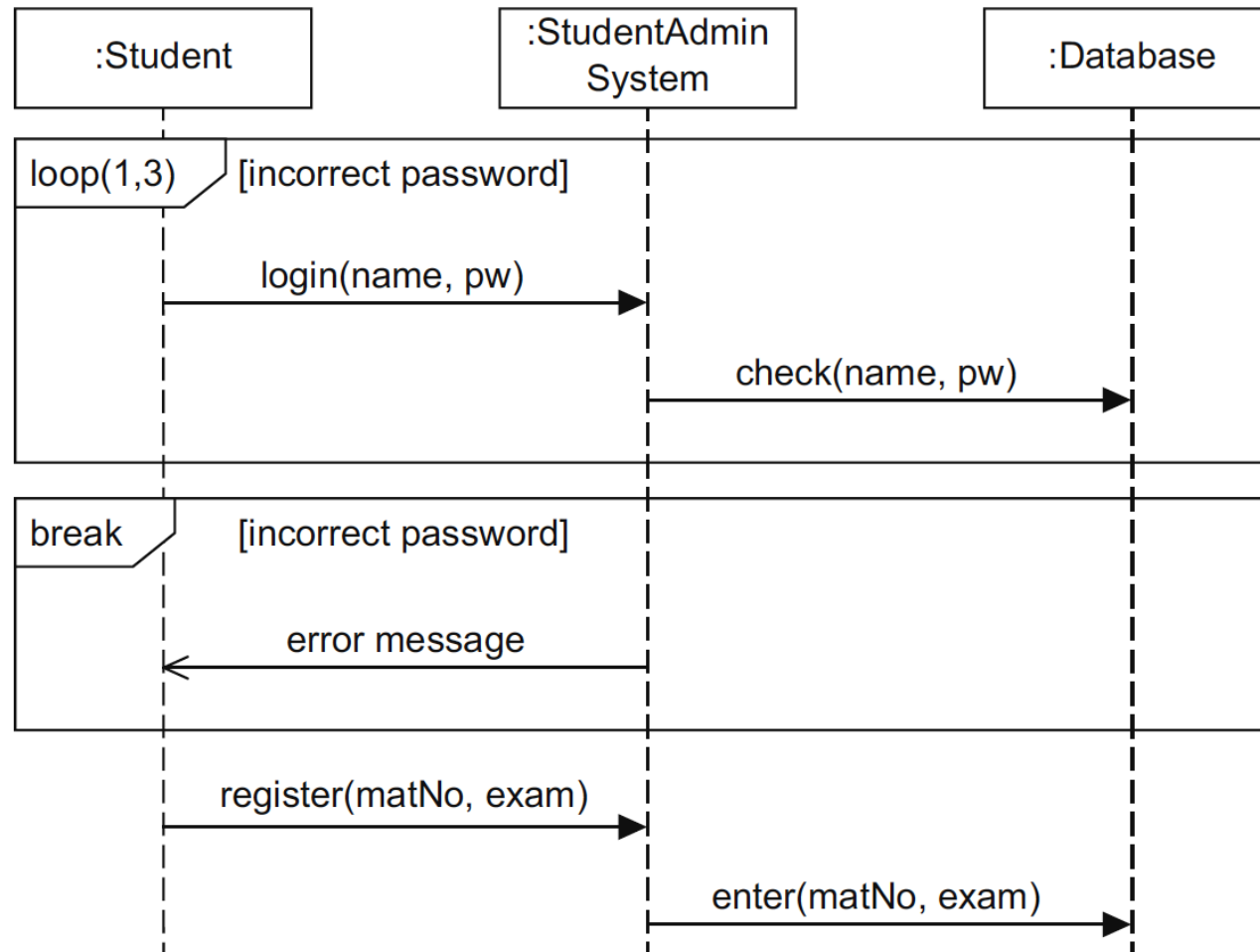
Concurrency and Order: `Seq`

- `Seq` fragment
  seq

  – Weak order
  – The ordering of events within each of the operands is maintained in the result.
  – Events on different lifelines from different operands may come in any order.
  – Events on the same life line from different operands are ordered such that an event of the first operand comes before that of the second operand.
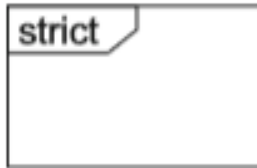
a->b->d
c->d->e

```
:A   :B   :C   :D
seq
     a →
     b
   ←
              ← c
        ← d
              e →
```
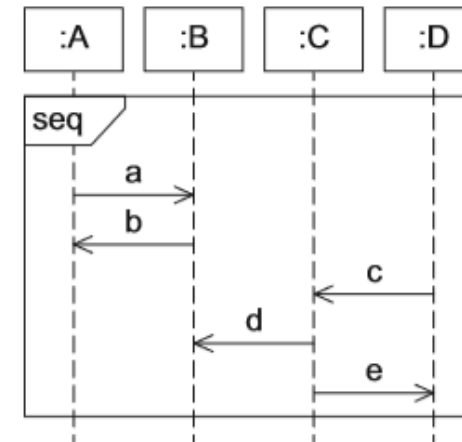
Traces:
T01: a → b → c → d → e
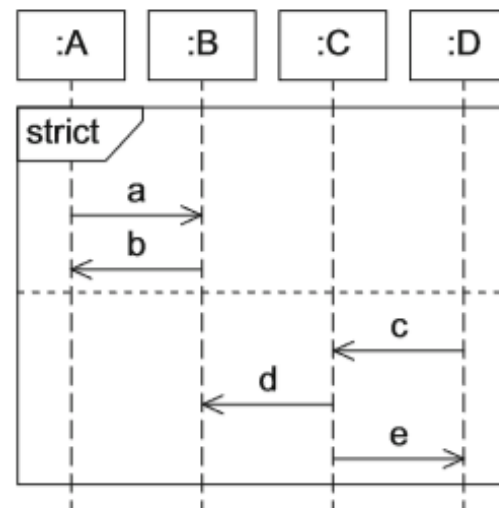T02: a → c → b → d → e
T03: c → a → b → d → e

Concurrency and Order (2): `Strict`

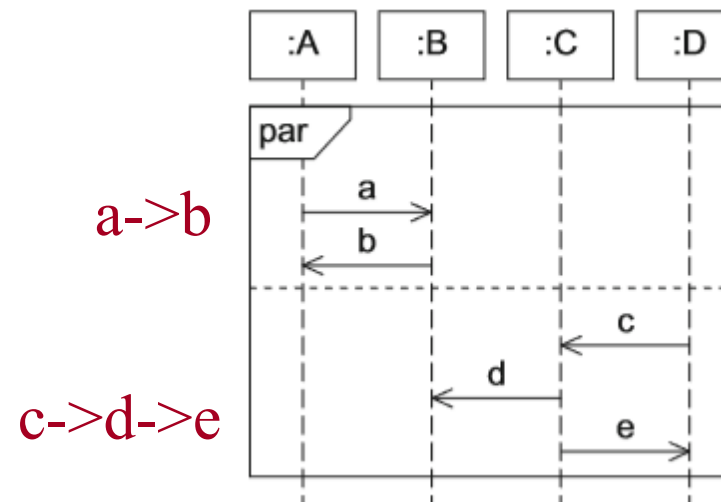- `Strict` fragment
  - Strong & strict order
  - Messages in an operand that is higher up on the vertical axis are always exchanged before the messages in an operand that is lower down on the vertical axis

Traces:
T01: a → b → c → d → e

Concurrency and Order (3): `Par`

- `Par` fragment
  - Order within the operands are respected
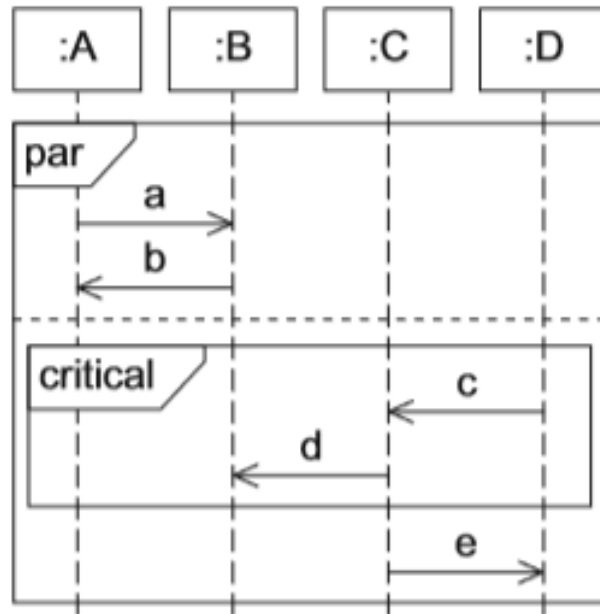 - The order of operands does not matter

a->b

c->d->e

Traces:
T01: a → b → c → d → e
T02: a → c → b → d → e
T03: a → c → d → b → e
T04: a → c → d → e → b
T05: c → a → b → d → e
T06: c → a → d → b → e
T07: c → a → d → e → b
T08: c → d → a → b → e
T09: c → d → a → e → b
T10: c → d → e → a → b

Concurrency and Order (4): `Critical`

- ## `Critical` fragment

  - Atomic interaction
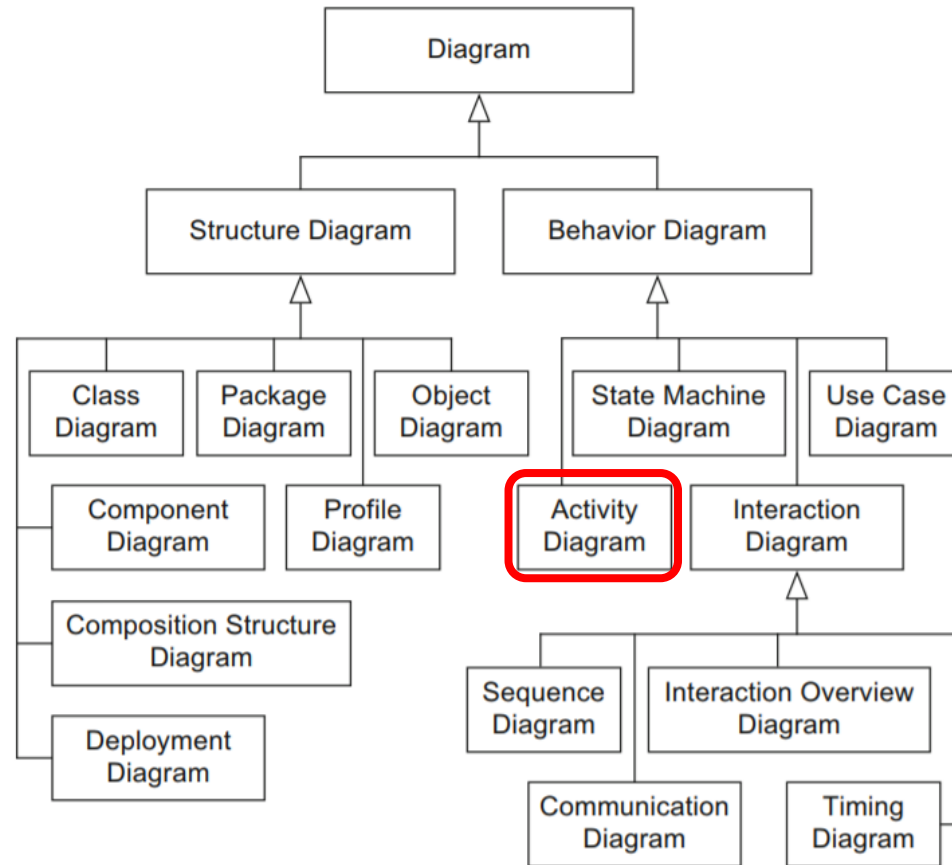
  - No other messages can happen during the execution

Example

StudentAdminSystem

+ newRegistration(date: Date, matNo: int, courseNo: int): void
+ getCourse(courseNo: int): Course
+ getStudent(matNo: int): Student

Course

+ setRegistration(registration: Registration): void

Student

Registration

+ setDate(date: Date): void
+ init(student: Student, course: Course): void

:StudentAdmin System

course:Course

student:Student

newRegistration (date, matNo, courseNo)

course = get Course(courseNo)

getStudent(matNo)

new()

registration :Registration

init(student, course)

setDate(date)

setRegistration (registration)

Activity Diagram

UML Diagrams

Activity Diagram

- What are the procedures of a system?

- At conceptual level: How to implement use case?

- At implementation level: How to implement an operation

- A flow-oriented language

Activity Diagram (2)

- An activity diagram allows you to specify user-defined behavior in the form of activities.

- An *activity* itself can describe the implementation of a use case.

- Activity
  - Parameters
  - Precondition
  - Postcondition
  - Actions
  - Edges

Input parameter

Output parameter

Activity name
«precondition»
«postcondition»
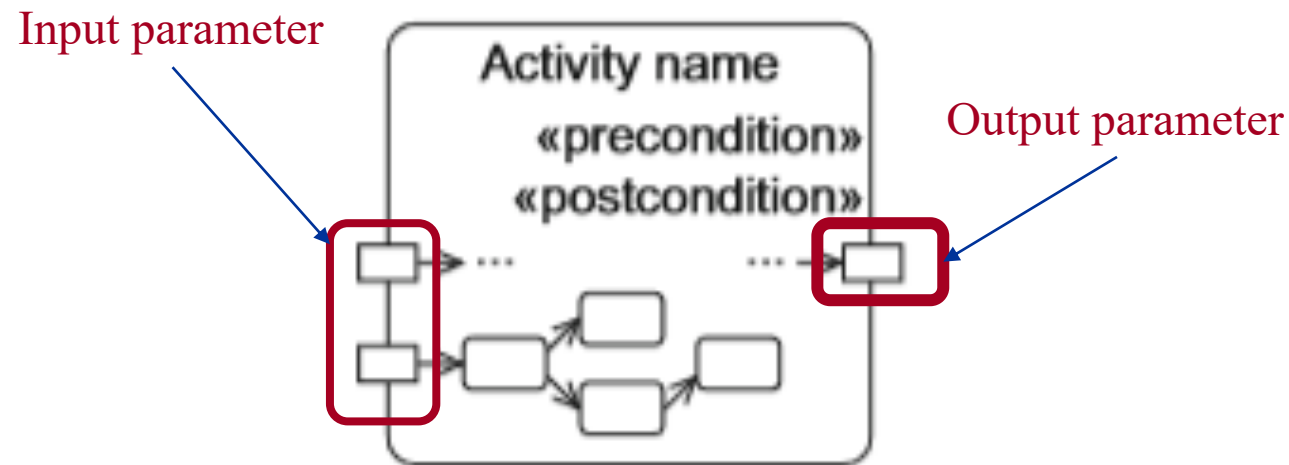
Precondition and Postcondition

- Precondition: the conditions have to be fulfilled before the activity is executed;

- Postcondition: the conditions have to be fulfilled after the activity is executed;

- E.g, Take exam
  - precondition: student is enrolled;
  - postcondition: student is graded
    (after take the exam)
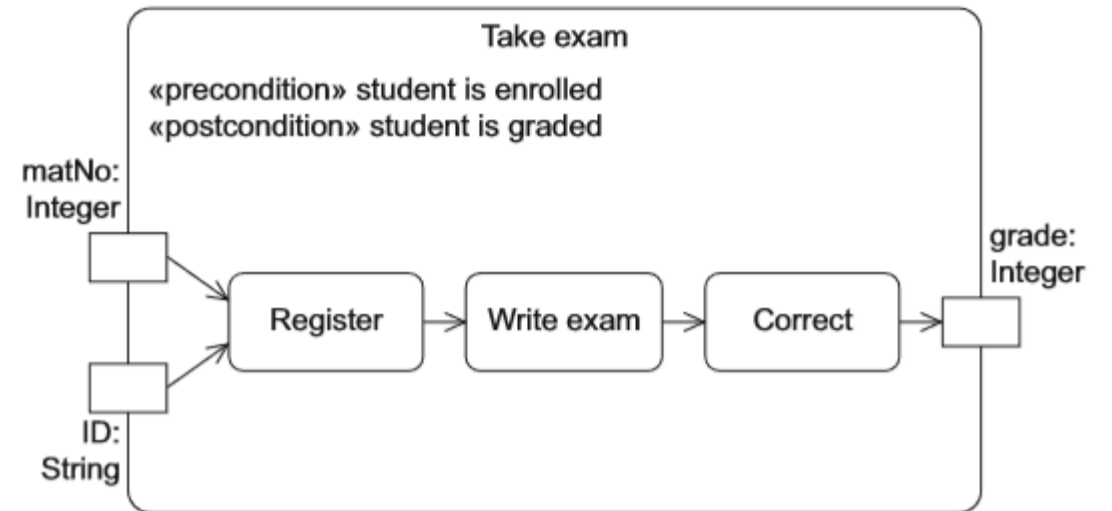
Actions
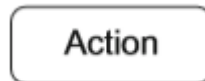
- Activity
  - Parameters
  - Precondition
  - Postcondition
  - Actions
    - No language restrictions
    - Atomic: may be further broken down in other contexts
  - Edges
    - Control flow edge: order between actions
    - Object flow edge: can exchange data
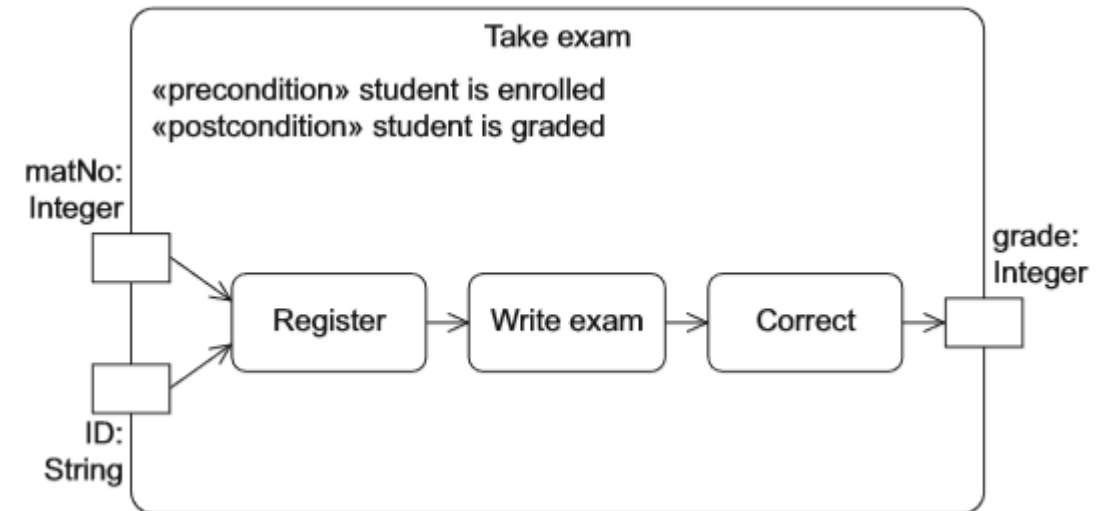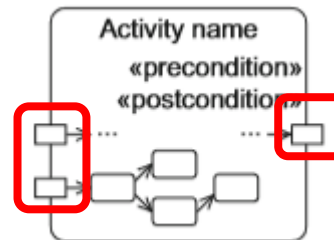
Actions (2)

Action

- The basic elements of activities are actions.

- An action is depicted as a rectangle with rounded corners.

  – The name of the action is positioned centrally within the rounded rectangle.

- There are no specific language requirements for the description of an action.

  – You can define the actions in natural language or in any programming language

Edges
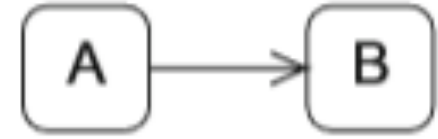
- Connect activities and actions to one another
- Types
  - Control flow edges
    - Define the order between nodes
  - Object flow edges
    - Used to exchange data or objects
    - Express a data/causal dependency between nodes
- Guard (condition)
  - Control and object flow only continue if guards in square brackets evaluate to true

Control flow

Object flow
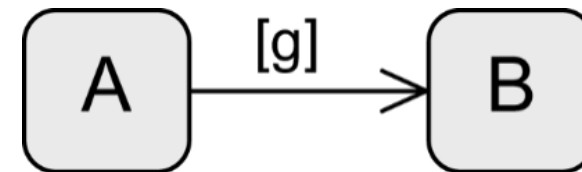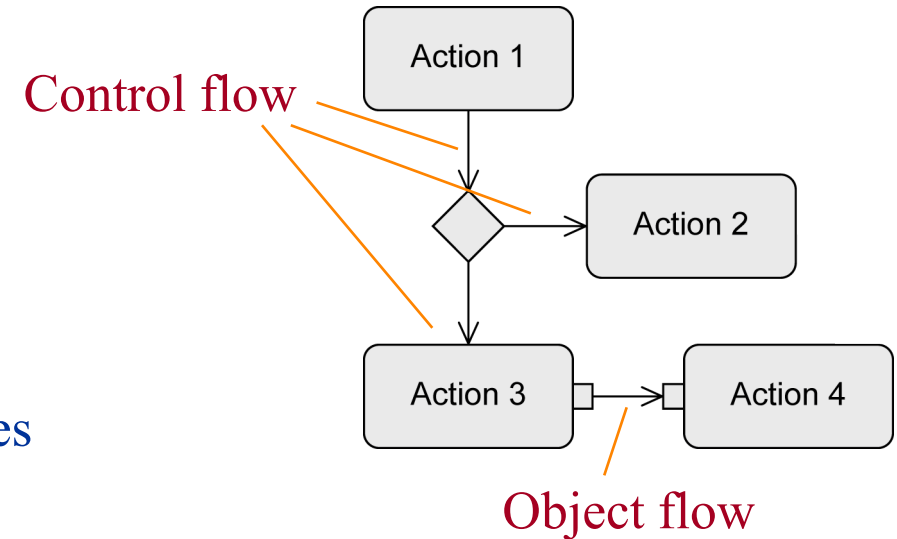
Predefined Actions: Event-based

- Accept event action

  - Wait for a specific event E

  - Use an accept event action to model an action that waits for the occurrence of a specific event.

  - E.g., Whenever a fire alarm is triggered, the lecture hall must be evacuated.
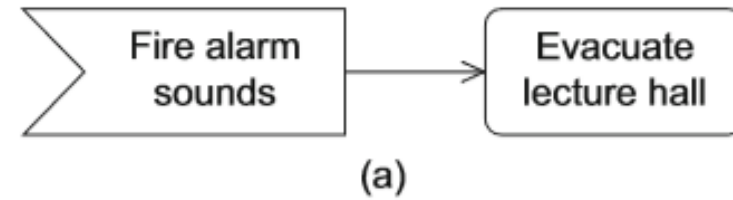
- Accept time event action

  - start when the corresponding event occurs.

  - E.g., At the end of a semester, certificates are issued.
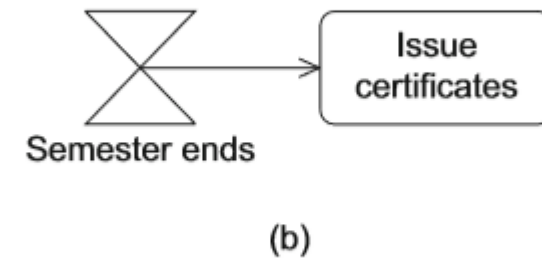
Predefined Actions: Event-based (2)

- Send signal action

  - To send signals, you can use send signal actions. Send signal actions are denoted with a "convex pentagon"—a rectangle with a tip that protrudes to the right.
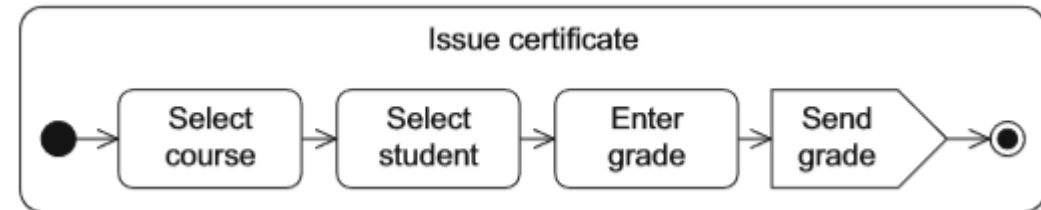
Predefined Actions: Call behavior actions

- Call behavior actions

- Actions can call activities themselves. These actions are referred to as call behavior actions and are marked with an inverted fork symbol.

- It symbolizes that the execution of this action starts another activity, thus dividing the system into various parts.

Name of the called activity

Inverted fork symbol
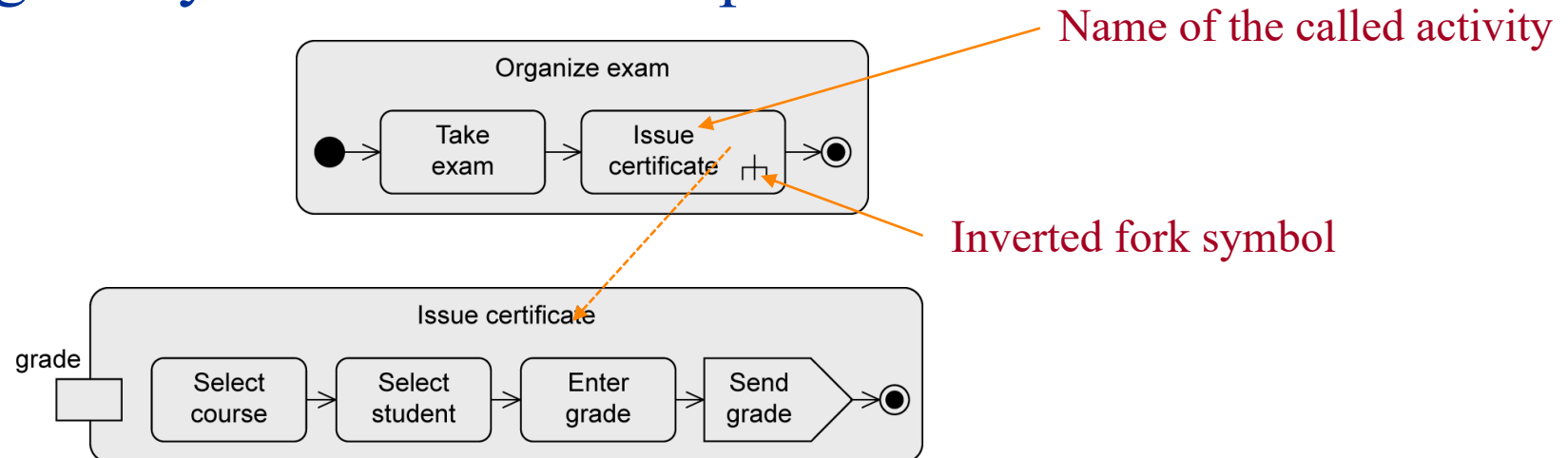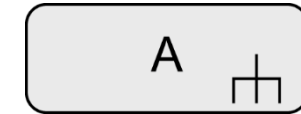
Control Flow

- Connector
  - Just to make the diagram clearer

- Control nodes
  - Initial node ●
  - Activity final node →◉
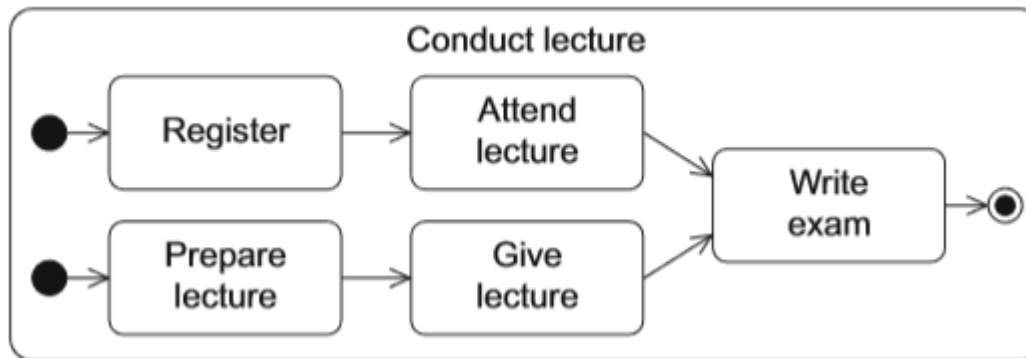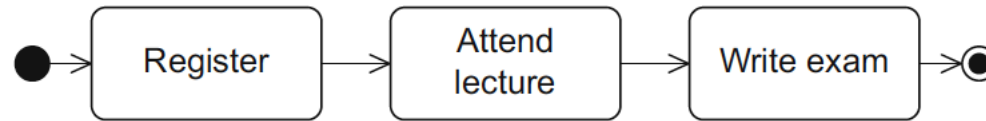
Control Flow

- ## Control nodes
  - Decision node

  - Decision behavior
    - Save space & provide clarity

 - Merge node

«decisionInput»
decision behavior

[no]

Exam
result

[yes]

Inspect
exam paper

Issue
certificate

«decisionInput»
Allow inspection
of exam paper?
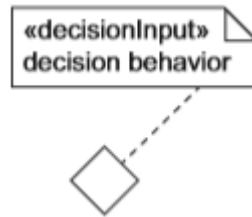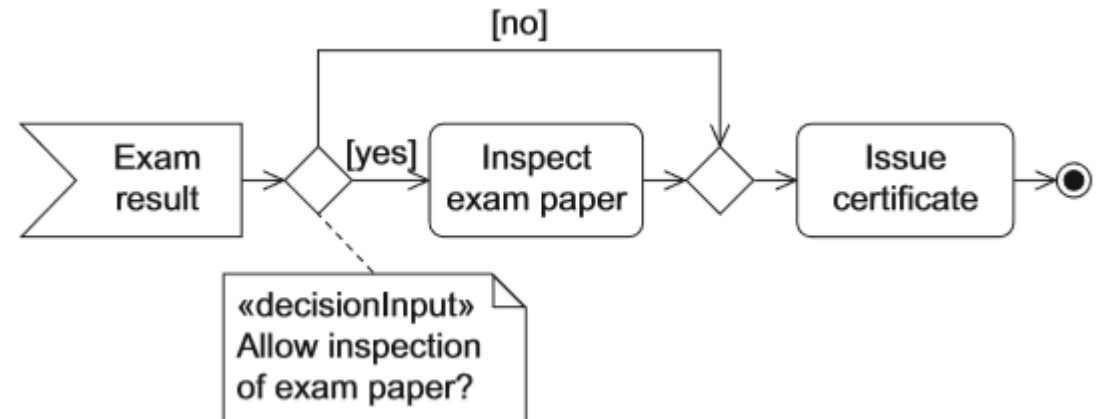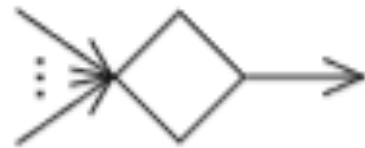
Control Flow (2)

- Parallelization & Synchronization node
  - Decision node can take only one edge

- If an execution path splits into multiple simultaneously active execution paths later on, you can realize this using a *parallelization* node.

- You can merge concurrent subpaths using a *synchronization* node. This node is the counterpart to the parallelization node
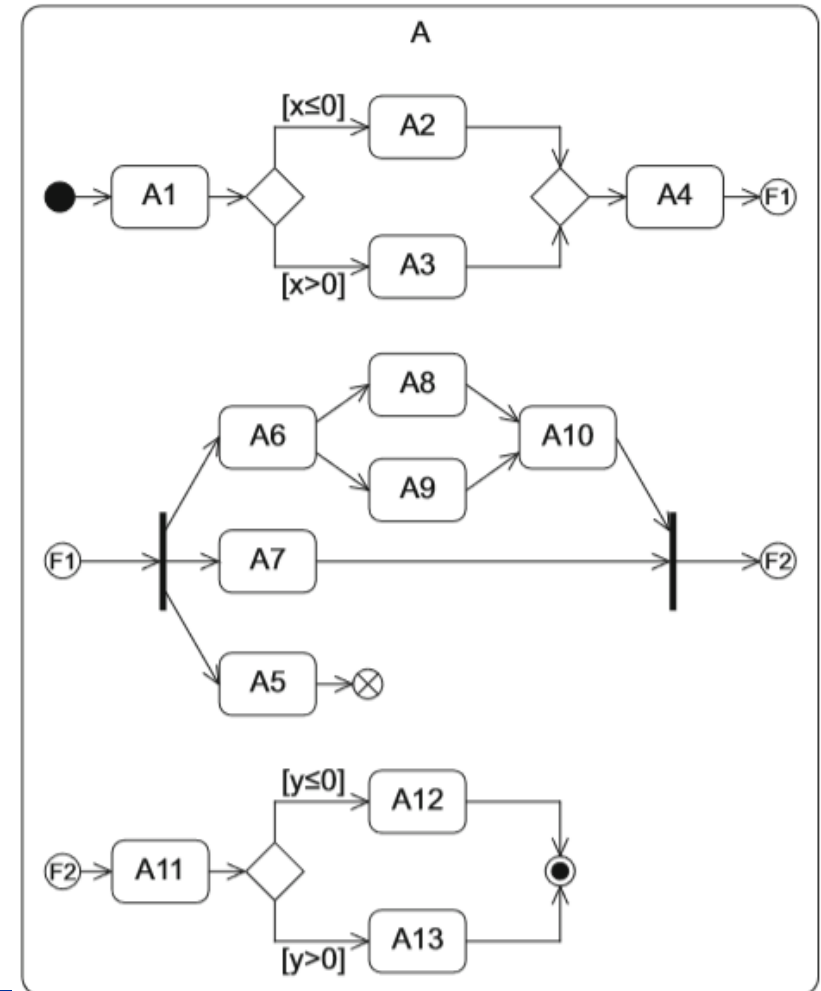
Parallelization node
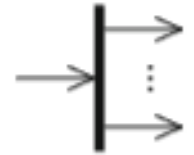
Synchronization node
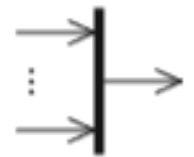
Control Flow (3)

- Activity final node ◉

  – Multiple final node: first reached final node terminates the activity

- Flow final node ⊗

  – For concurrent activities only
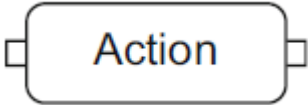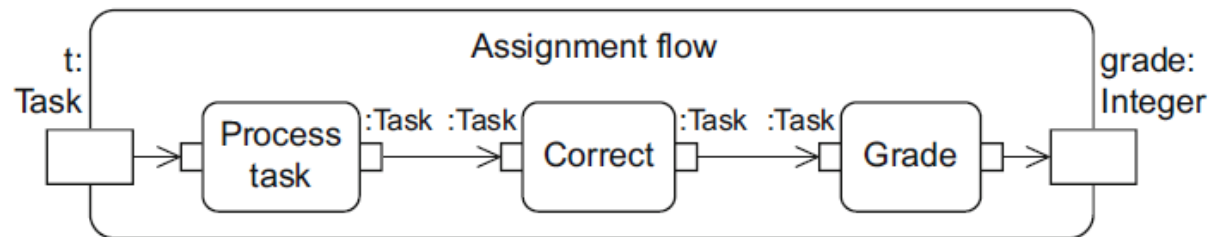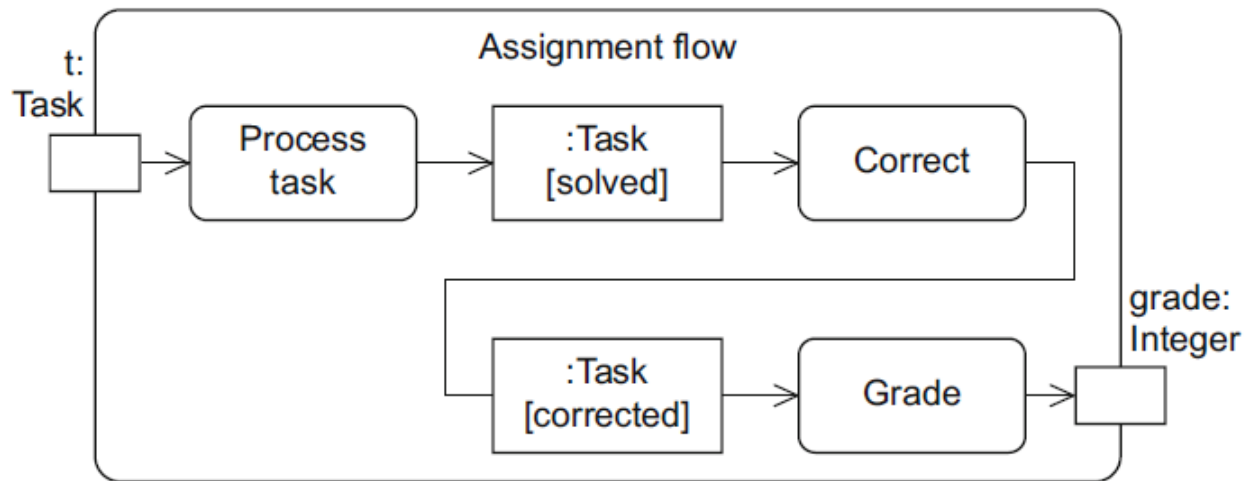
  – Only terminate one concurrent path
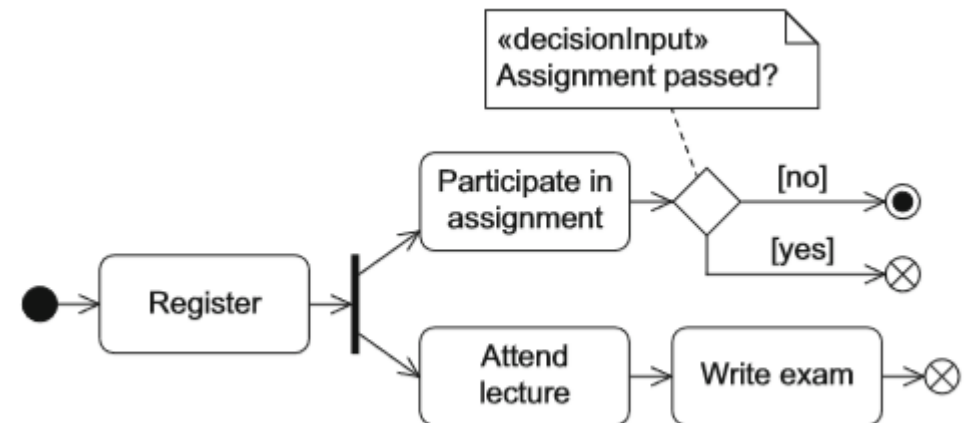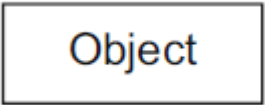
Example: Control flow

- If A5 is still executing when the activity final node is reached, A5 is interrupted
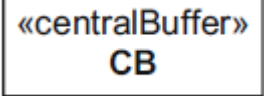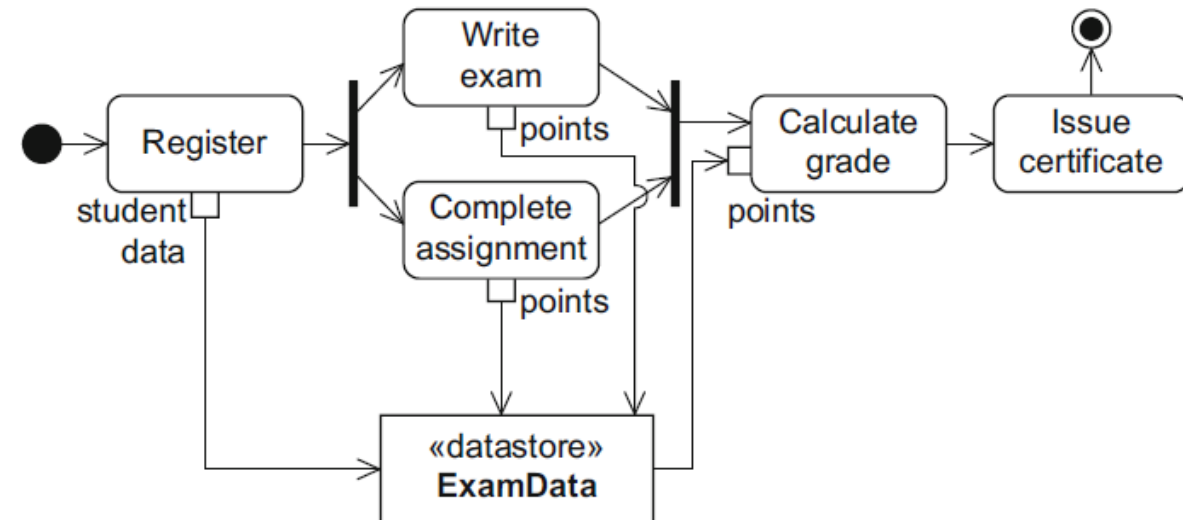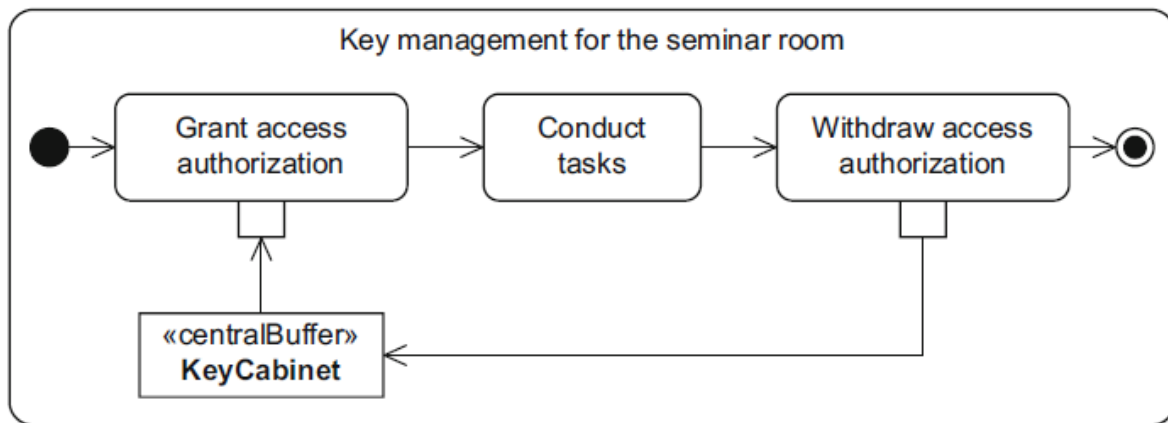
Object Flow: Syntax

- Exchange of data among actions
- Object
- Pin notation
  - Parameters are also objects

Object Flow: Syntax (2)

- ## Central buffer «centralBuffer» **CB**

  - – Data exited the central buffer is no longer in there

- ## Data store «datastore» **DS**

  - – Data exited the data store still has a copy in there
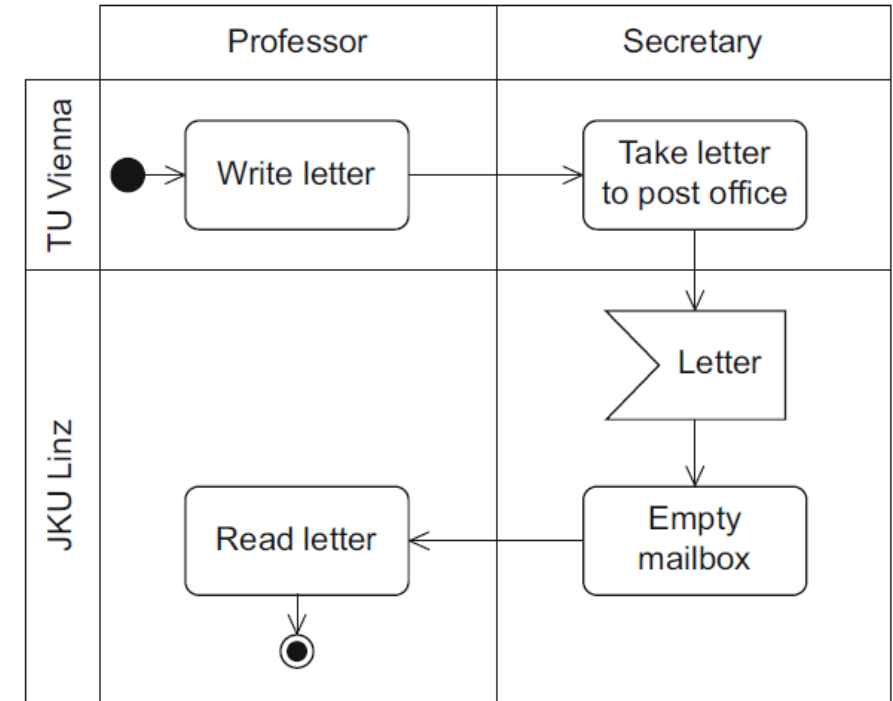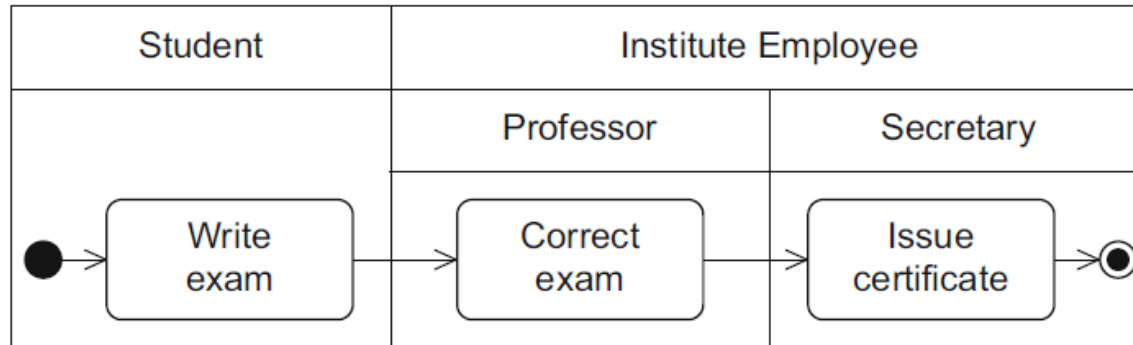
Swimlane/Partition

- Group actions in terms of who's performing them
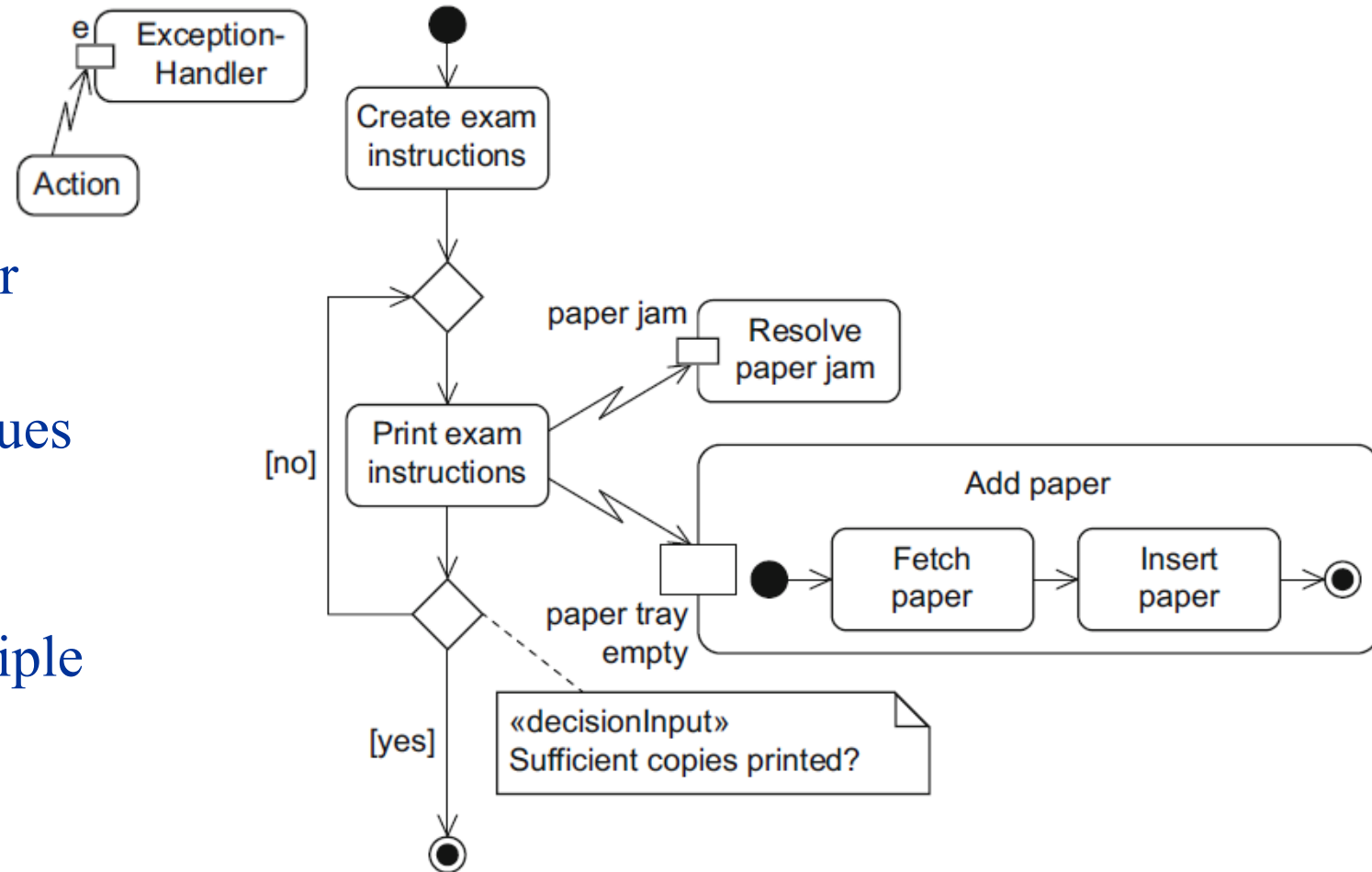- A much clearer view of the activity diagram
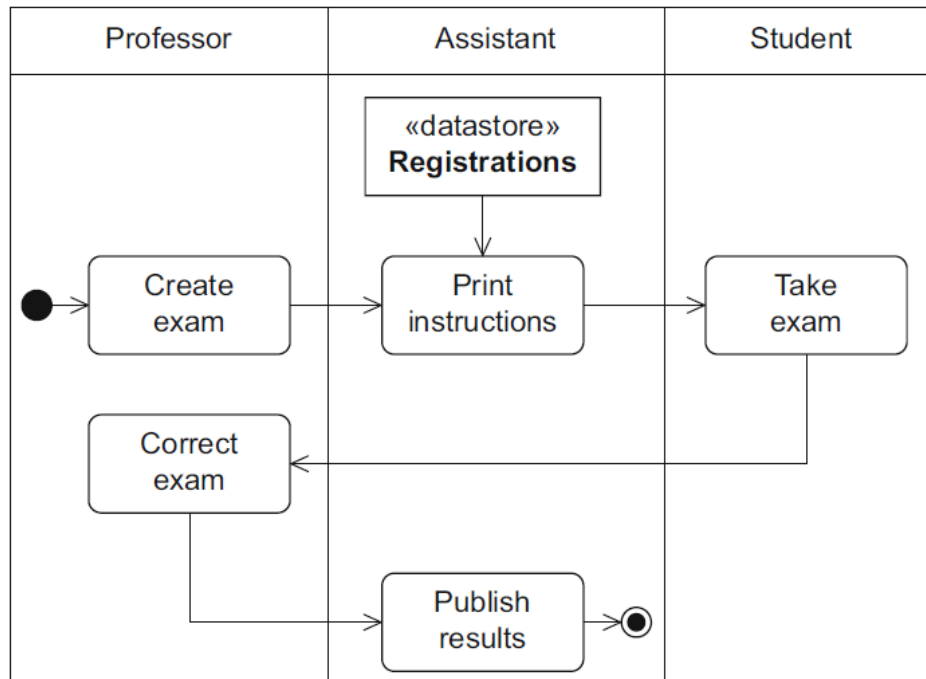
Swimlane (2)

- Swim lanes can have sub-partitions
- Swim lanes can also have multiple dimensions

Exception Handling

- Exception handler node
  - Error situation e
  - Execute exception handler when e happens
  - Then the sequence continues as if the action ended normally
 - One action can have multiple exception handlers

Interruptible activity region

- Interruptible activity region
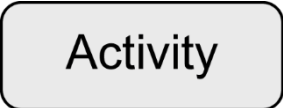- Activities within the dashed region terminate immediately when E occur
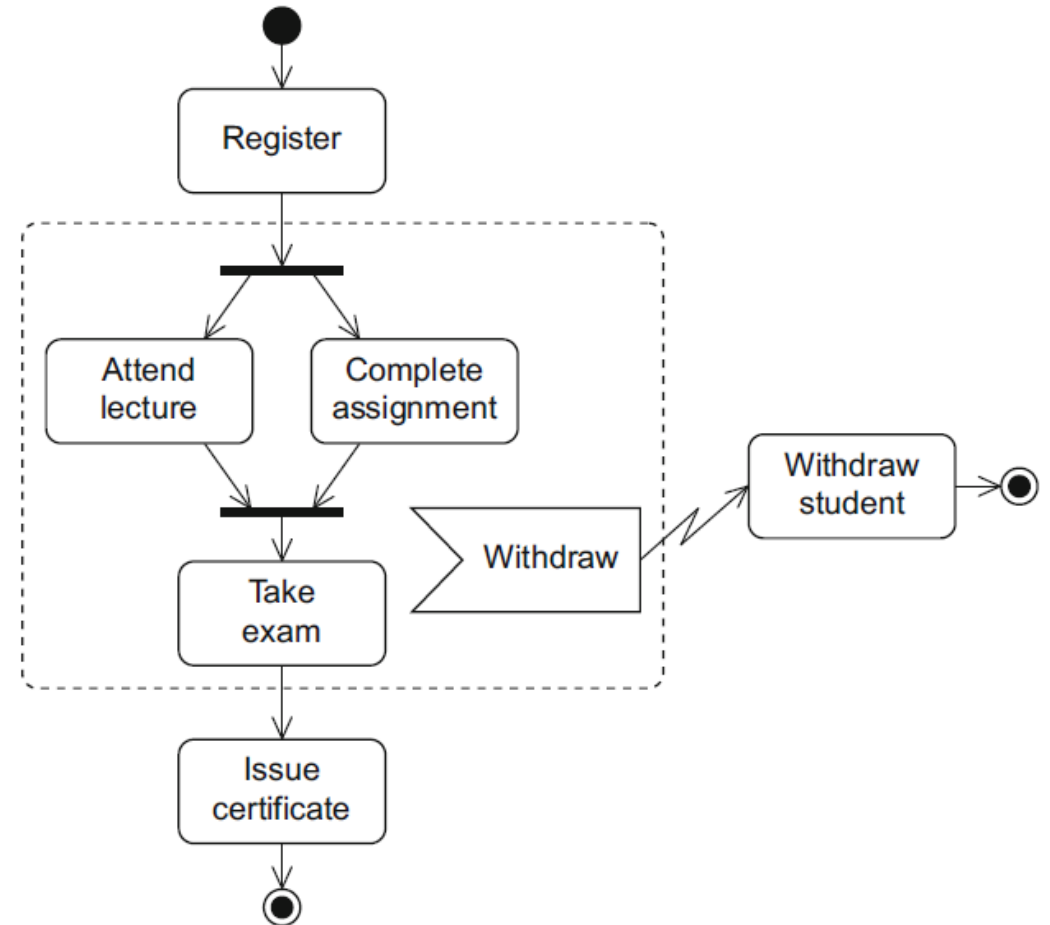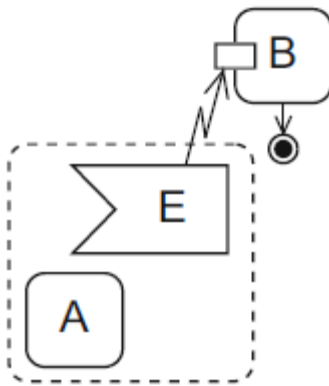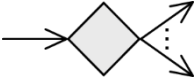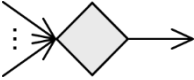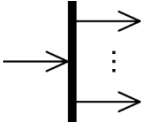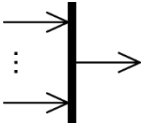
Example: Student ID

- To obtain a student ID, the student must request this ID from an employee of the student office.

- The employee hands the student the forms that the student has to fill out to register at the university.

- These forms include the student ID itself, which is a small, old-style cardboard card.

- The student has to enter personal data on this card and the employee confirms it with a stamp after checking it against certain documents.

- Once the student has filled out the forms, the student returns them to the employee in the student office and hands over documents such as photo identification, school-leaving certificate, and birth certificate.

- The employee checks the documents. If the documents are incomplete or the student is not authorized to receive a student ID for the university, the process is terminated immediately.

- If the documents are all in order, the employee checks whether the student has filled out the student ID correctly.

- If there are any errors, this ID is destroyed and the student has to fill out another one. Otherwise the ID is stamped.

- However, the student ID is not valid until it bears the semester label sent to the student by post.
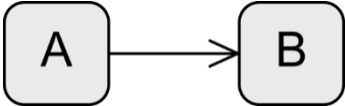
Control Flow

Object Flow
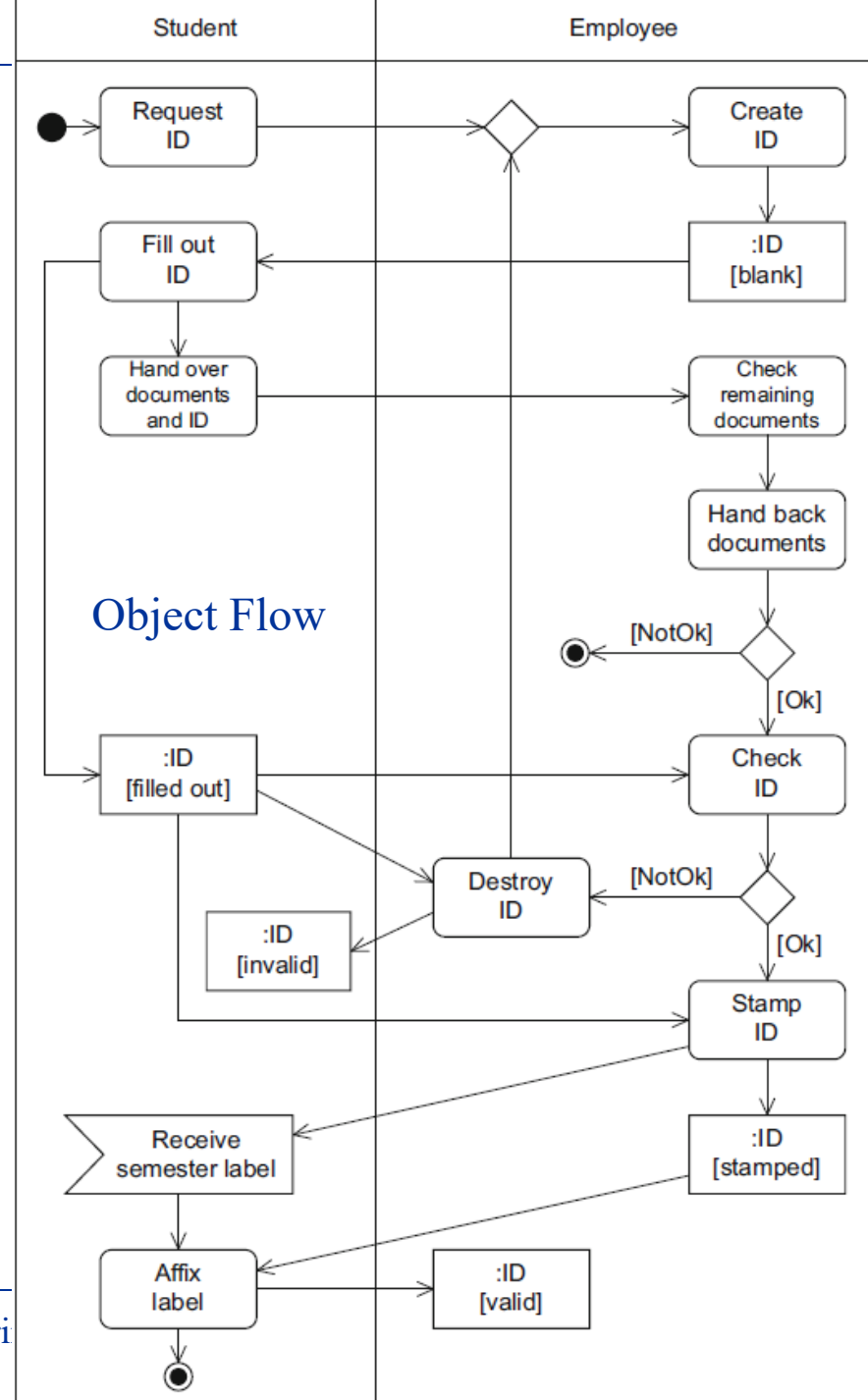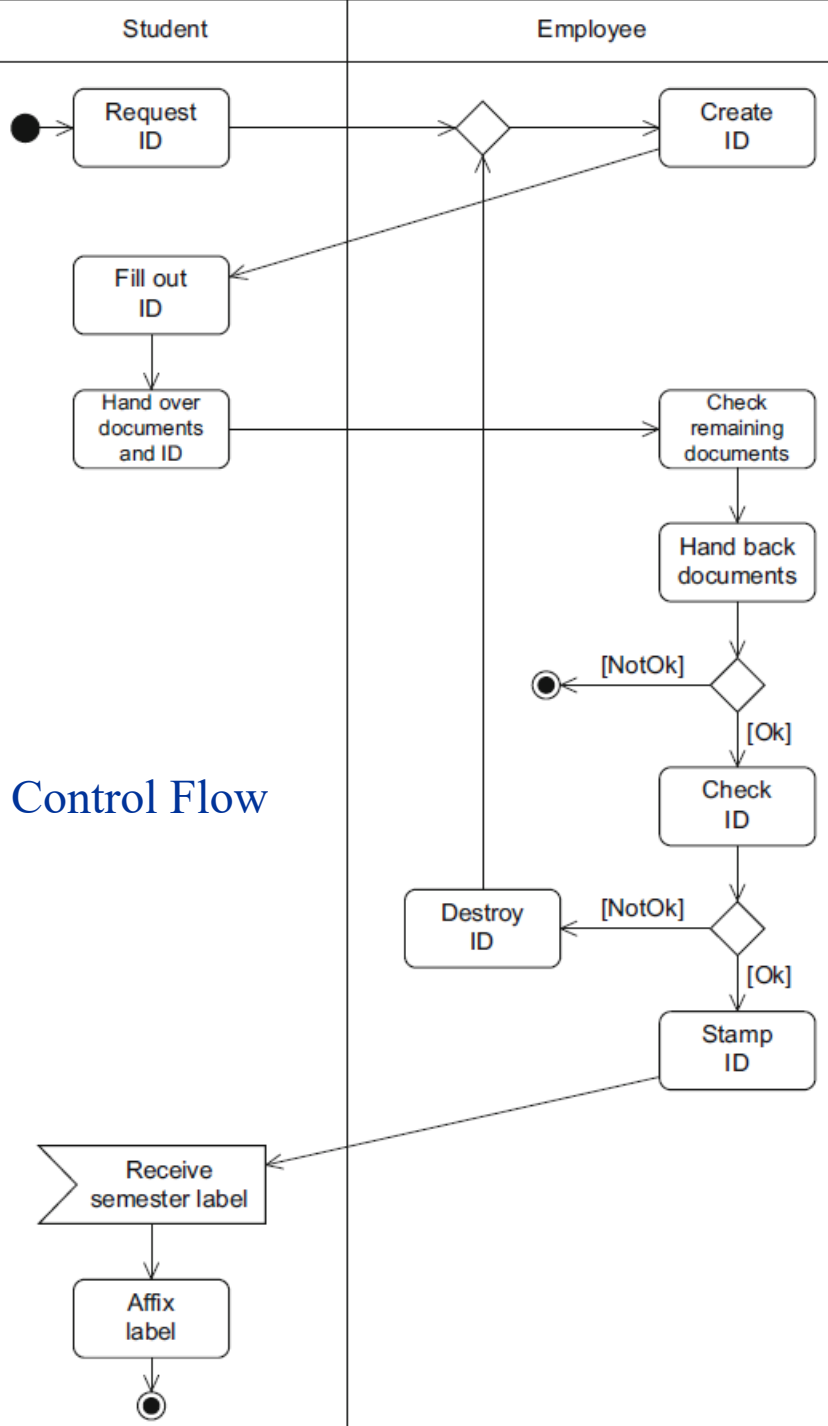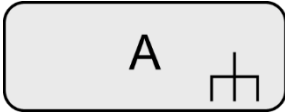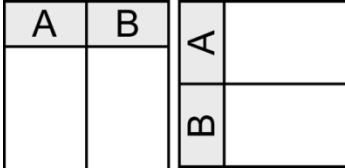
Notation Element

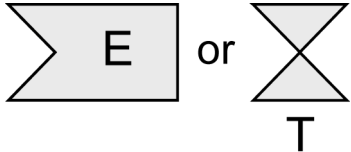| Name	Notation	Description
Action node	Action	Represents an action (atomic!)
Activity node	Activity	Represents an activity (can be broken down further)
Initial node	●	Start of the execution of an activity
Activity final node	◉	End of ALL execution paths of an activity
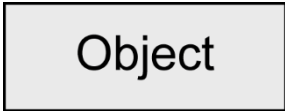
Notation Element (2)

Name	Notation	Description
Decision node		Splitting of one execution path into alternative execution paths
Merge node		Merging of alternative execution paths into one execution path
Parallelization node		Splitting of one execution path into concurrent execution paths
Synchronization node		Merging of concurrent execution paths into one execution path
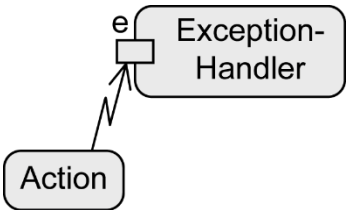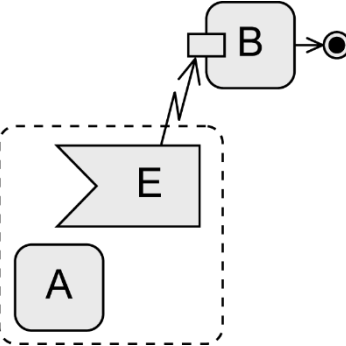
Notation Element (3)

Name	Notation	Description
Flow final node	⊗	End of ONE execution path of an activity
Edge	A → B	Connection between the nodes of an activity
Call behavior action	A	Action A refers to an activity of the same name
Partition	A B / A / B	Grouping of nodes and edges within an activity

Notation Element (4)

Name	Notation	Description
Send signal action	S	Transmission of a signal to a receiver
Asynchronous accept (timing) event action	E or ⋈ T	Wait for an event E or a time event T
Object node	Object	Contains data or objects
Parameter for activities	Activity	Contains data and objects as input and output parameters
Parameter for actions (pins)	Action	

Notation Element (5)

Name	Notation	Description
Exception Handler		Exception handler is executed instead of the action in the event of an error e
Interruptible activity region		Flow continues on a different path if event E is detected