# Software Engineering

## UML—Unified Modeling Language

Silesian University of Technology

# Agenda



# Model

## Model

A model is a simplified abstract view of complex reality.

A model stresses important features and neglects the unimportant ones.

# Model

## Model

A model is a simplified abstract view of complex reality.
A model stresses important features and neglects the unimportant ones.

- A model simplifies the reality
- A model is not universal $\rightarrow$ should be used in the scope it has been created for

# Methodics

## Methodics

A standardised approach to solve a problem in some domain (it involves notation, models, techniques, . . . )

# Methodics

## Methodics

A standardised approach to solve a problem in some domain (it involves notation, models, techniques, . . . )

- Formal notation—strictly defined way of recording the information (from the analytic phase)—essential technique
- Auxiliary techniques
  - editorial tools
  - data repositories
  - CASE tools

# Objectives of models in software engineering

- Why do we build the model of the system?
  - to better understand the systems (natural help in comprehension of complex problems)
  - to simulate the behavior of the system

# Objectives of models in software engineering

- Why do we build the model of the system?
  - to better understand the systems (natural help in comprehension of complex problems)
  - to simulate the behavior of the system
- The models enable
  - better comprehension of the problem domain
  - better comprehension of the systems themselves
  - elaboration of the templates for future use
  - collection of knowledge, experience, and documentation of actions
  - consistency control and detection of contradictions

# Modeling rules

1. Each task has its own characteristics

# Modeling rules

1. Each task has its own characteristics
2. Each task needs proper adjustment of means to achieve the objective: small system $\rightarrow$ simple model, big system $\rightarrow$ complex model

# Modeling rules

1. Each task has its own characteristics
2. Each task needs proper adjustment of means to achieve the objective: small system $\rightarrow$ simple model, big system $\rightarrow$ complex model
3. Each task may be analysed at various levels of abstraction $\rightarrow$ model can be elaborated at various levels of detail

# Modeling rules

1. Each task has its own characteristics
2. Each task needs proper adjustment of means to achieve the objective: small system $\rightarrow$ simple model, big system $\rightarrow$ complex model
3. Each task may be analysed at various levels of abstraction $\rightarrow$ model can be elaborated at various levels of detail
4. A good model corresponds to the reality
5. A model does not create the reality, but it reflects the reality

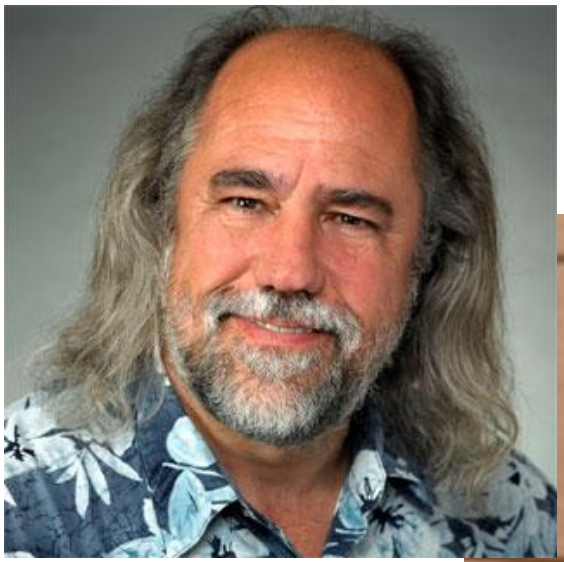
# Modeling rules

1. Each task has its own characteristics
2. Each task needs proper adjustment of means to achieve the objective: small system $\rightarrow$ simple model, big system $\rightarrow$ complex model
3. Each task may be analysed at various levels of abstraction $\rightarrow$ model can be elaborated at various levels of detail
4. A good model corresponds to the reality
5. A model does not create the reality, but it reflects the reality
6. No model is precise enough, as a model simplifies the reality

# Agenda



# UML—Unified Modeling Language



Grady Booch

James Rumbaugh

Ivar Jacobson

# UML—why to learn?


# Diagrams

## Diagram

Graphical, symbolic representation of information

- A diagram presents some view of the system

# UML diagrams

- *Structure diagrams*
  - class diagram
  - object diagram
  - package diagram
  - composite structure diagram
  - component diagram
  - deployment diagram
  - profile diagram
- *Behavioral diagrams*
  - use case diagram
  - activity diagram
  - state machine diagram
  - *interaction diagram*
    - sequence diagram
    - communication diagram
    - timing diagram
    - interaction overview diagram

# Agenda



# Structure diagrams

- Structure diagrams present the internal structure of the system, its parts, packages, components, classes, objects, . . . , and relations between them
- *Structure diagrams*
  - class diagram
  - object diagram
  - package diagram
  - composite structure diagram
  - component diagram
  - deployment diagram
  - profile diagram

# Agenda

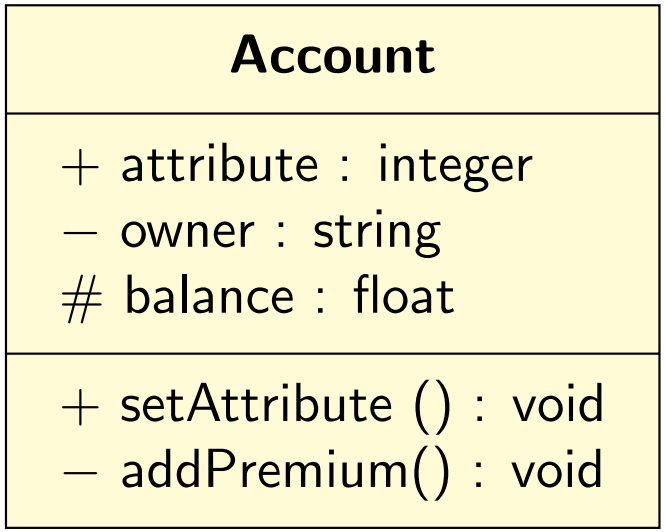


# Class diagram

- Depicts the structure of the system
- The most frequent diagram
- Class diagram presents
  - classes
  - interfaces
  - cooperations
  - and their relations

# Class diagram

- Top box: name of the class in bold
- Middle box: attributes
- Bottom box: methods (operations)

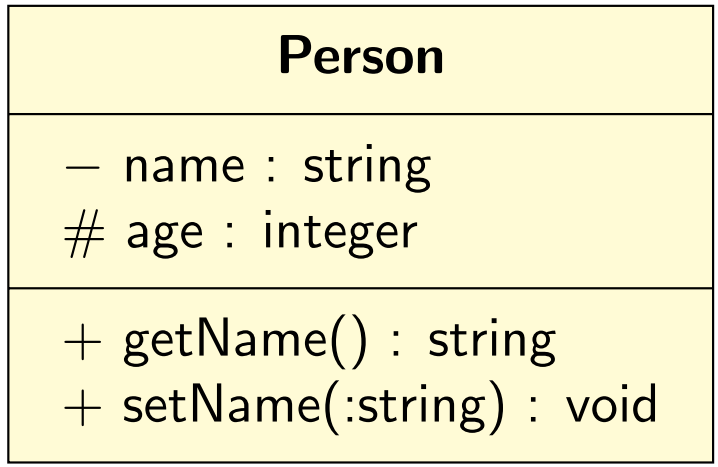



# Visibility

- An important feature
- Levels of visibility
  - public (+) – default in UML
  - protected (#)
  - private (−)

# Visibility

```
class Person
{
  private:
    string name;
  protected:
    int age;
  public:
    string getName();
    void setName(string);
};
```



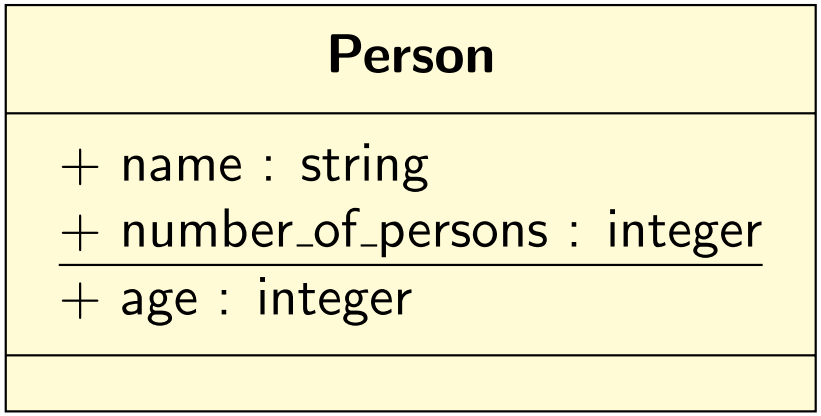

# Scope

**instance scope** each instance of the class has its own element

**class scope** (underlined) the element is common for all instances of the class

# Scope

```
class Person
{
  public:
    string name;
    static int
        number_of_persons;
    int age;
};
```

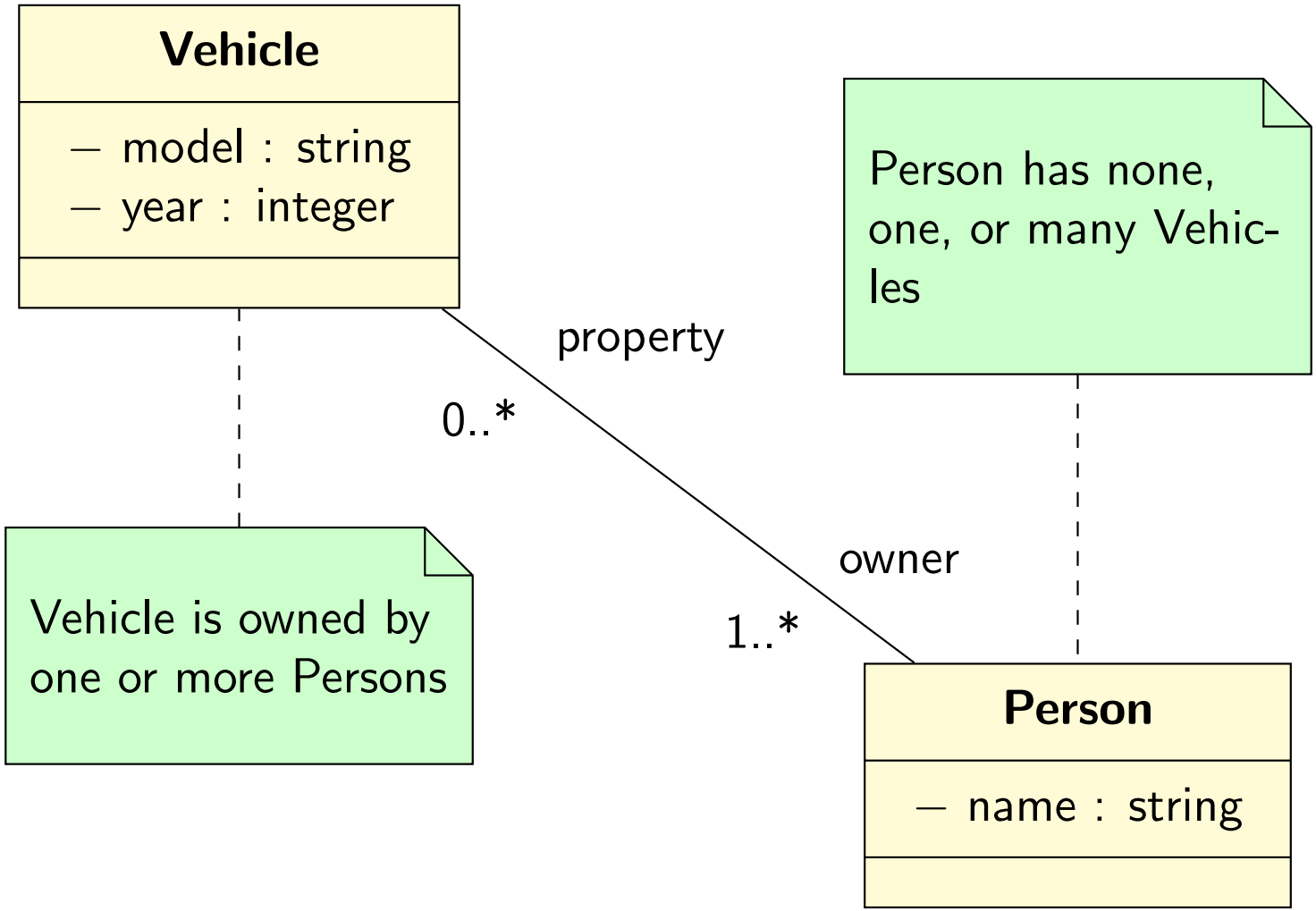



# Relations

Essential information on relation

- type—obligatory
- name
- roles' names
- multiplicity

# Example


Vehicle
– model : string
– year : integer
property
0..*
owner
1..*
Person has none, one, or many Vehicles
Vehicle is owned by one or more Persons
Person
– name : string

# Association

- Relatively weak relation
- General relation, not specialized
- The objects in relation communicate, collaborate


# Aggregation

- Reflects the phrases: *is built of*, *consists of*, *is a part of*
- Types of aggregation
  - (weak) aggregation
  - strong aggregation, composition

# Aggregation

- Weak aggregation
- Lifetimes of objects in relation are independent
- Does not imply ownership


# Aggregation


```
class Train
{
    std::vector<Wagon *> m_cars;
    // ...
};
```

# Composition

- Strong aggregation
- Destruction of an owner implies destruction of its parts (components)
- Components are owned by a composite


# Composition


```
class Window
{
    std::vector<Button> m_buttons;
    // ...
};
```

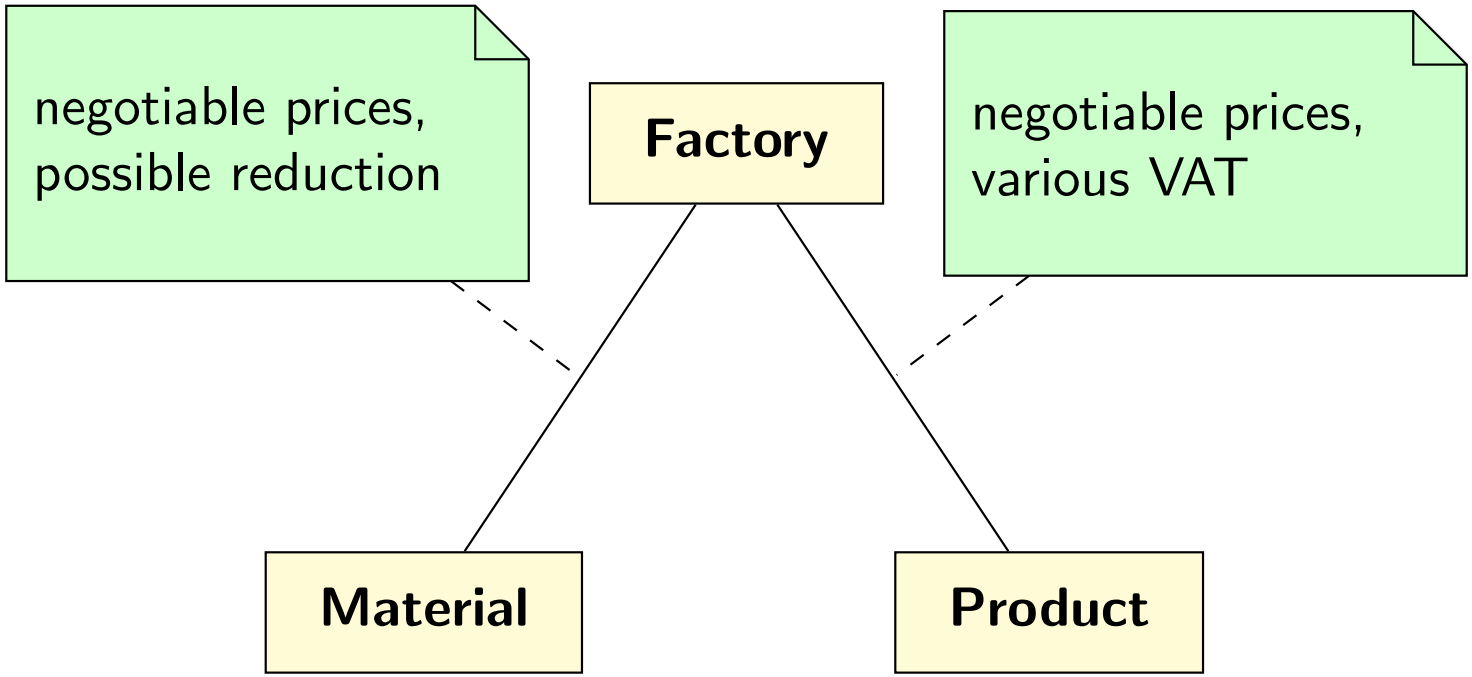
# Association class

## Association class

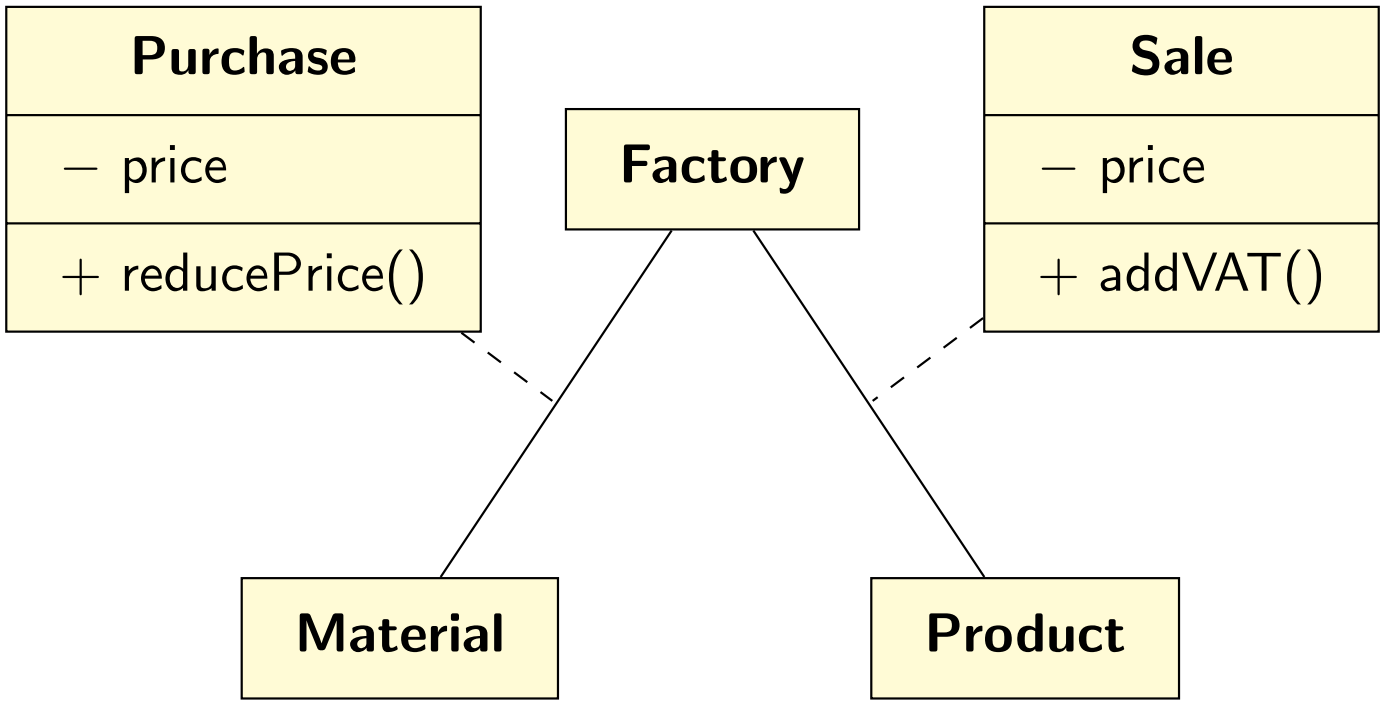
A class which has the properties of the relation

- Models the attributes and operations of the relation
- Models the $n$-to-$m$ relations

# Association class

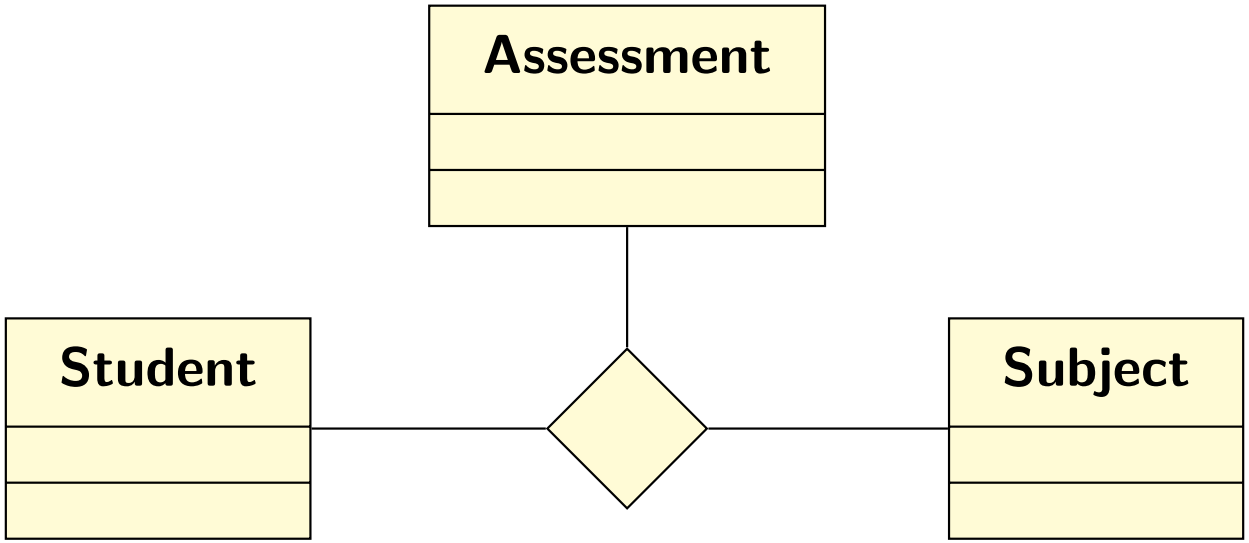



# Association class



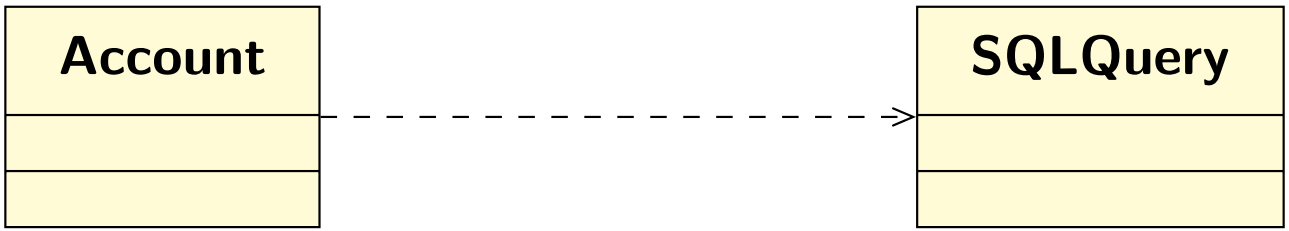

# $n$-ary association

- Relation between $n > 2$ classes
- Always bidirectional

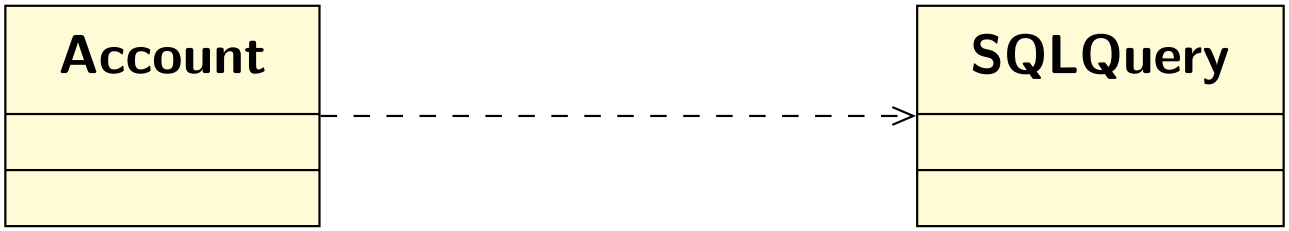



# Dependency

- The weakest relation
- One object uses the object of another class




# Dependency

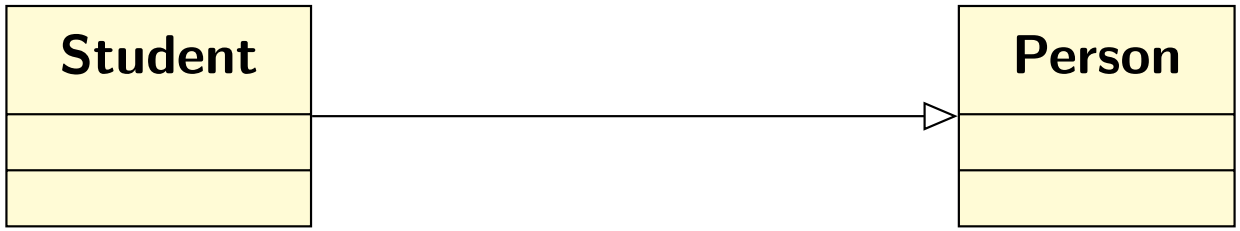



```
void Account::save()
{
    SQLQuery * query = new SQLQuery();
    query->Execute();
    // ...
    delete query;
}
```

# Generalisation

- Represents the generalisation and specialisation
- Inheritance (in programming languages)
- Reflects the phrase: *is a kind of*

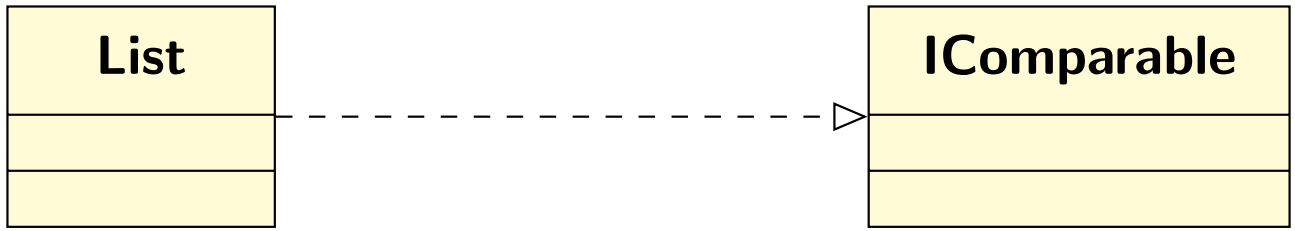

# Generalisation



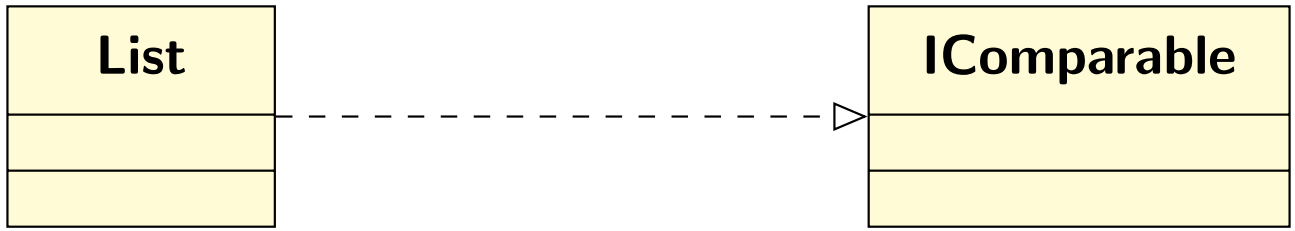

```
class Person
{
    // ...
};

class Student : public Person
{
    // ...
};
```

# Realisation

1. Implementation
2. The class inherits abstract features and implements them
3. Usually the base class is an interface

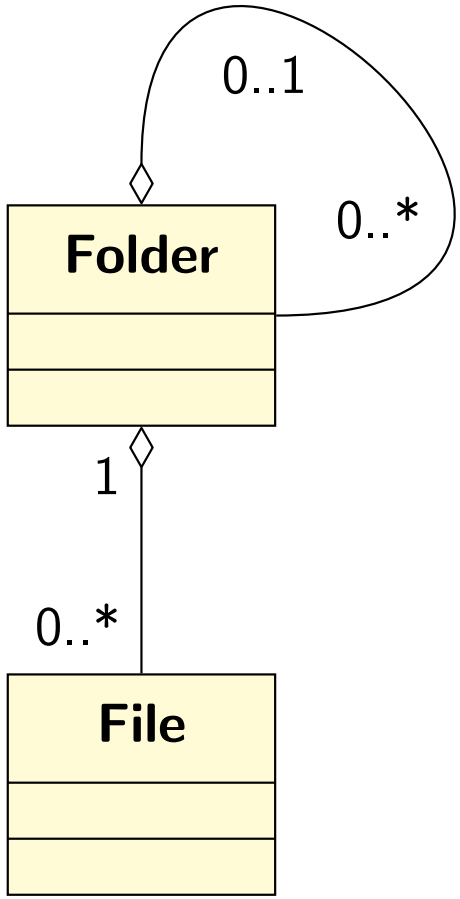



# Realisation

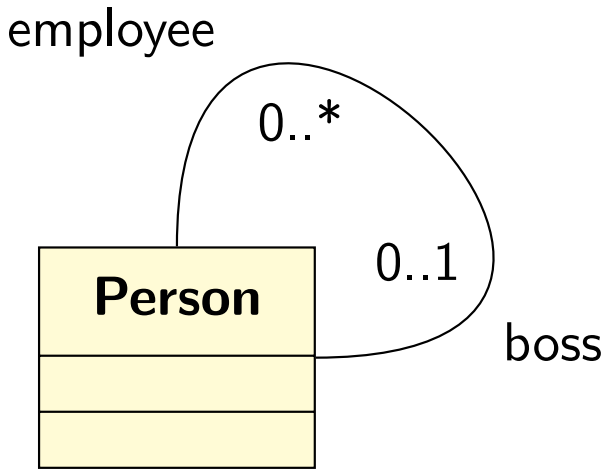



```
class IComparable
{
    public: int Compare ( /* */ ) = 0; // pure virtual
};
class List : public IComparable
{
     public: int Compare ( /* */ )
                {
                    // implementation
                }
};
```

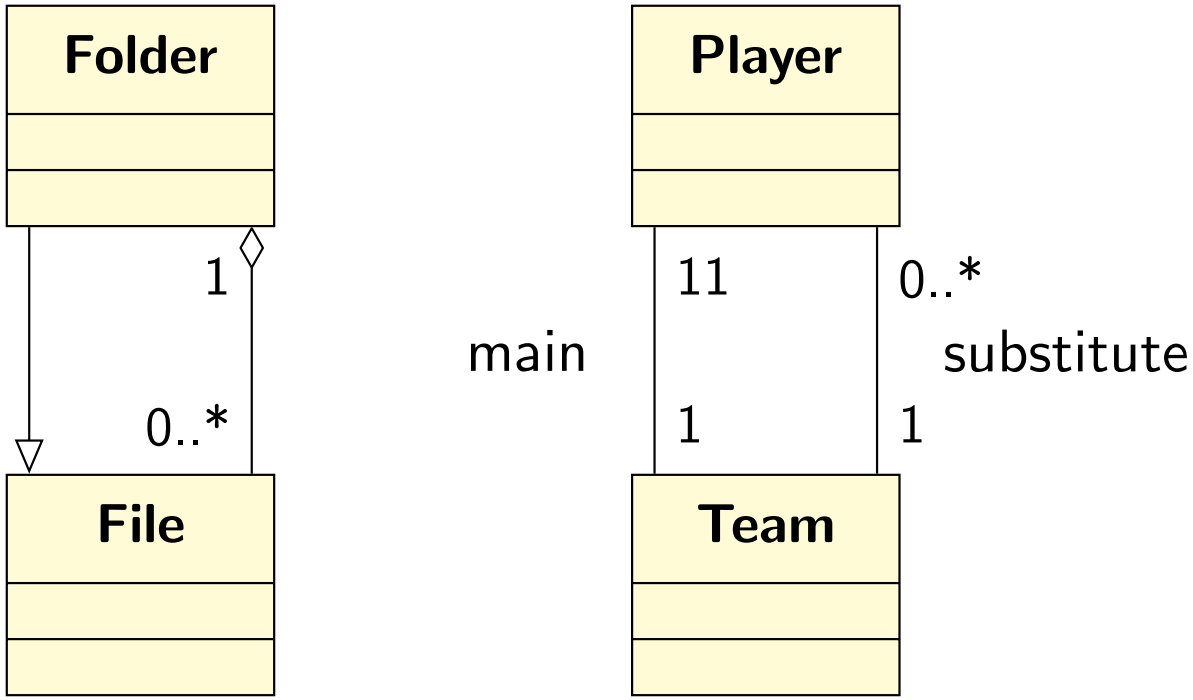
# Examples (tree-like structures)




# Examples (tree-like structures)




# Example (multiple relations)

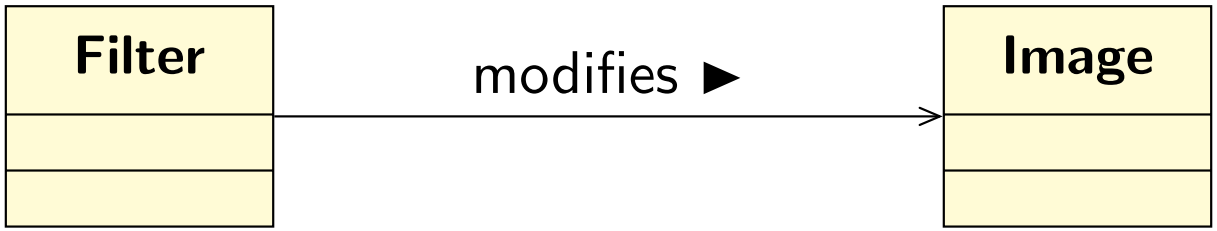


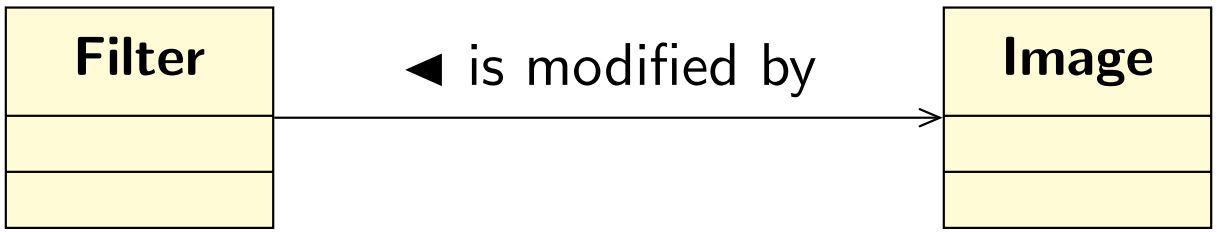

## Navigability

Typical direction of relation


# Navigability

The direction of reading a name is independent from the navigability



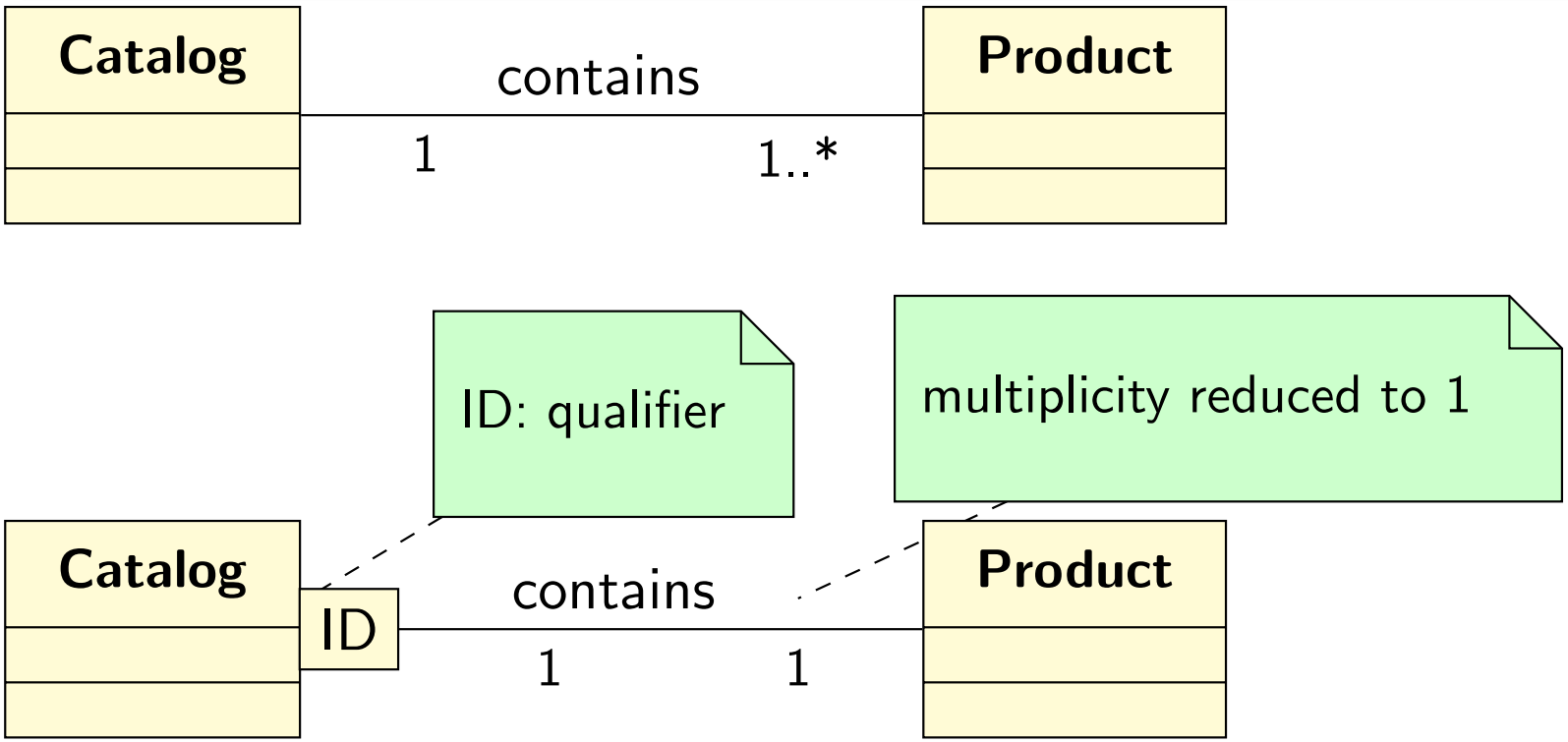

# Navigability

- The navigability is usually redundant and obfuscates the diagram
- Should be used for unidirectional relations

# Qualified association

## Qualified association

Attribute of the relation used for identification of the objects

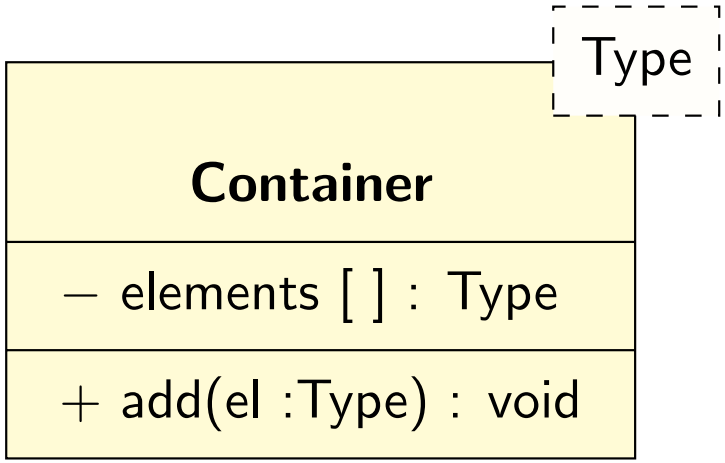



# Class template

## Class template

A generic class, class with parameterized types of attributes or operations

```
template <class Type>
class Container
{
    Type * elements;
  public:
    void add(Type el);
};
```




# Classifier

## Classifier

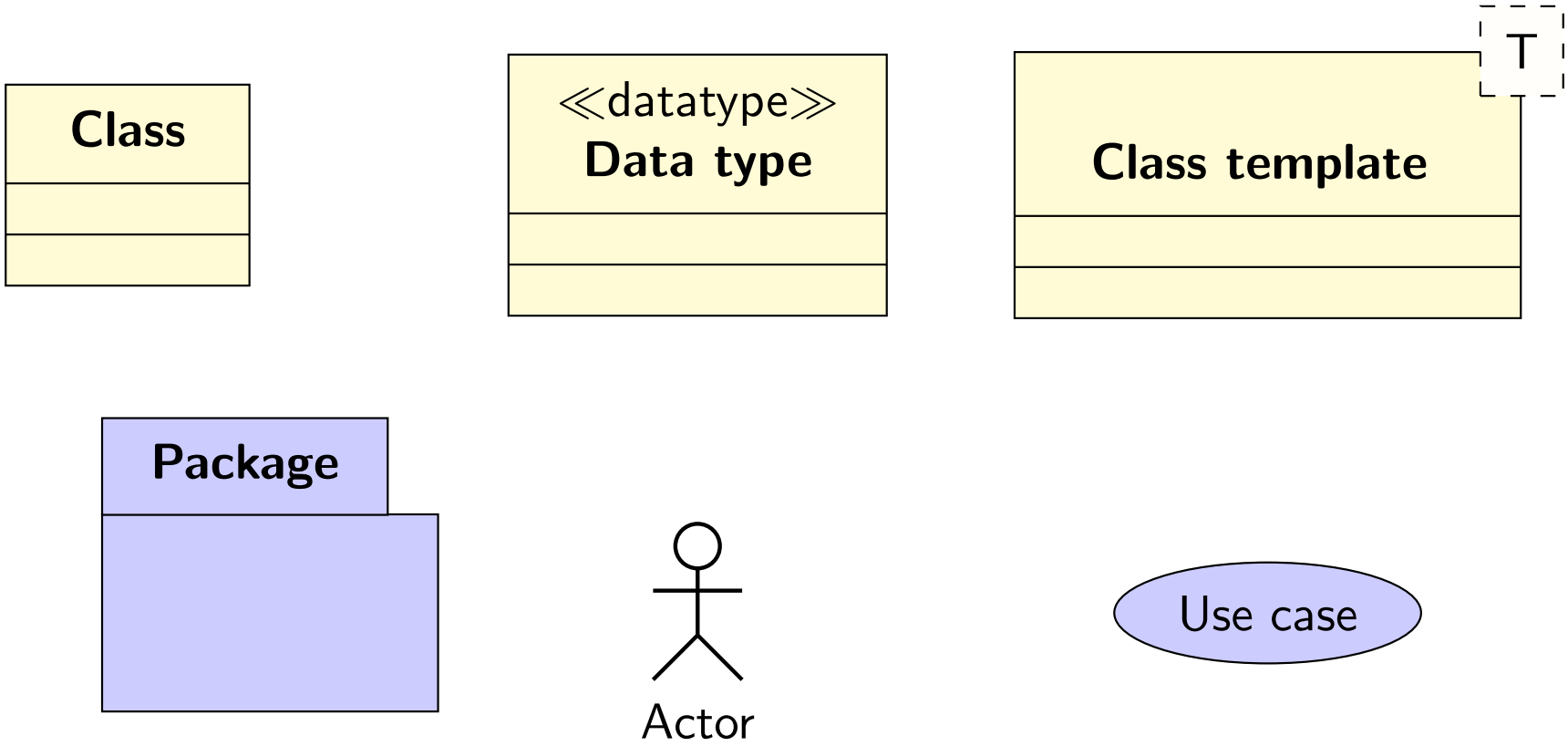
An abstract UML metaclass to support classification of instances according to their features (or an element that can have instances)

- A classifier has attributes and (often) operations
- All instances of a classifier have the same set of attributes and operations

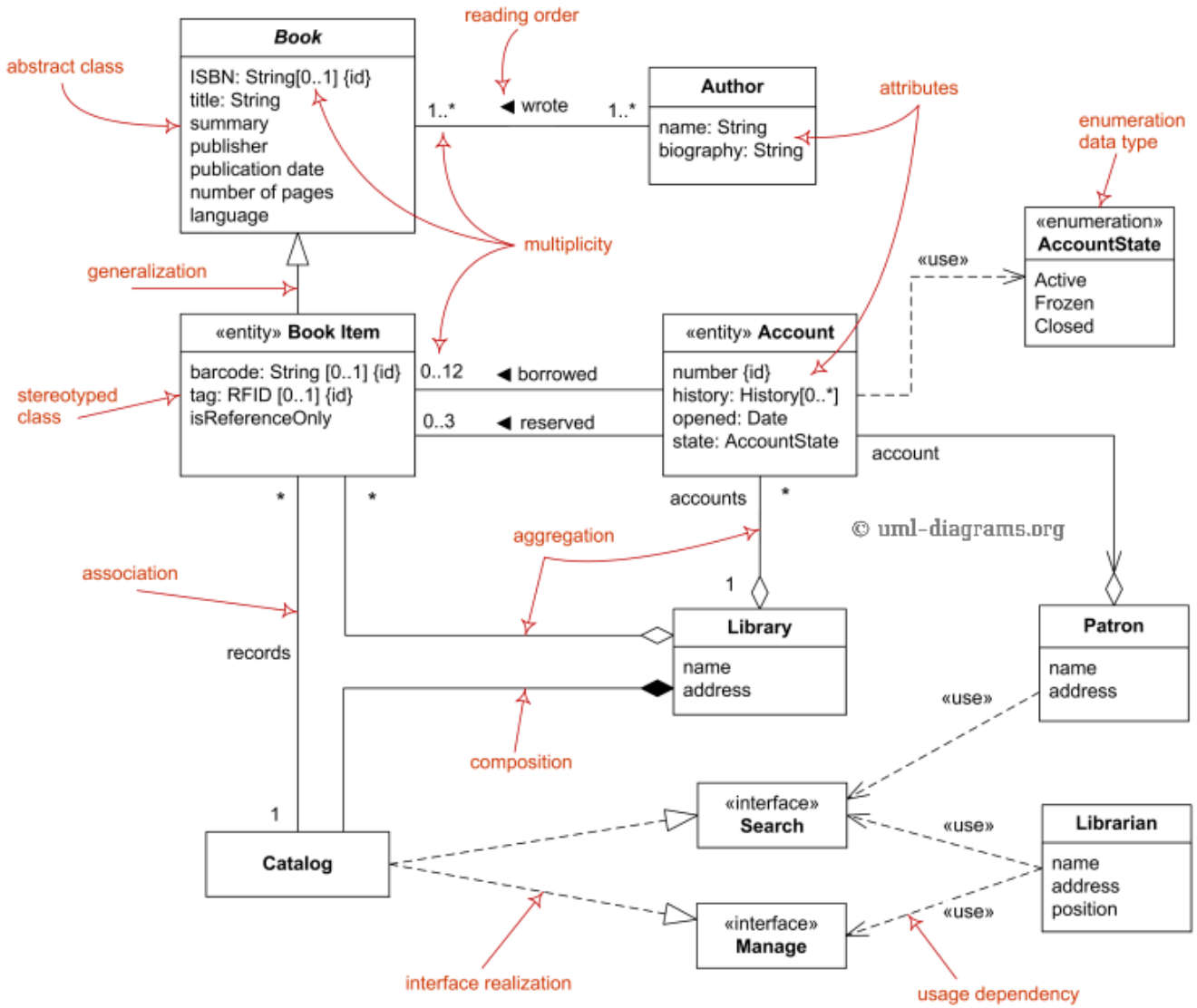
# Classifier—types

| | |
|---|---|
| Class | description of objects with the same set of attributes, operations, relations, and meaning |
| Relation | between instances of classifiers |
| Interface | set of operations |
| Data type | built-in types and enumerations |
| Signal | passed between instances of classifiers |
| Component | physical, replaceable part of system |
| Node | physical element with some calculation power |
| Use case | list of steps needed to achieve goals |
| Subsystem | components forming an integrated whole |

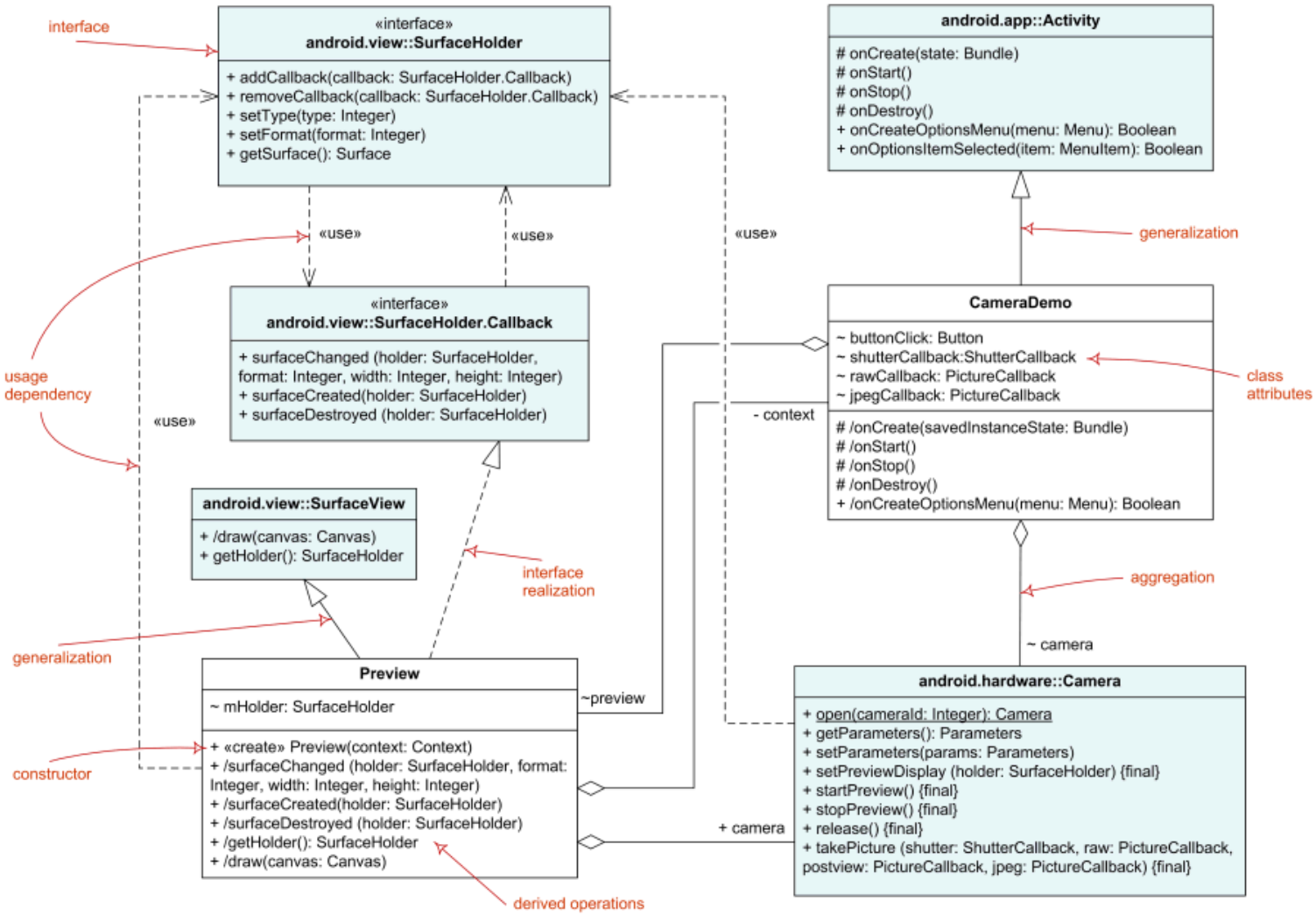
# Symbols for classifiers




# Class diagram—example




# Class diagram—example



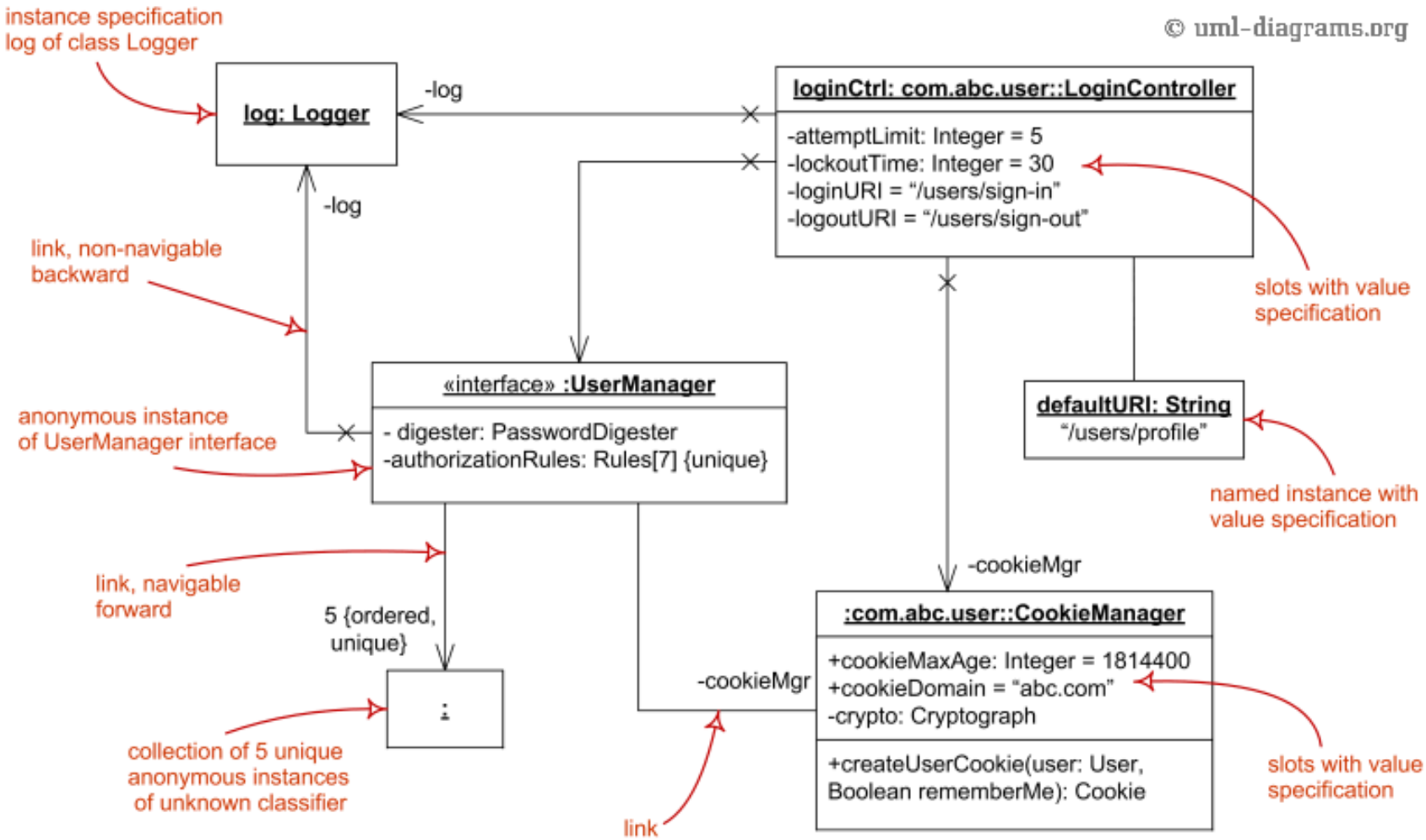

# Agenda



# Object diagram

- Object diagram
  - an instance of a class diagram
  - a snapshot of a system at a given time
- Purpose of object diagrams
  - verification of class diagrams
  - modeling specific cases during the system lifetime
  - usually redundant

# Object diagram



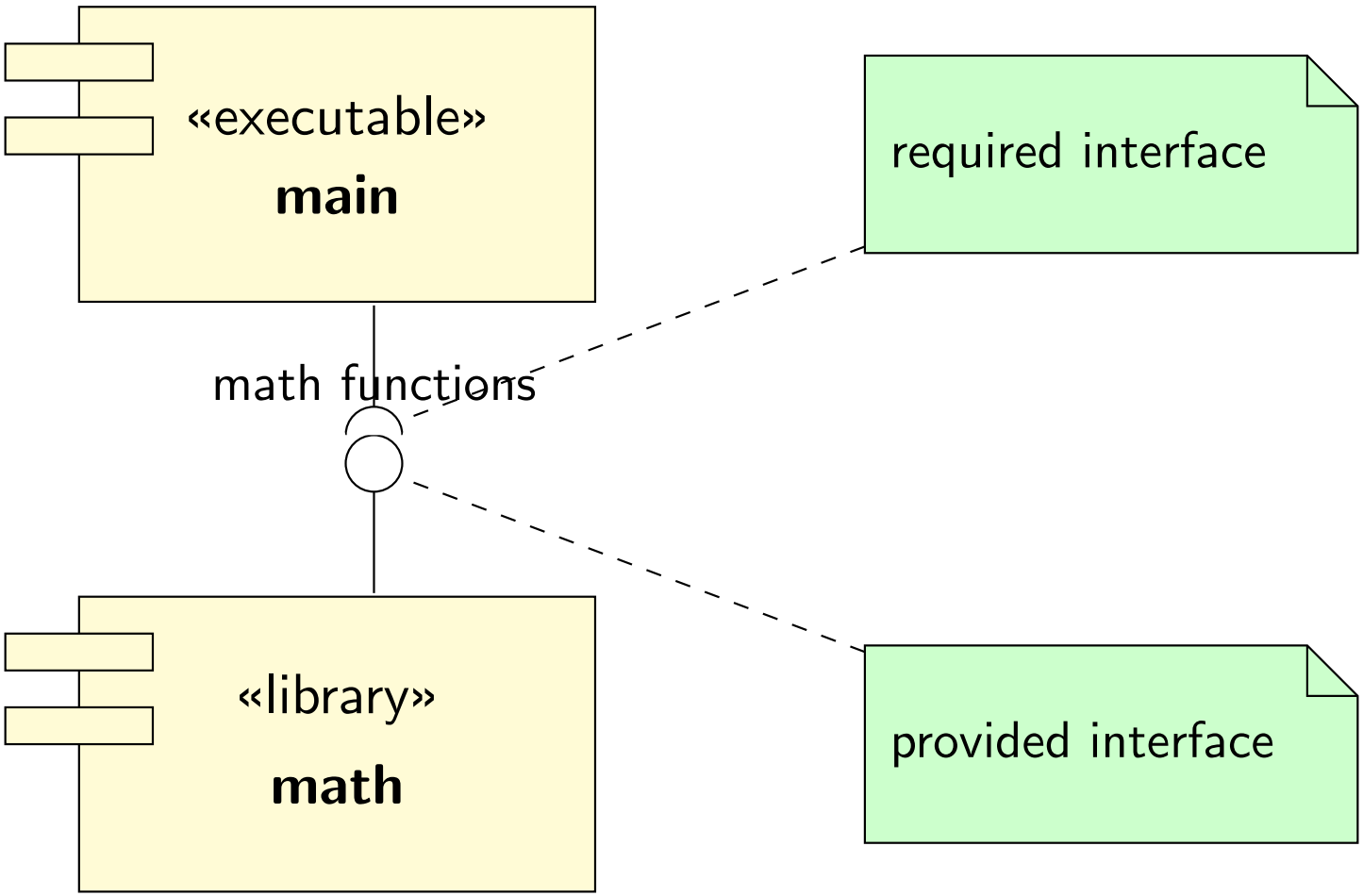

# Agenda



# Component diagram

- Component diagram
  - both logical and physical perspective
    - logical (classes, interfaces, . . . )
    - physical (libraries, files, programs, . . . )
  - higher level than a class diagram

# Component diagram




# Agenda

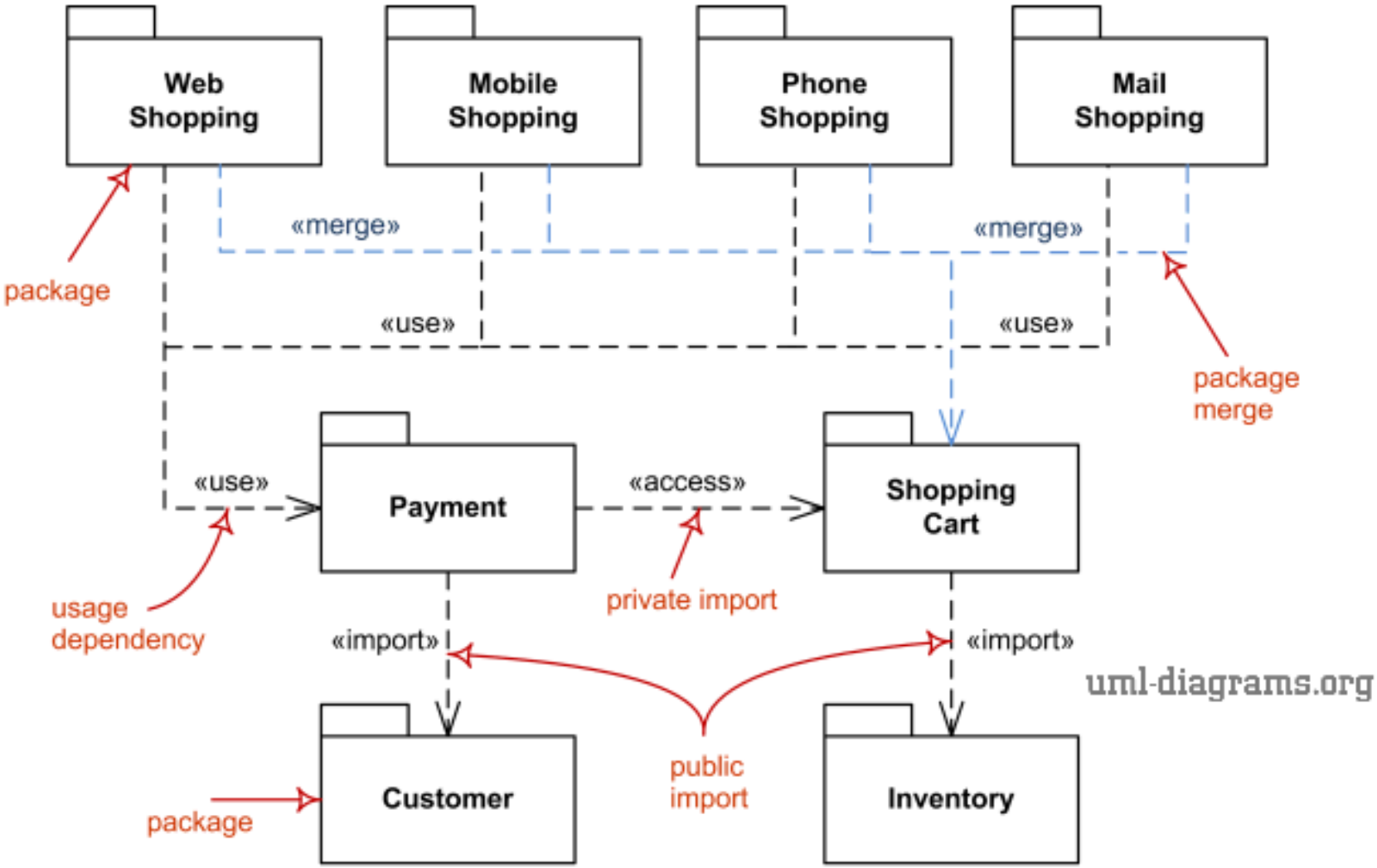


# Package diagram

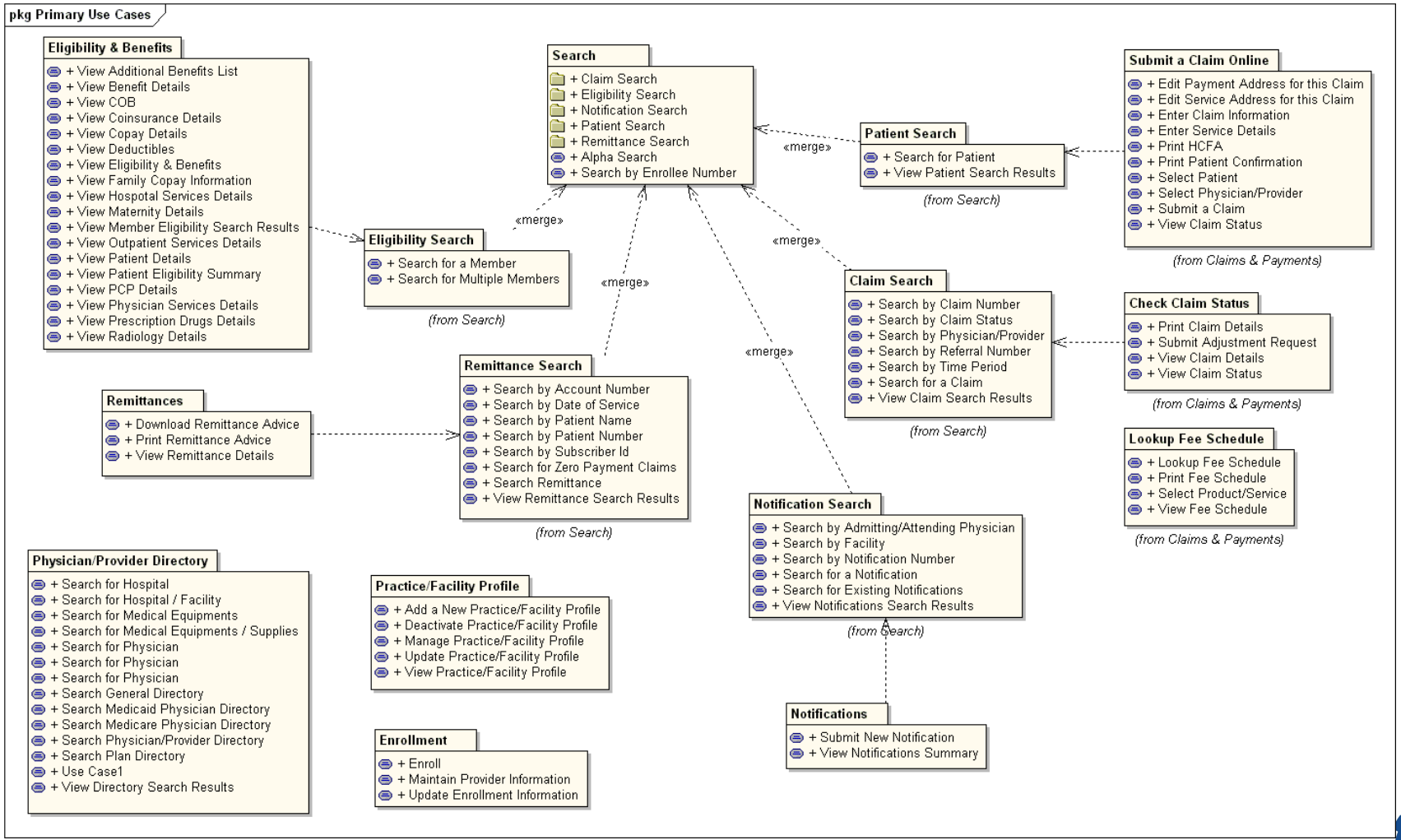
A package diagram presents the relations between the packages

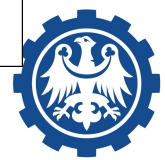
- relations:
  1. `<<import>>`
     - the elements of the imported package are visible in the importing package
     - a default relation
  2. `<<merge>>`
     - merging of packages
     - the features of the elements of both packages are mutually visible
  3. `<<access>>`
     - one package requires help from functions of other package
- difference between component and package diagram
  - component diagram—models interfaces
  - package diagram—models large-scale groups

# Package diagram



# Package diagram



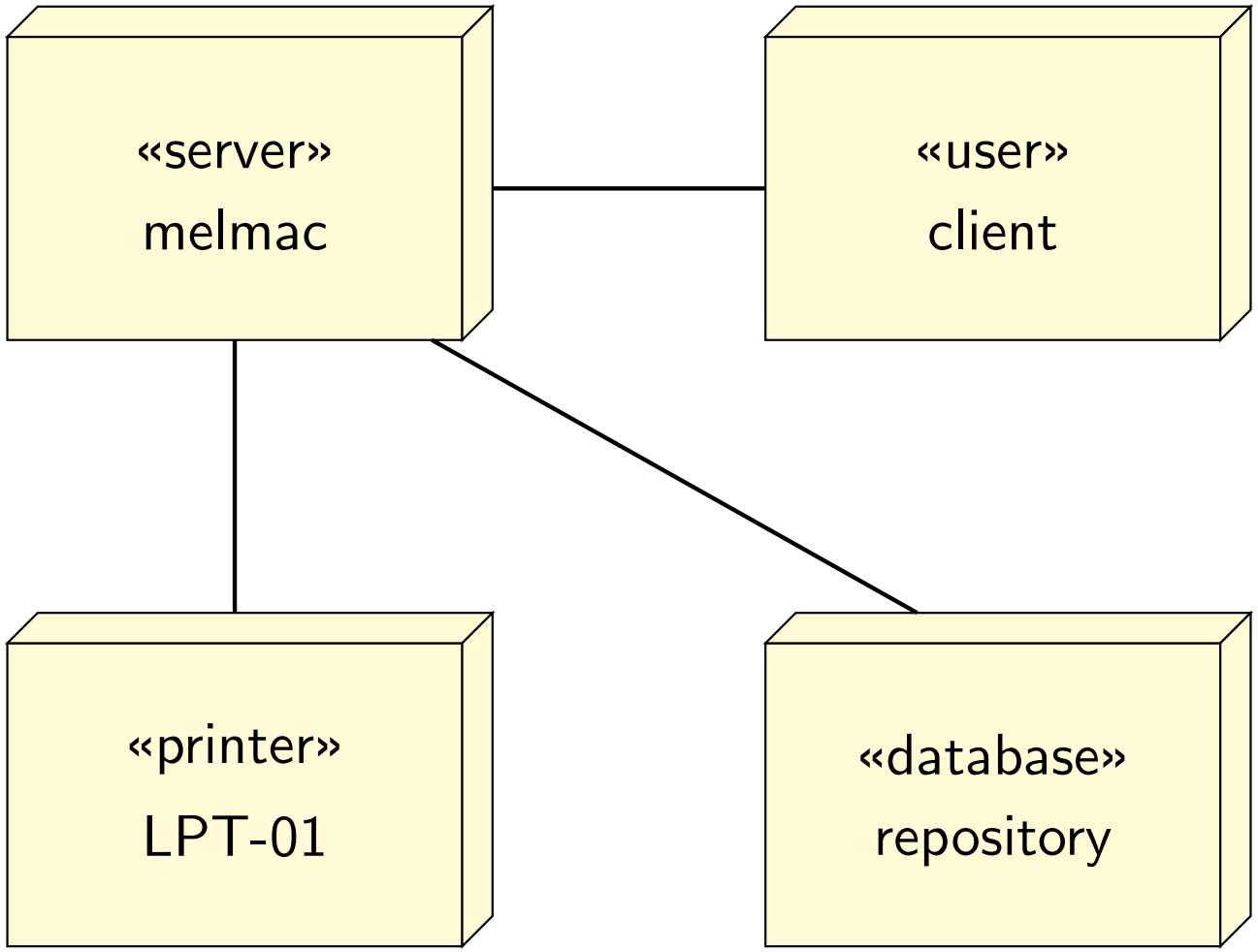
# Agenda



# Deployment diagram

- Presents physical aspects of the system
- Shows physical nodes that hold the components of the system
- Easy to read for clients

# Deployment diagram




# Agenda



# Behavior diagrams

- Behavior diagrams model the behavior of a system
- Types of behavior diagrams
  - use case diagrams
  - activity diagrams
  - state machine diagrams
  - *interaction diagrams*
    - sequence diagrams
    - communication diagrams
    - timing diagrams
    - interaction overview diagrams

# Agenda



# Use case diagram

- Use case diagram
  - specify the expected behavior of a system
  - does not specify the exact method (and order) for providing the expected behavior
  - useful in specifying functional requirements
  - presents actors and use cases alongside their relations
  - does not model all the use cases (should not be more than 20 use cases)

# Use case diagram—items

**A use case**

An action of the system aimed at providing a desired result to an actor

- Strictly linked with the system functionality
- Shows what system does, not how
- Located inside the system

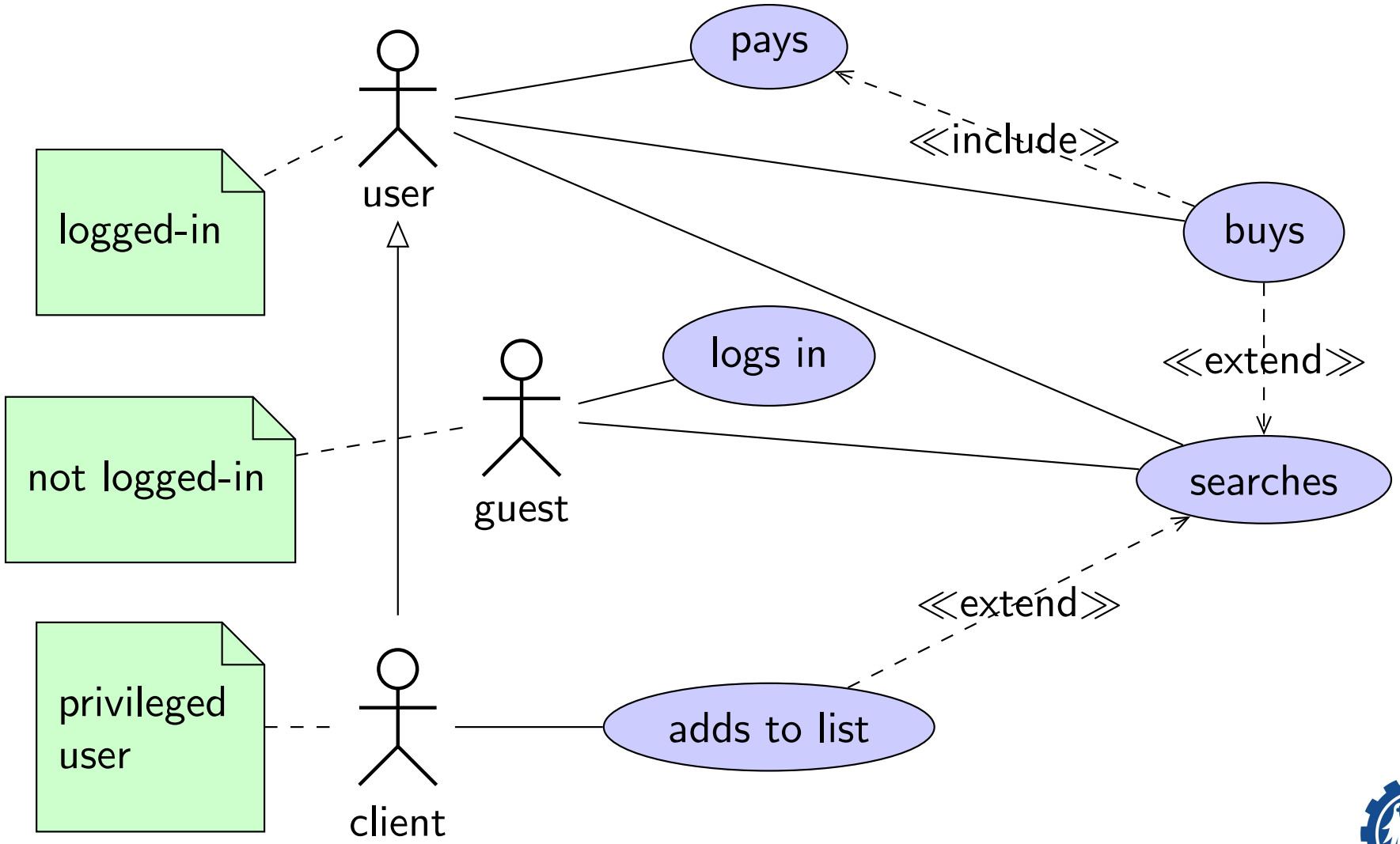
# Use case diagram—items

## An actor

Represents an entity that interacts with the system

- A logical item
- A user (human) or other system
  - several entities can act as a single actor
  - one entity can act as several actors
- An actor is not a part of our system

# Use case diagram



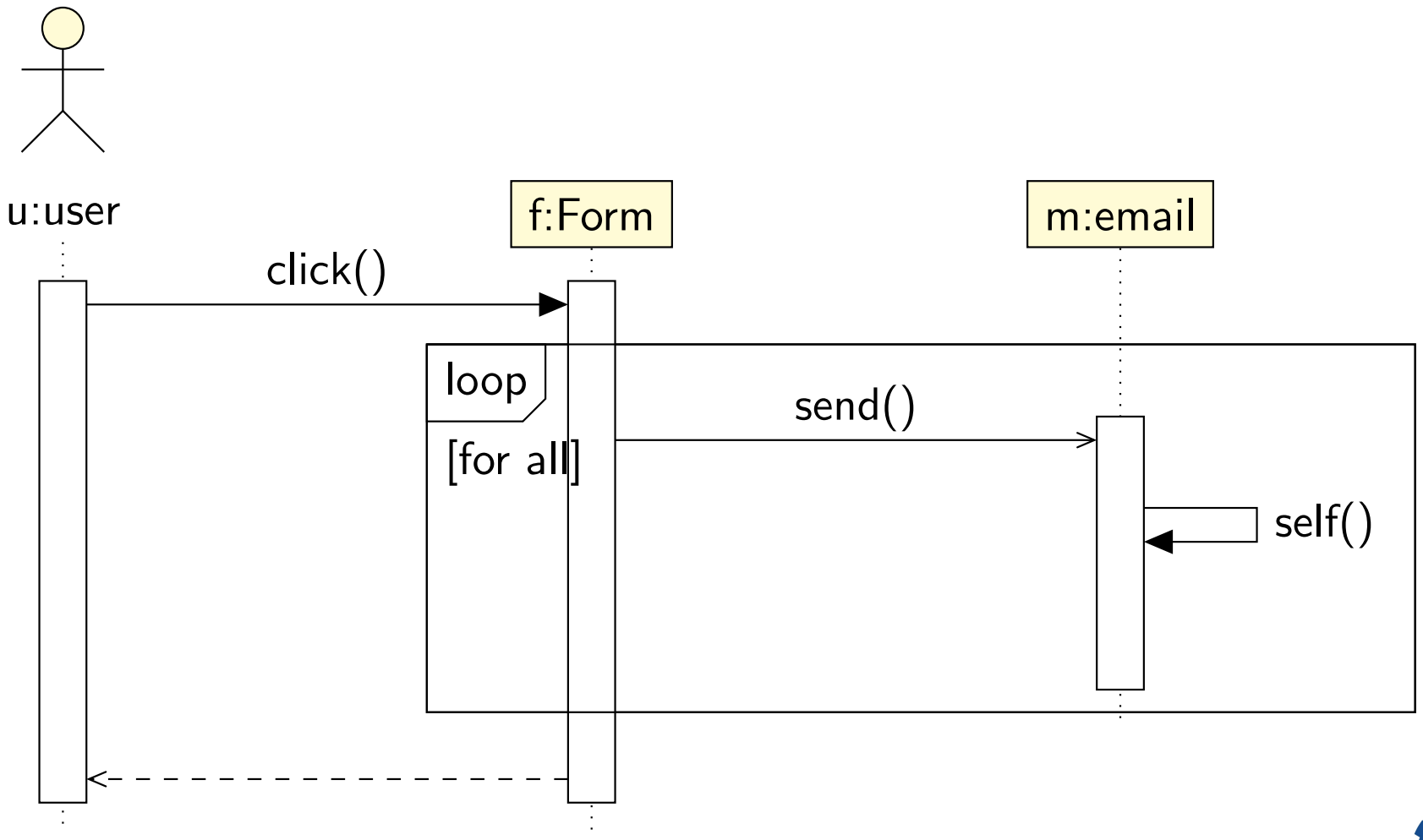

# Agenda



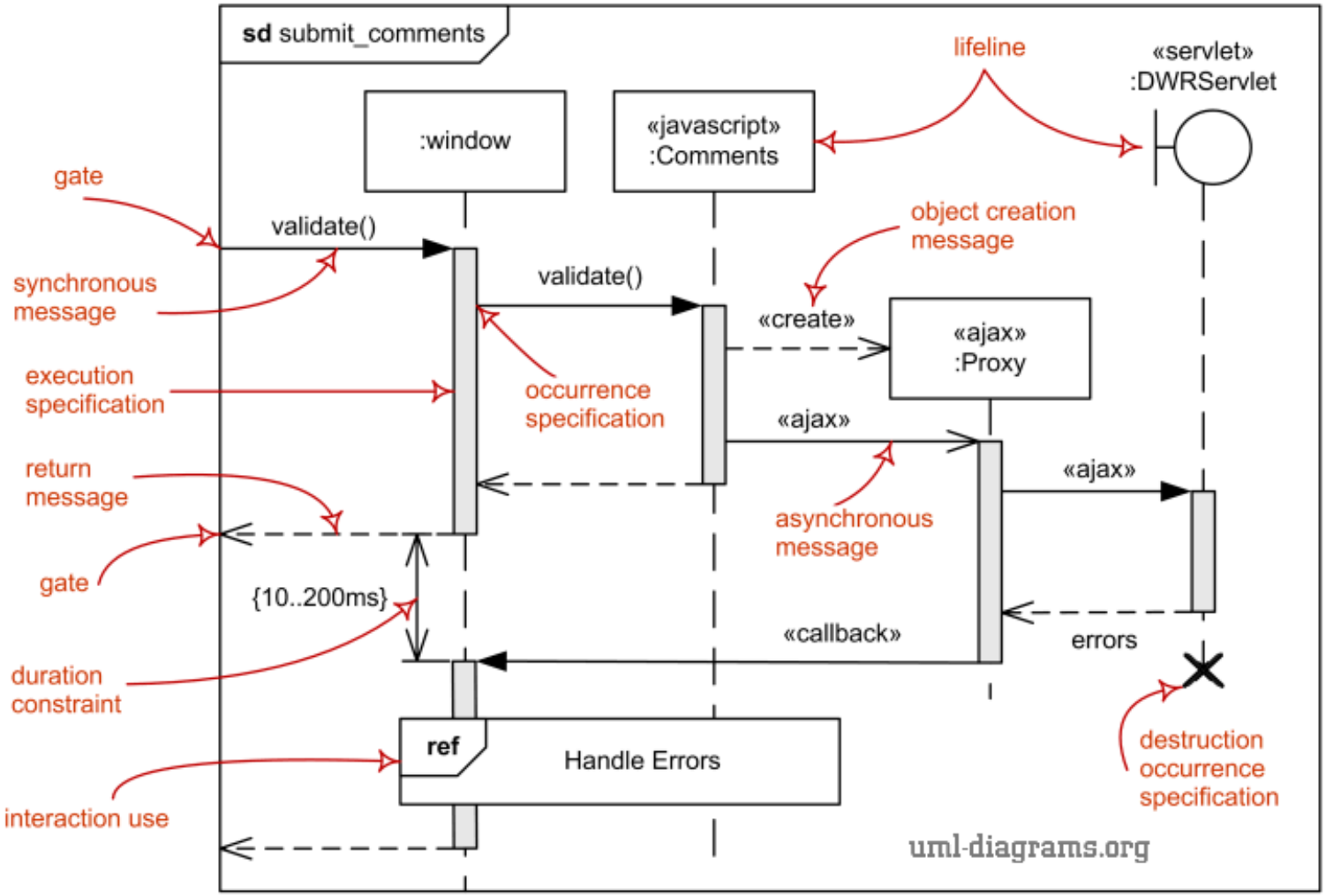
# Sequence and communication diagrams

- Two diagrams to model interaction
  - sequence diagrams
  - communication diagrams (collaboration diagram)
- Collaboration between objects, their interactions, and messages
- Sequence diagrams show object interactions arranged in time sequence
  - a diagram can model a single use case
- Communication diagrams stress the structure of cooperation

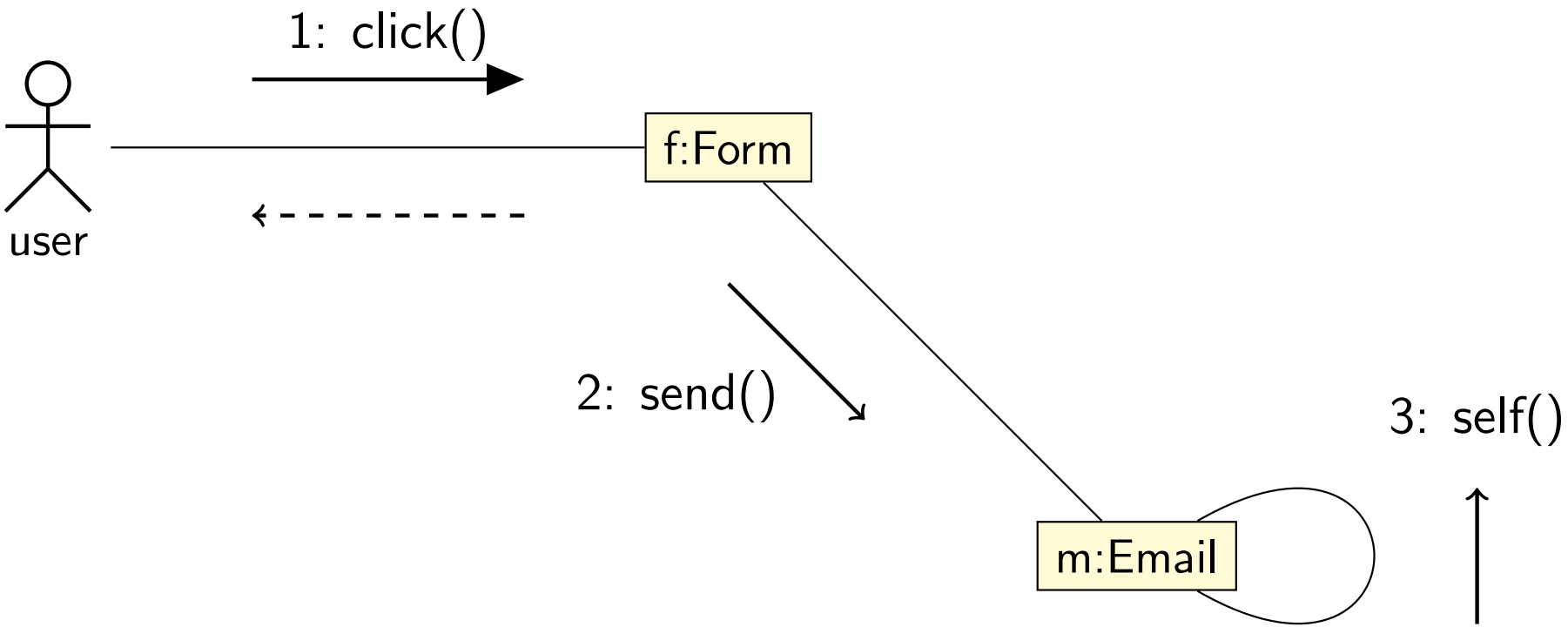
# Sequence diagram




# Sequence diagram



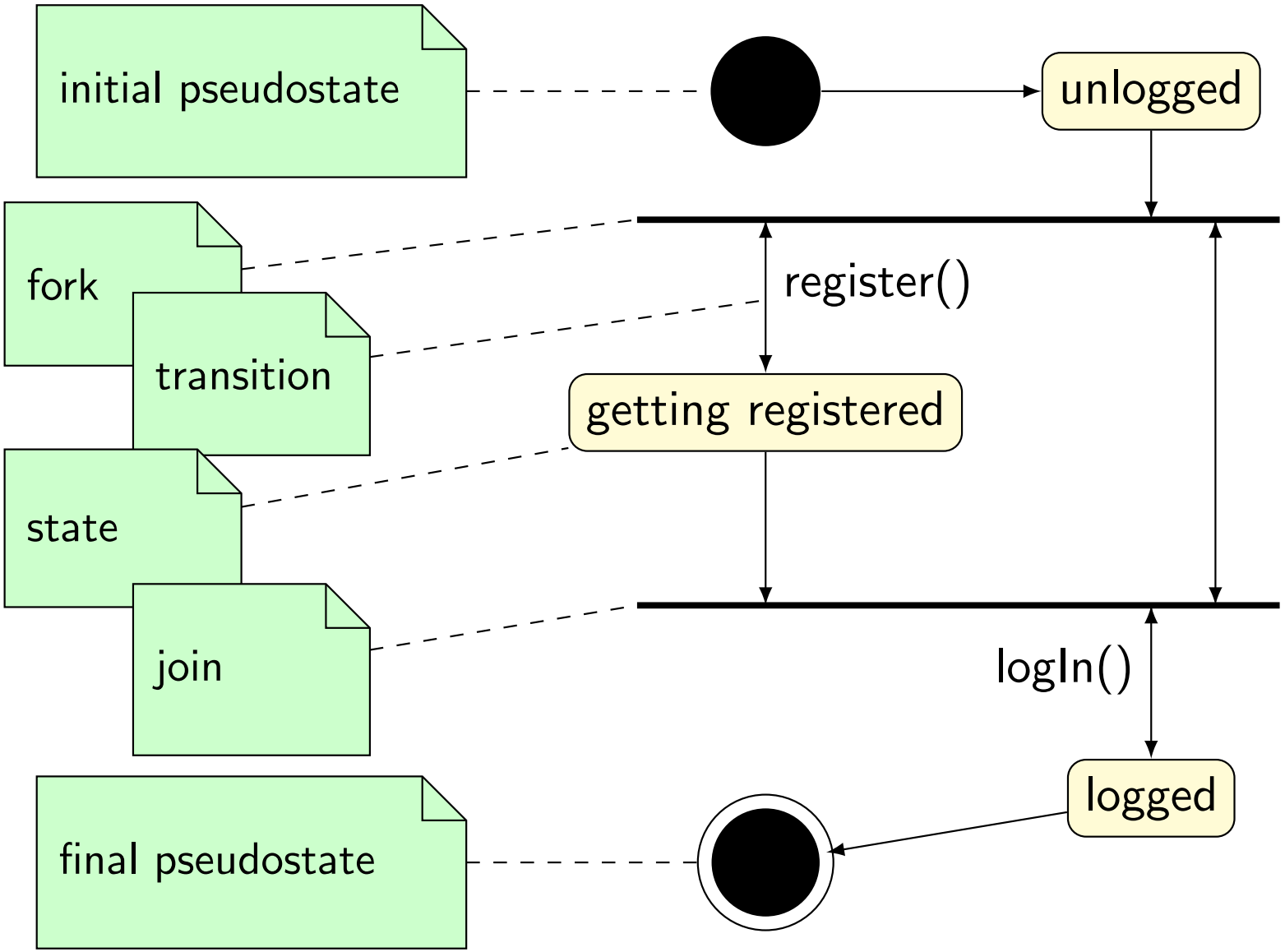

# Communication diagram



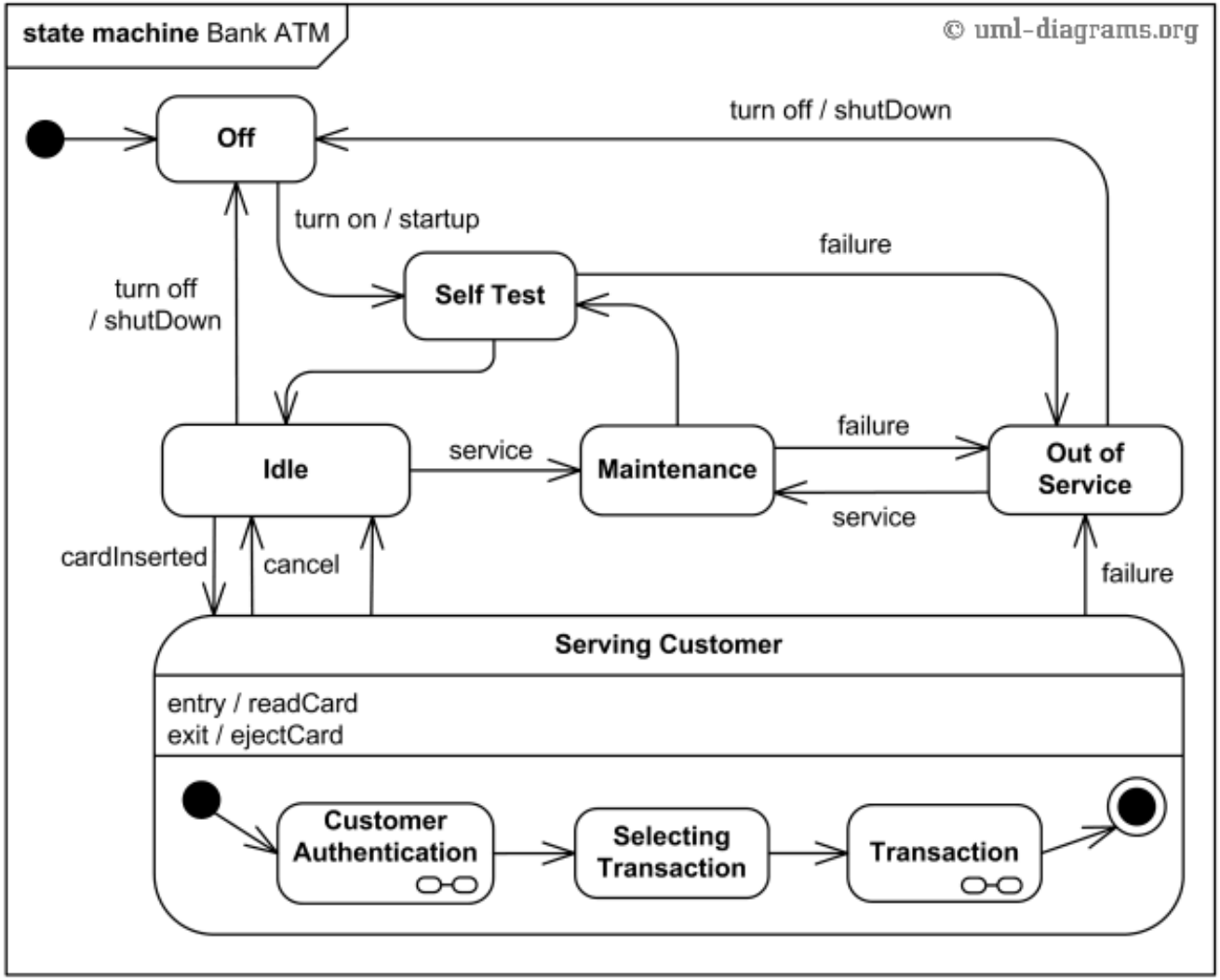

# Agenda



# State Machine Diagram



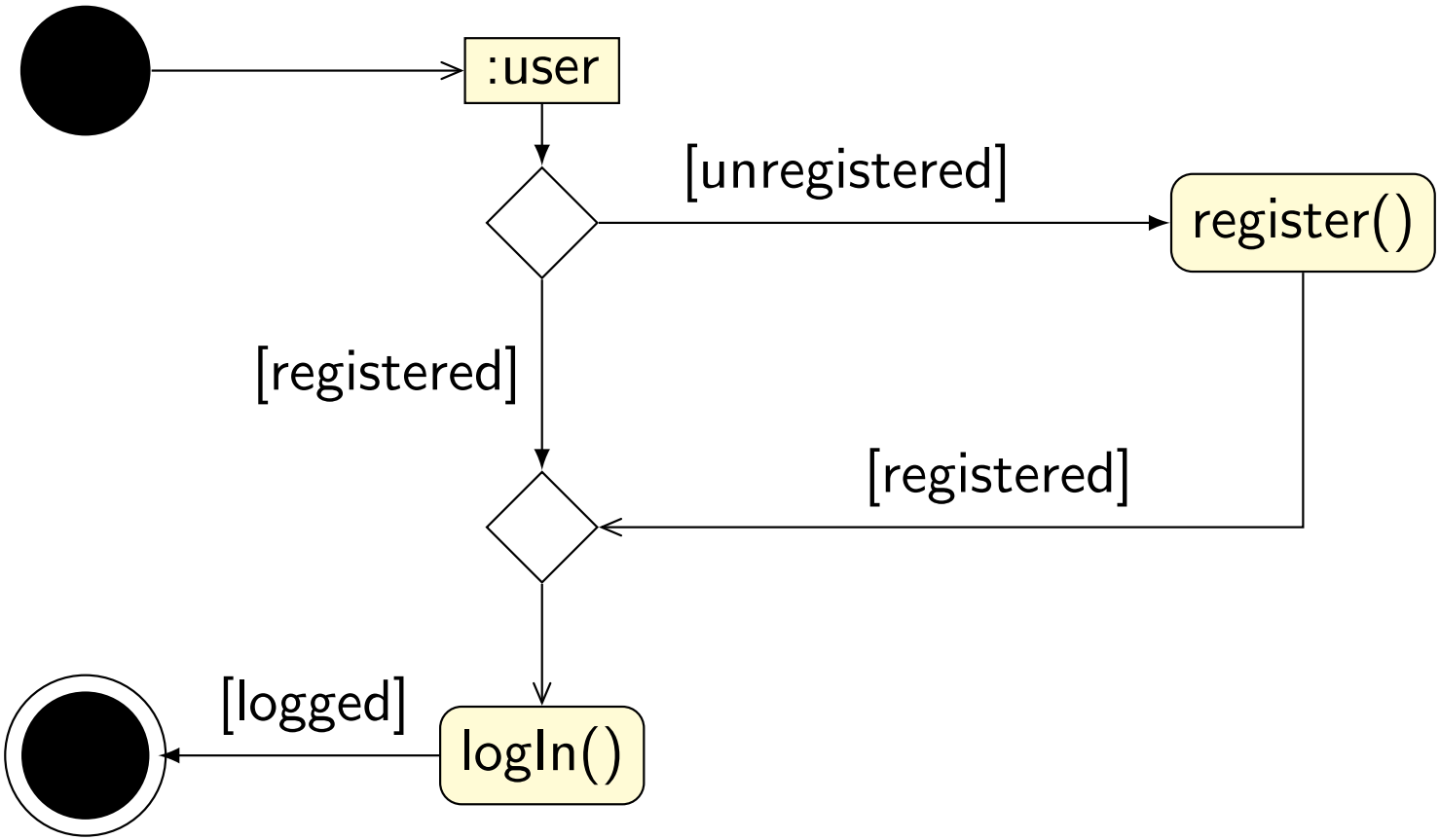
# State Machine Diagram




# Agenda



# Activity Diagram



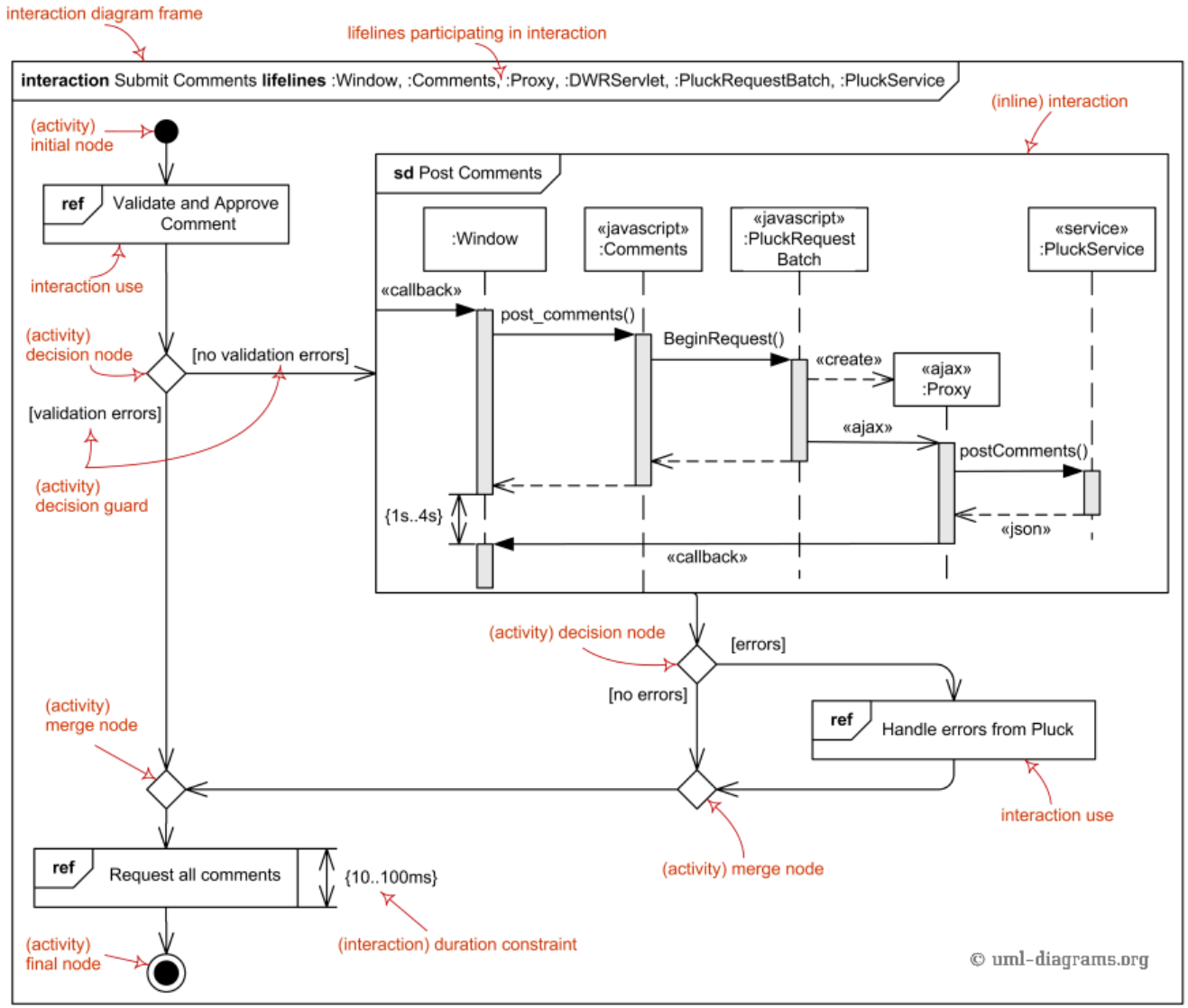
# Agenda



# Interaction Overview Diagram

- Interaction overview diagrams
  - combine the activity and sequence diagrams
  - an activity diagram whose nodes may contain sequence diagrams

# Interaction Overview Diagram

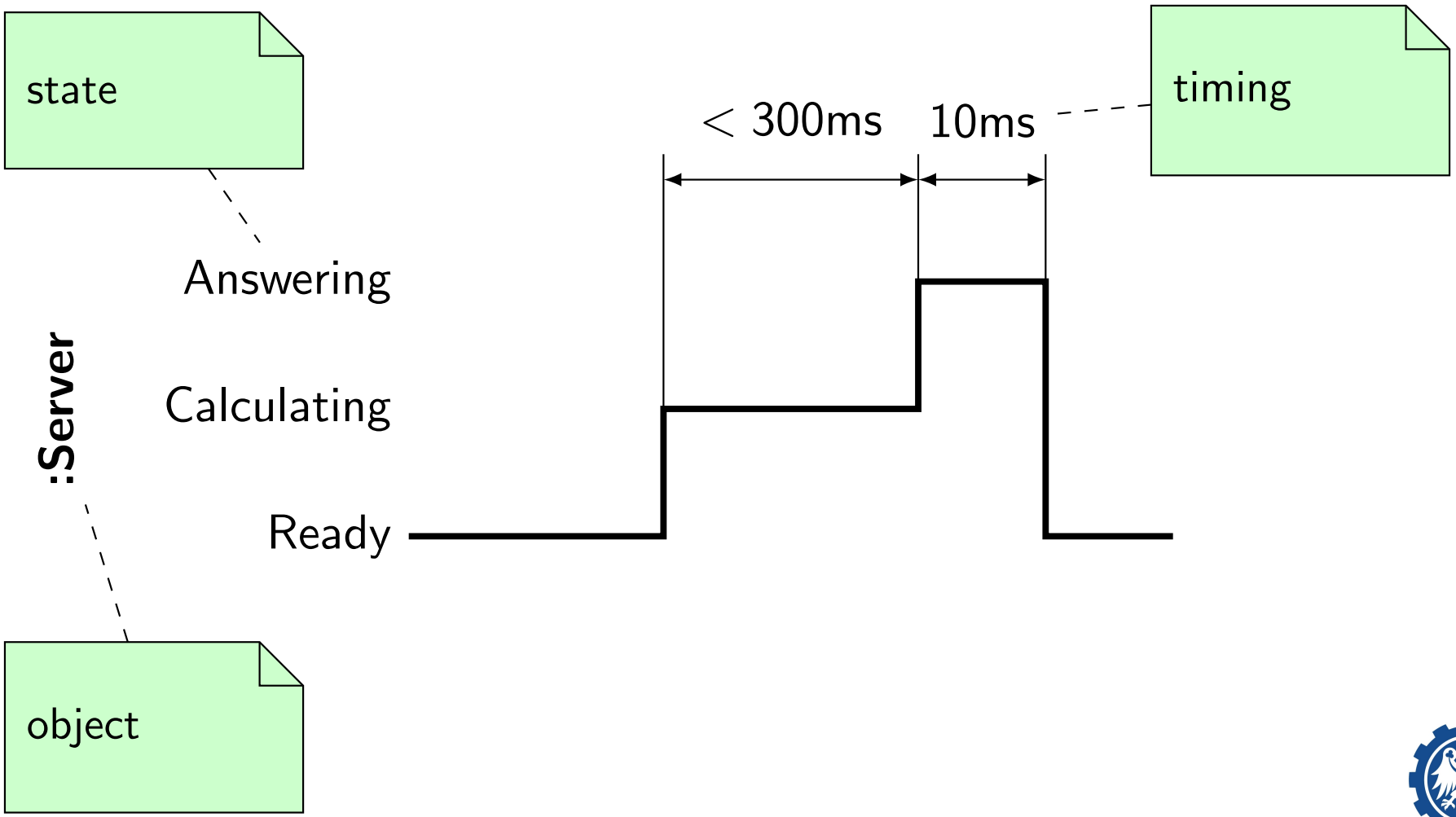



# Agenda

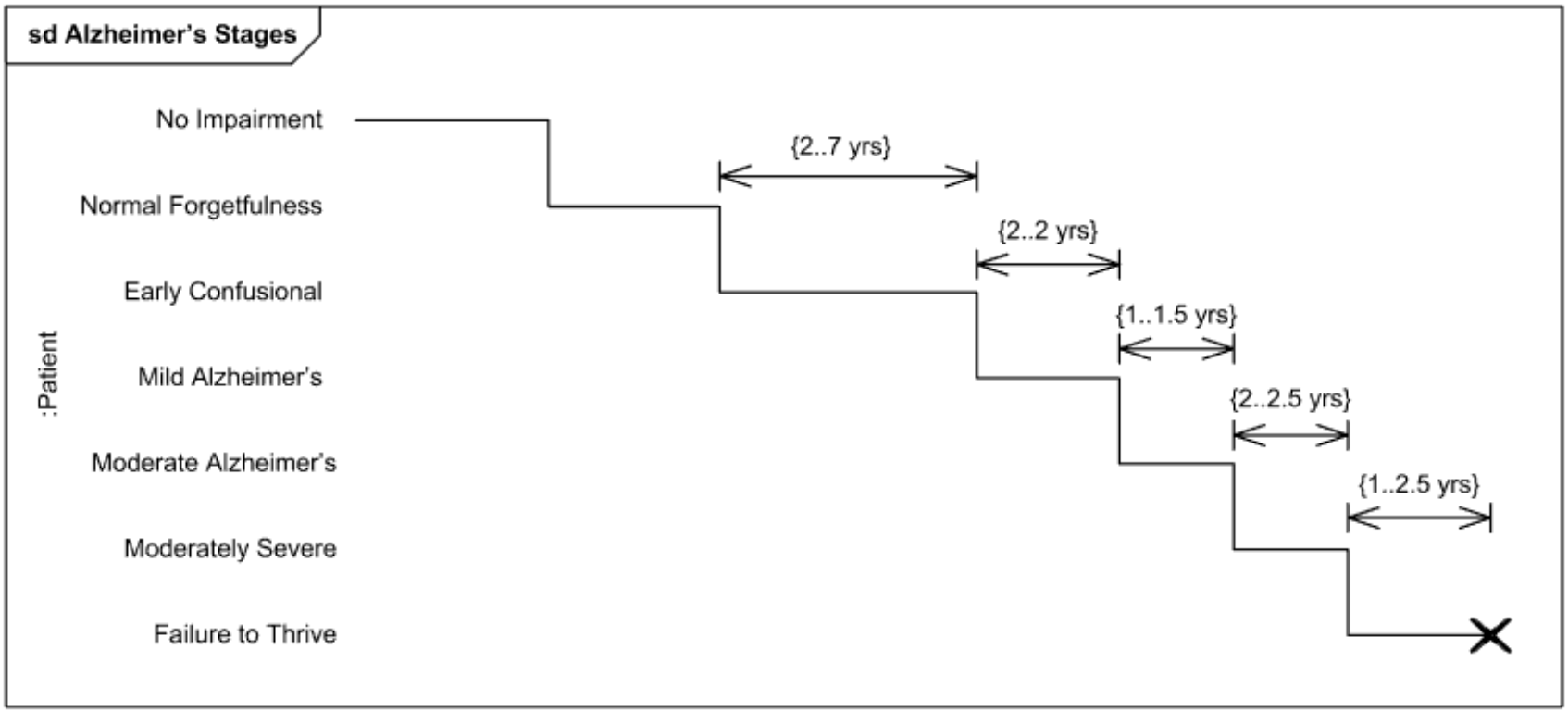


# Timing Diagram

- primary objective: timing of changes of object's states



# Timing Diagram




# Agenda



# Acknowledgements

The content of this presentation is based on:

1. Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling Language User Guide, 2nd Edition, 2005
2. J. Szedel, M. Kolano – Zaawansowane Zagadnienia Inżynierii Oprogramowania, Lectures 1 and 2, PowerPoint presentation for Computer Science students. Silesian University of Technology in Gliwice, Institute of Informatics. (in Polish)
3. `http://www.uml.org`
4. `http://www.uml-diagrams.org`
5. `http://visual-paradigm.com`

For more examples, see
`https://circle.visual-paradigm.com/diagram-examples`

# Software Engineering

## UML—Unified Modeling Language

Silesian University of Technology