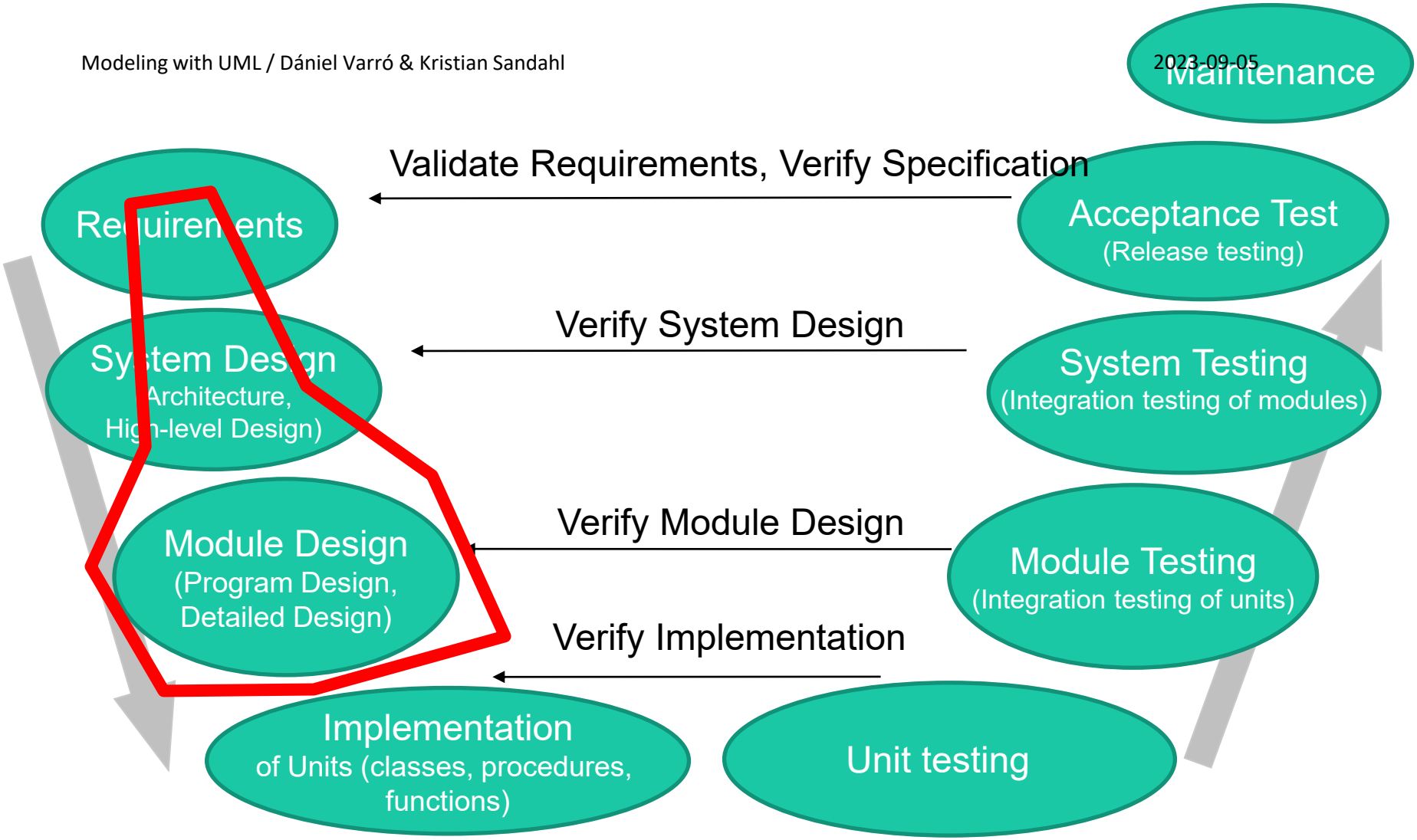
Modeling with UML

Dániel Varró / Kristian Sandahl

LINKÖPING UNIVERSITY

UML in Software Engineering


Maintenance
Validate Requirements, Verify Specification
Requirements
Acceptance Test
(Release testing)
Verify System Design
System Design
(Architecture,
High-level Design)
System Testing
(Integration testing of modules)
Verify Module Design
Module Design
(Program Design,
Detailed Design)
Module Testing
(Integration testing of units)
Verify Implementation
Implementation
of Units (classes, procedures, functions)
Unit testing
Project Management, Software Quality Assurance (SQA), Supporting Tools, Education

The goals of module design

- Provide the expected function
- Prepare for change:
  - Separation of concern
  - Testability
  - Understandability
- Contribute to quality, e.g.:
  - Performance
  - Usability
  - Reliability
  - …
- Map for the implementers, testers, and maintainers
- Provide detailed specs for the internal content and interface of a module

Modelling software

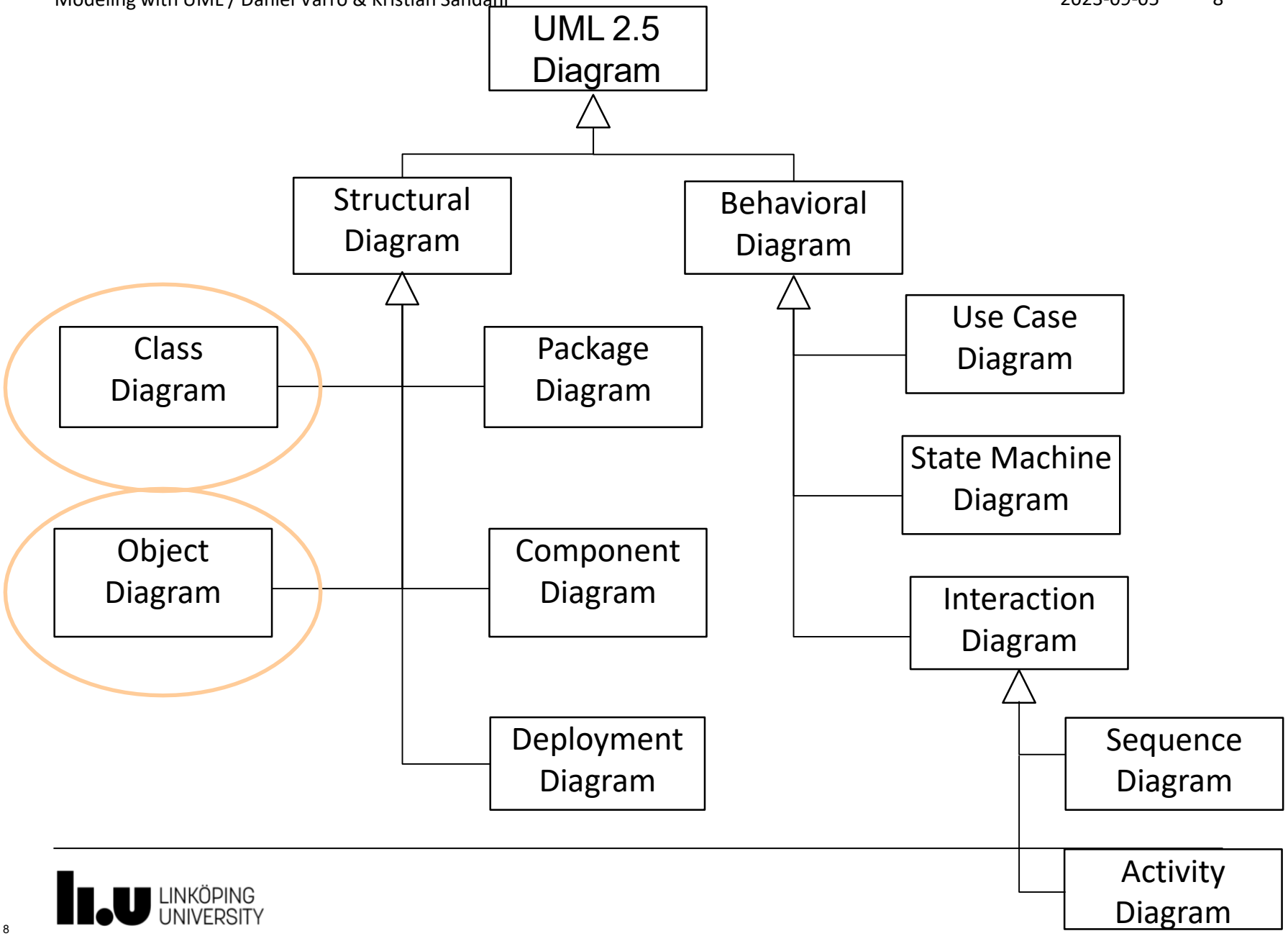
- Models **supplement** natural language
- Models support both elicitation and design
- Models can generate code and test cases
- The boundaries between specification and design have to be decided
- **UML** has become the standard notation
- Industry interest in SysML –
  extends UML (and defined in UML)

Unified Modeling Language

- Wide-spread standard of modeling software and systems
- Several diagrams and perspectives
- Often needs a text of assumptions and intenti
- Many tools tweak the standard, we use UML

UML Class and Object Diagrams

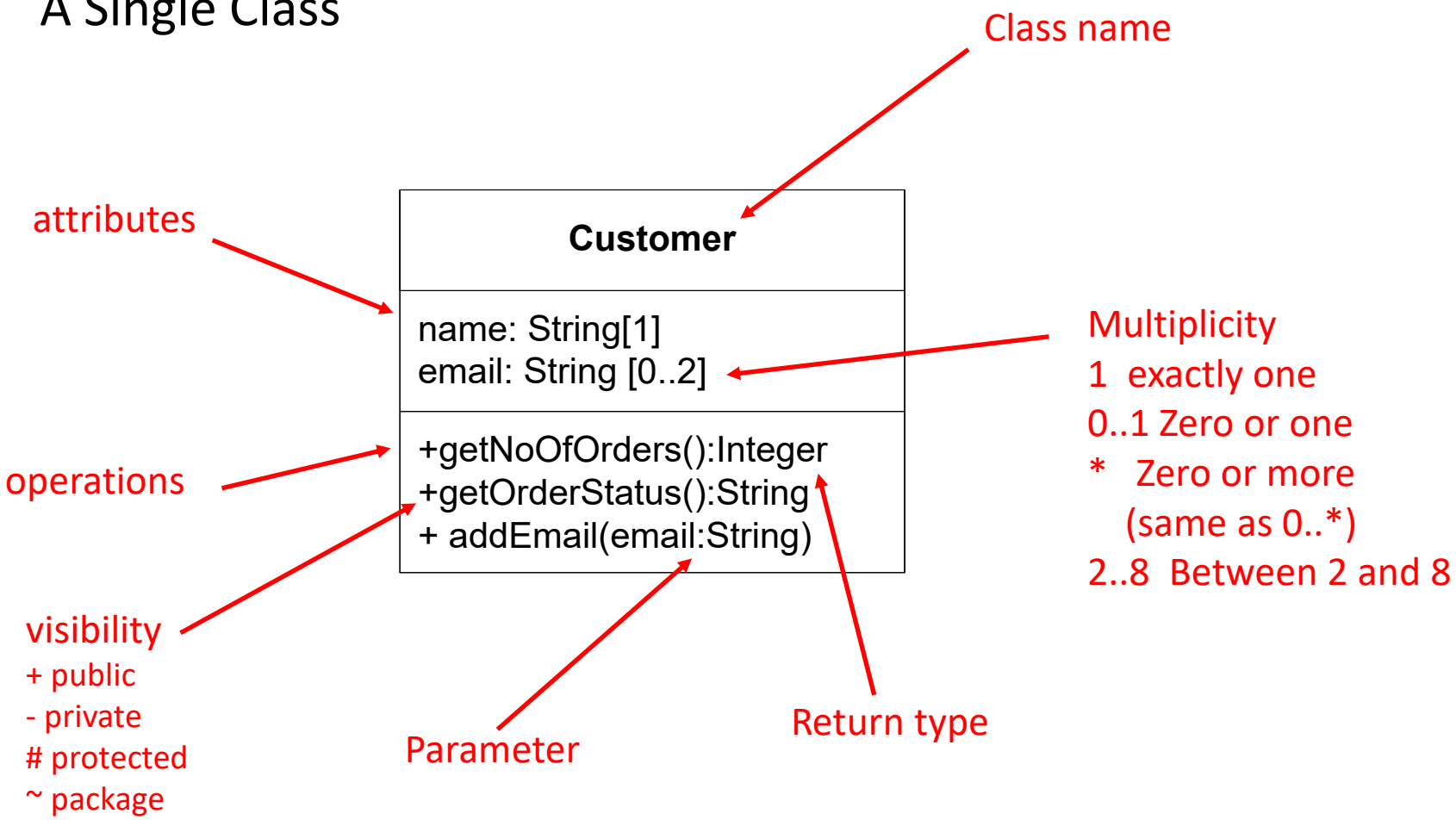
Well-known Diagrams of UML



Where to use Class diagrams?

- **Domain modeling**: Capture key concepts and relations in a domain
  - Ontologies
  - Metamodels
- **Database design**:
  - E.g. used by object-relational mappings (Hibernate)
  - User code manipulates objects → serialized in Rel DB
- **Component / module design**:
  - Internal structure of components / modules
- Defines structure of various serialization formats
  - XMI: XML Metadata Interchange (modeling tools), JSON

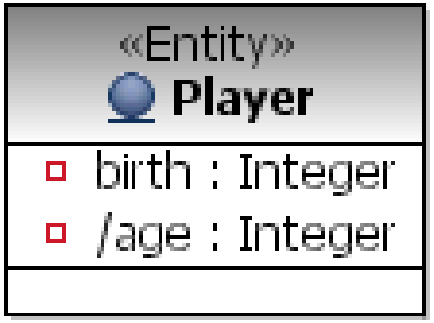
A Single Class



Attributes

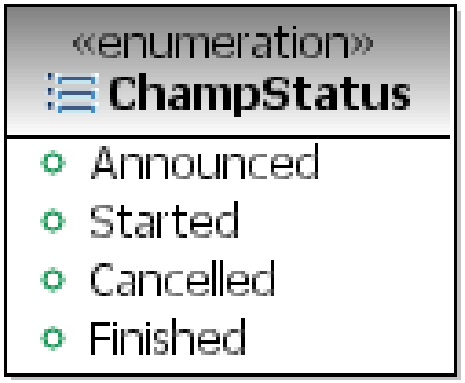
- Each attribute shall have
  - Name: e.g. birth
  - (Primitive) Type:
    - E.g. String, Integer, Real, Date, …
  - Example:
    - `Integer birth;`
- Each attribute may
  - Specify default value
  - Be derived: e.g. age
    - Calculated from other values

    `age = currYear – birth`

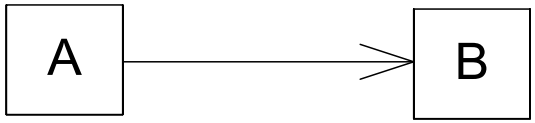


Enumerations

- Enumeration:
  - a fixed set of symbolic values
  - represented as a class with values as attributes
- Usage:
  - Frequently define possible states
  - Use enumerations instead of hard-wired String literals whenever possible

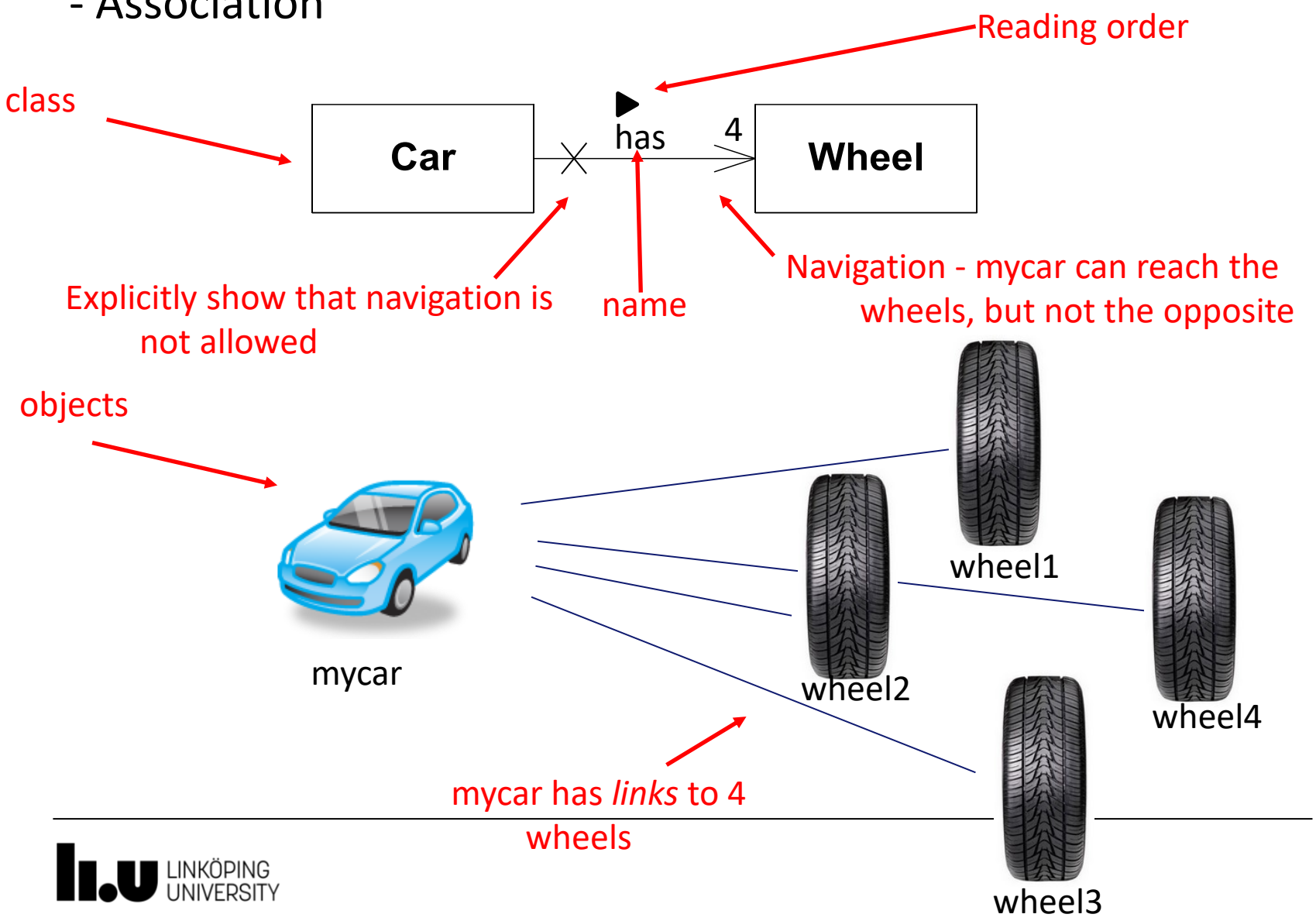


Relationships (1/6) - overview and intuition - Association

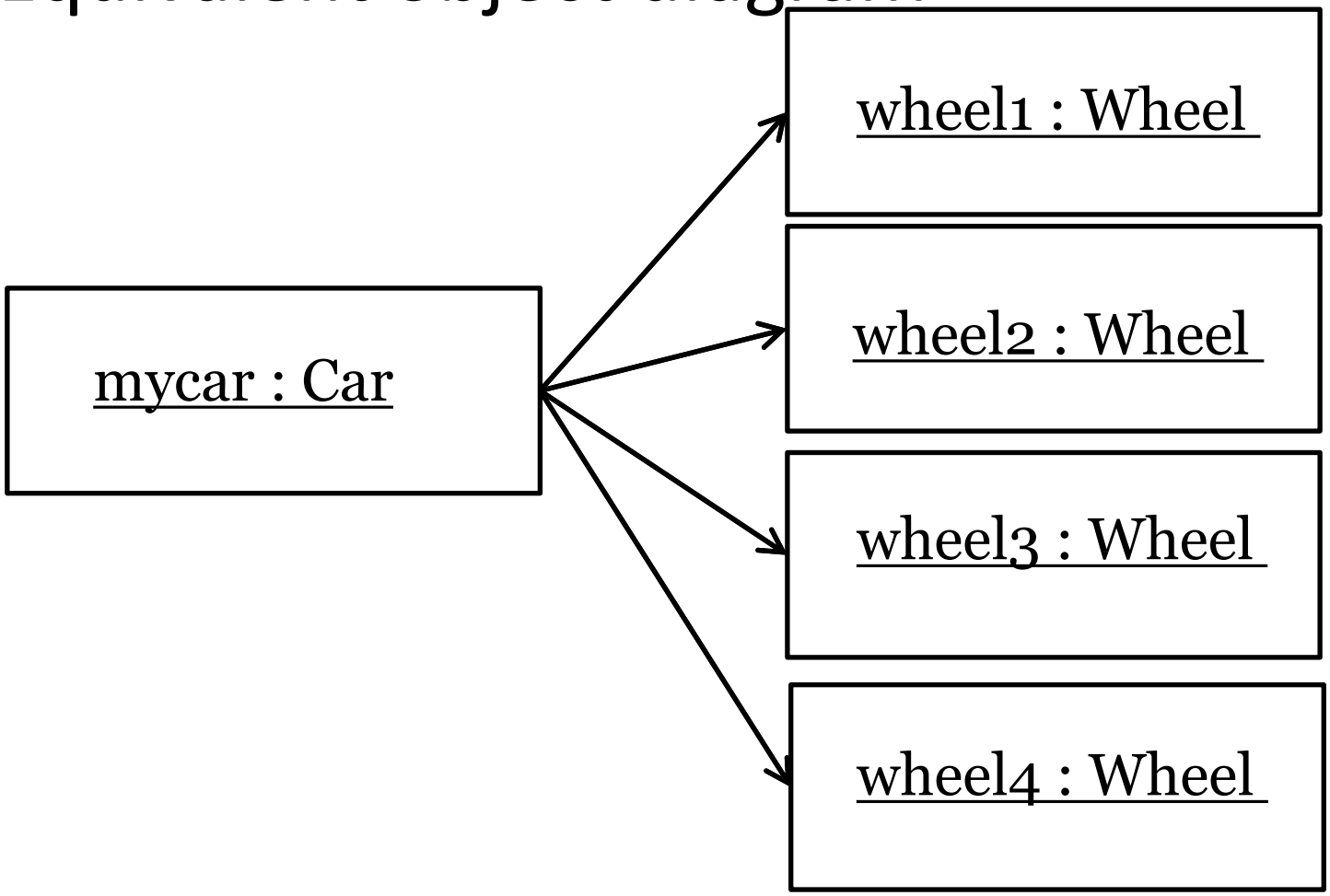


Association
(with navigability)

Relationships (1/6) - overview and intuition - Association

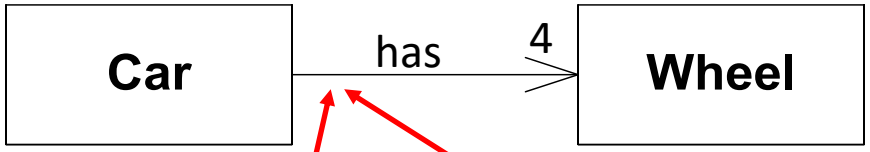


Equivalent object diagram



Relationships (1/6) - overview and intuition

- Association



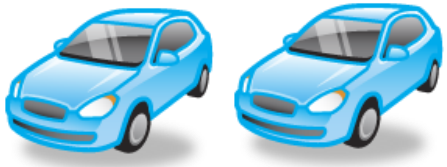
What does it mean to have a * here?

What if we have multiplicity 1 instead?

"*"

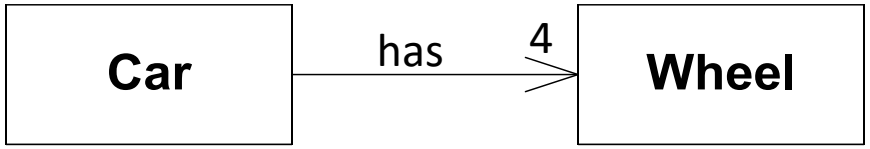
A wheel can be linked to more than one car instance

"1"



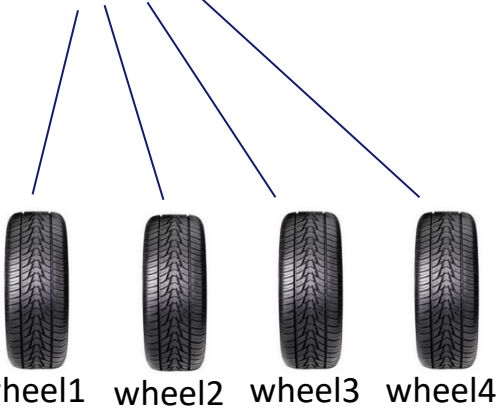
A wheel can only be liked to one car instance

Relationships (1/6) - overview and intuition - Association



Associations are the "glue" that ties a system together

mycar1

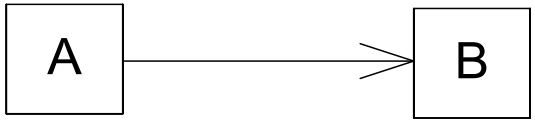


association instance = *link*

A link (association instance) describes a *relation* between objects at run-time.

{(mycar1,wheel1),
(mycar1,wheel2),
(mycar1,wheel3),
(mycar1,wheel4)}

Relationships (2/6) - overview and intuition - Aggregation



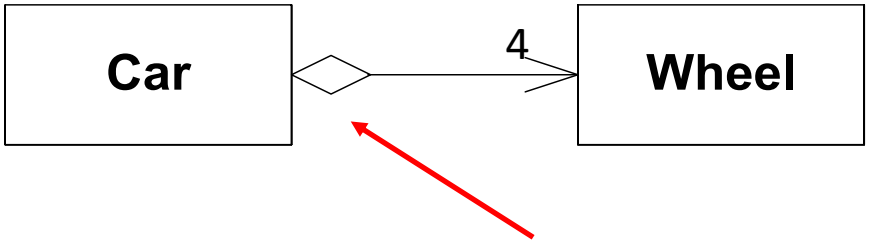
Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

Aggregation

Relationships (2/6) - overview and intuition - Aggregation

Common vague interpretations: "owns a" or "part of"



What does this mean? What is the difference to association?

Vague definitions → Inconsistency and misunderstandings

Aggregation was added to UML with little semantics. Why?

Jim Rumbaugh
"Think of it as a modeling placebo"

Recommendation: - Do not use it in your models.
- If you see it in other's models, ask them what they actually mean.

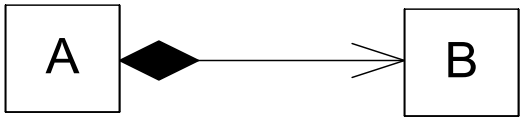
Relationships (3/6) - overview and intuition - Composition

Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

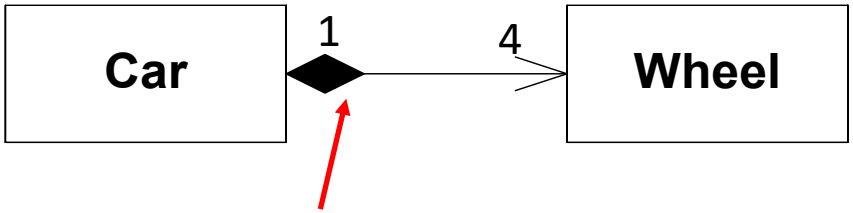
Aggregation

Avoid it to avoid misunderstandings



Composition

Relationships (3/6) - overview and intuition - Composition

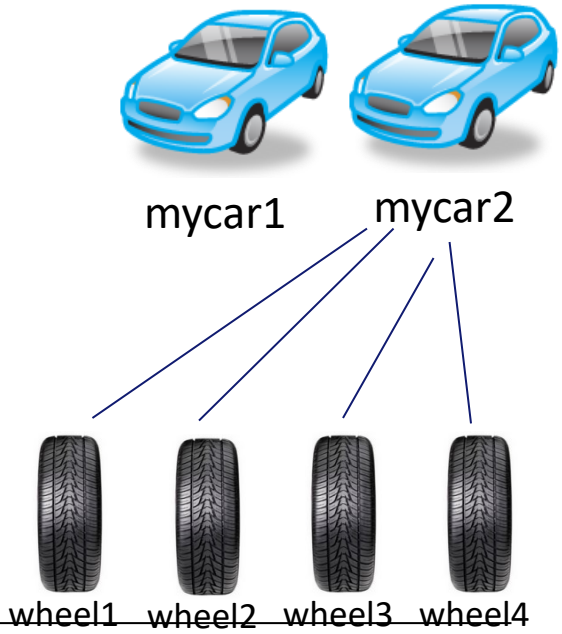


Any difference to association?

Yes! First, multiplicity must be 1 or 0..1. An instance can only have one owner.

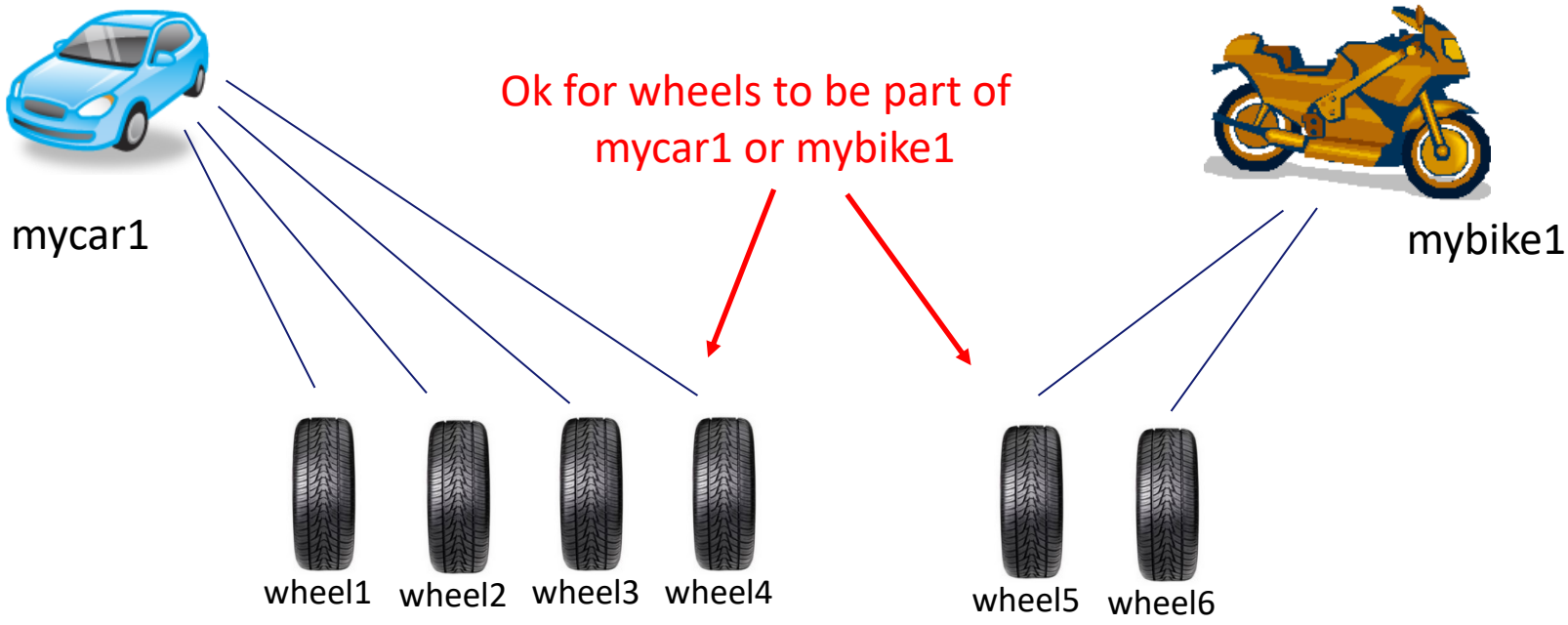
But, isn't this equivalent to what we showed with associations?

Well, in this case...



Relationships (3/6) - overview and intuition - Composition

Using composition...



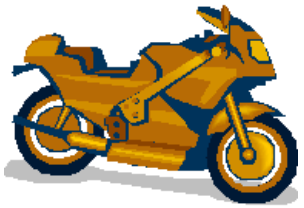
Relationships (3/6) - overview and intuition - Composition

Using composition...

Can mycar1 and mybike1 share the same wheels?

NO!
Not with composition!

mycar1



mybike1

wheel1 wheel2 wheel3 wheel4

Key concepts
- "No sharing" rule
- The owner is responsible for managing its parts, e.g. allocation and deallocation.

Relationships (3/6) - overview and intuition

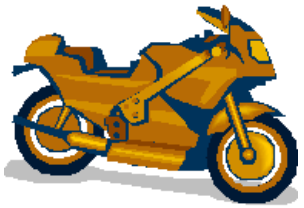
- Composition

(Note the difference. The diamond is removed.)

Using associations...

Can mycar1 and mybike1 share the same wheels this time?

mycar1



mybike1

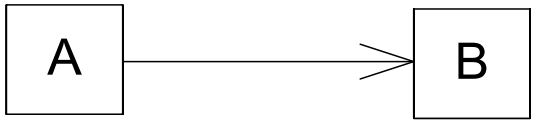
Yes! Associations do not have a "no sharing" rule.

However, in this case it is a strange model...

wheel1 wheel2 wheel3 wheel4

wheel5 wheel6

Relationships (4/6) - overview and intuition - Generalization

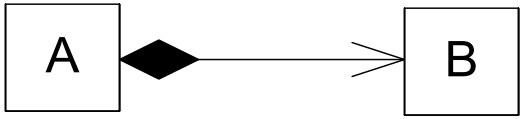


Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

Aggregation

Avoid it to avoid misunderstandings



Composition

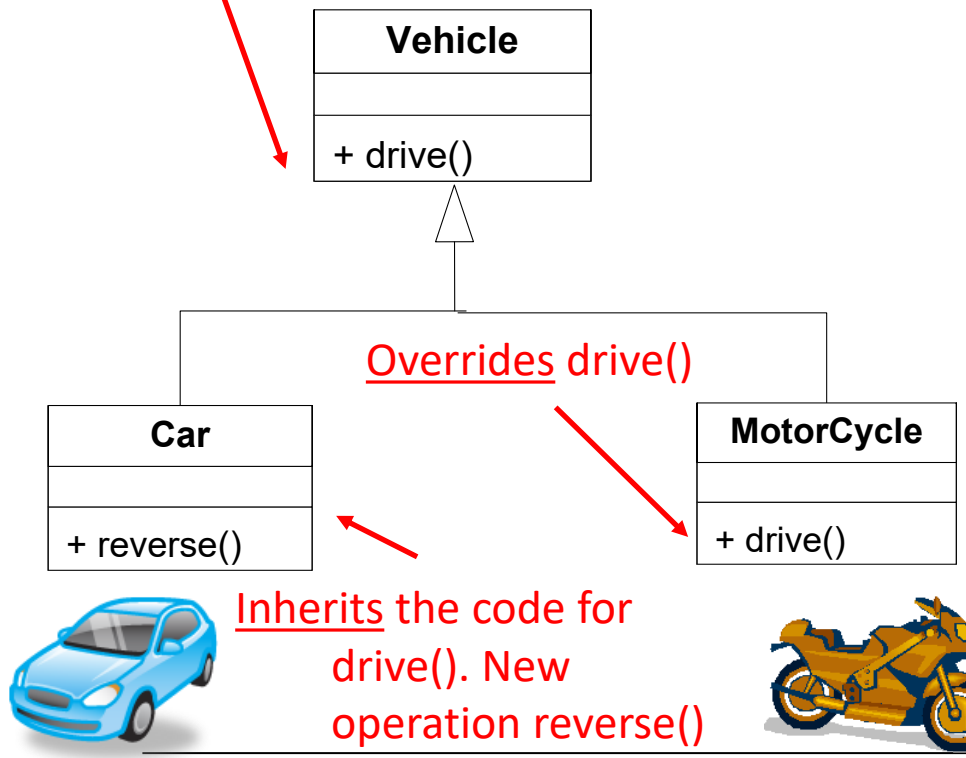
An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

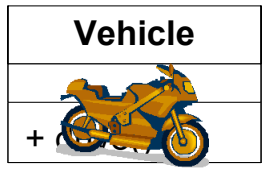
Generalization

Relationships - (4/6) overview and intuition - Generalization

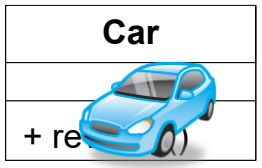
Class with code for the drive() operation

1. **Inheritance**
~ relation implementation

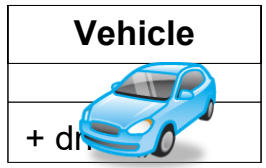




Visible Type: *Vehicle*.
Instance of: *MotorCycle*.
Can we drive()? Can we reverse()?



Visible Type: *Car*.
Instance of: *Car*
Can we drive()? Can we reverse()?



Visible Type: *Vehicle*.
Instance of: *Car*
Can we drive()? Can we reverse()?

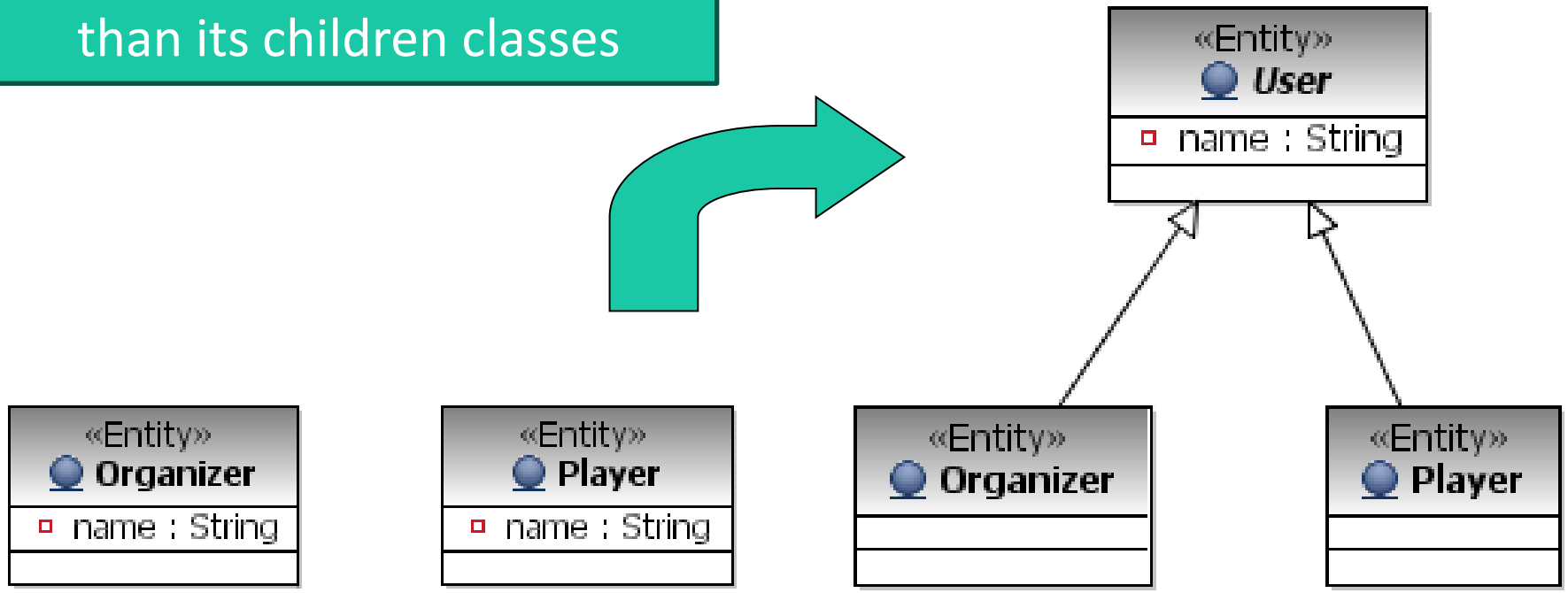
An instance of a class can have many types
= (subtyping) polymorphism

2. **Subtyping**
~ relation on interfaces

static typing: safe substitution

Typical Use of Generalization

Parent class is more general than its children classes



Aim: Lift up common attributes and methods to the superclass

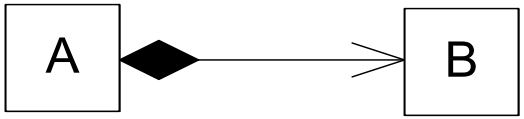
Relationships - (5/6) overview and intuition - Realization

Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

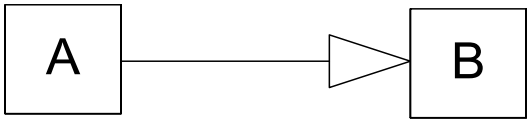
Aggregation

Avoid it to avoid misunderstandings



Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

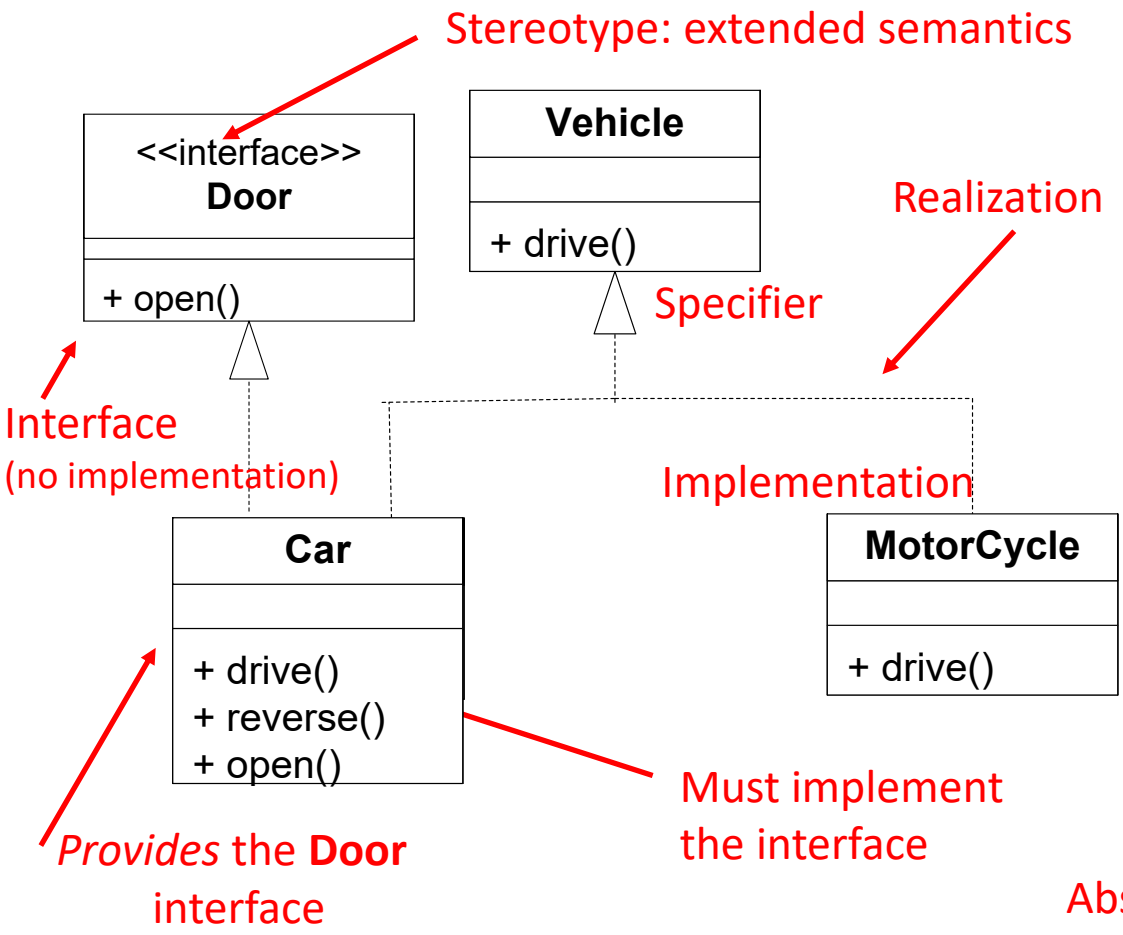


Generalization

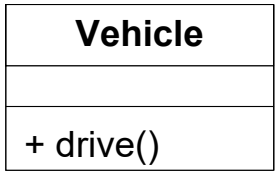
1) "A" inherits all properties and operations of "B".
2) An instance of "A" can be used where a instance of "B" is expected.

Realization

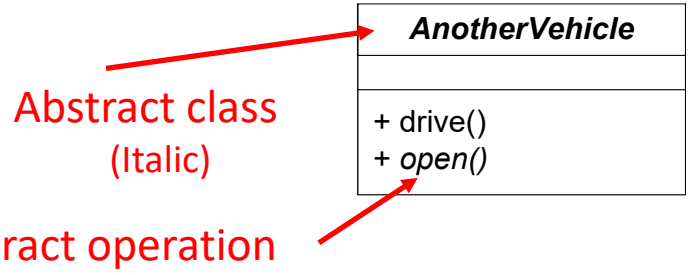
Relationships - (5/6) overview and intuition - Realization



Can we create an instance of Vehicle?

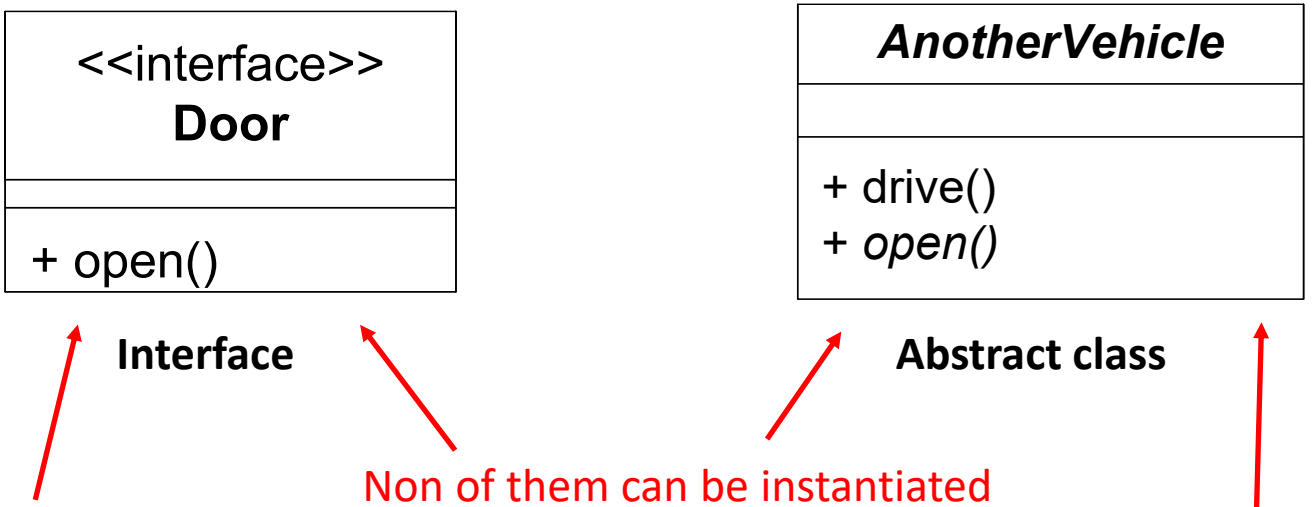


Can we create an instance of AnotherVehicle?



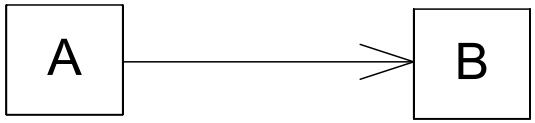
Relationships - (5/6) overview and intuition - Realization

What is the difference between an interface and an abstract class?



An abstract class with <u>only abstract operations</u> is conceptually the same as an interface

Relationships - (6/6) overview and intuition - Realization

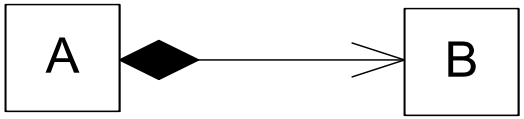


Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

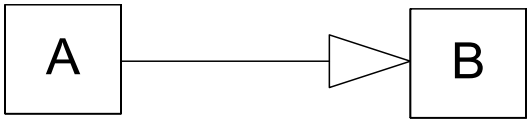
Aggregation

Avoid it to avoid misunderstandings



Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.



Generalization

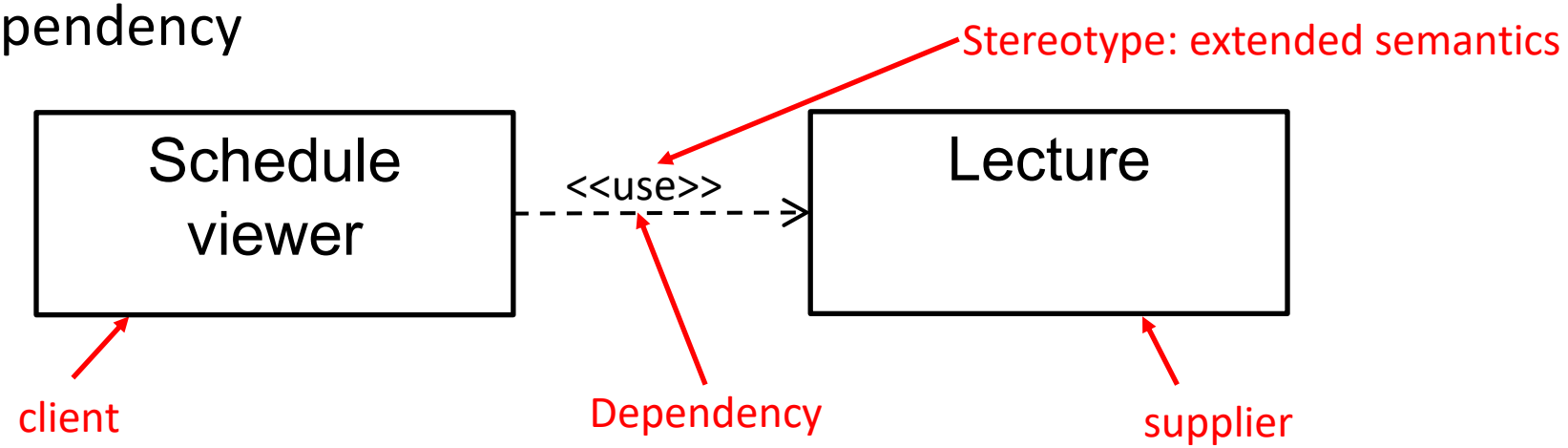
1) "A" inherits all properties and operations of "B".
2) An instance of "A" can be used where a instance of "B" is expected.

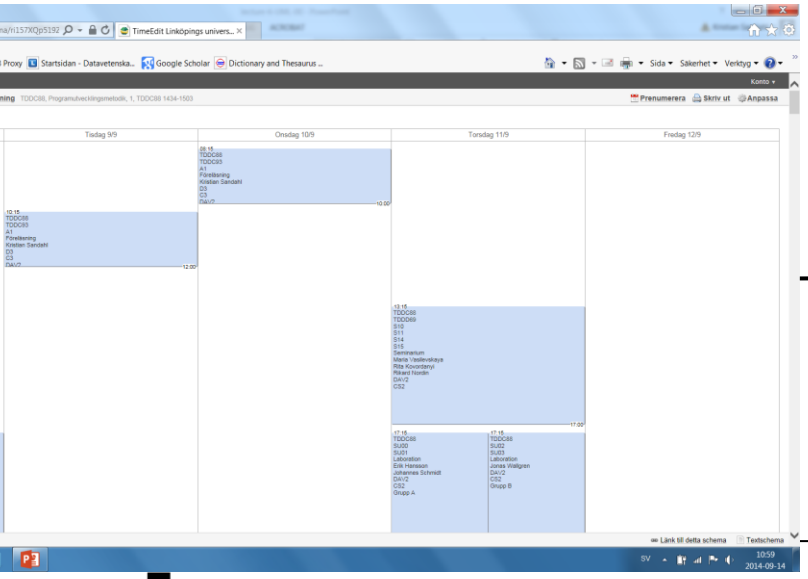
Realization

"A" provides an implementation of the interface specified by "B".

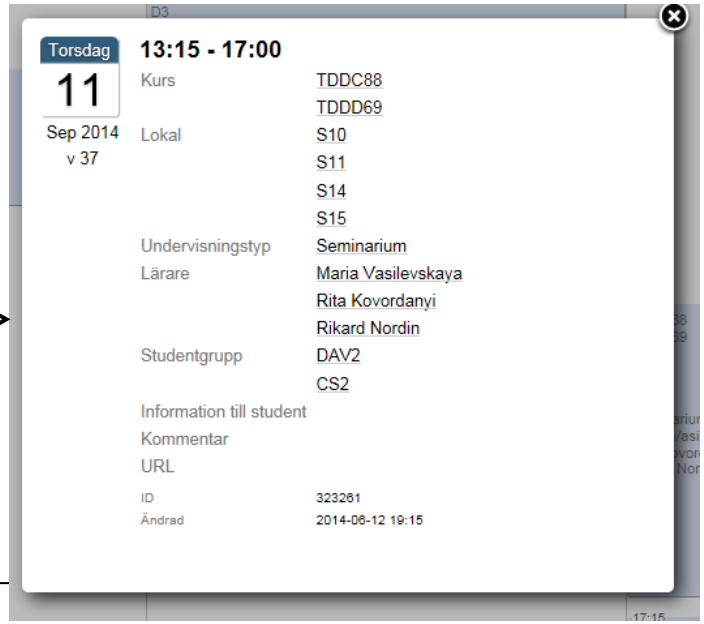
Dependency

Relationships - (6/6) overview and intuition - Dependency





<<use>>



Relationships - overview and intuition

Conceptual models, domain models

Association
(with navigability)

"A" has a reference(s) to instance(s) of "B". Alternative: attributes

Aggregation

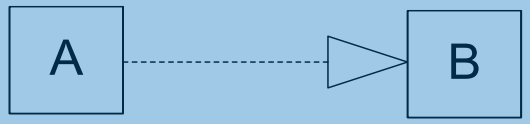
Avoid it to avoid misunderstandings

Composition

An instance of "B" is part of an instance of "A", where the former is not allowed to be shared.

Generalization

1) "A" inherits all properties and operations of "B".
2) An instance of "A" can be used where a instance of "B" is expected.



Realization

"A" provides an implementation of the interface specified by "B".

~~**Dependency**~~

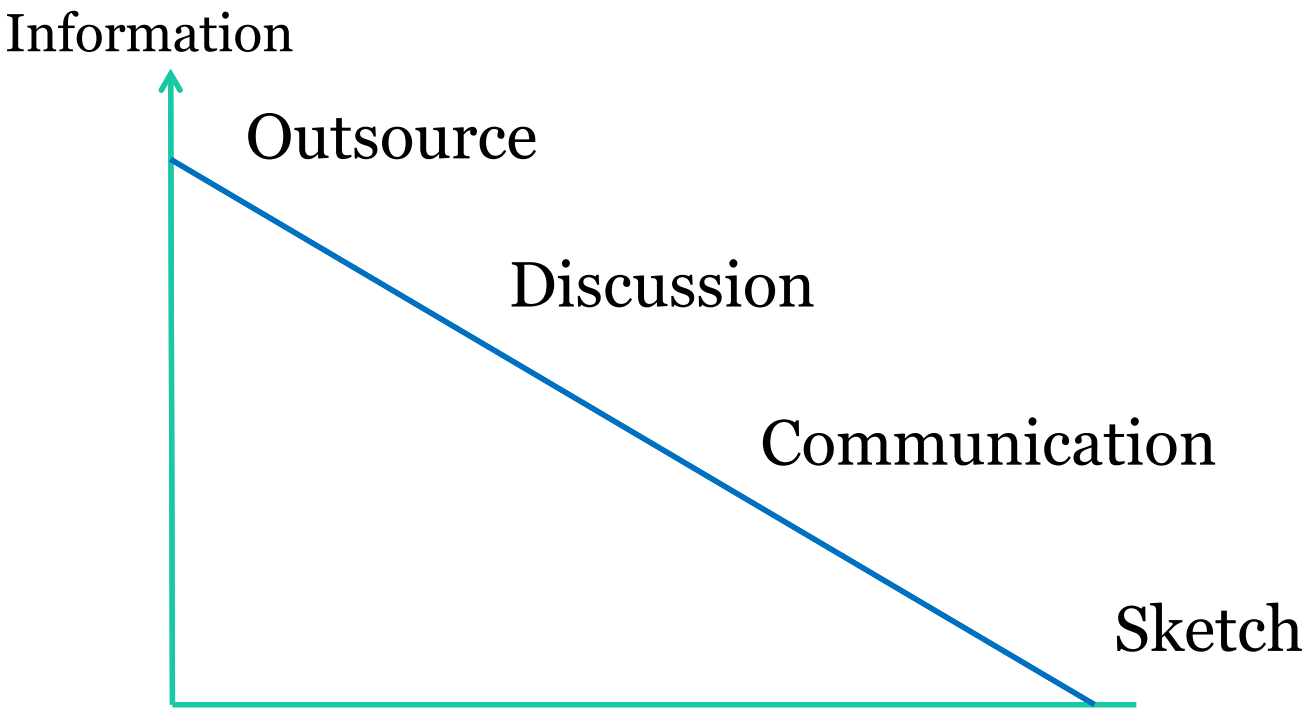
"A" is dependent on "B" if changes in the definition of "B" causes changes of "A".

LINKÖPING UNIVERSITY

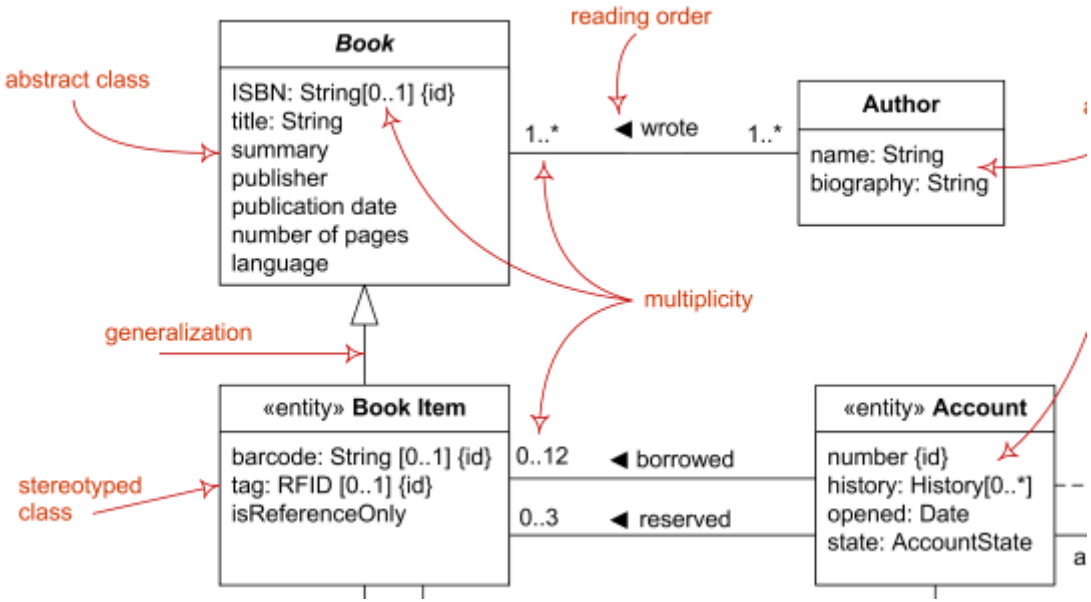
Design models, architecture models, implementation models

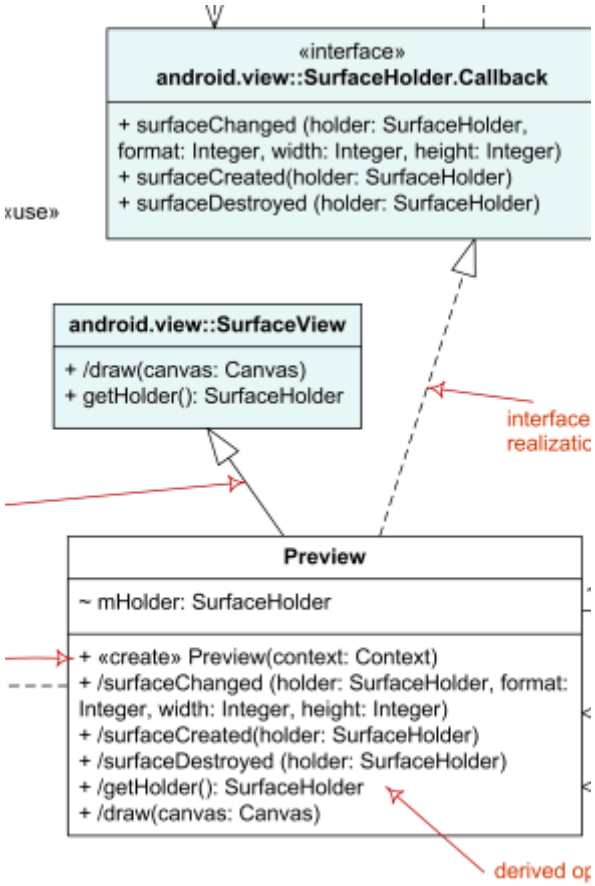
Domain Models
vs
Implementation Models

What you model depends on the recipient and the perspective

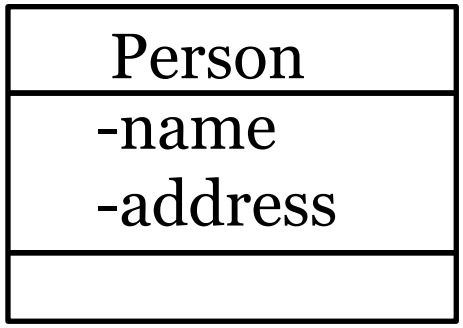


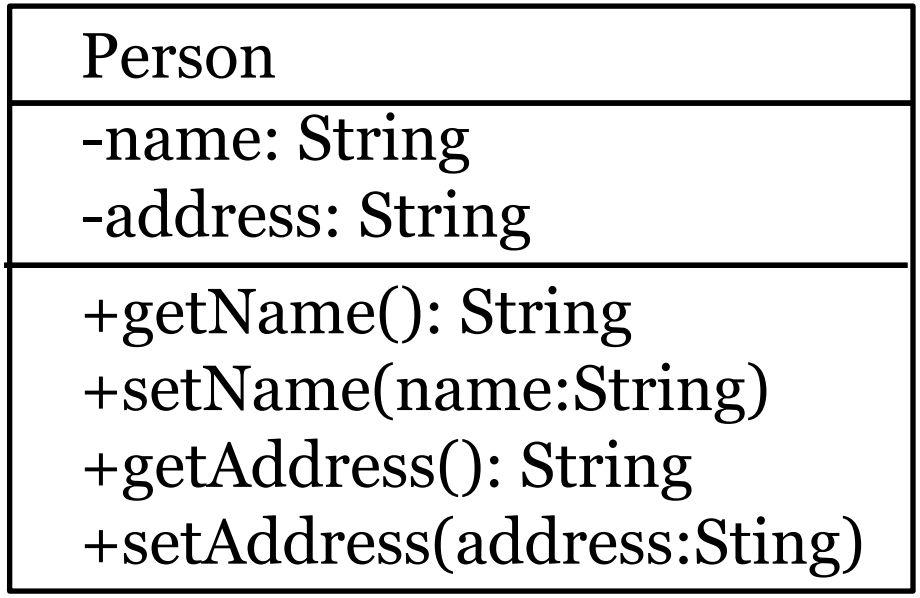
Perspectives: Domain modeling vs. Implementation





Domain model vs. implementation model





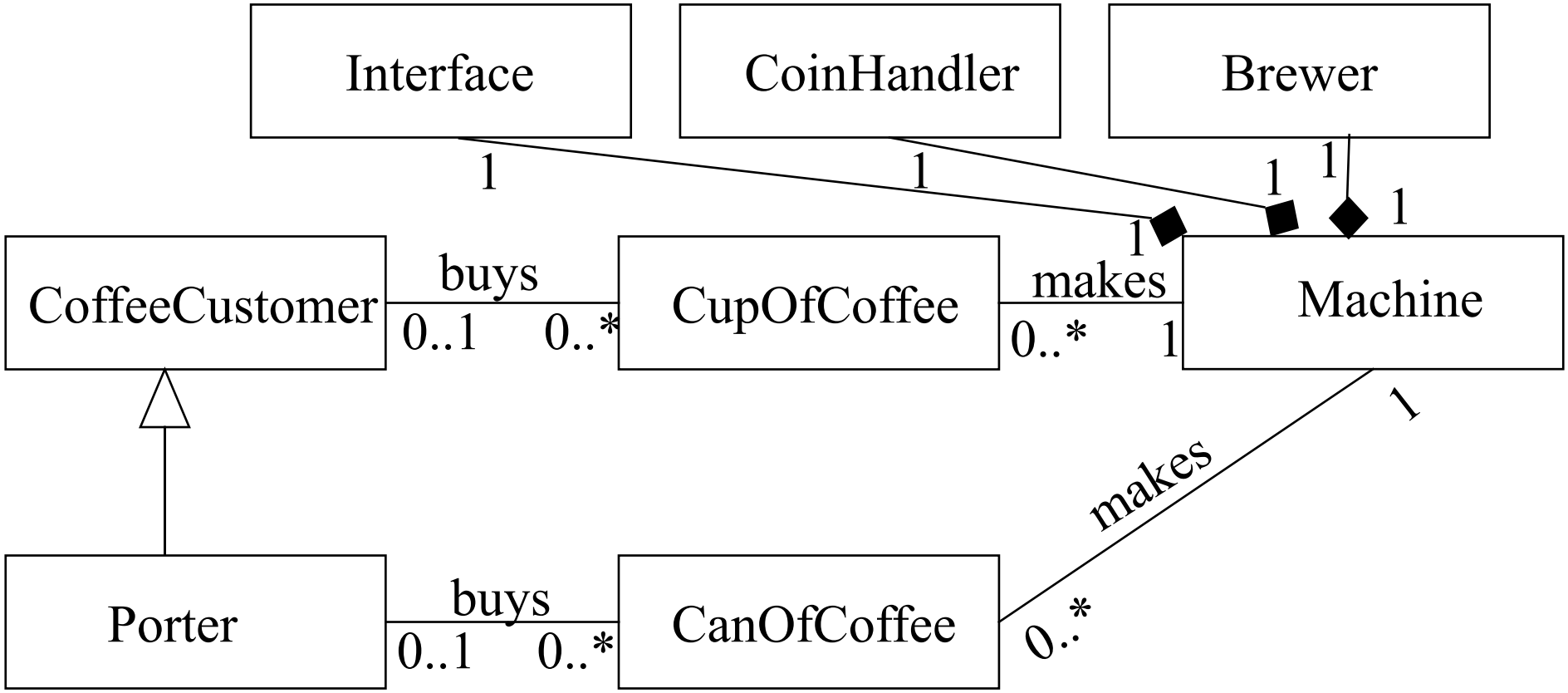
In this course: domain model = conceptual model

Identifying classes: noun analysis

A CoffeeDrinker approaches the machine with his cup and a coin of SEK 5. He places the cup on the shelf just under the pipe. He then inserts the coin, and press the button for coffee to get coffee according to default settings. Optionally he might use other buttons to adjust the strength and decide to add sugar and/or whitener. The machine processes the coffee and bell when it is ready. The CoffeeDrinker takes his cup from the shelf.

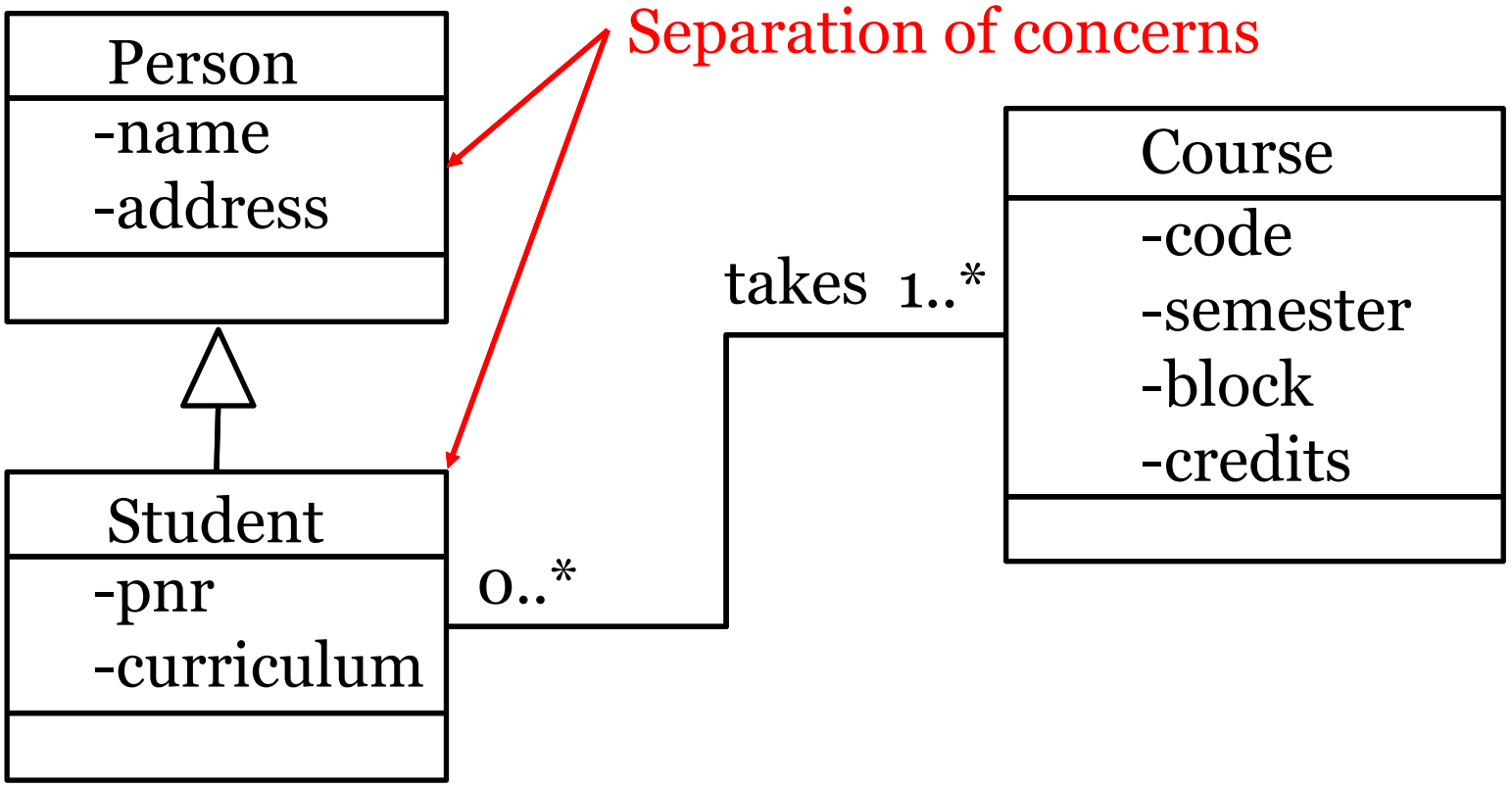
- **machine – real noun handled by the system**
- cup – unit for beverage
- coin – detail of user and machine
- shelf – detail of machine
- pipe – detail of machine
- **button– handled by the system**
- sugar – detail of coffee
- whitener – detail of coffee
- **cup of coffee – handled by the system**
- **indicator – not discovered**

The coffee machine class model



Domain modeling example

Identify key domain concepts and relations
Design data model for persistent data



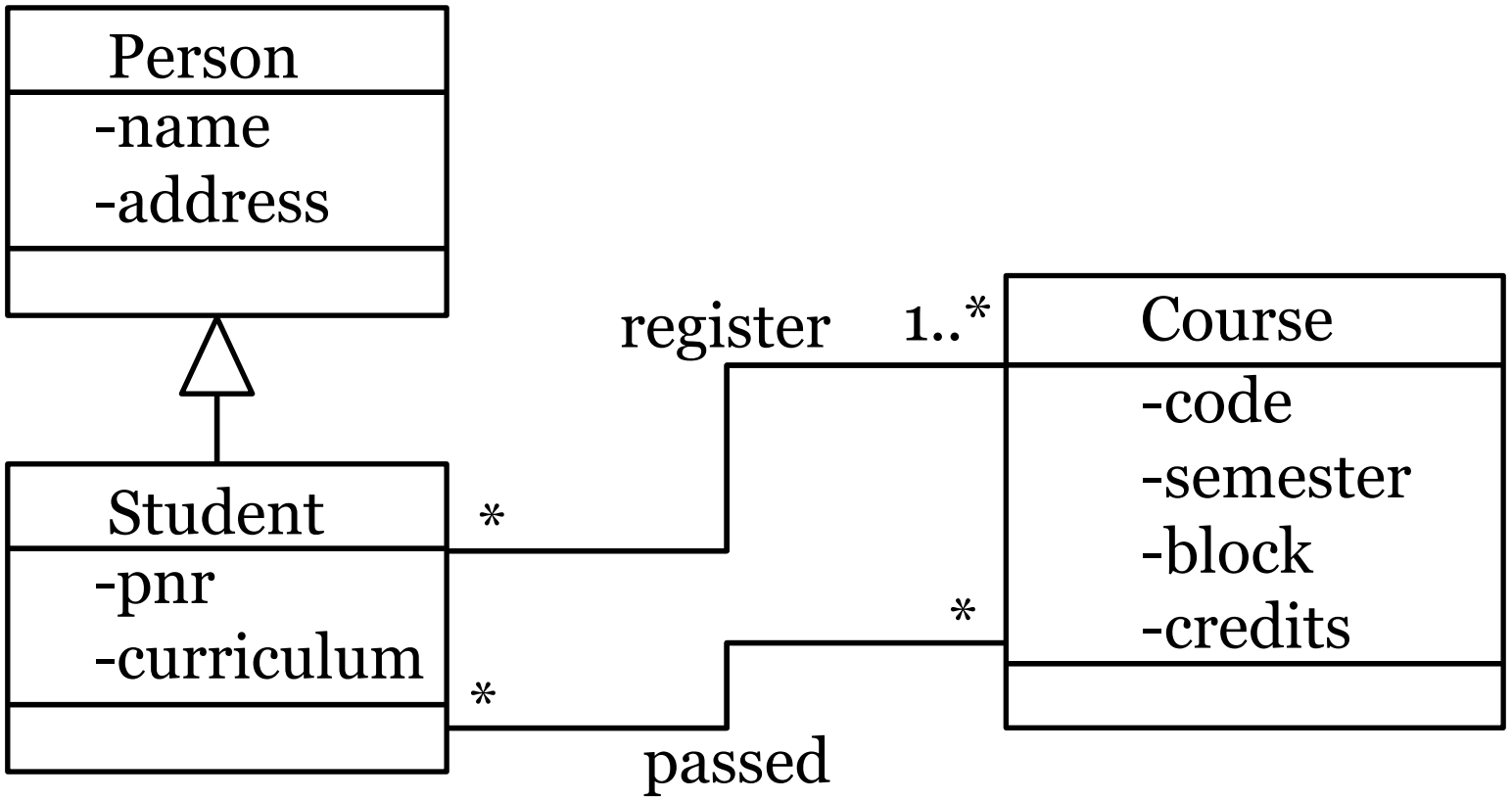
More use-cases and the Class Responsibility Card (CRC)

Use-case name:
Students register for courses
Students passes courses for degree

Student

Responsibilities	Collaboration
Register for courses	Course
Keep track of passed courses	Course
…	…

The model becomes



More requirements

The university stores registrations
The university stores passed courses

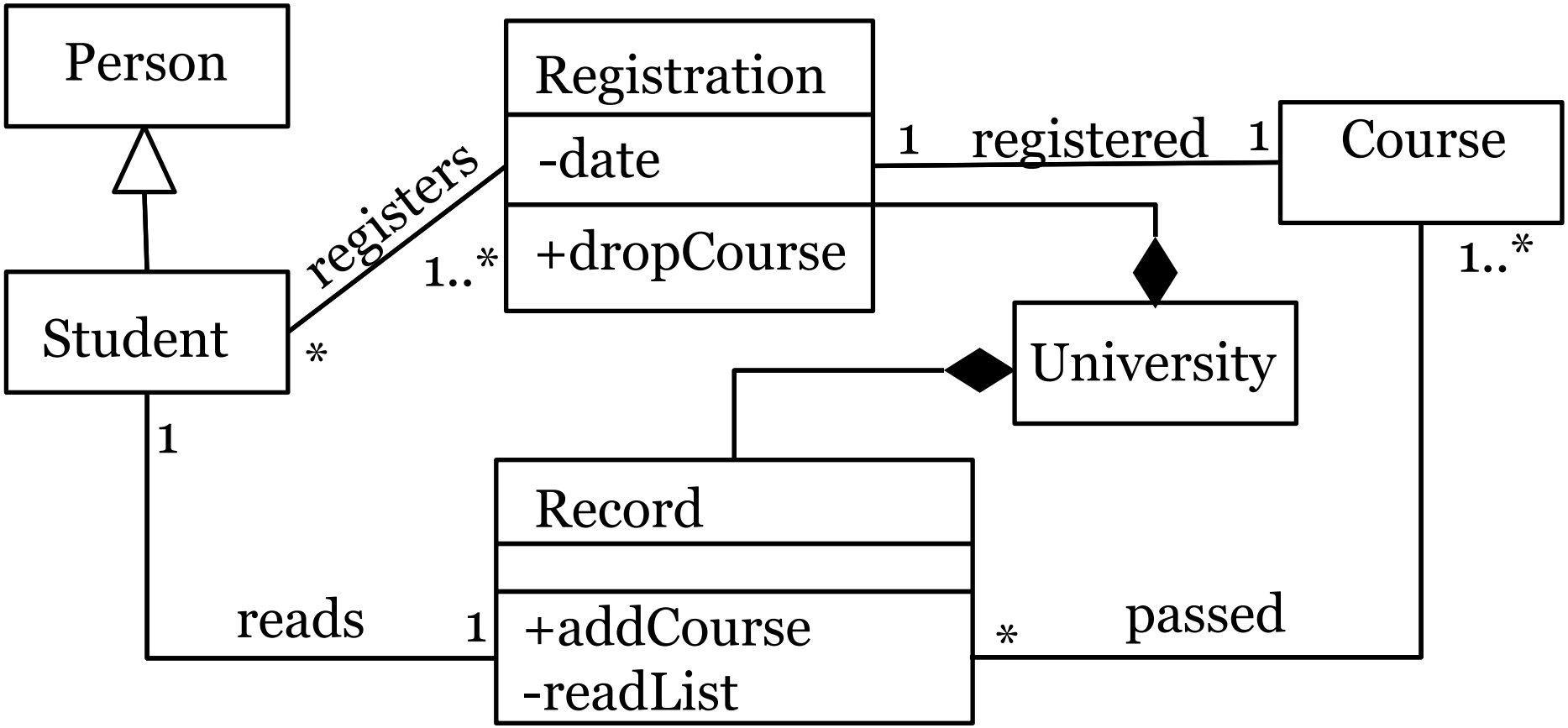
Student

Responsibilities	Collaboration
Register for courses	Registration

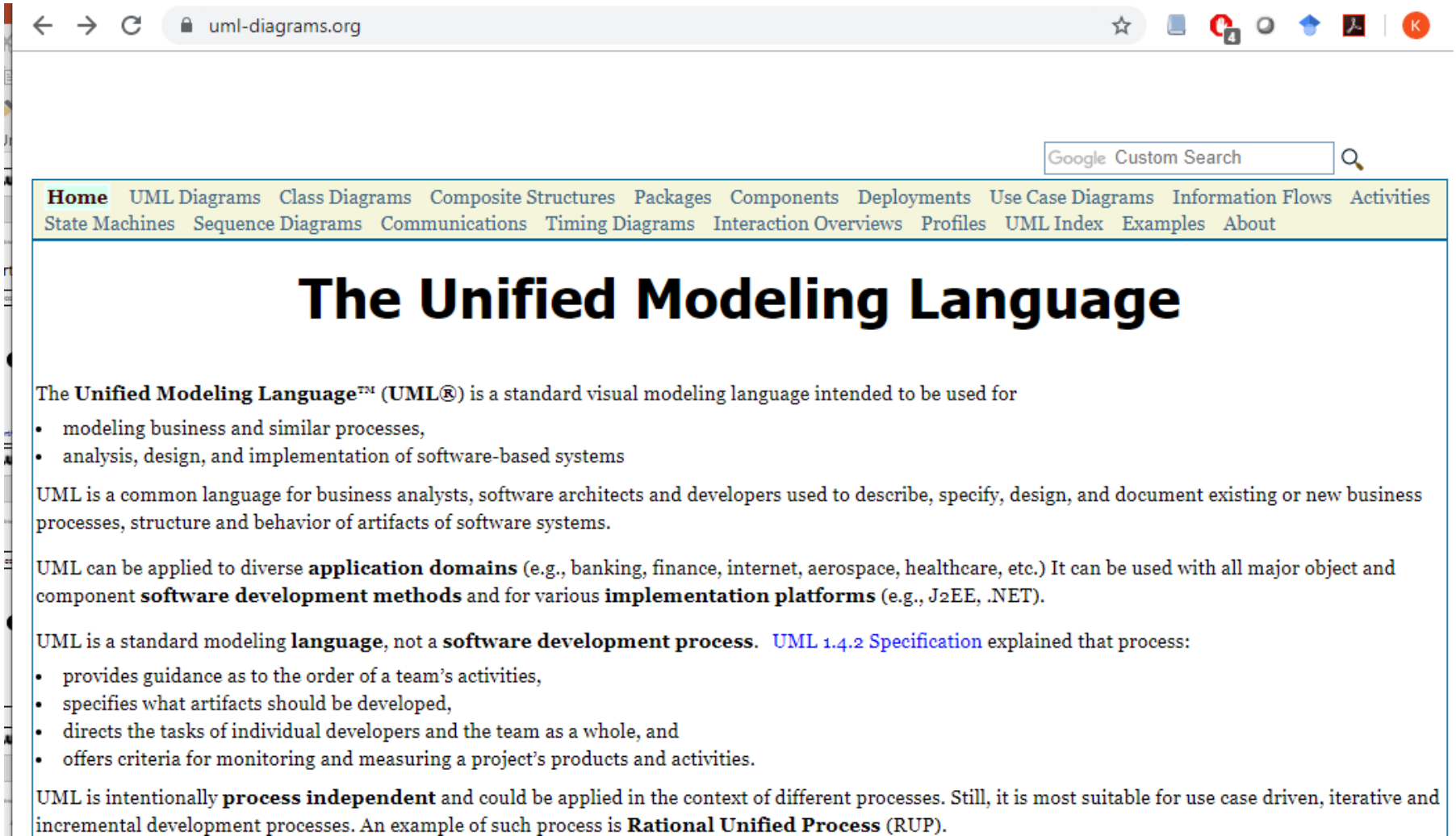
University

Responsibilities	Collaboration
Store registrations	Registration
Store passed courses	Record

Refined model (details suppressed)

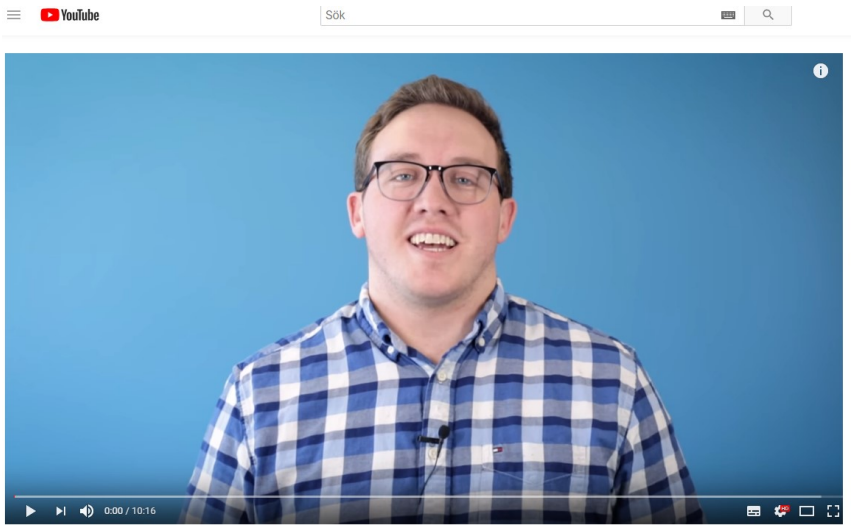


A complete, comprehensive guide to UML 2.5



Rehearsal and a little example

- https://www.youtube.com/watch?v=UI6lqHOVHic

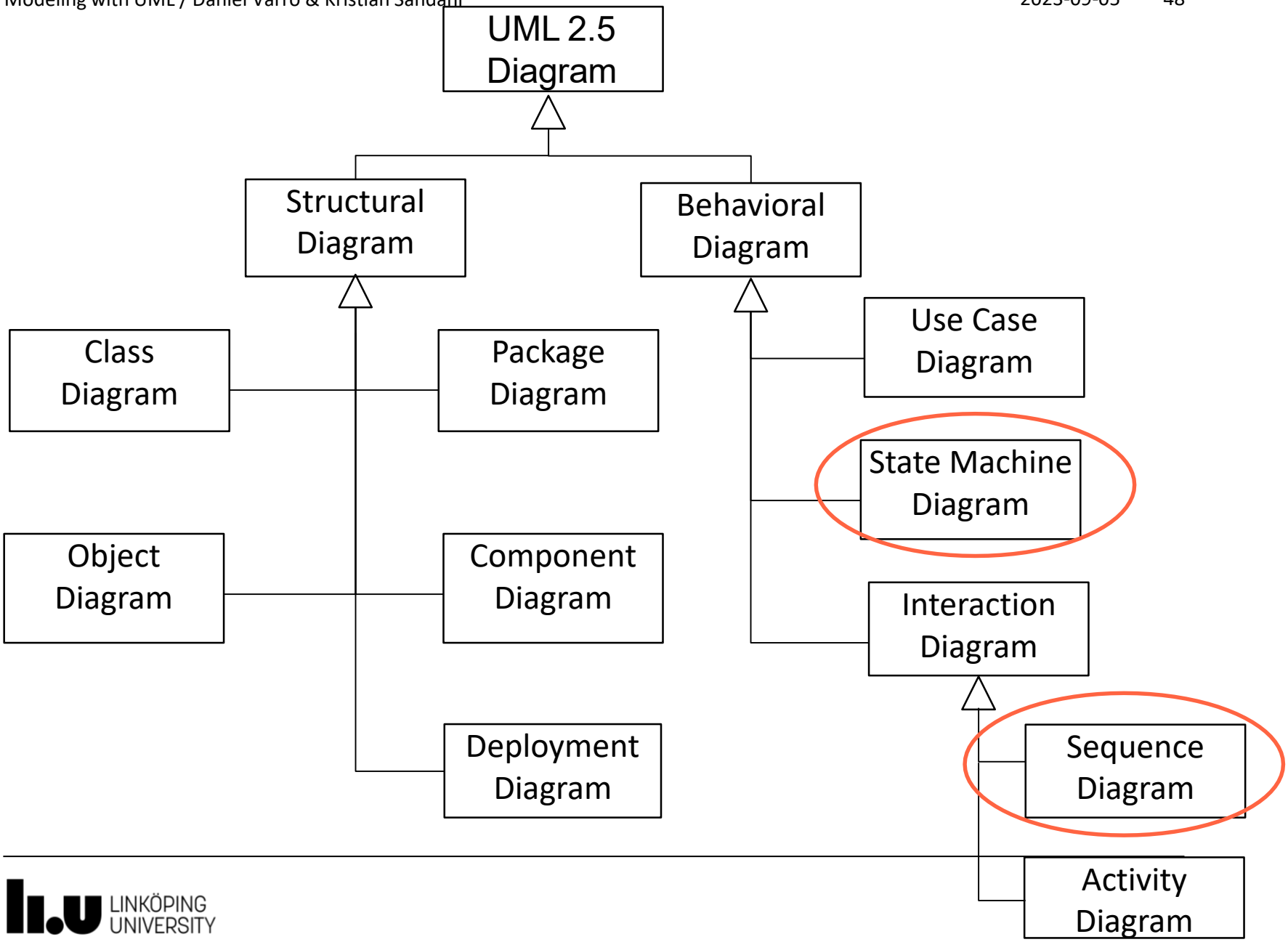


UML Behavior Modeling
(Sequence Diagrams)

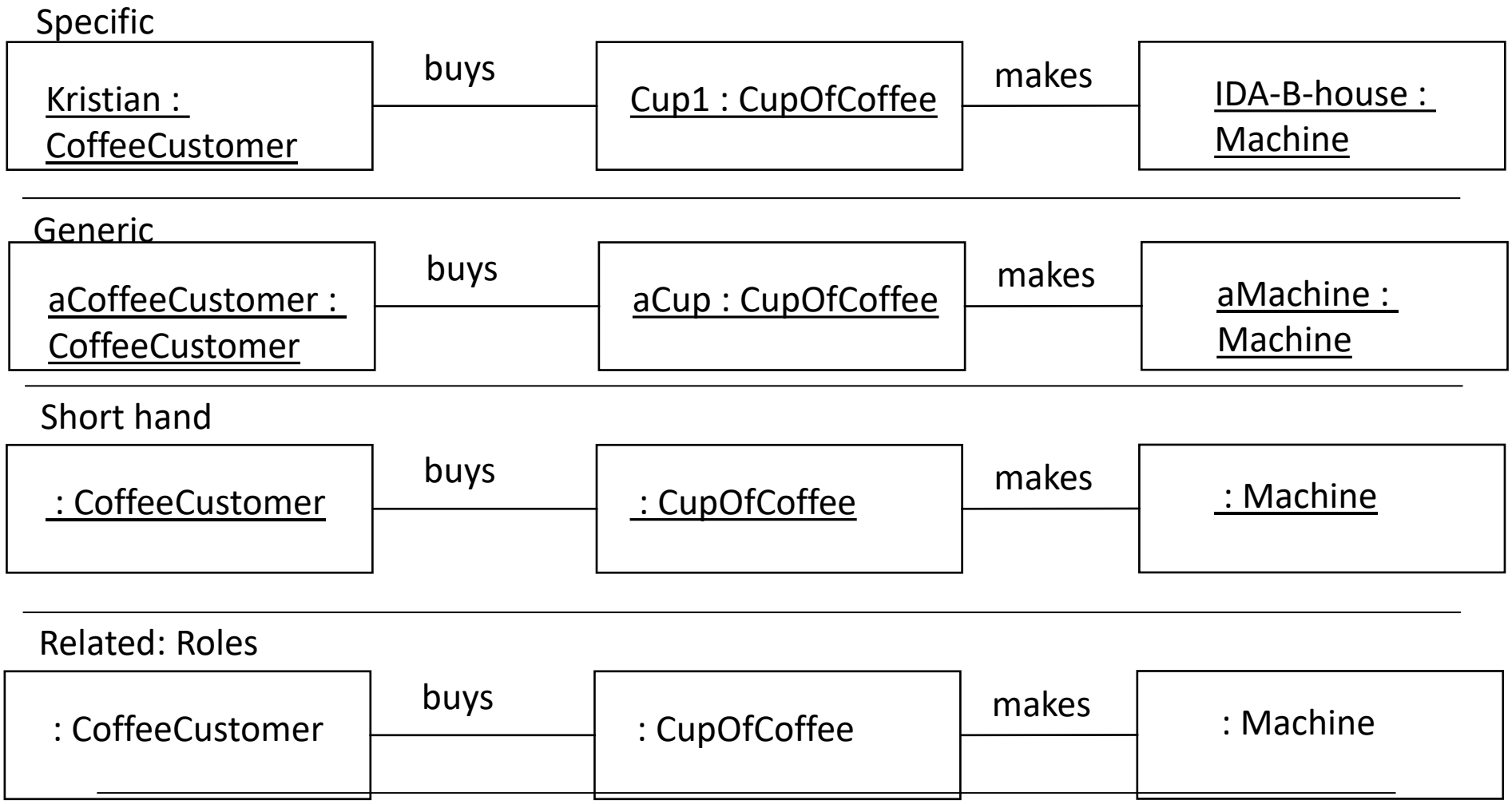
Provide a description of the dynamic behavior as interactions

- between actors and the system and
- between objects within the system

Well-known Diagrams of UML

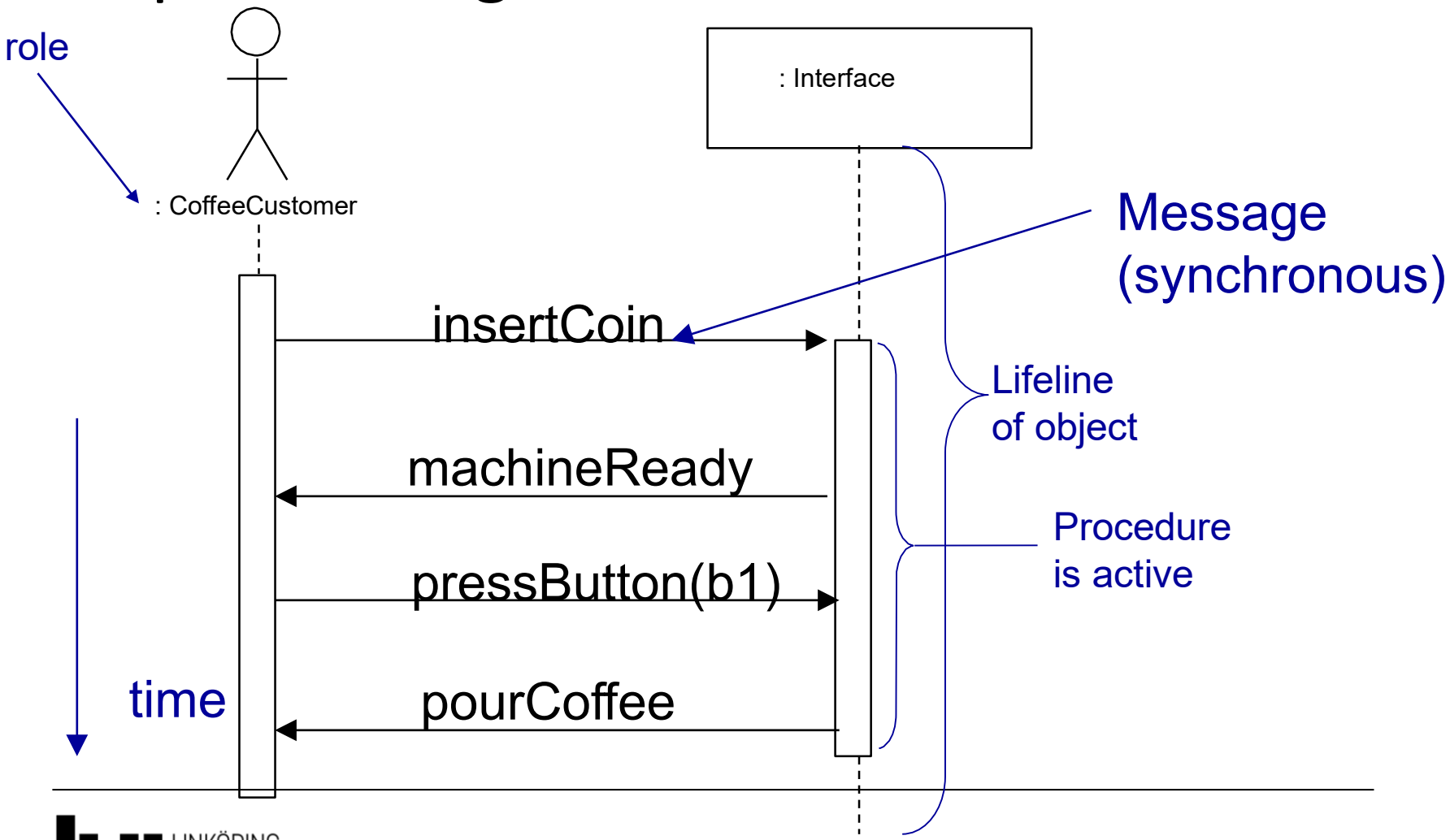


Different instance models

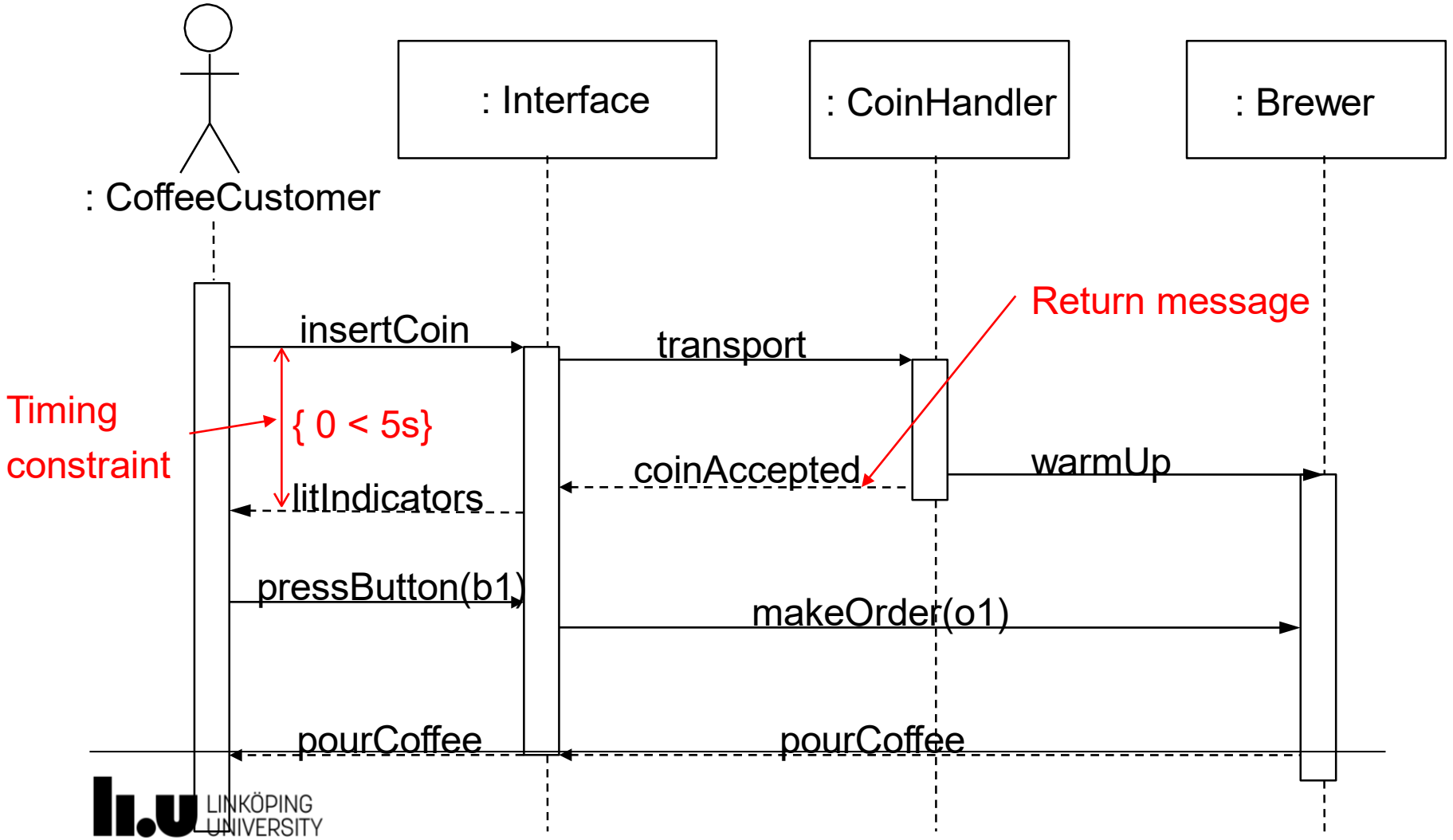


How to capture their internal behavior and interaction?

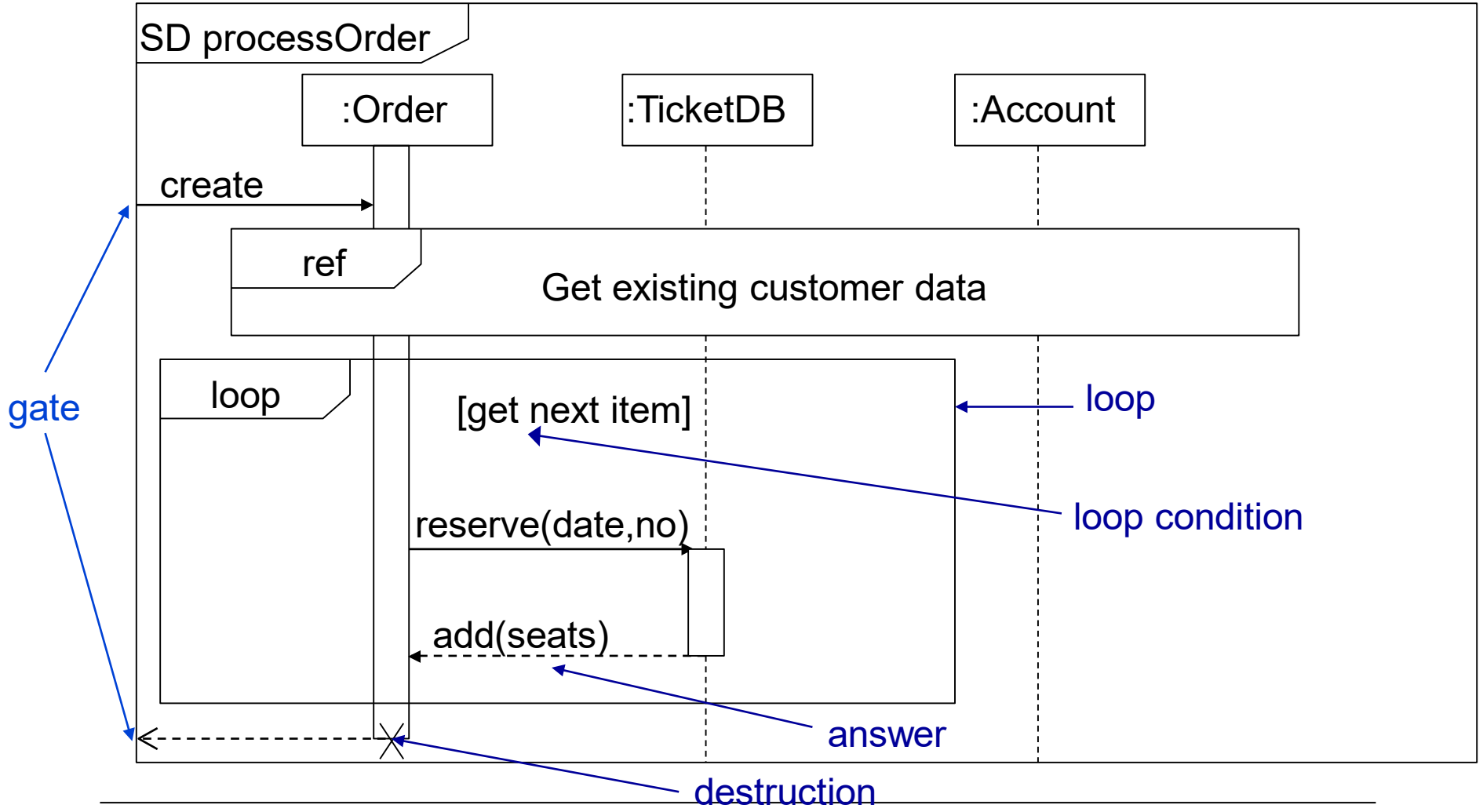
Sequence diagram



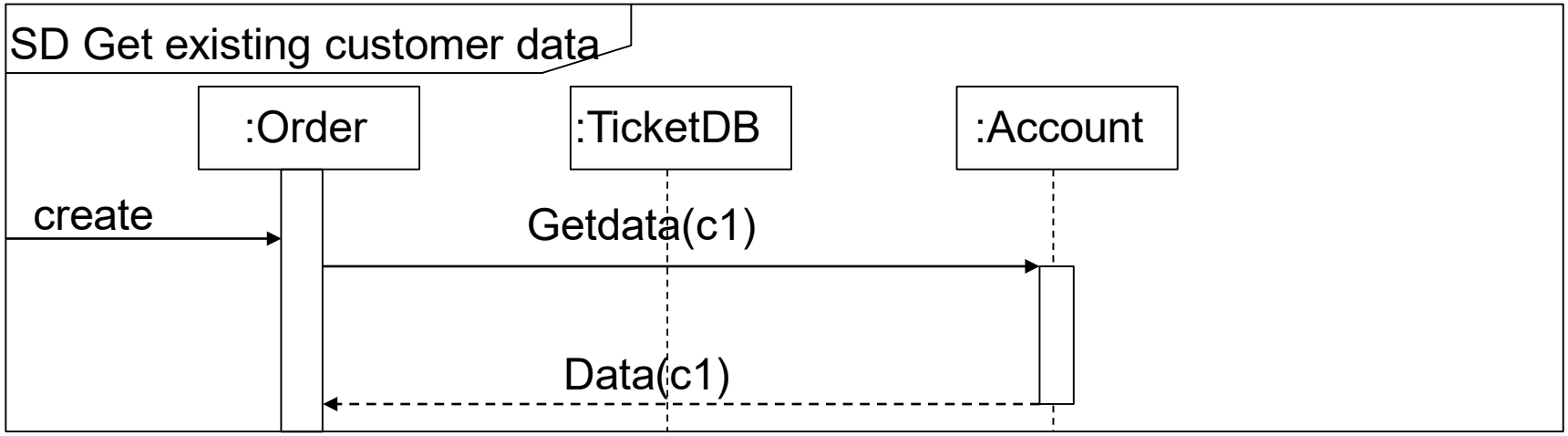
Sequence diagram with several roles



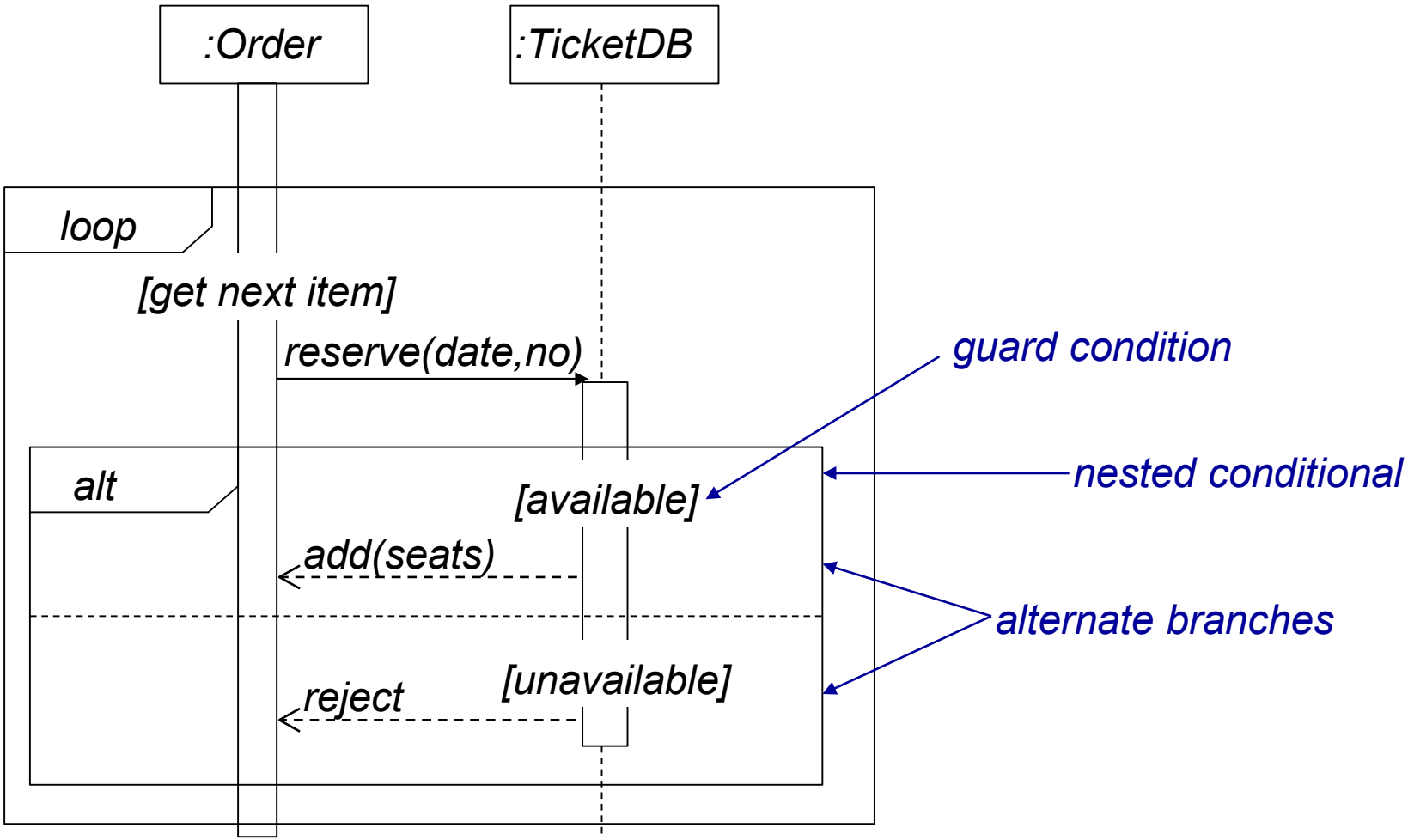
Combining fragments of sequence diagrams



Combining fragments of sequence diagrams

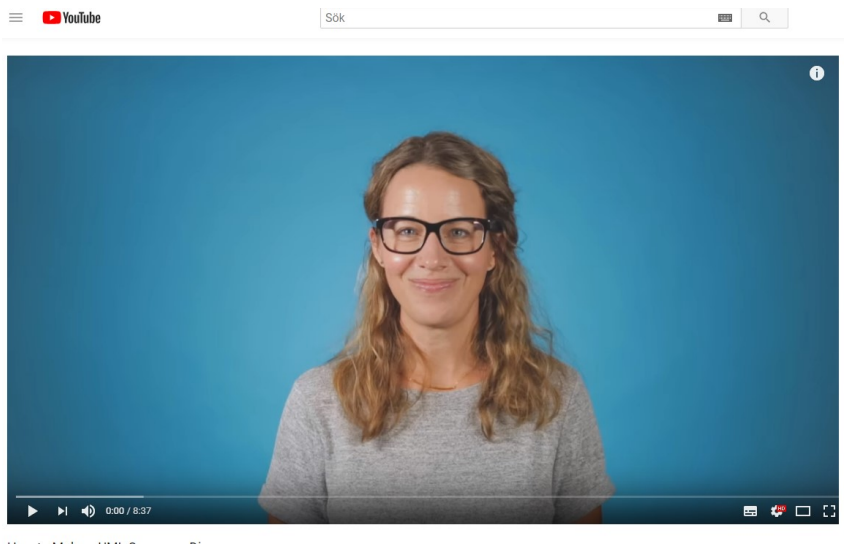


More fragments of sequence diagrams



Rehearsal and a little example

https://www.youtube.com/watch?v=pCK6prSq8aw&t=7s



Two flaws:
Objects preceded with ":"
Eject card after invalid card or invalid PIN shall terminate transaction.

UML Behavior Modeling
(State Machine Diagrams)

For defining reactive behavior of objects
by executing state transitions and actions
in response to events

State-based Behavior Modeling

- **State partition** (AKA **state space**)
  - A set of distinguished **system states**
  - Examples
    - Days of Week: {Mon, Tue, Wed, Thu, Fri, Sat, Sun}
    - States of microwave oven: {full power, defrost, off}
  - **DEF**: A state partition is a set, exactly one element of which characterises the system at any time.
- **Current state**
  - E.g. today is Wed, the microwave is on defrost, etc.
  - **DEF**: At any given moment, the current state is the element of the partition which is currently valid.

Example: Abstract & Concrete States of a Stack

- **Concrete states** of a stack
  - $Stack_1$
    - Length = 2
    - Element[0] = String(„Winter 2023”)
    - Element[1] = String(„Fall 2023”)
  - $Stack_2$
    - Length = 2
    - Element[0] = String(„Winter 2024”)
    - Element[1] = String(„Fall 2024”)
- **Abstract states** of a stack:
  - **empty** : boolean isEmpty() {return length==0;}
  - **full** : boolean isFull() {return length==MAX;}
  - **hasContent**: boolean hasContent()
    {return length > 0 && length < MAX;}

Are these stacks in a different concrete state?
YES!

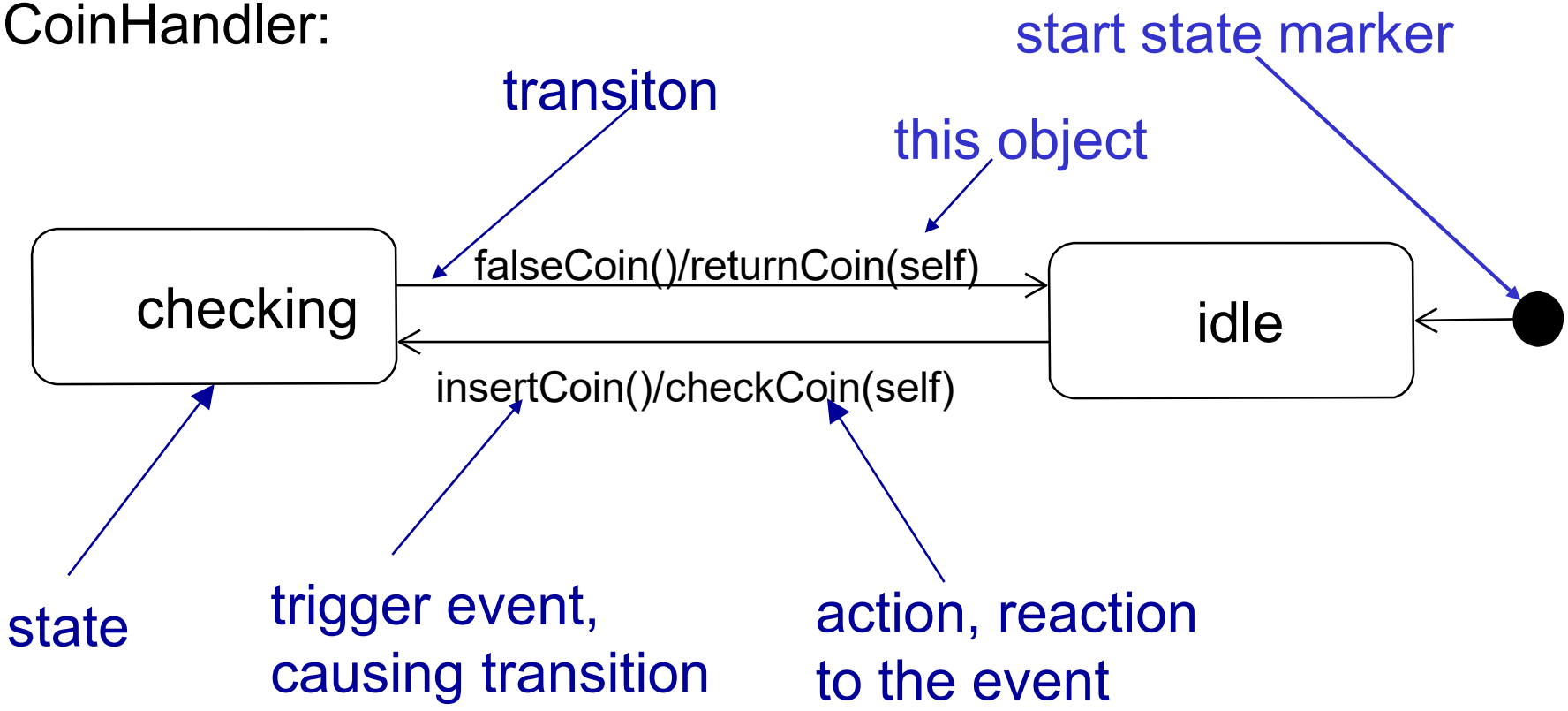
Are these stacks in a different abstract state?
NO!

Abstract State vs. Concrete State

- **Concrete state** of an object:
  - Current value of each of its attributes
  - Concrete state space:
    - Combination of possible values of attributes
    - May be infinite
- **Abstract states** of an object:
  - Predicates over concrete states
  - One abstract state ← many concrete states
  - Potentally: state hierarchies

State machine diagram

For class CoinHandler:



Specification

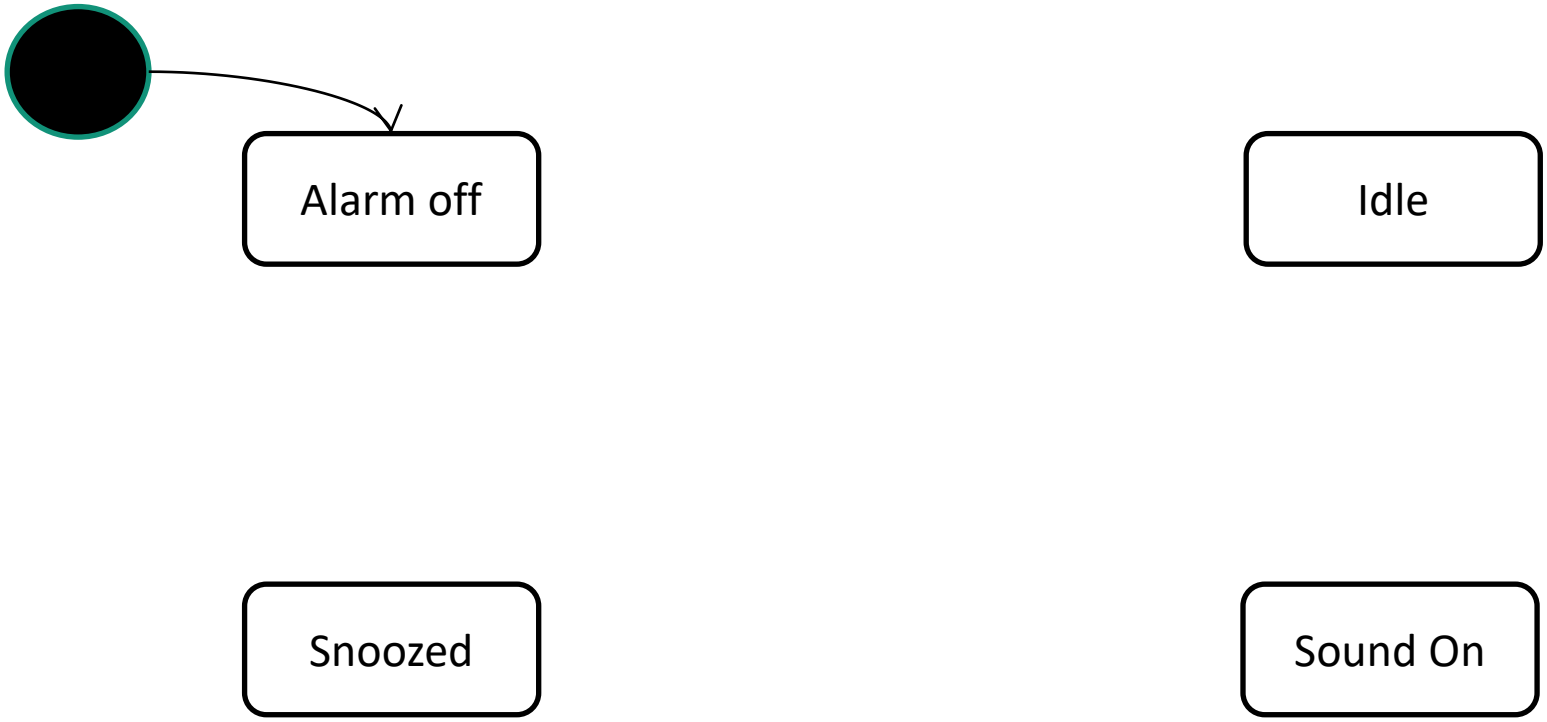
- Kristian's alarm clock starts sounding at 6.00 with a nasty signal. He can now do either of three things:

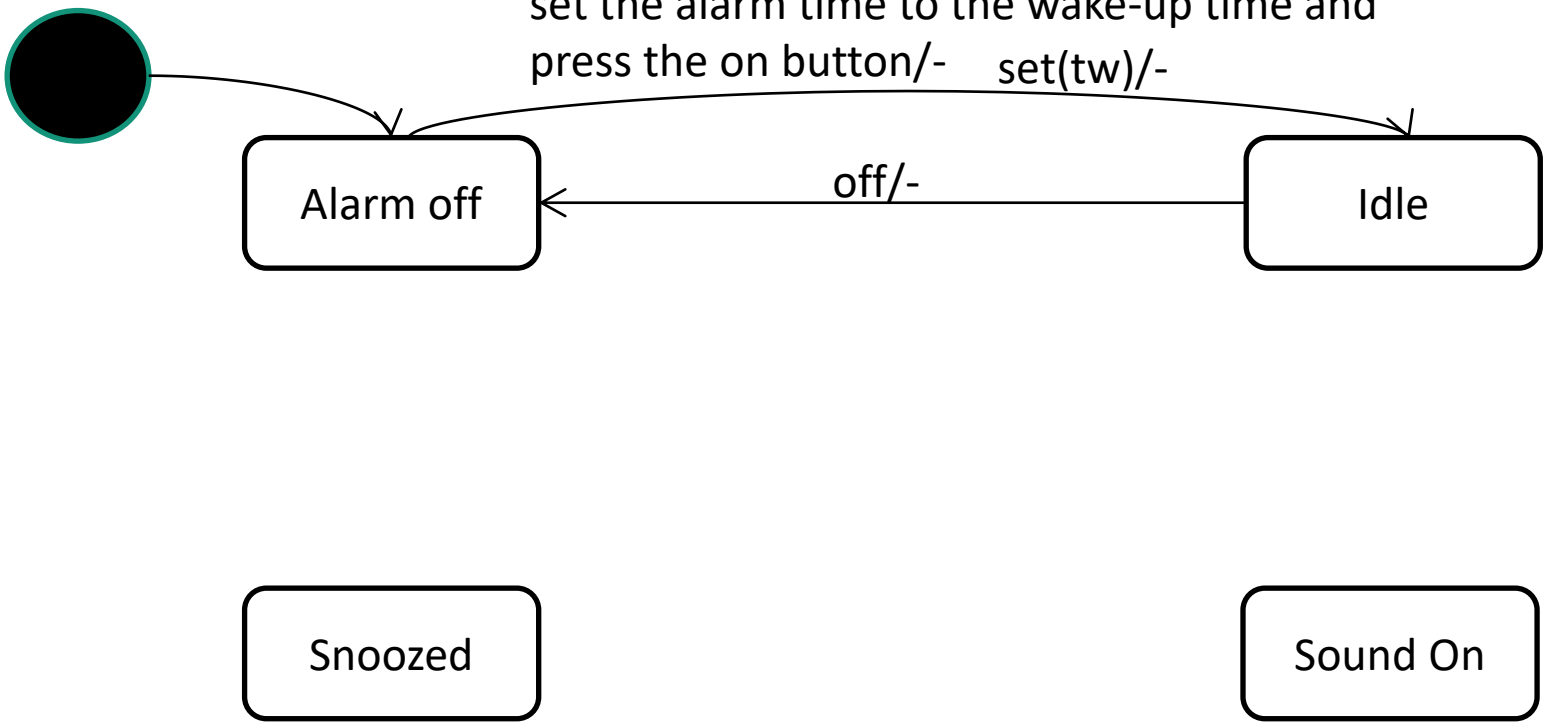
  a) Turn the alarm off;

  b) Press the snooze button; or

  c) Do nothing.
- If the snooze button is pressed the signal will turn off and start sounding after 5 minutes again.
- When an hour has passed from the first time the alarm sound started, the snooze button has no effect.
- After that the alarm sound starts, the signal will last for 2minutes.
- If no action has been taken during these 2 minutes, the absence of action will have the same effect as if the snooze button were pressed exactly when the alarm stopped to sound

- Task: design a UML state machine of the class *AlarmClock*

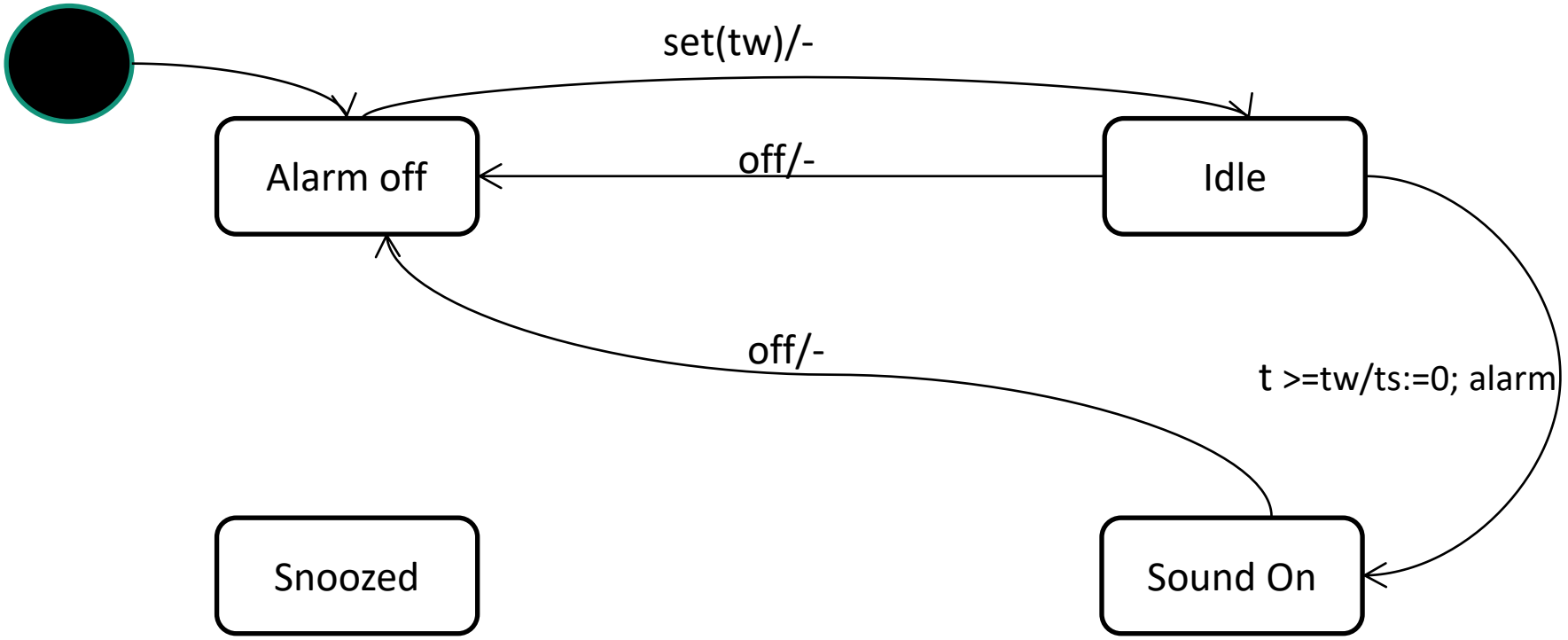

Alarm off
Idle
Snoozed
Sound On



off = off button pressed

tw = wake-up time

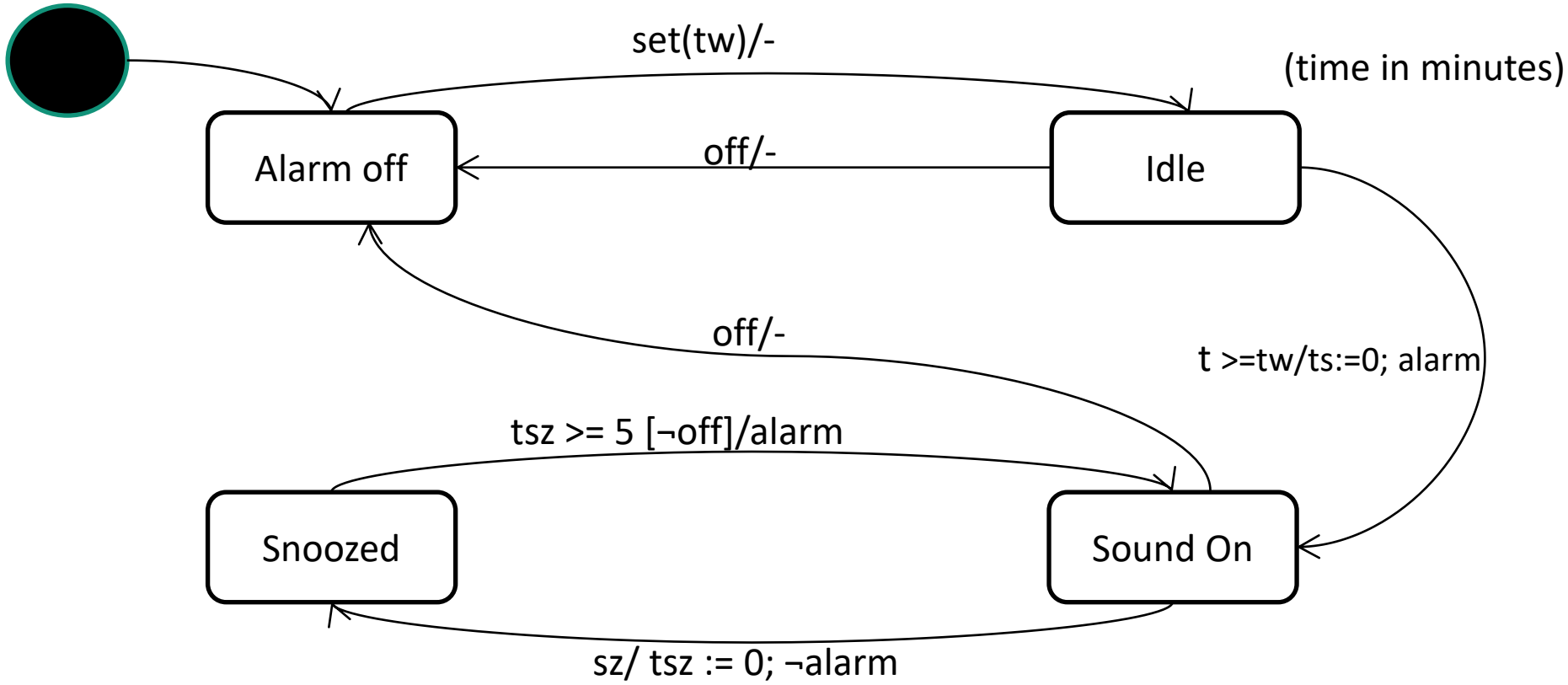
set(tw) = set the alarm time to tw and press the on button



t = current time	off = off button pressed
tw = wake-up time	
ts = sound start-time	alarm = play sound signal
	set(tw) = set the alarm time to tw and press the on button

Specification

- Kristian's alarm clock starts sounding at 6.00 with a nasty signal. He can now do either of three things:
  a) Turn the alarm off;
  b) Press the snooze button; or
  c) Do nothing.
- If the snooze button is pressed the signal will turn off and start sounding after 5 minutes again.
- When an hour has passed from the first time the alarm sound started, the snooze button has no effect.
- After that the alarm sound starts, the signal will last for 2minutes.
- If no action has been taken during these 2 minutes, the absence of action will have the same effect as if the snooze button were pressed exactly when the alarm stopped to sound



t = current time	off = off button pressed
tw = wake-up time	sz = snooze button pressed
ts = sound start-time	alarm = send signal
tsz = time snooze pressed	set(tw) = set the alarm time to tw and press the on button

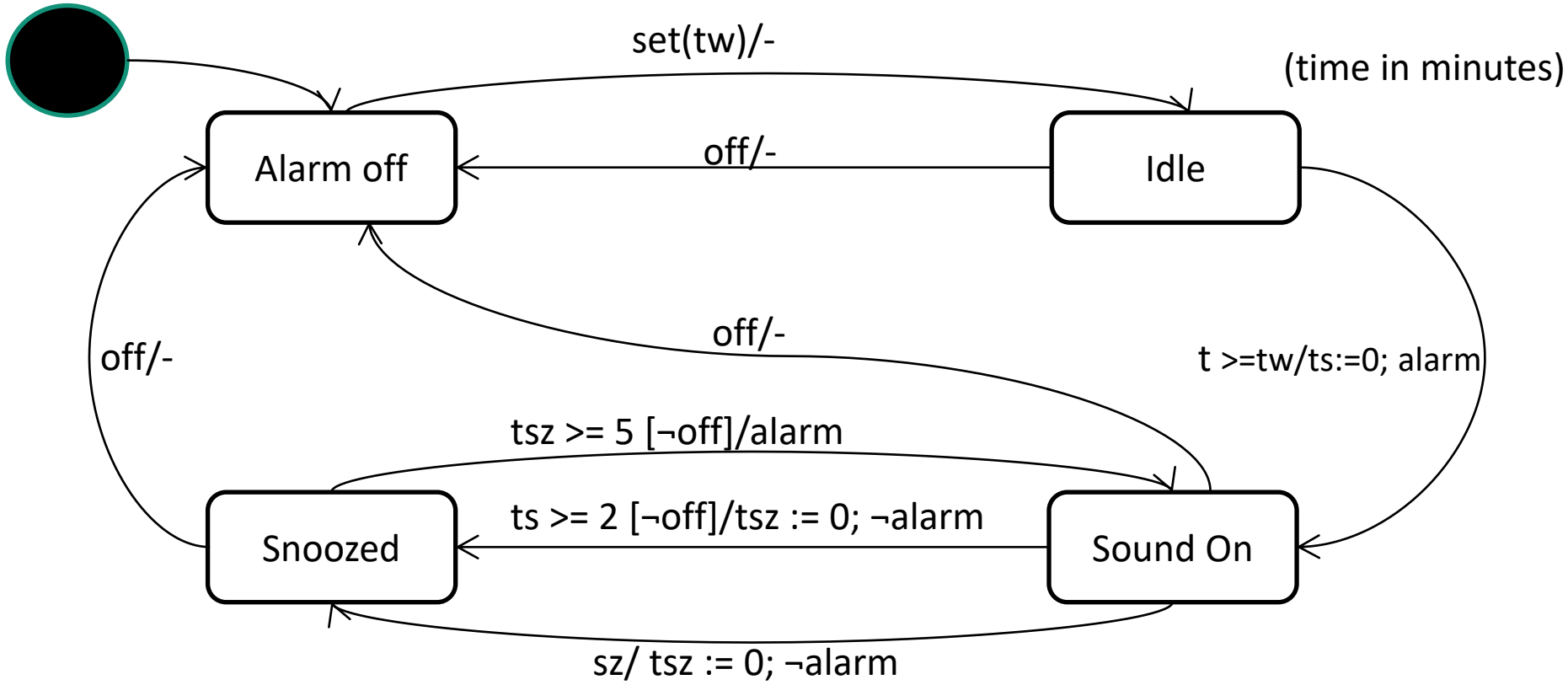
Specification

- Kristian's alarm clock starts sounding at 6.00 with a nasty signal. He can now do either of three things:

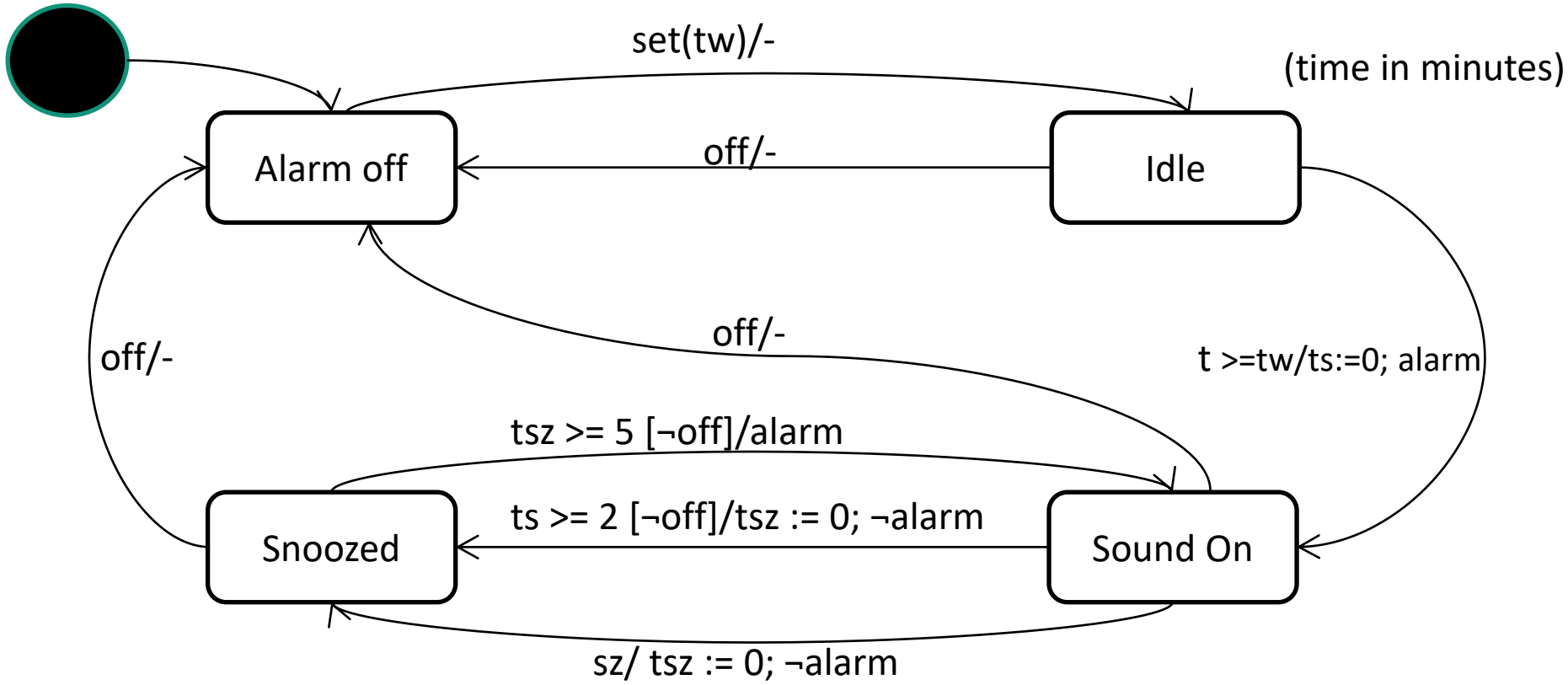
  a) Turn the alarm off;

  b) Press the snooze button; or

  c) Do nothing.
- If the snooze button is pressed the signal will turn off and start sounding after 5 minutes again.
- When an hour has passed from the first time the alarm sound started, the snooze button has no effect.
- After that the alarm sound starts, the signal will last for 2minutes.
- If no action has been taken during these 2 minutes, the absence of action will have the same effect as if the snooze button were pressed exactly when the alarm stopped to sound

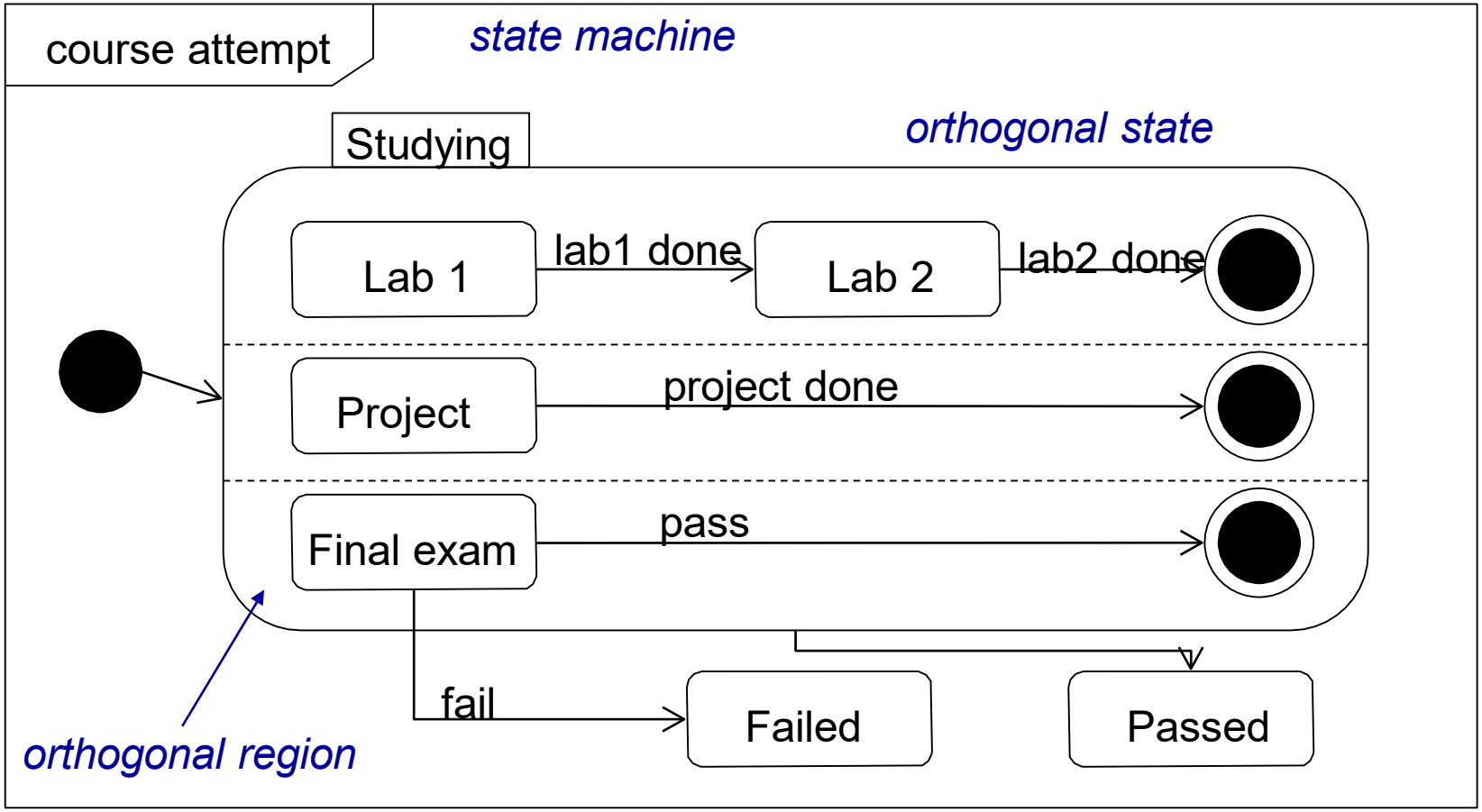


t = current time	off = off button pressed or t >= tw+60
tw = wake-up time	sz = snooze button pressed
ts = sound start-time	alarm = send signal
tsz = time snooze pressed	set(tw) = set the alarm time to tw and press the on button

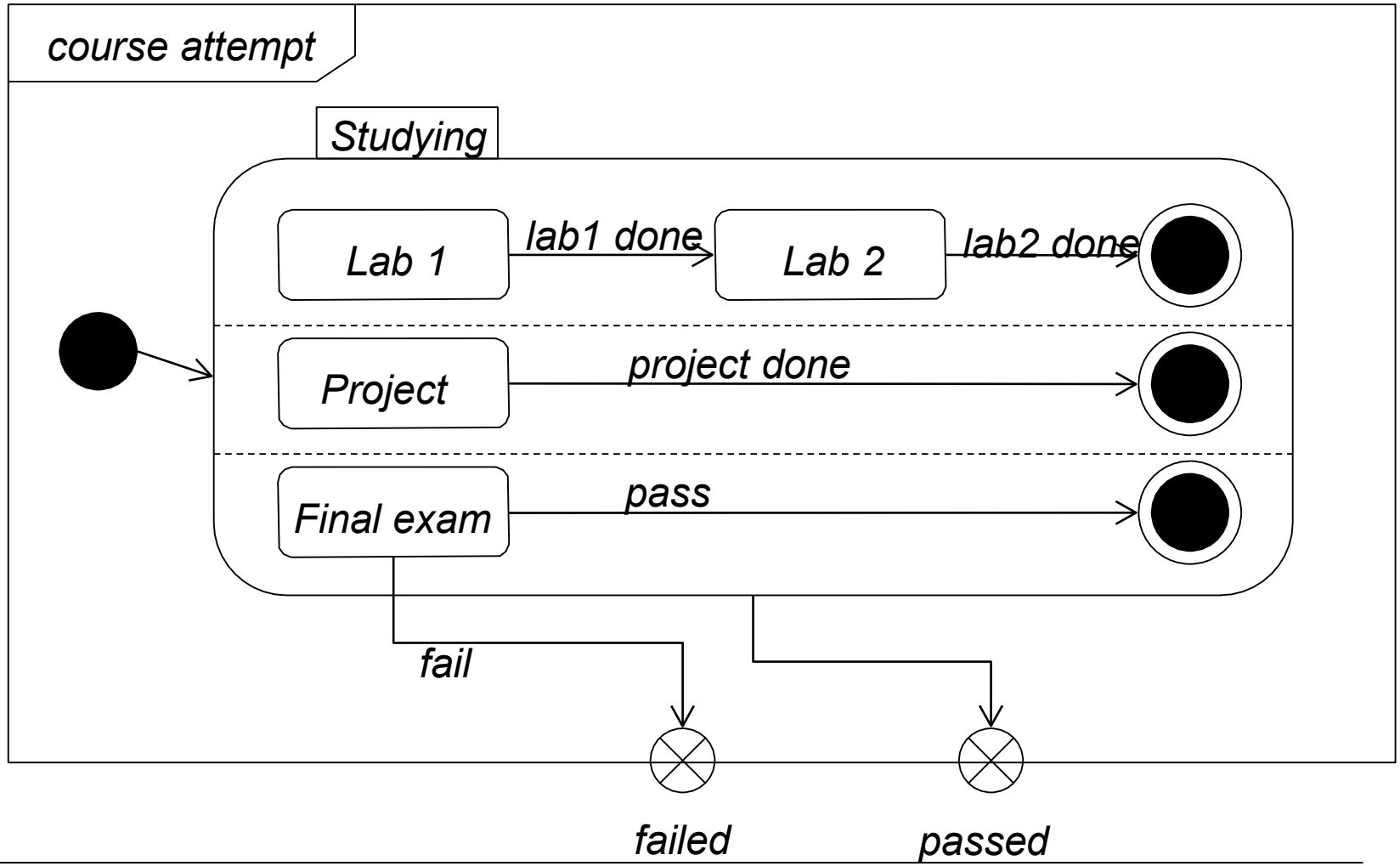


t = current time	off = off button pressed or t >= tw + 60
tw = wake-up time	sz = snooze button pressed
ts = sound start-time	alarm = send signal
tsz = time snooze pressed	set(tw) = set the alarm time to tw and press the on button

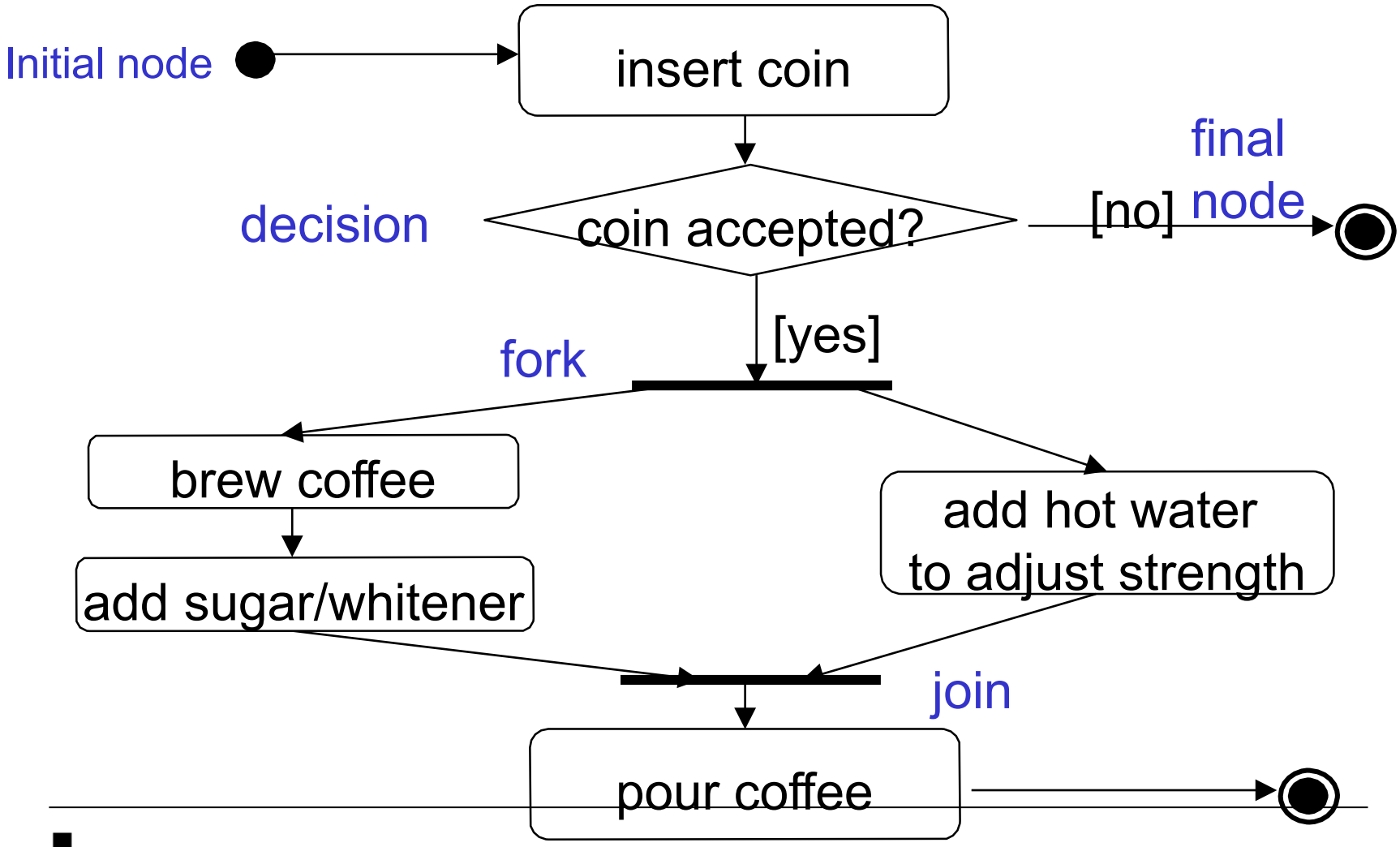
Orthogonal, composite states



Explicit exit points



Activity diagram ≠ State diagram



Summary

- Structural diagrams
  - Class vs. Objects, Attributes, Relationships
- Behavioral diagrams
  - Sequence diagram
  - State machine diagram
- Domain analysis vs implementation

Preparation for Friday (Modeling Practice)

A review management system (REMS) help the review of scientific journal papers submitted by researchers. Authors submit a paper by using a form to specify a title, an abstract, a list of keywords and a first version as PDF document. They may also suggest names for excluded reviewers. When a new submission is received, REMS assigns a qualified editor to manage its review process by matching the keywords of the paper with editors' expertise. An editor sends invitation to several reviewers (not excluded by the authors) who either accept or decline this invitation. When two reviewers agree to review the paper, no further reviewers will be invited. A reviewer needs to complete a review which includes a textual critic and a recommendation: accept, minor revision, major revision or reject. Based upon the recommendations of the reviews, the editor makes a decision on the paper (which is also one of accept, minor revision, major revision and reject). If the decision is major revision, the authors need to resubmit a revised version of the paper, and the editor initiates a 2nd round of review, which is identical with the 1st round, except for excluding major revision as a possible outcome.

Write a **functional requirement** to capture that *only qualified editors will handle any paper*.
Write an **non-functional requirement** on *the availability* of the REMS system.

Draw a ***use case diagram*** for the REMS system highlighting key actors, use cases and their relations.

Draw a UML Class Diagram as domain model for the REMS system showing the domain concepts, their relationships and potential generalizations. Specify multiplicities for your associations and arrange all objects into a containment hierarchy by appropriate composition relations between classes.

Describe the **high-level workflow** of the *paper review process* using a UML Activity Diagram. You may assume that the successful invitation of a reviewer is separated into an activity called *Invite-and-Accept-Review* which you may use in your diagram. Your actions should have direct traceability to use cases!

Describe the **state-based behavior** of the "*Paper*" class by a UML Statechart Diagram. Use operations derived from use cases as triggering events of transitions. (The Paper class represents a submission that is handled by REMS for review.)

Modeling with UML /
Dániel Varró & Kristian Sandahl

www.liu.se

LINKÖPING UNIVERSITY