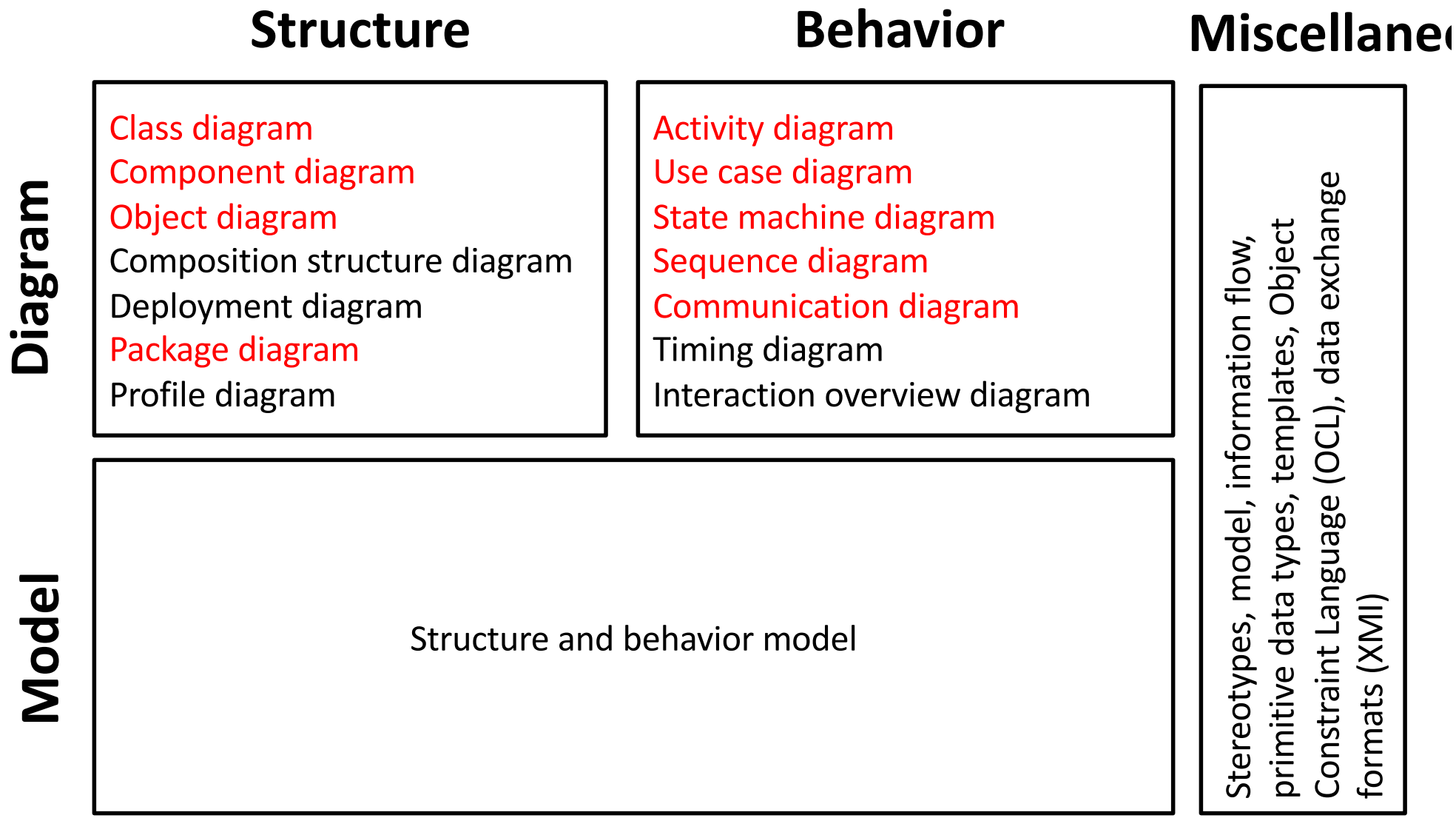
Systems Engineering and Project Management (Unified Modeling Language)

Prof. Dr. Franz Wotawa

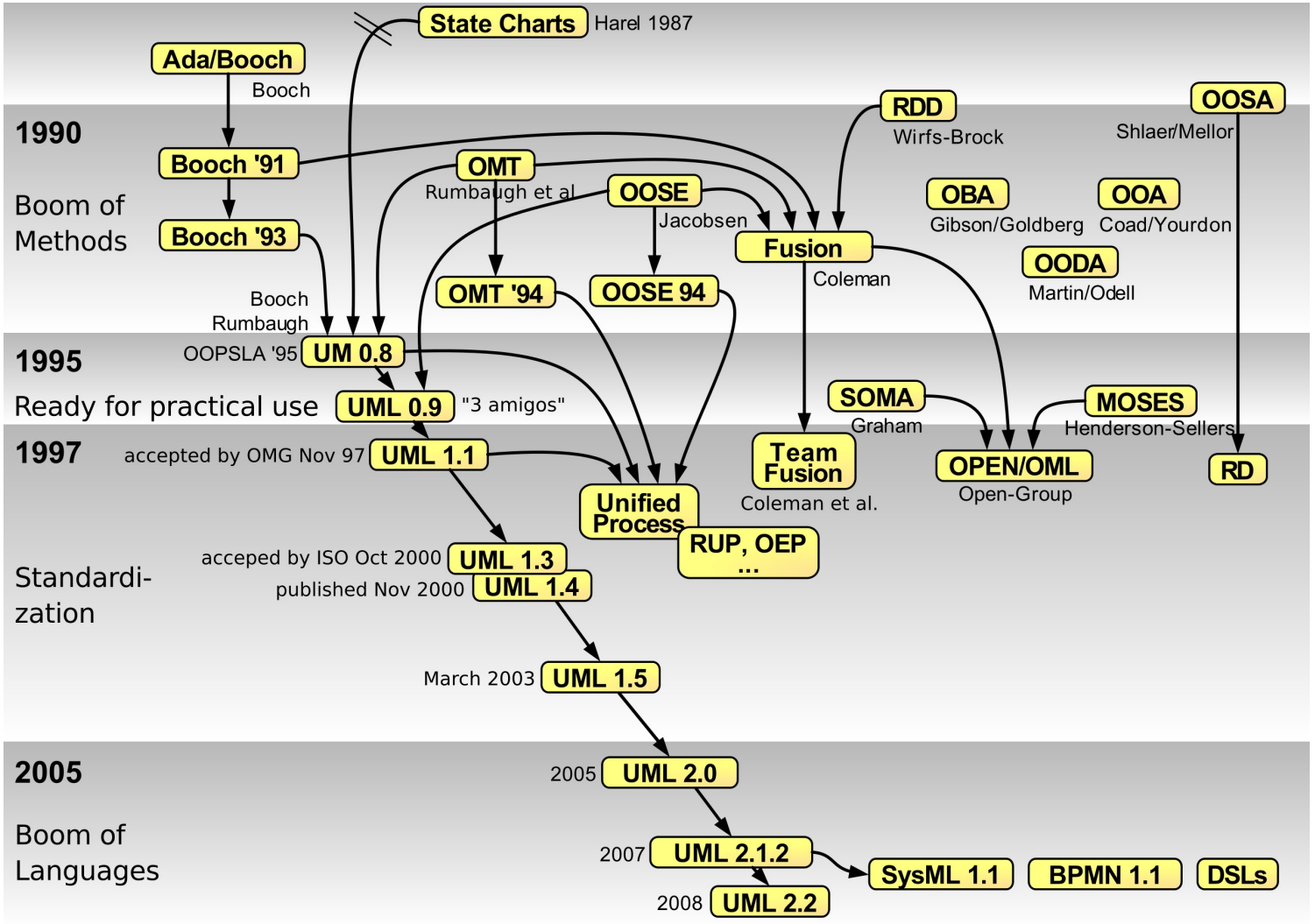
Institute of Software Engineering and Artificial Intelligence

wotawa@tugraz.at

Structure of UML



UML History



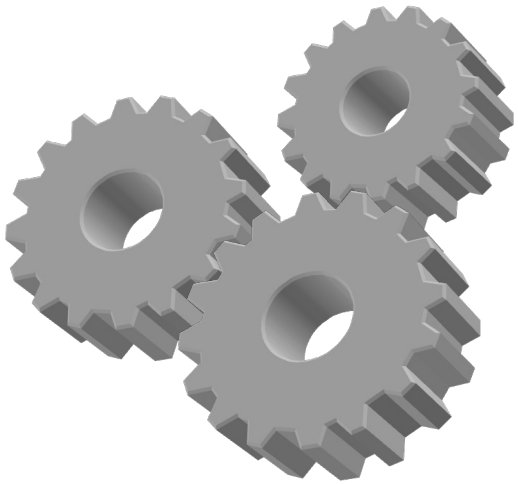
By Guido Zockoll, Axel Scheithauer & Karland90 - Translation from File:OO-historie-2.svg, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=109300755

Purpose and intent of UML

- Representation of systems and their components
- Different perspectives on the system through different diagrams
- Reduction of complexity through decomposition into subsystems and step-by-step refinements
- Provision of a basis for discussion with customers (use cases) and developers
- Defined semantics of the diagrams

Notes

- It is not necessary to describe a system using all possible diagram types.
- Diagrams should be checked for their usefulness!
- Almost always useful:
  - Use cases
  - Class diagrams



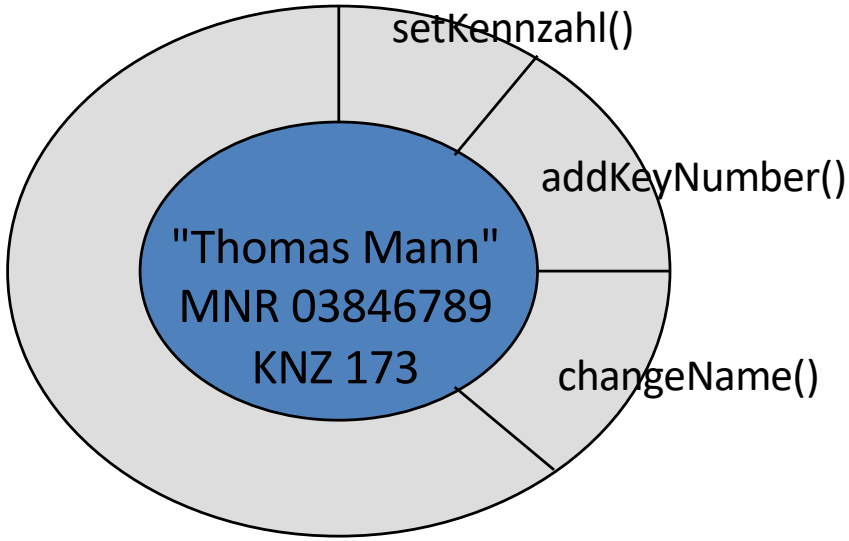
CLASS DIAGRAM
(CLASS DIAGRAM)

Basic Terms

- Object
  "*A discrete entity with a well-defined boundary that encapsulates state and behavior; an instance of a class*"
  [UML Reference Manual (Rumbaugh)]
- Attributes
  - Combination of data and functions (operations (analysis), methods (design))
  - Data hiding (through functions)
  - Each object is unique
  - Attributes store the object data

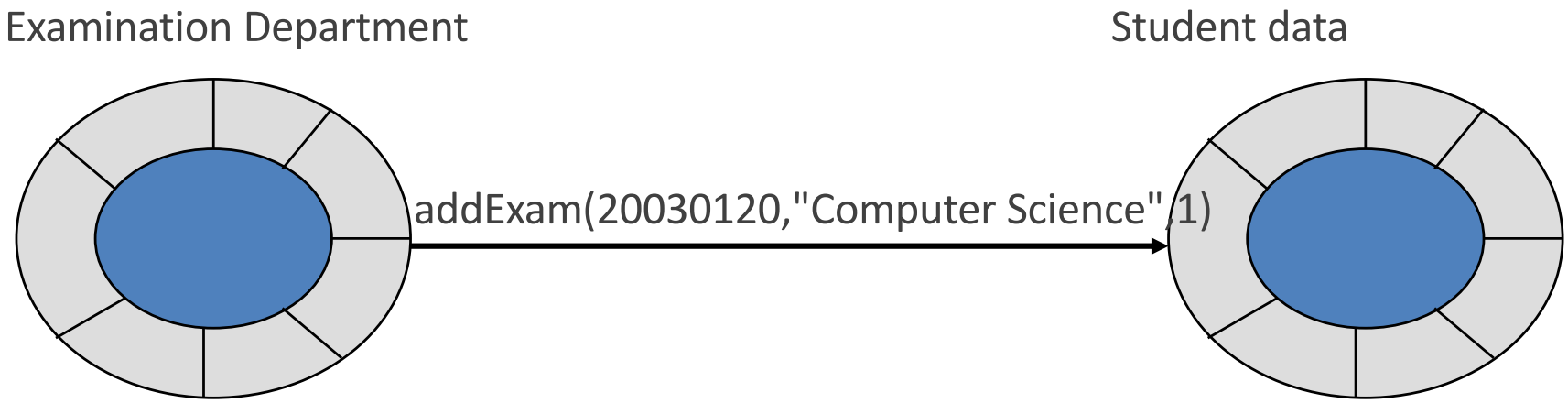
Encapsulation

- The object state is defined by the object attributes and can only be changed by object functions.
- The object behavior is defined by the object functions.

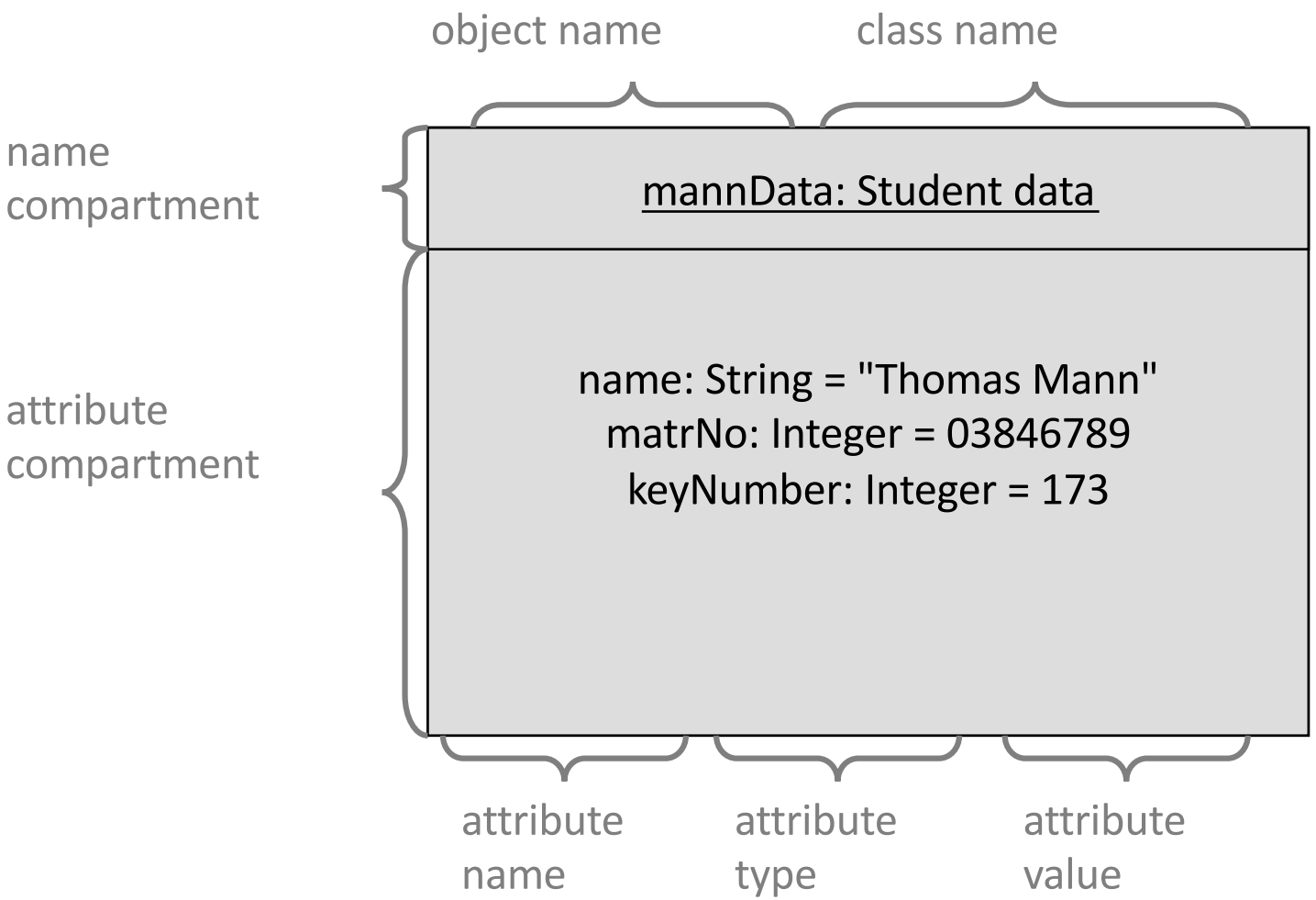


System behavior

- System behavior is generated by objects sending messages. Messages are sent via links between objects. [Collaboration]



UML Object representation

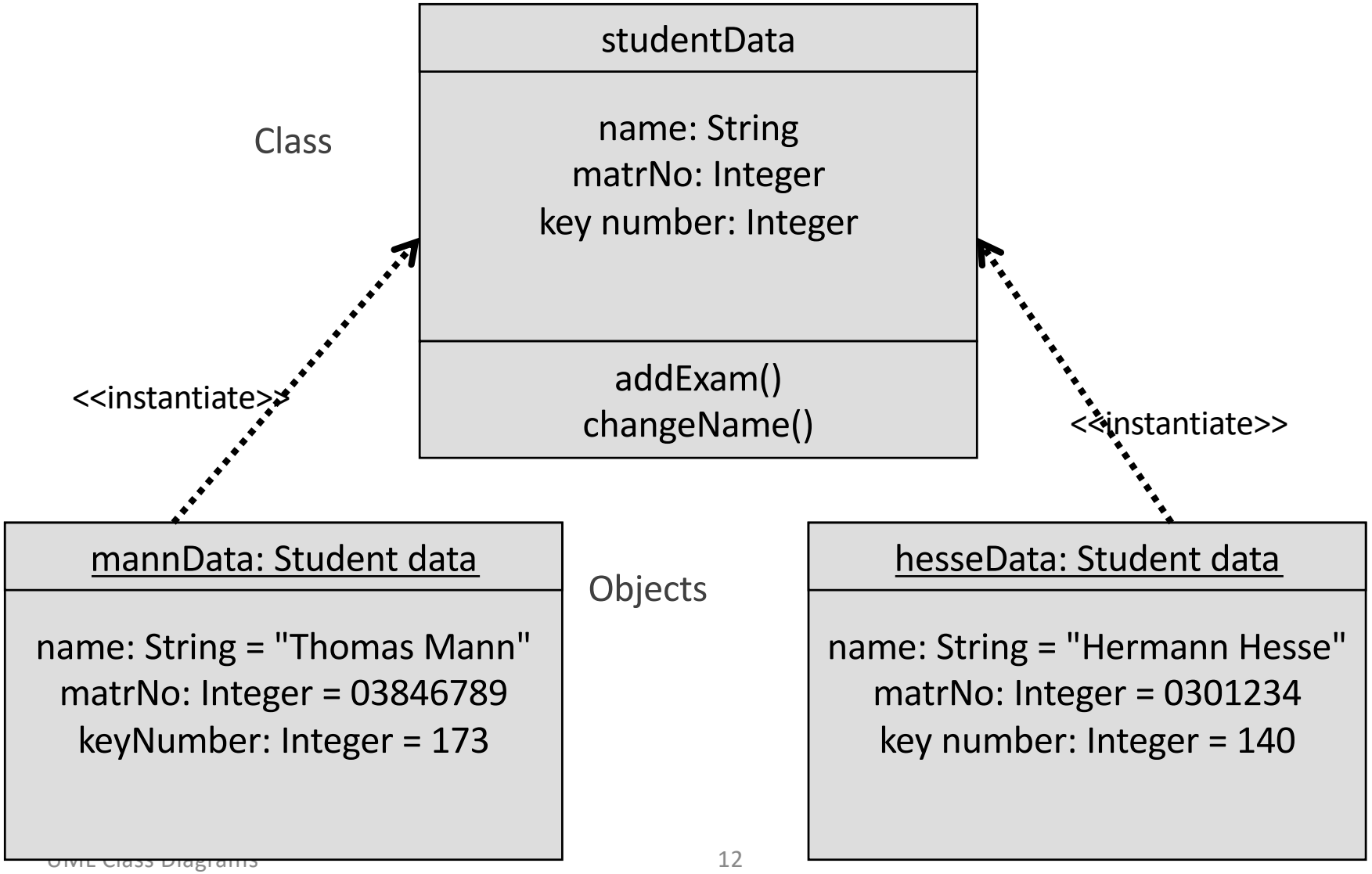


Classes

1. A class describes the behavior of a set of objects
   "The descriptor for a set of objects that share the same attributes, operations, methods, relationships, and behavior"
2. Each object is an instance of a class

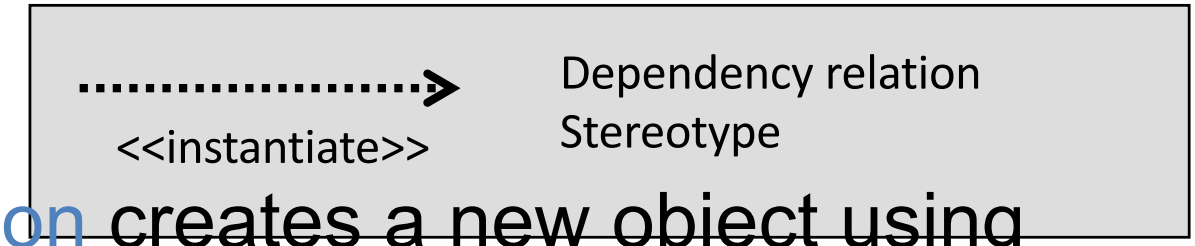
- The correct classification scheme is key to successful OO analysis

Relationship Classes - Objects

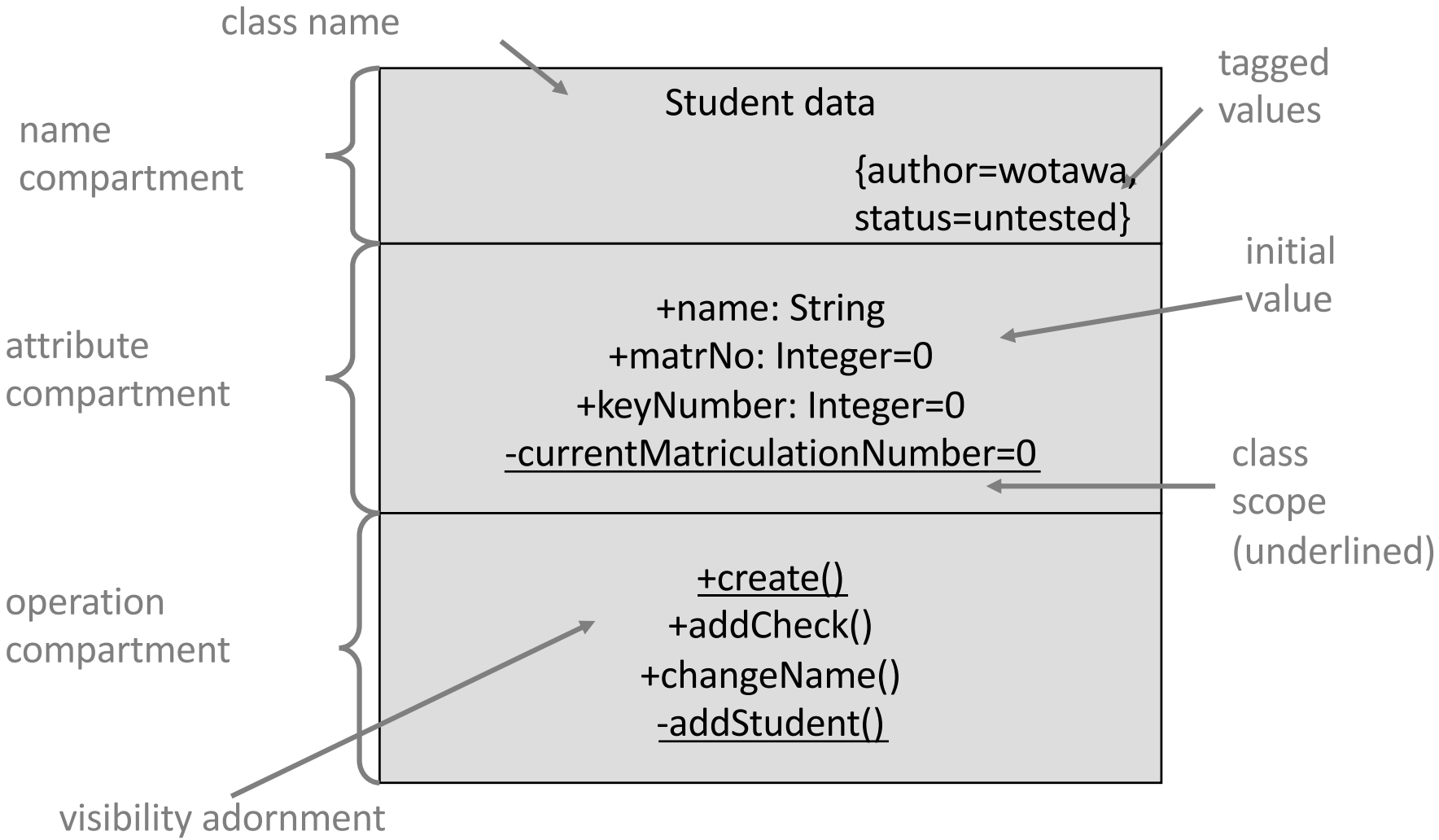


Relationships

- Connection between things
  "A connection between modeling elements."
- Dependency Relation
  "A change in the supplier has an effect on the client."
- Object instantiation creates a new object using its class as a template



UML class representation



Name Compartment

- There are no naming conventions in UML!
- Good names follow two principles:
  - Class names begin with a capital letter and then have upper and lower case letters. Capital letters are only used at the beginning of words.
  - Do not use abbreviations!
- **Examples**: StudentRecord, Account, etc., but **not** StdRec or Acnt!

Attributes Compartment

- Following form
  visibility name multiplicity : type = initialValue
- Visibility

+	Public	All
-	Private	Only the class
#	Protected	The class and its subclasses
~	Package	All in the same package

- Multiplicity

```
address[3]: String
emailAddress[0..1]: String
```

Operation Compartment

- Function is defined by
  - name
  - Parameter list
  - return type

  (signature).

  visibility name (parameterName: parameterType,...) : returnType
- Example:

```
+add(argument: Integer): Integer
```

Scope

- We distinguish between two types:
  - Instance Scope
    Attributes and operations that are available for specific objects.
  - Class Scope
    Attributes and operations that are valid for a class.
- **Example**: `add` is available for every object (defined by Integer). A method that instantiates a new object (`create()`) is defined for the class.

Object Construction and Destruction

- Constructors are special methods (on the class side) that create a new object of a class.
  `BankAccount()`
  `create()`
- Destructors are special methods that are called when an object is no longer needed.
  `~BankAccount()`
  `finalize()`

Relationships

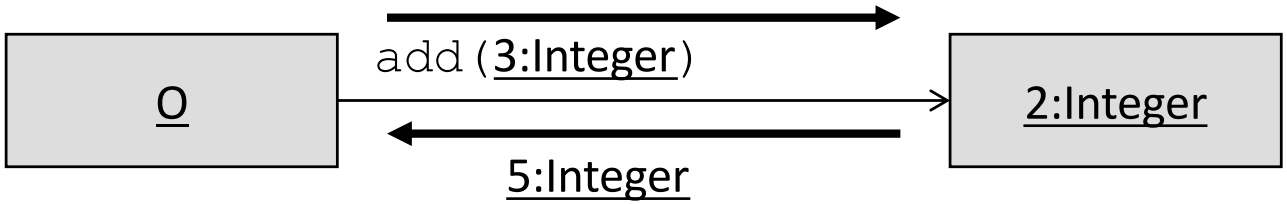
- Relationships between things in UML
- Connect things with each other
- Examples:
  - Between actor and use case (association)
  - Between use case and use case (generalization, `<<include>>`, `<<extend>>`)
  - Between actor and actor (generalization)

Links and Object Diagrams

- Objects communicate (by sending messages) via links. Links are connections between objects.
- Object diagrams describe the state of objects and their relationships at a specific point in time.

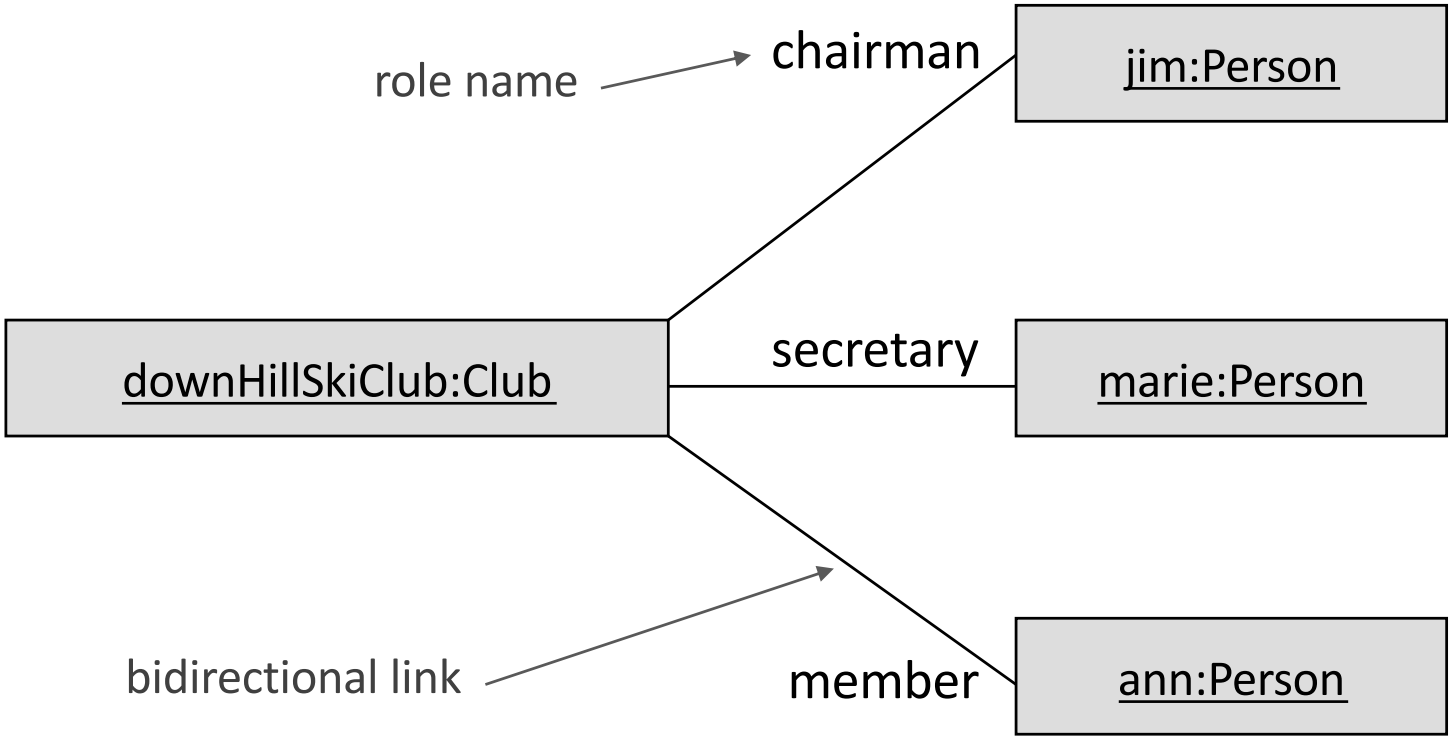
Example - Links

In a method m of object O, there is 2 + 3



1. The method `add` with parameter 3:Integer is sent to the object 2:Integer
2. The return value is sent back to O.

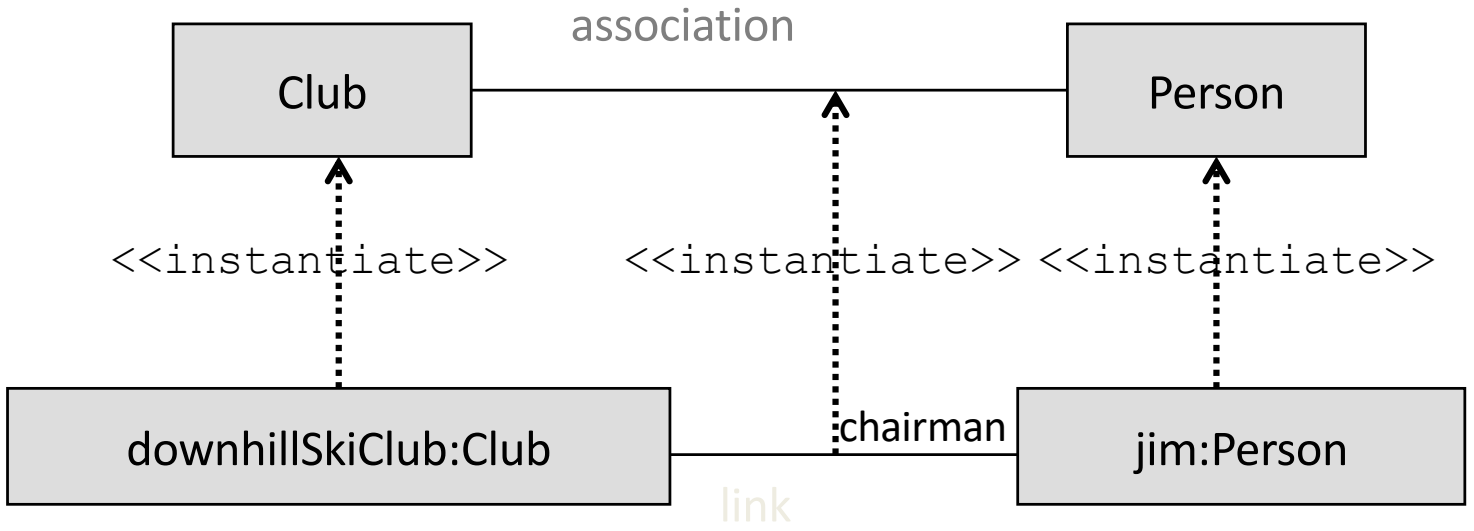
Example - Object Diagram



Snapshot of an executing OO system

Associations

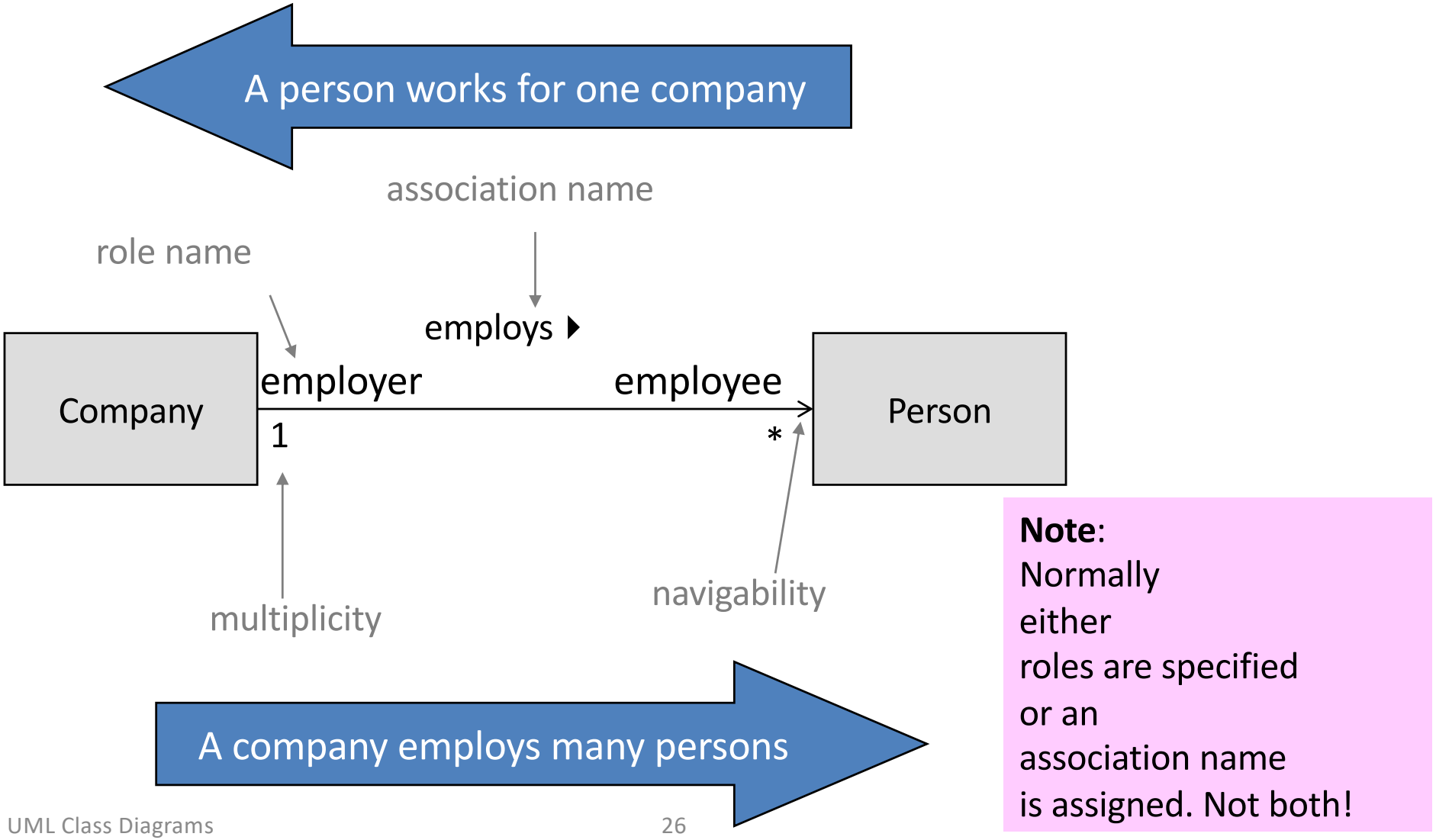
- Associations are connections between classes.
- Links depend on associations.



UML Syntax of Associations

- An association is represented by a line between classes and can also contain the following information:
  - Name
  - Role names
  - Multiplicity
  - Navigability

Example



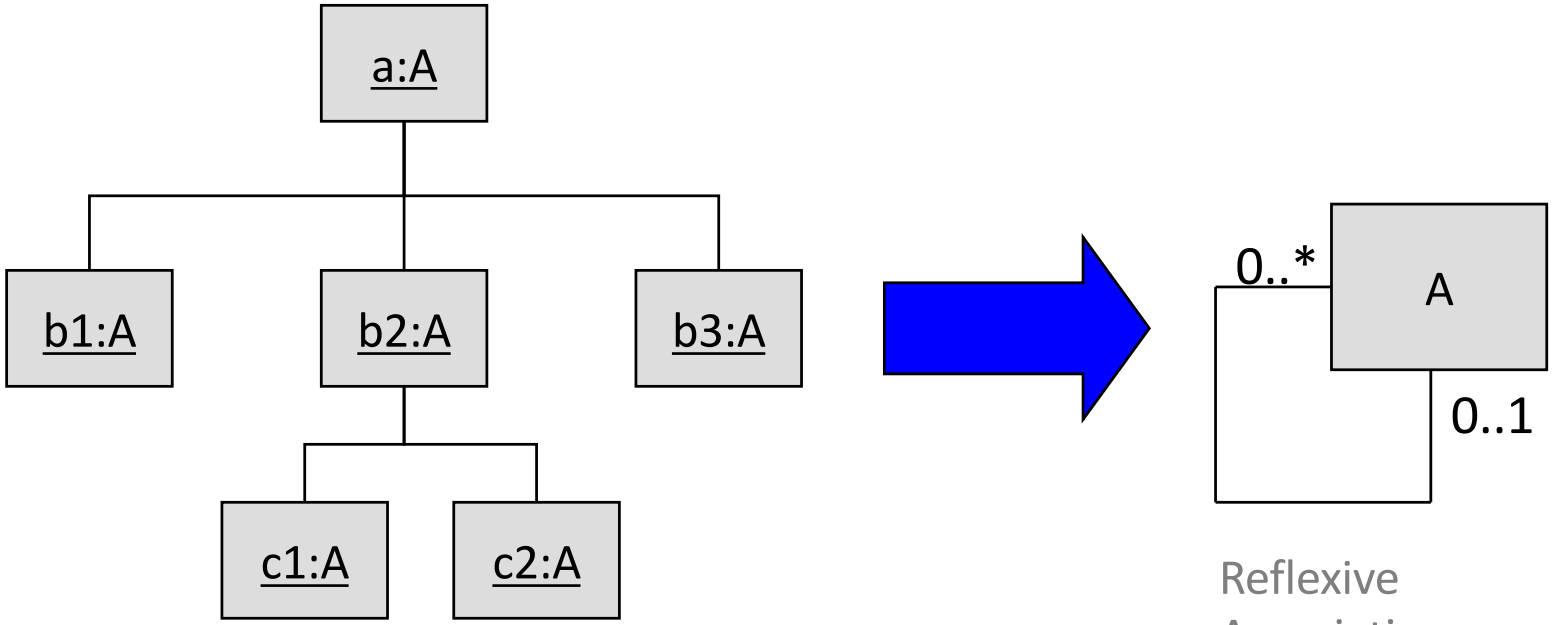
Note:
Normally
either
roles are specified
or an
association name
is assigned. Not both!

Multiplicity

- If not specified, it has not yet been decided
- UML diagrams must be read exactly as they were drawn!

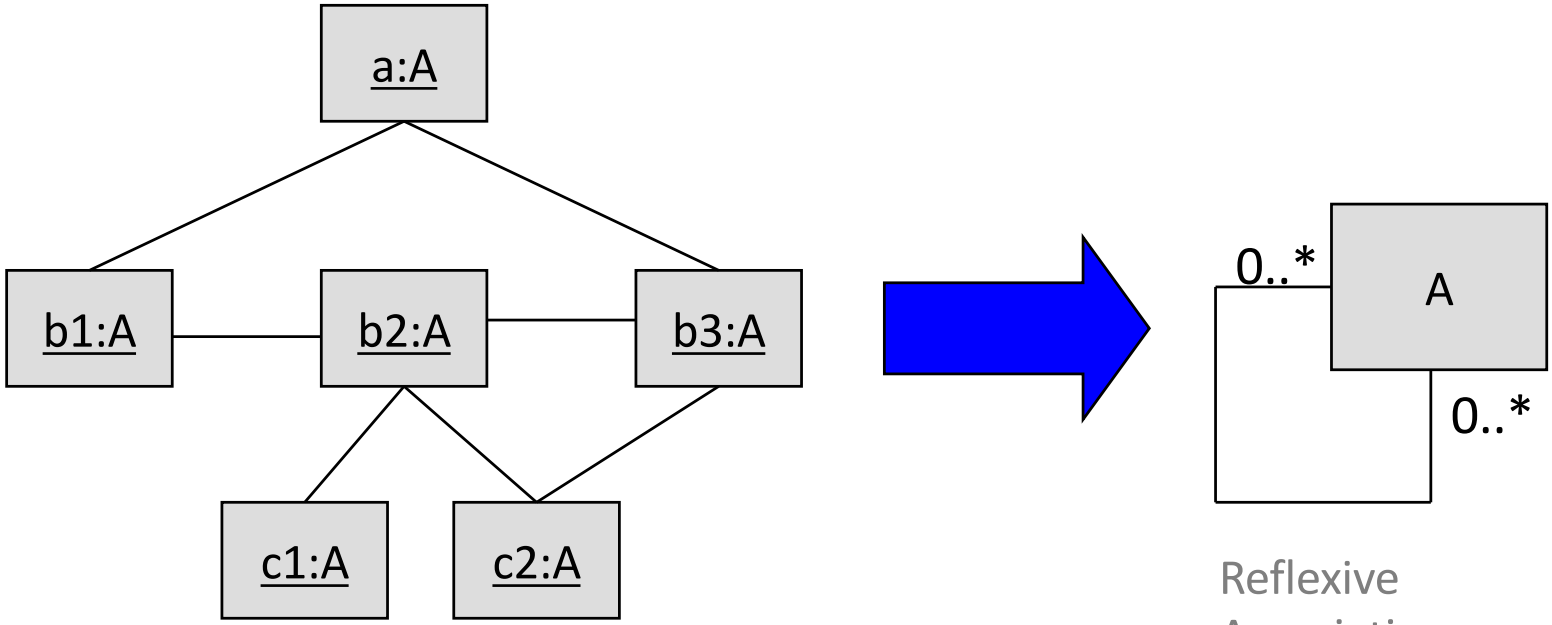
0	0 or 1	1	1 or more
1	Exactly 1	1..6	1 to 6
0..*	0 or more	1..3,7..10,15, 17,*	1-3, 7-10, 15, 17 or more
*	0 or more		

Representation of hierarchies



Reflexive Association

Representation of Networks



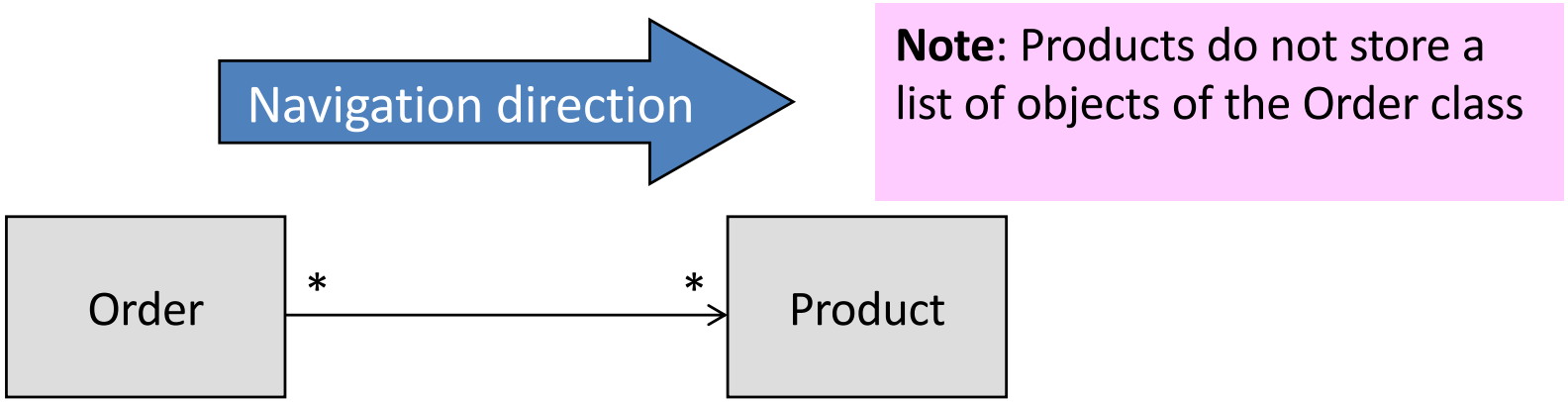
Reflexive Association

Note on navigability

- Navigability shows how to get from one class to another.
- *"Messages can only be sent in the direction of the arrow."*
  (Source -> Target)
- **Example**: `Order` objects can send data to `product` objects (but not vice versa).

Purpose of Navigability

- Minimizing class coupling

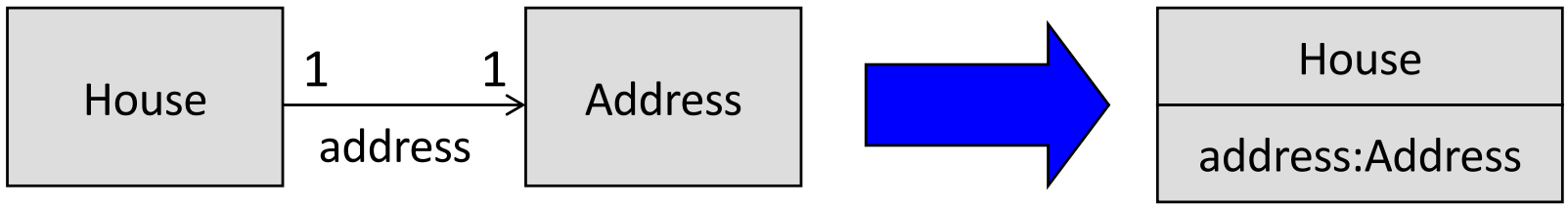


Note: Products do not store a list of objects of the Order class

In good OO designs, the number of class couplings should be minimized.

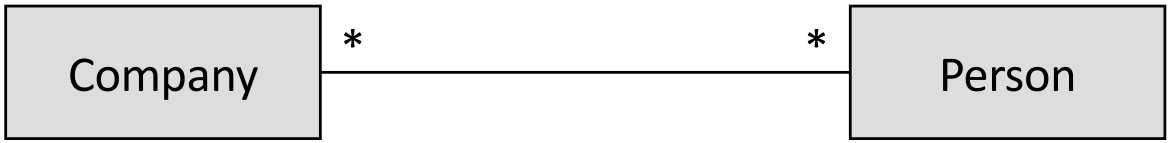
Associations and Attributes

- If there is an association between a source class and a target class, this means that the source class can store an object of the target class.
- This is done automatically during code generation!



Association Class

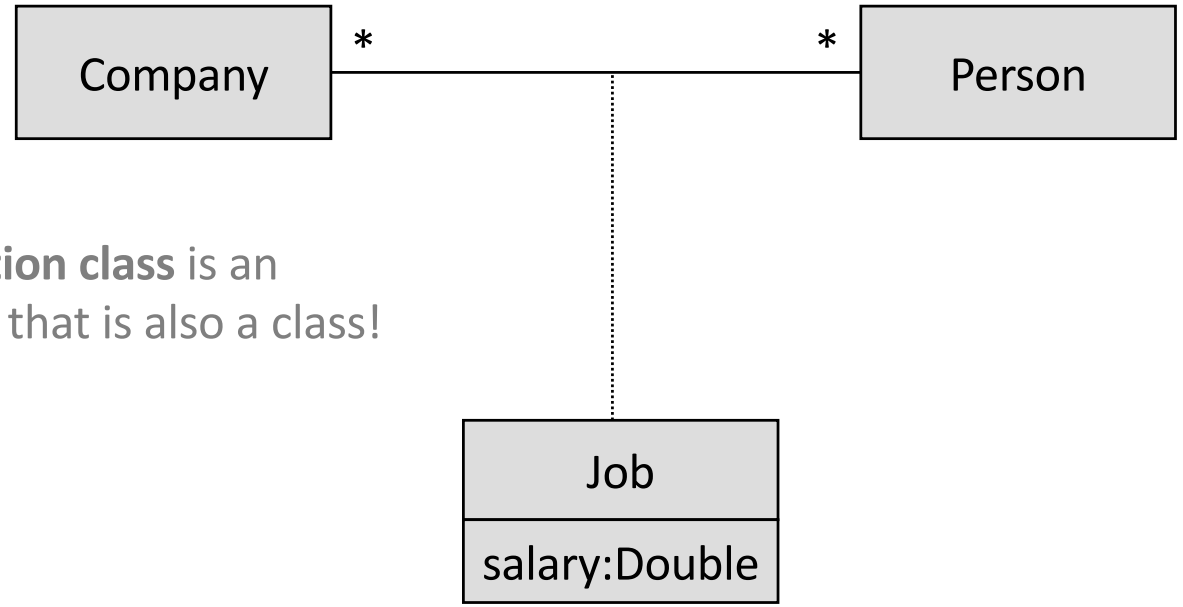
- Many-to-many relationships can cause problems that can be solved by creating a separate class.



Problem: Where is the salary stored?

Salary is a property of the association!

Association Classes 2

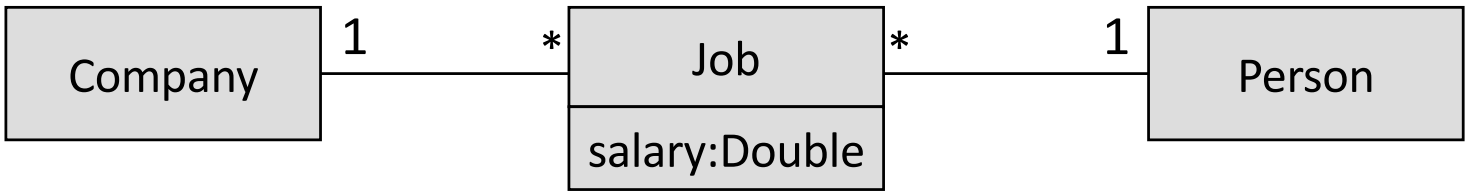


An **association class** is an association that is also a class!

Semantics: There is only one link between two objects at any given time!

Reified Association

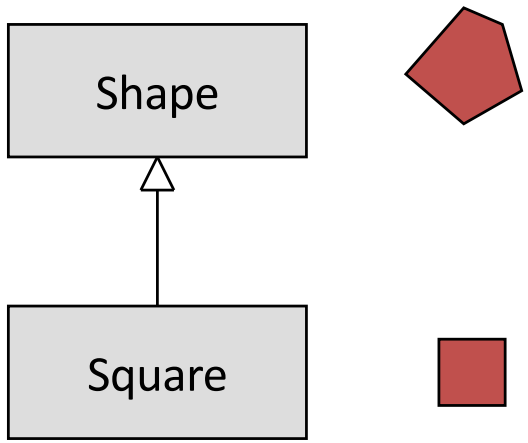
Unlike the association class, the actual introduction of a class allows any number of links between two objects.



A person can now be employed multiple times in one company!

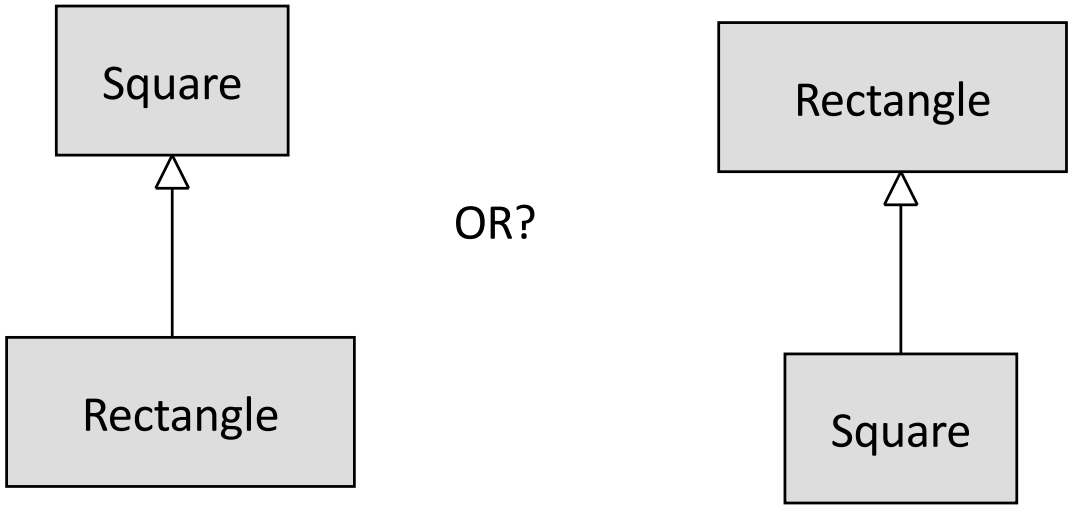
Inheritance

- Class hierarchies based on generalization.
- Generalization is a relationship between a more specific and a more general thing.



Class hierarchies

- Generalization of specialized things
- Specialization of more general things



Inheritance

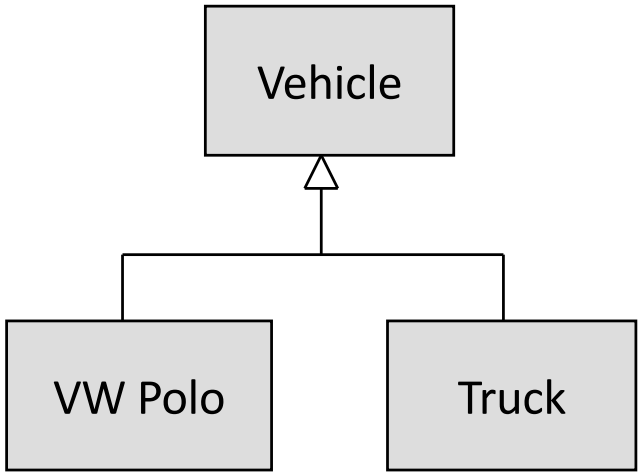
- Subclasses inherit all features from the superclass.
  - Attributes
  - Operations
  - Relationships
  - Constraints
- The features can be overwritten in the subclass.
  - Introduce operation with the same signature

Abstract Classes

- Provide an "interface"
- Do not have a concrete implementation for at least one operation.
- The implemented functionality is in the subclasses.
- Cannot be instantiated.

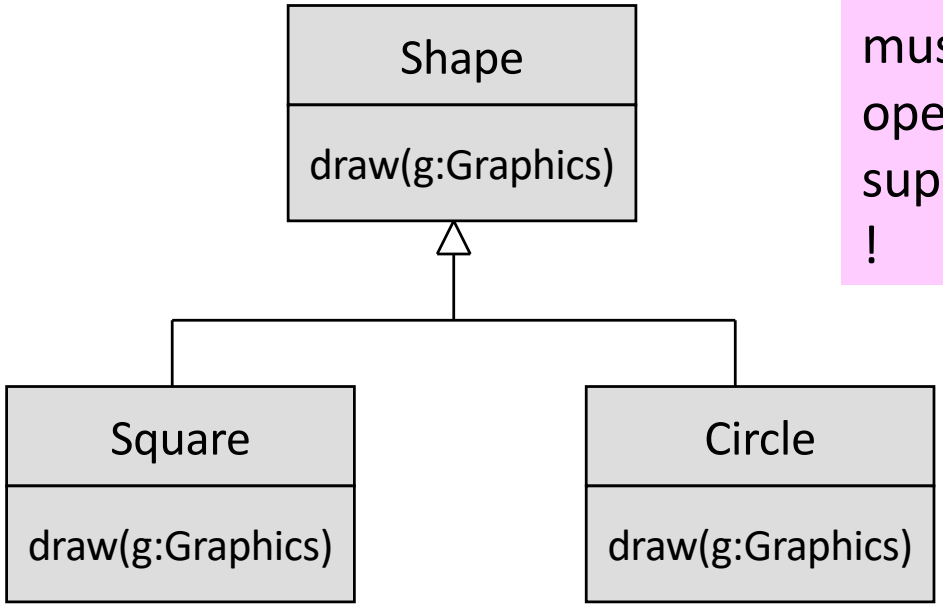
Notes

- Things on the same (graphical) level of abstraction should also represent the same abstraction.
- Example: (bad)



Polymorphism

- A polymorphic operation has many implementations.



Concrete subclasses must implement all abstract operations of the superclass
!

FINDING CLASSES

Analysis and Design

- Analysis Goal: *Understanding the problem domain*

- Design Goal: *Describing the software architecture*

Finding classes

- Analysis Goal: To find an appropriate description of problem domain
  - The classes
  - Their relationships
  - Thinking about implementation is not allowed!
- Think about the user, about the interfaces to the external world. Everything else is implementation.
- Close connection to use cases

Finding Classes

1. Brainstorming
2. Noun method
3. CRC cards

- A creative process...

Classes

- Common types of classes:
  - Physical Objects (Calendar, Display, Page)
  - Conceptual Entities (Appointment, Reminder)

Finding classes: Brainstorming

- Think of terms relevant to the problem
  - e.g., Terms often used by experts
- Good classes
  - Encapsulate data
  - *Are* something, not just *do* something
  - Different in behavior from other classes
- Bad classes
  - Outside the system boundary
  - “Manager” classes

Finding Classes: Noun Method

- Underline all nouns in requirement document
- Finds important terms
- Criticism: grammar determines your design!
  - *For every appointment entry, a database record is made*
  - *For every appointment that is entered, a database entry is made.*
- The noun method is good, but be critical!

Refining Classes

- After classes have been found, refine by
  - Constructing class diagrams
  - Discussion of use cases
    - *Can use cases be explained using the class diagram?*

Class Diagrams

- A useful tool for visualizing an analysis or design.
- Goal: capture important objects and their relations
- Don't be too formal about UML in the early stages! Make notes, shift things around, create different possibilities, different views.
- For initial deliberations, a blackboard is much easier than Rose
  - For later refinement and for presentation, tools are helpful

Example

Use Case: Insert Appointment

ID: UC1.1

Actors: owner

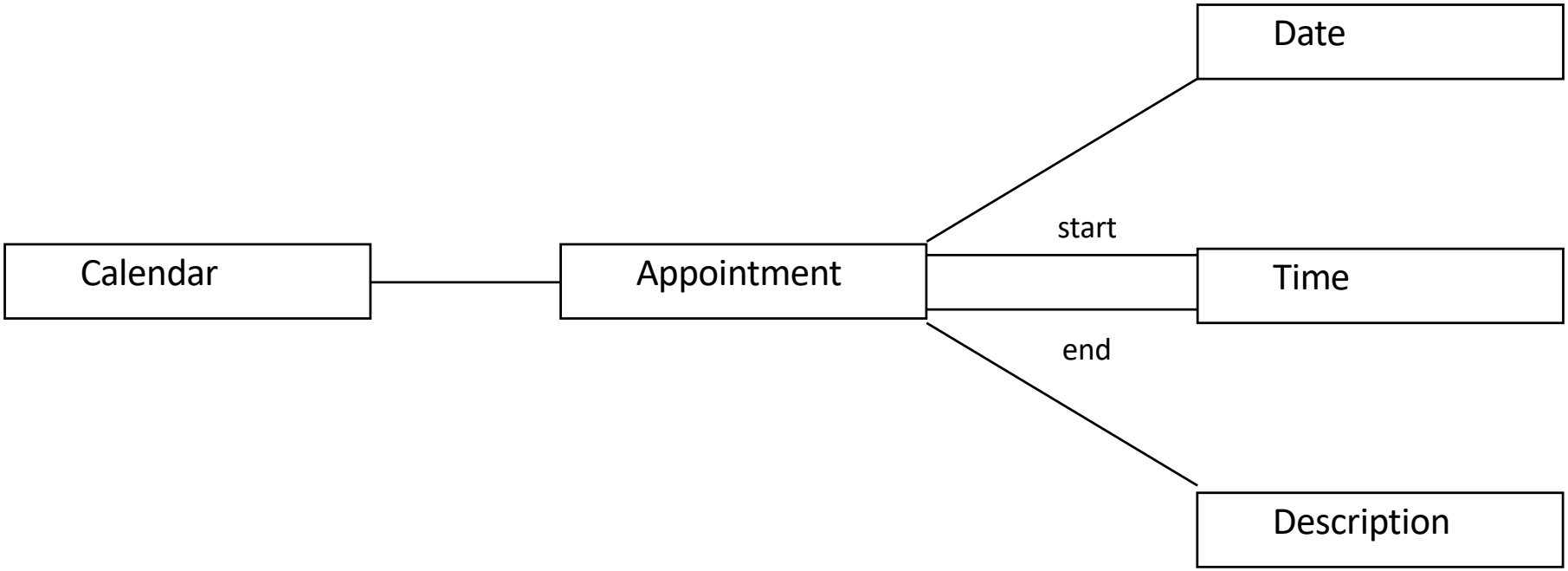
Preconditions:

Flow of events:

1. The user specifies the date, start time, end time, and description of the appointment.
2. The calendar warns about conflicting appointments
3. The appointment is entered in the calendar

Postconditions: The appointment is in the calendar

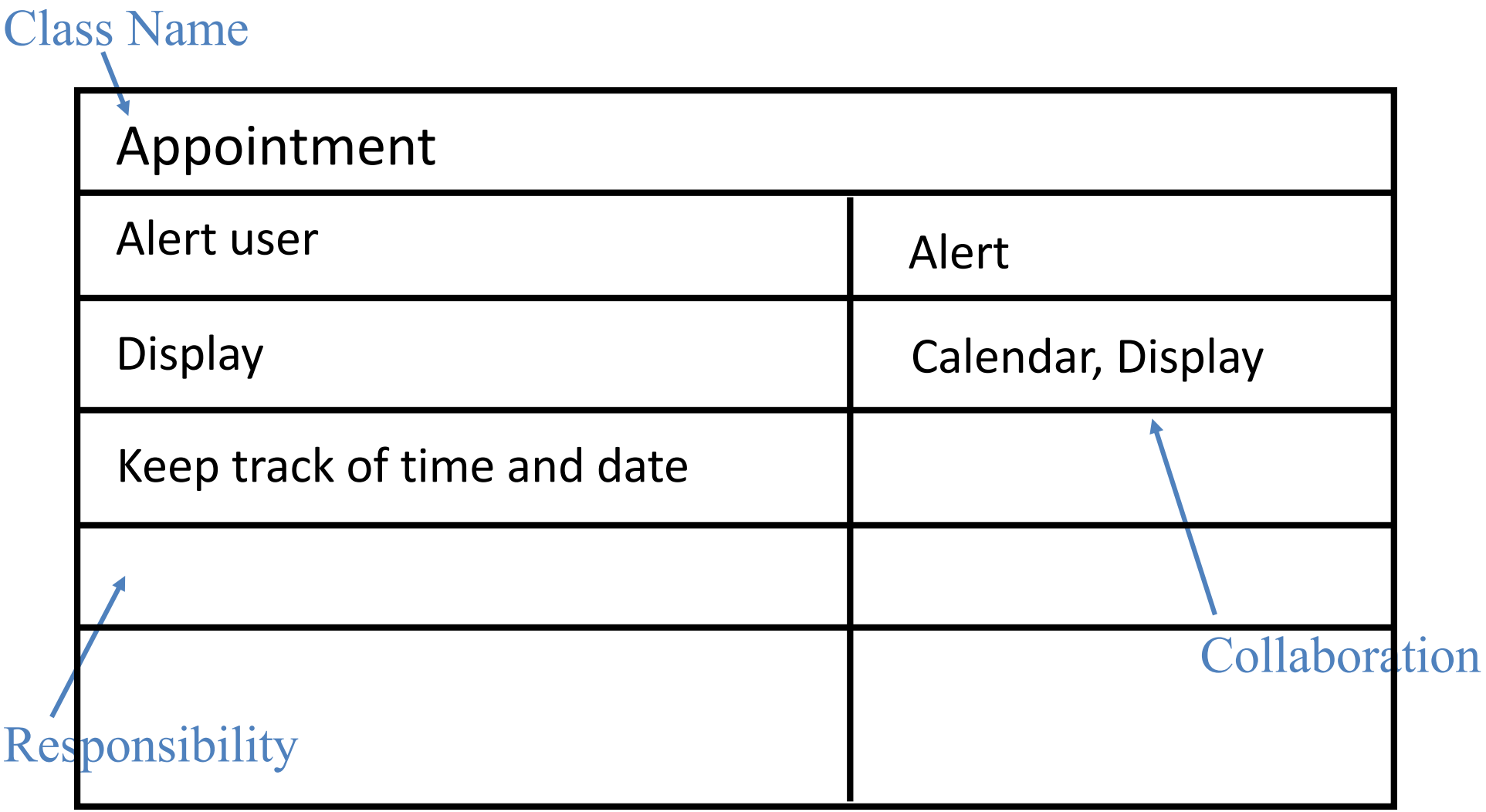
Class Diagram



Class-Responsibility-Collaboration Cards

- Class-Responsibility-Collaboration on a 10 by 15 cm index card.
- Find out if classes work well together.
- Responsibilities are more important than interfaces; classes collaborate with other classes to fulfill them.
- Works well for analysis *and* design

CRC Cards


Class Name
Appointment
Alert user
Alert
Display
Calendar, Display
Keep track of time and date
Collaboration
Responsibility

CRC cards, example

Appointment	
Alert user	Alert
Display	Calendar, Display
Keep track of time & date	

To-do	
Display if not yet done	Calendar, Display
Keep track of date	To-do

Calendar	
Display appointments	Appointment
Display To-dos	To-do
Warn for collisions	Appointment

CRC Cards

- Visual:
  - Close together means collaborate closely
  - Below means part of (or is supervised by)
- Tactile
  - You can point to a card and take it in your hand when discussing its responsibility.

CRC and Scenarios

- Scenarios and use cases

- Go through a scenario
  - Explain how the system reacts to the scenario
  - Keep track of responsibilities on CRC cards

Note on Inheritance

- Inheritance may come naturally from the problem domain
  - *We have two types of appointments...*
- Similarities may become apparent during analysis
  - Appointments and to-do items have similar responsibilities
- Especially important in design

Design With CRC Cards

- Conflict:
  - Need to clearly separate design and analysis
  - Need to have a smooth transition

CRC and Scenarios

- Scenarios and use cases

- Go through a scenario
  - Explain how the system reacts to the scenario
  - progressively make your explanations more detailed.
  - Keep track of responsibilities on CRC cards
  - When interaction understood, document with interaction diagrams

CRC: Merging and Splitting

- If a class has too many responsibilities (more than 3), either
  - restate the responsibilities at a higher level, or
  - split the class into two
- If a class has too few responsibilities, merge it into another class

CRC Cards for Analysis & Design

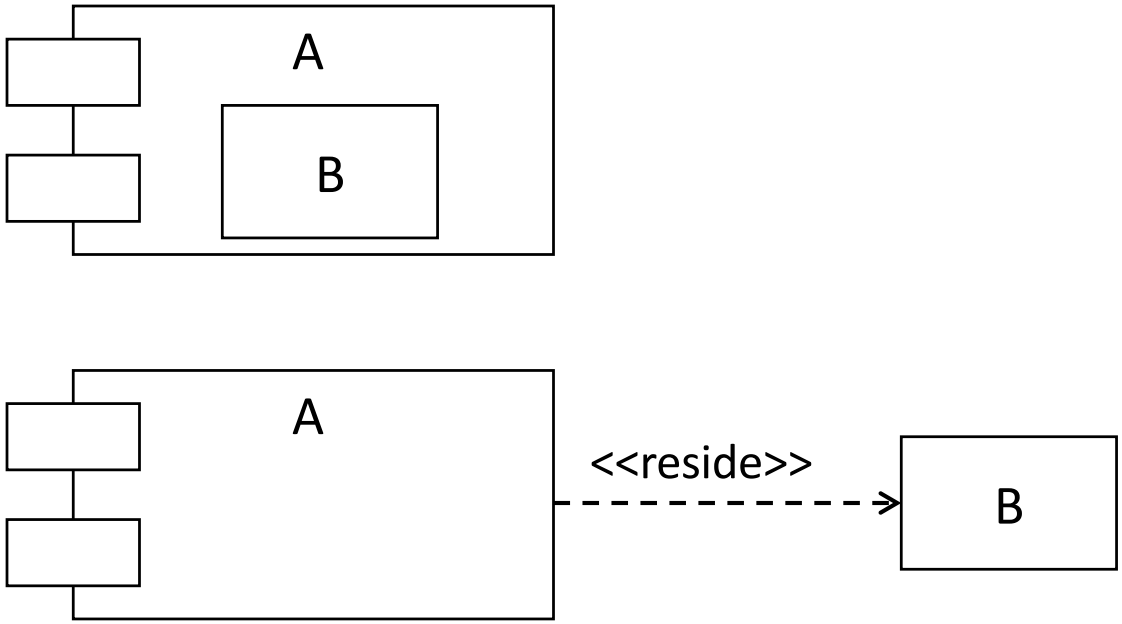
- Analysis:
  - Go through some scenarios (responsibilities are high level, lots of *magic*)
- Design
  - Go through scenarios in more detail
  - Explain how magic happens, until you can explain everything in sufficient detail (trivial to implement)

ADDITIONAL STRUCTURE DIAGRAMS

Component Diagrams

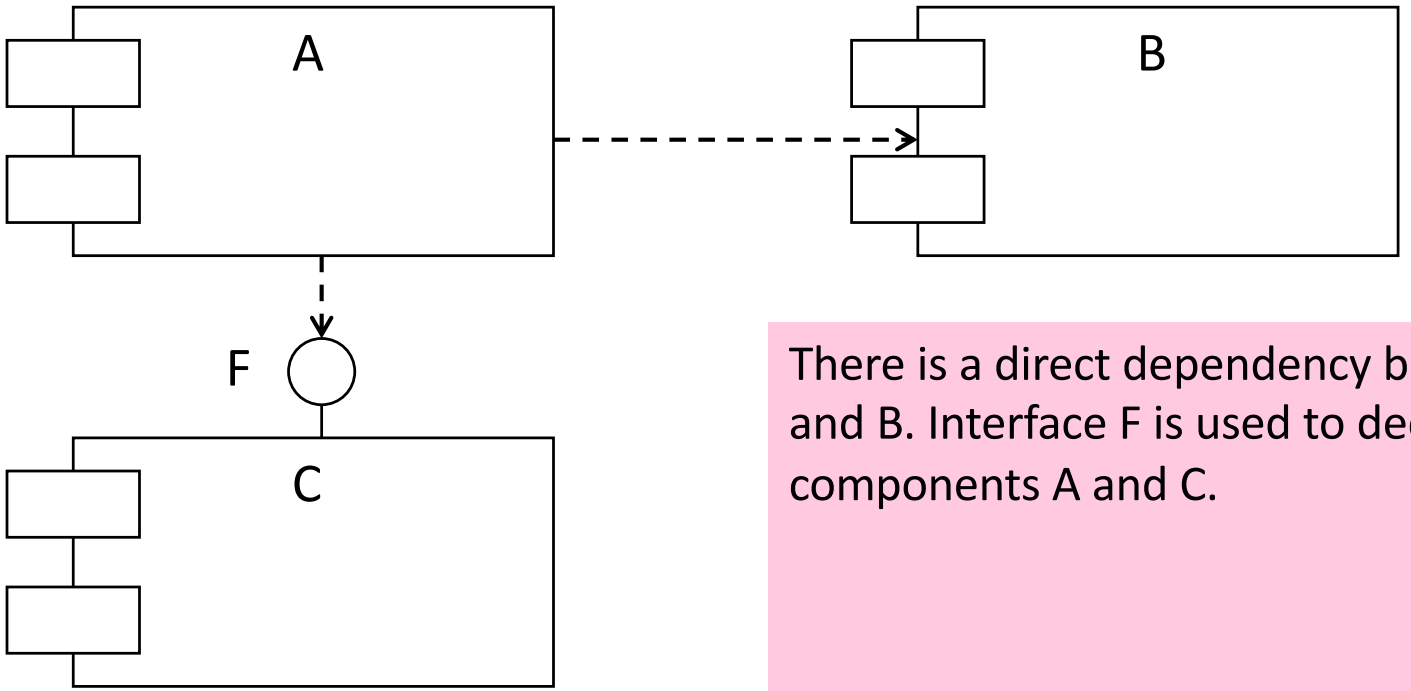
- "A component is a physical, replaceable part of a system that packages implementation, and conforms to and provides the realization of a set of interfaces."
- Unit of Reuse!
- Source File, ActiveX control, JavaBeans, Java servlets,...
- Contain many classes and interfaces

Syntax - Component Diagrams



"Class B is part of component A."

Another example



There is a direct dependency between A and B. Interface F is used to decouple components A and C.

Shortcut for: A class in C implements interface F.

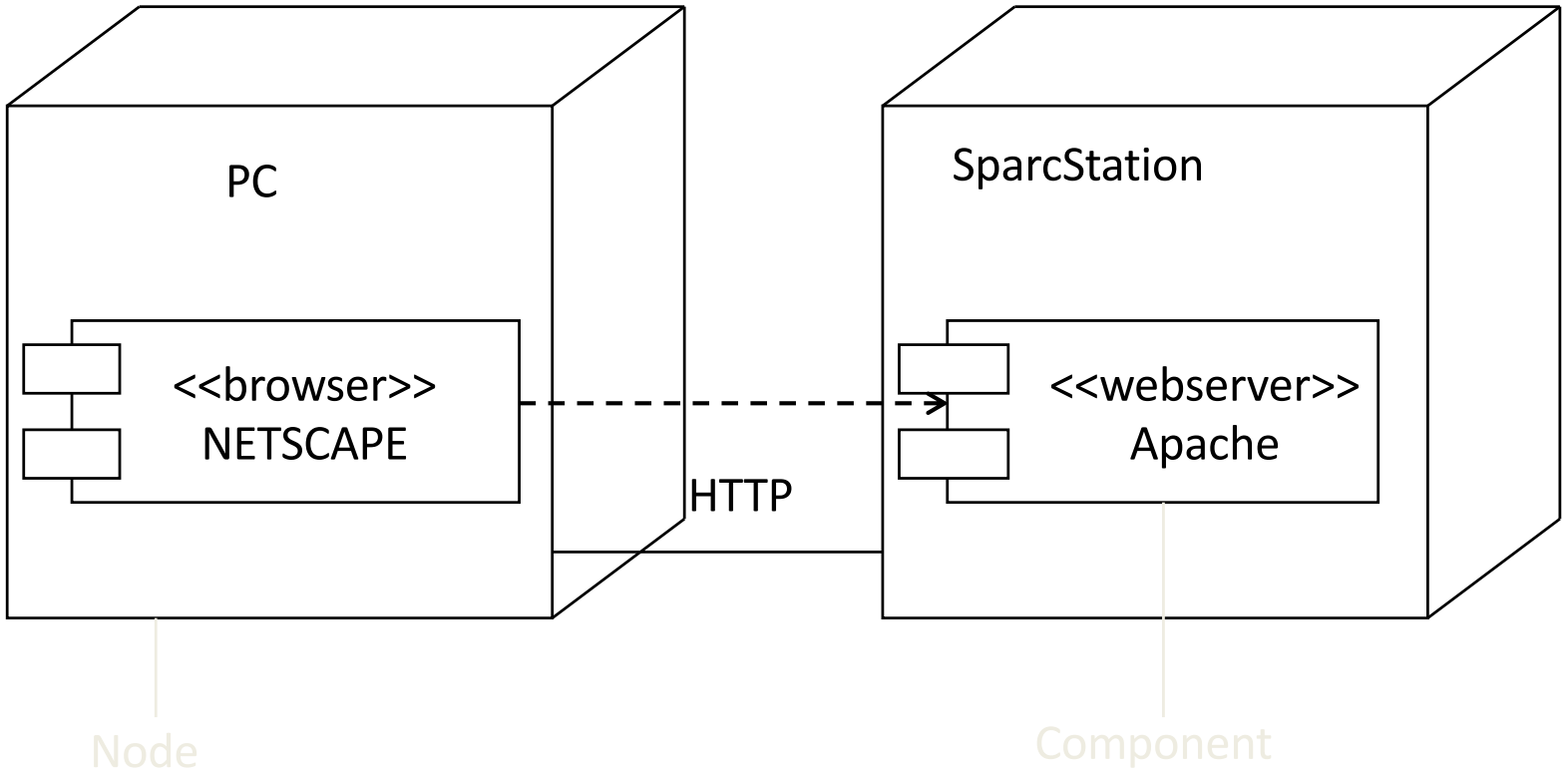
Deployment Diagrams

- Show the hardware on which the software is to run, as well as the distribution of the software on this hardware.
- 2 Types of Deployment Diagrams
  - Descriptor form: Nodes, their relationships, and components
  - Instance form: Node instances, their relationships, and component instances

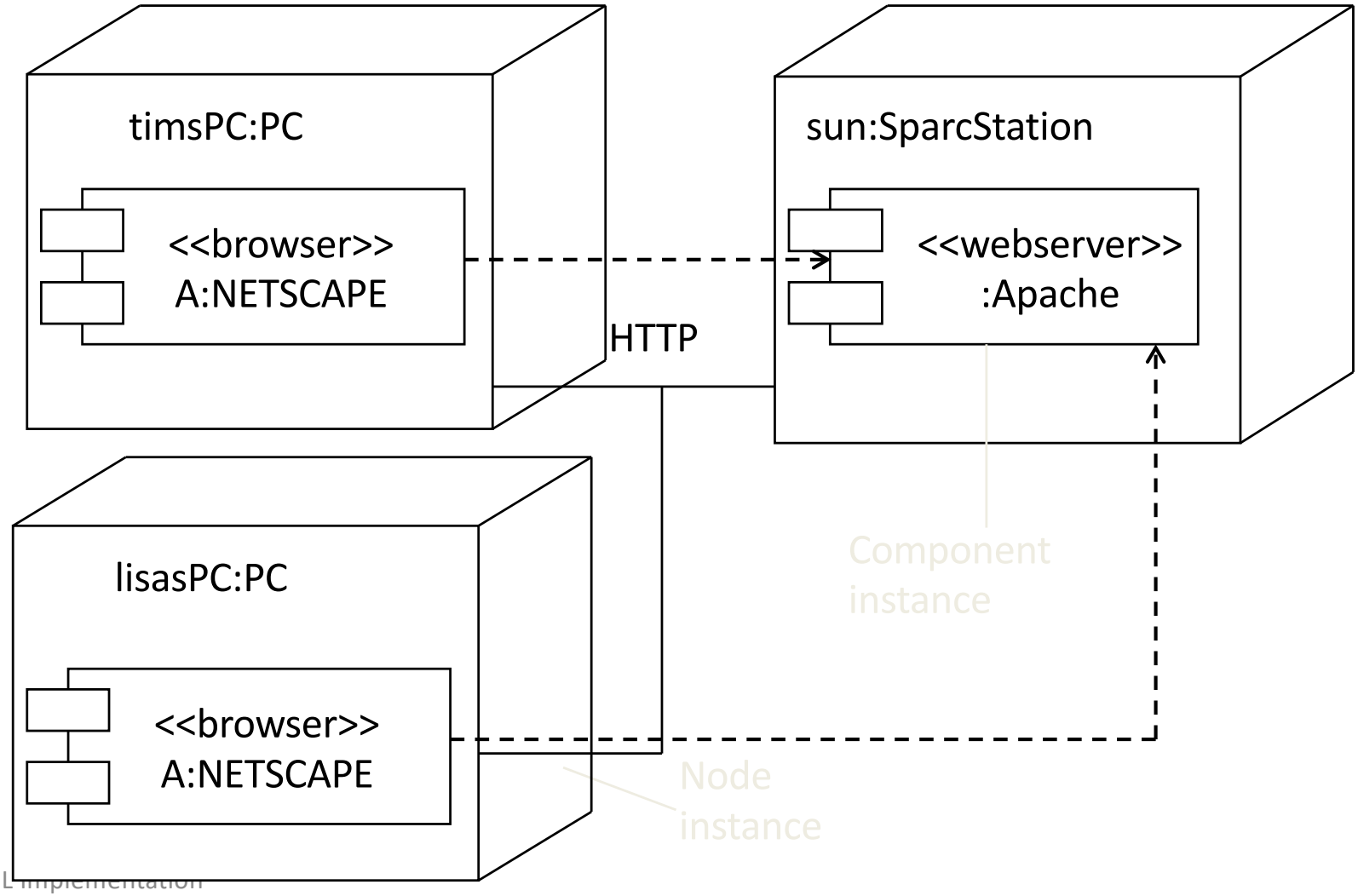
Use of Deployment Diagrams

- 2-step process
  - Design Workflow
    - Focus on nodes, node instances, and their relationships
    - Goal: Determine hardware architecture
  - Implementation Workflow:
    - Assign component instances to node instances
    - Assign components to nodes

Example - Descriptor Form

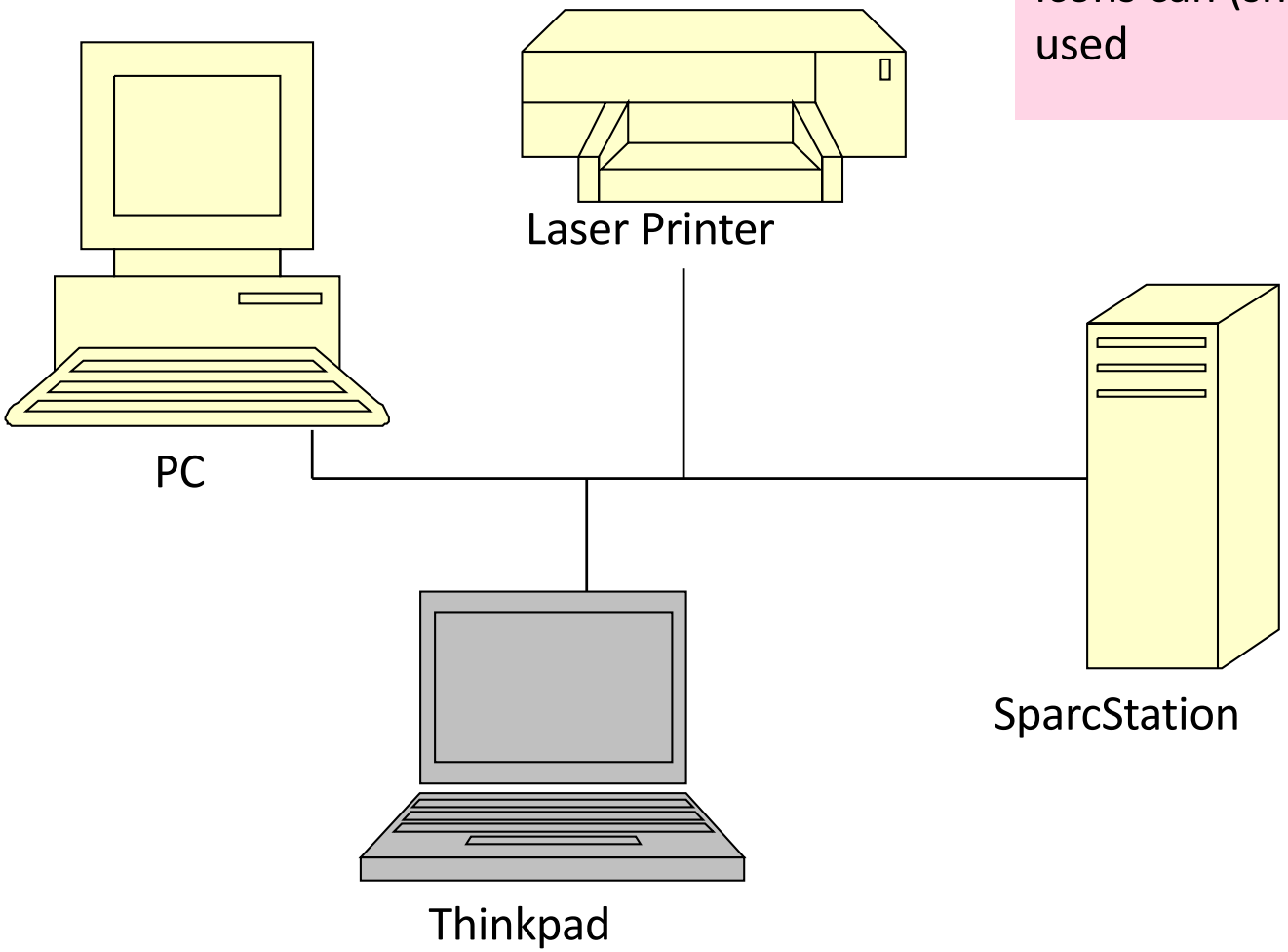


Example - Instance Form



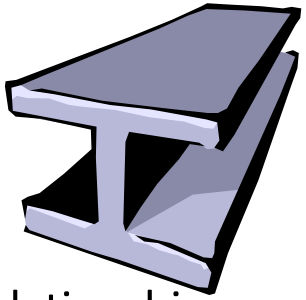
Note

Icons can (should) used



OTHER RELATIONSHIPS & DIAGRAMS

Relationships



Relationships

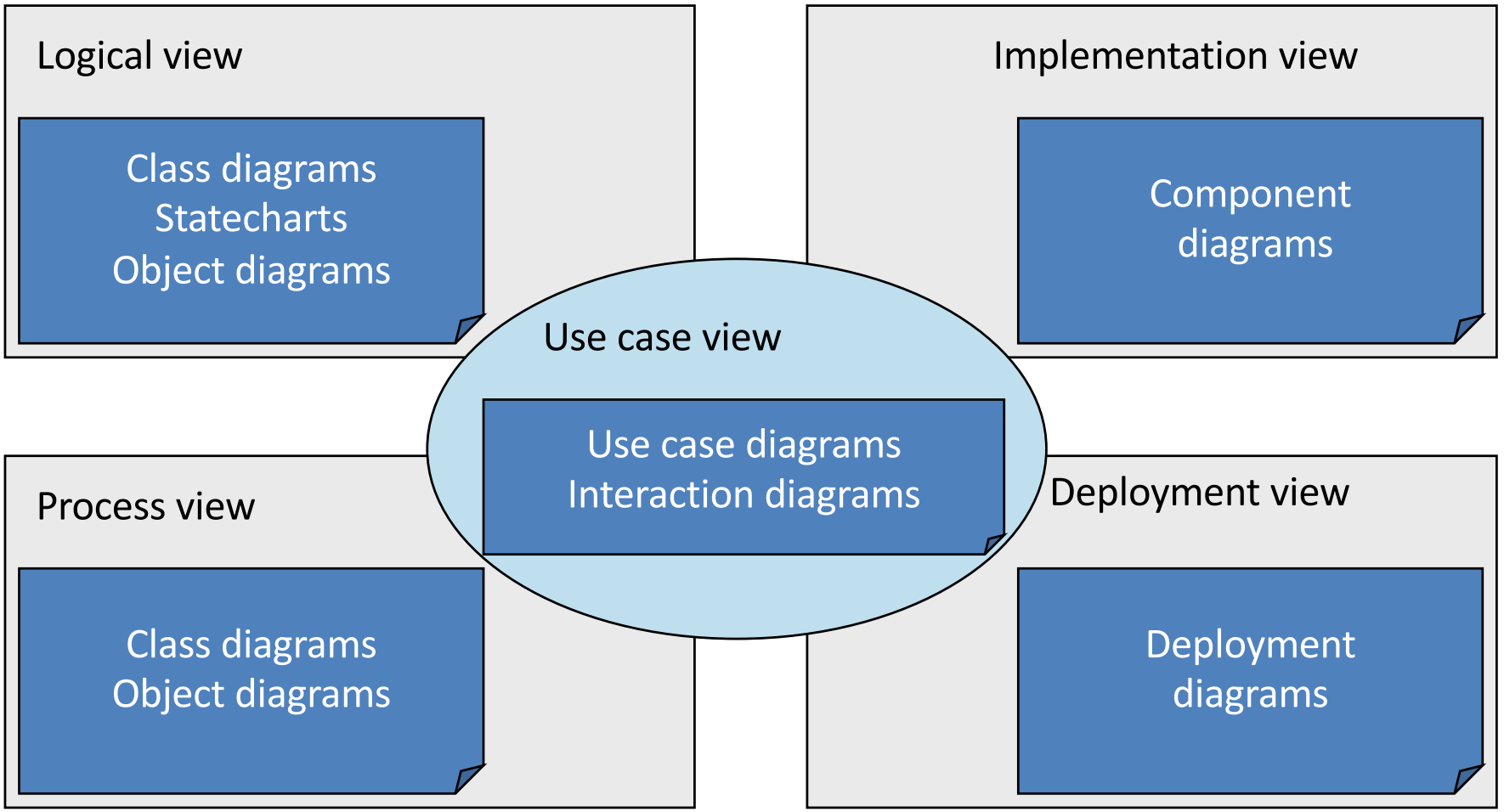
Association

Dependency

Generalization

Realization

Architecture (4+1)

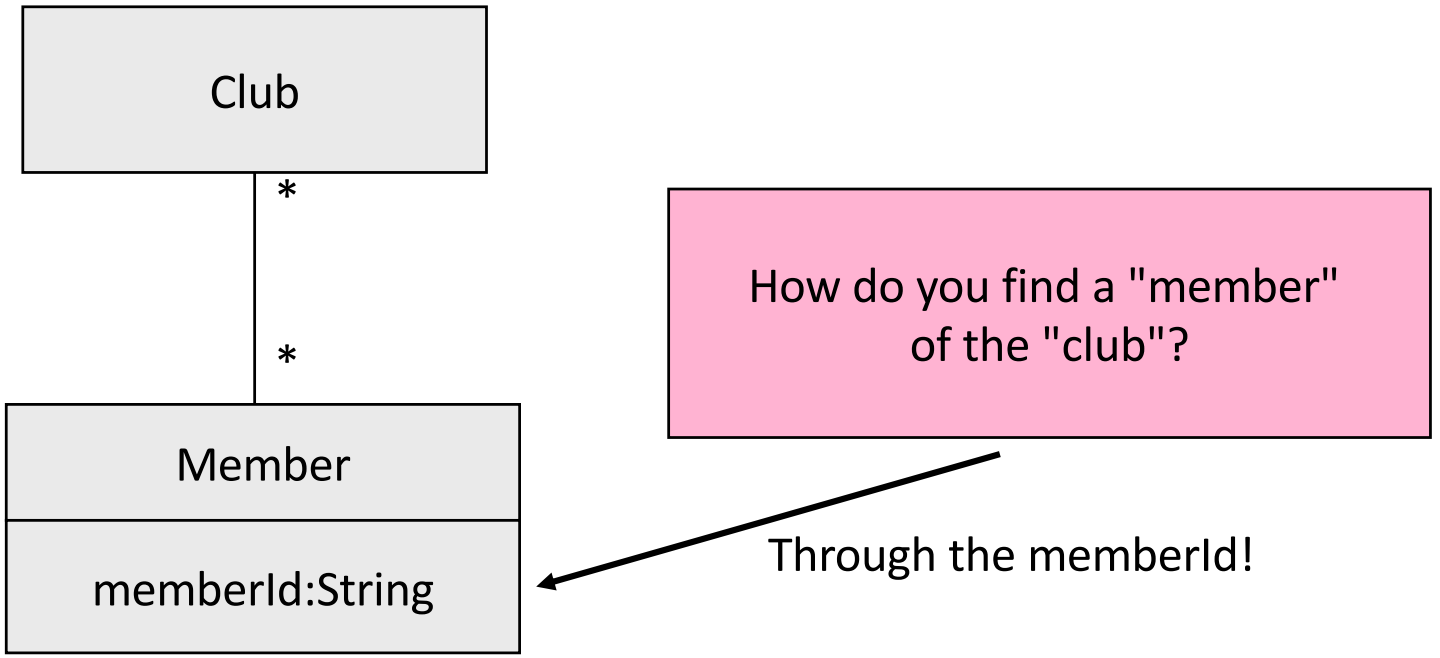


Other Relationships

- Qualified Associations
  - Reduction of n-to-n associations to n-to-1 associations
- Dependencies
  - The client depends on the supplier in a specific way.
  - **Example**: An object of one class is sent as a parameter to a method of another class.
- Further relationships will be introduced later...

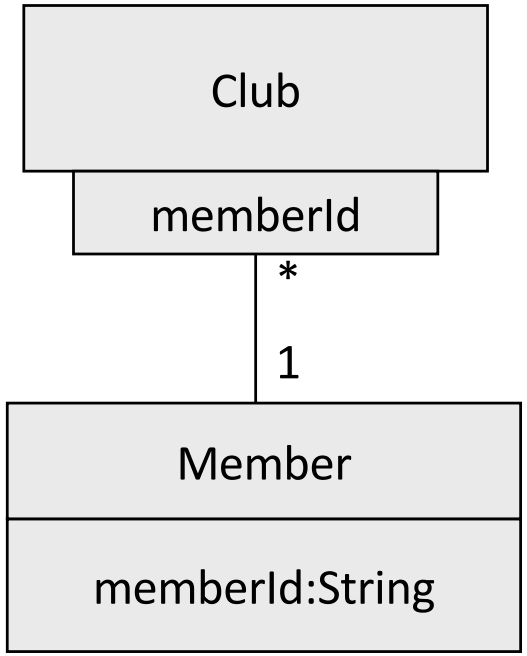
Qualified Associations

- A qualified association selects an element from a target set.



Other representation...

- Can always be used when a specific object can be selected from a set using a UNIQUE key.



Dependencies

- "A dependency is a relationship between two elements where a change to one element (the supplier) may affect or supply information needed by the other element (the client)."
- The client depends on the supplier!

Types of Dependencies

- Usage
  - The client uses a service provided by the supplier.
- Abstraction
  - The supplier is more abstract than the client.
- Permission
  - The supplier allows (restricted) access by the client.
- Binding
  - Only for languages that implement templates.

Usage Dependencies

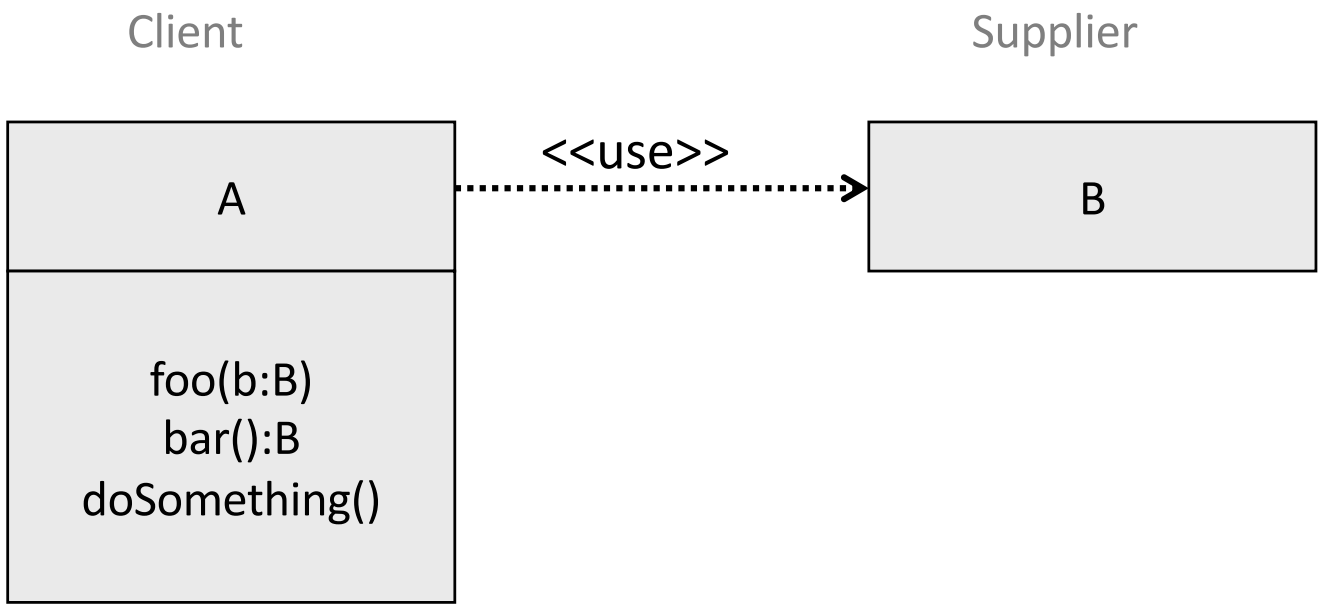
- <<use>>
  1. An operation of A requires a parameter of class B
  2. An operation of A returns a value of B
  3. An operation of A uses an object of B in a method (but not as an attribute)
- <<call>>

  The client operation calls a supplier operation.
- <<parameter>>

  The supplier is a parameter or return value of a client operation
- <<send>>

  The client sends the supplier (= signal) to an unspecified destination.
- <<instantiate>>

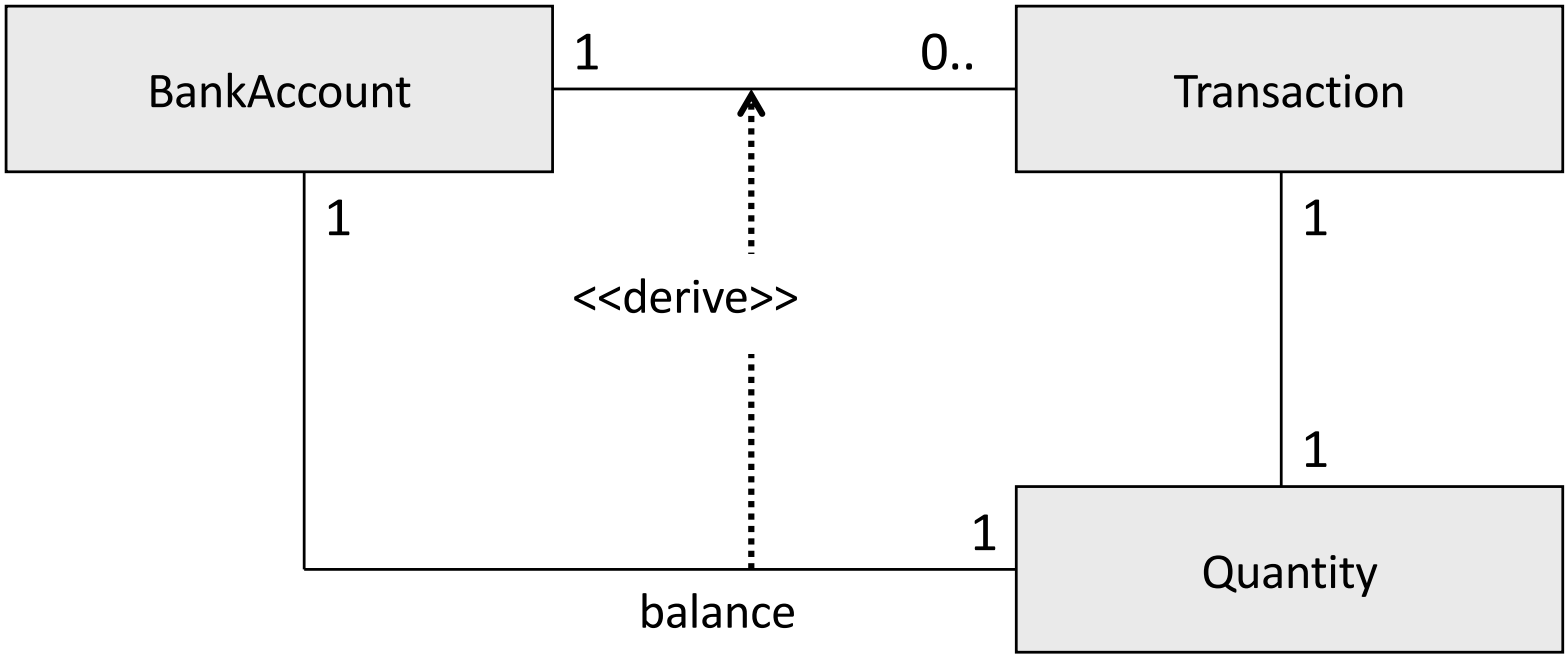
Example Usage Dependency



Abstraction Dependencies

- <<trace>>
  - To show that the supplier was developed at an earlier stage of the development process than the client. (Analysis vs. Design)
  - Or to map requirements to a use case (that supports this requirement).
- <<refine>>
  - To show that the client is a "refined" version of the supplier.
- <<derive>>
  - To show that one thing can be derived from another.

Example <<derive>>



Permission Dependencies

- <<access>>

  The supplier package allows a client package to access the PUBLIC content (the namespaces remain separate).

- <<import>>

  Same as <<access>>, but the supplier's namespace is connected to the client's namespace (in the client).

- <<friend>>

  The client element has access to the supplier element (regardless of visibility).

PACKAGE DIAGRAM

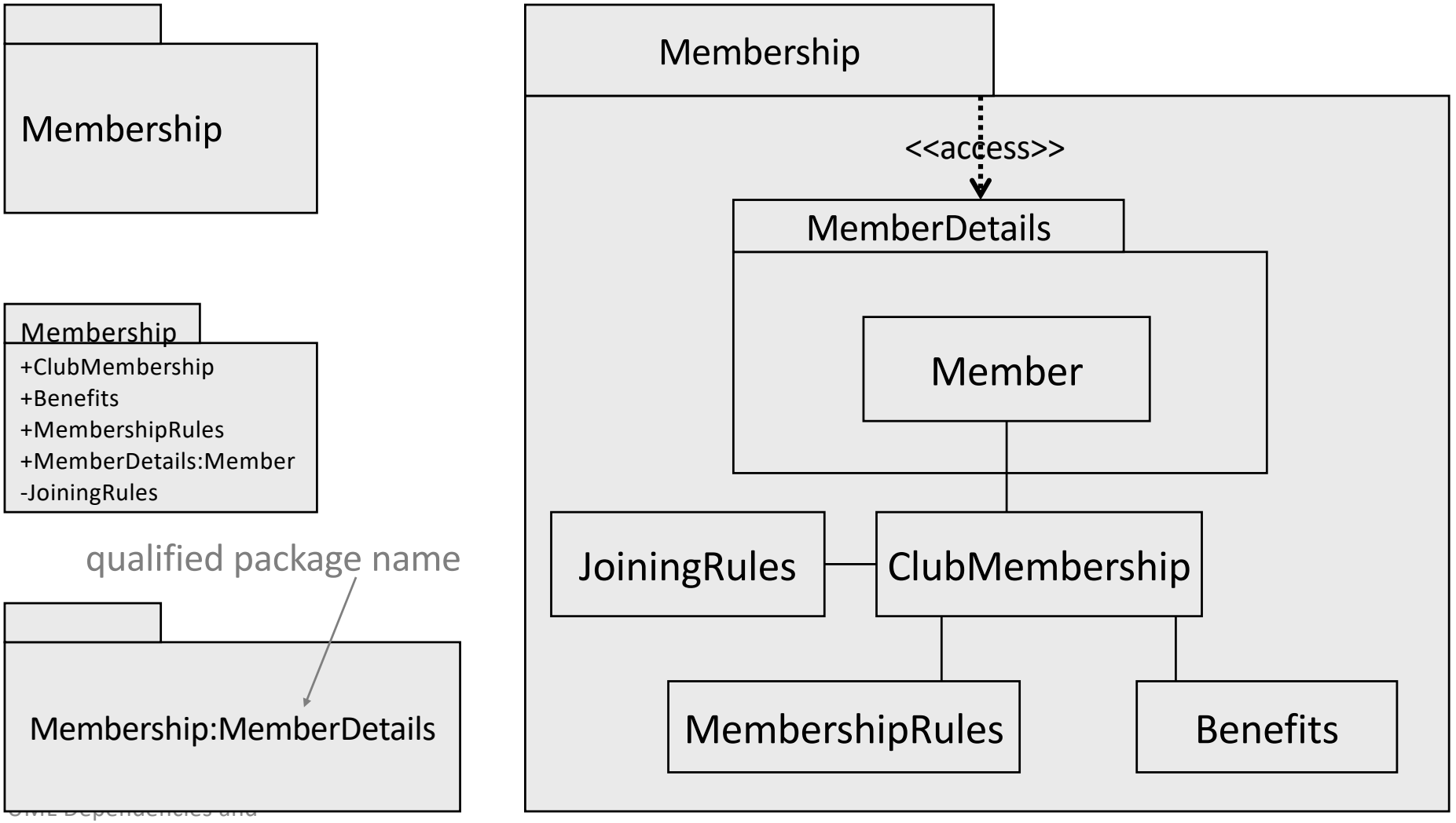
Why packages?

- To organize UML elements and diagrams into groups
- Uses:
  - To group elements with similar semantics
  - Delimitation of "semantic areas"
  - Units for configuration management
  - Units for parallel work
  - Separation of name spaces

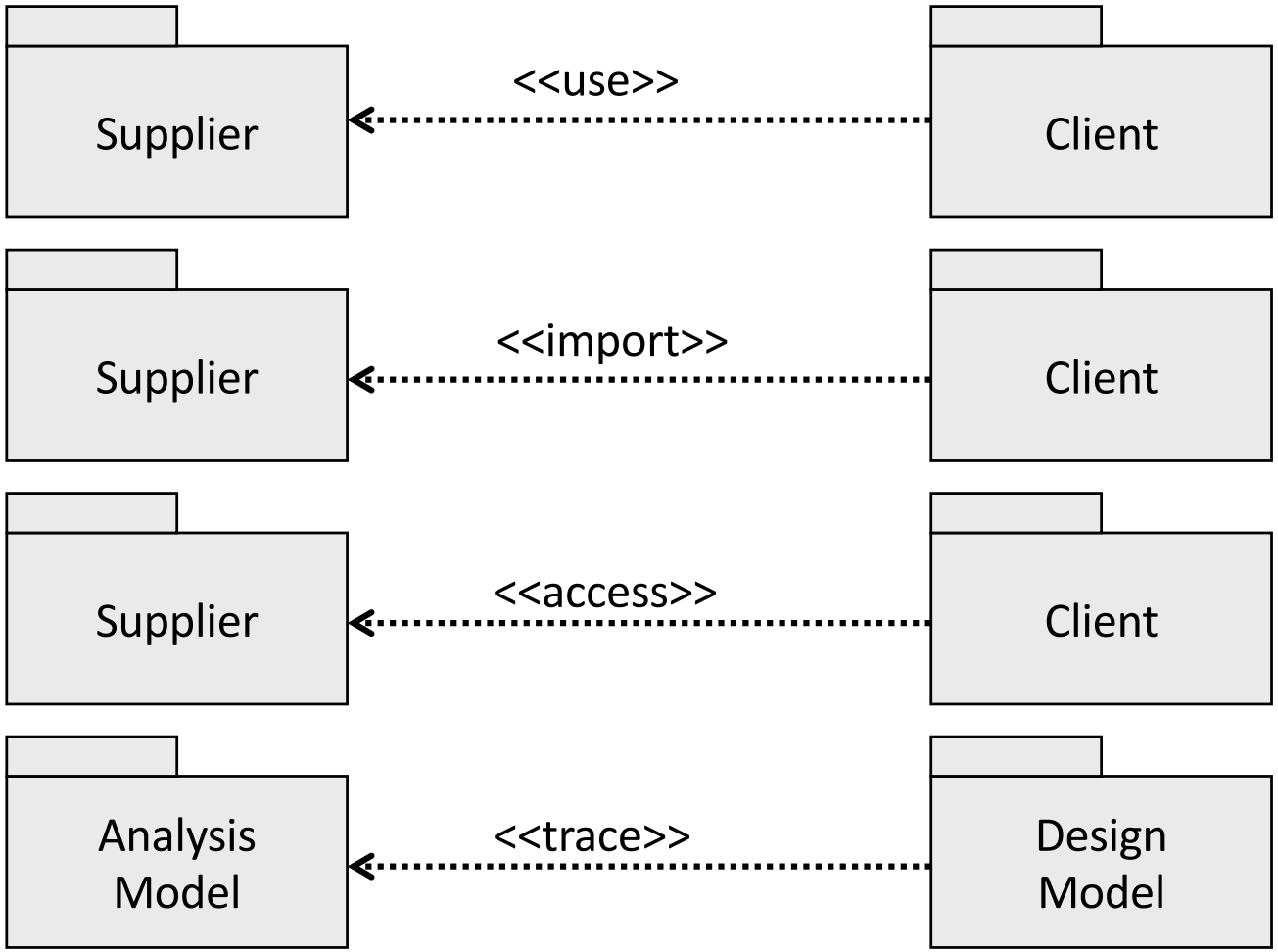
Packages – Structure and Content

- Each element or diagram belongs to a package
- Packages form a hierarchy.
- Top-level package <<topLevel>>
- Example: Analysis package with
  - use cases
  - Analysis Classes
  - Use case realizations
- Visibility for packages determines whether elements can be seen from outside or not.

Packages - Example

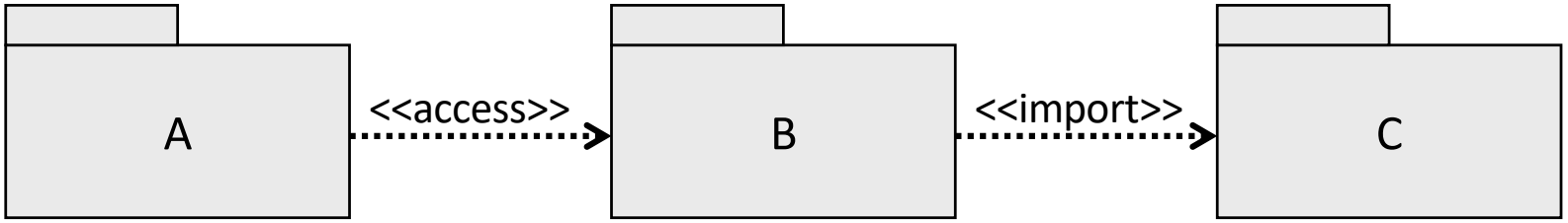


Package Dependencies



Transitivity

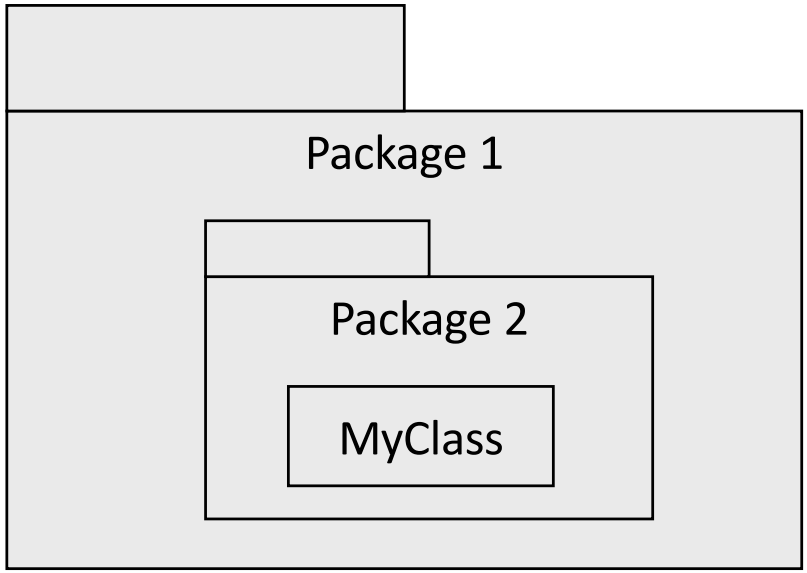
- <<access>> and <<import>> are NOT transitive!



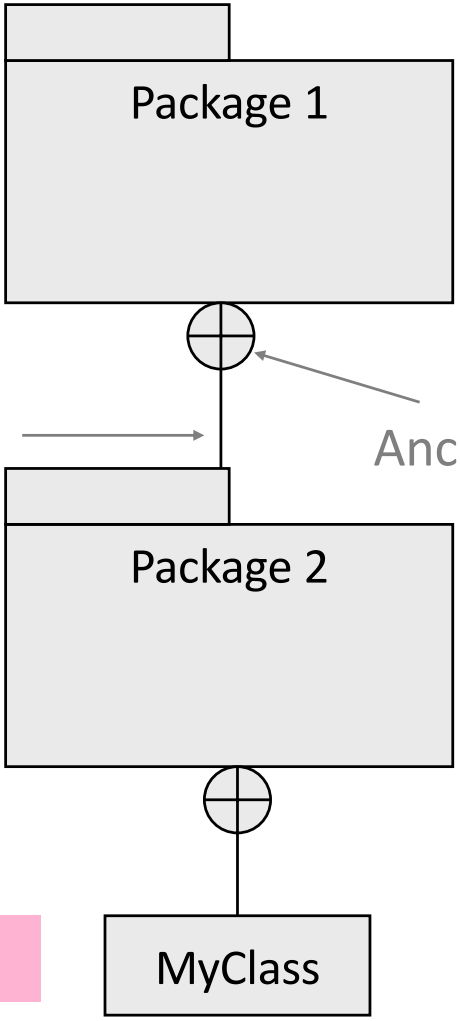
Elements of A have no access
to elements of C

Nested Packages

A package can contain the public elements of its containing package (but not vice versa).

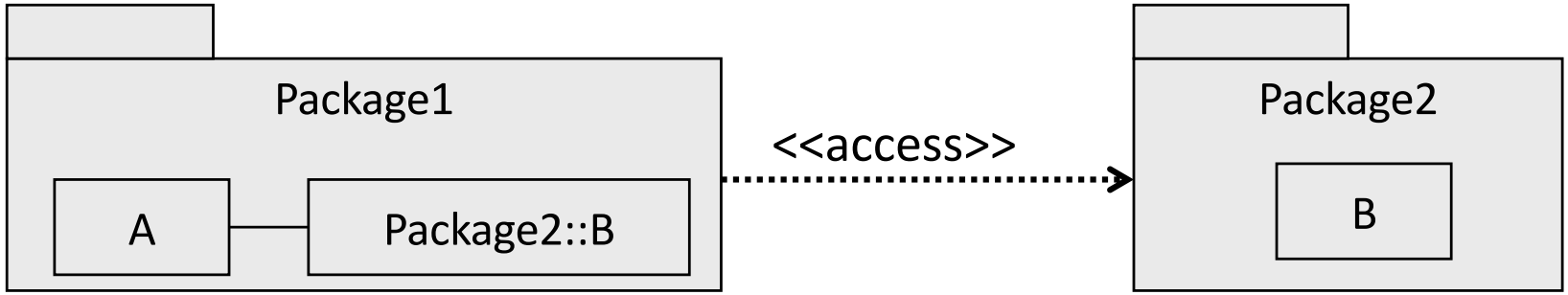


Containment Relationship



Pathname, e.g.: Package1::Package2::myClass

Example - Element Access



Generalization and Stereotypes

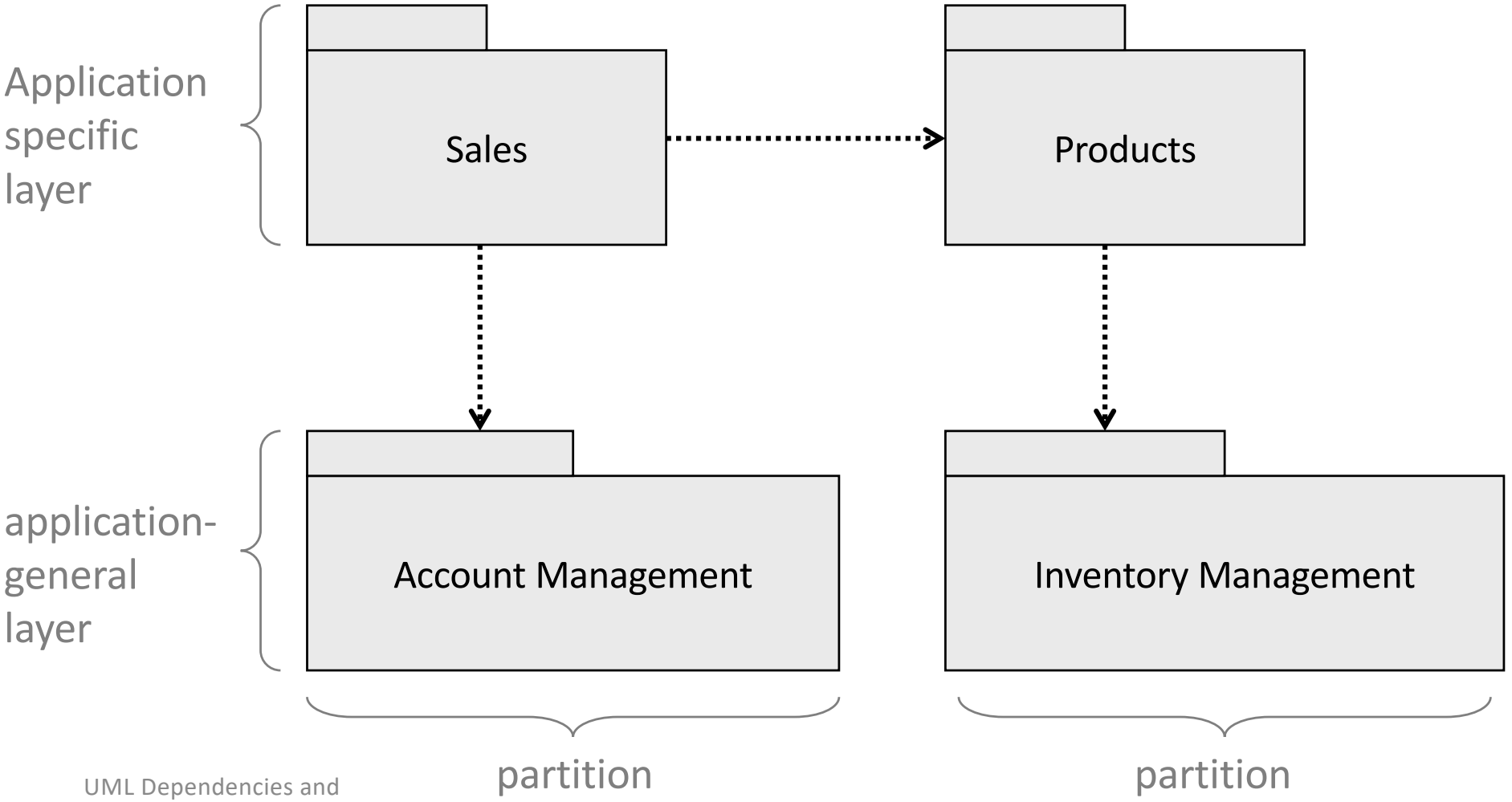
- Packages can be arranged in an inheritance hierarchy (just like classes).
- Stereotypes to define the semantics of packages more precisely.

<<system>>	The system to be modeled
<<subsystem>>	An independent subsystem
<<façade>>	A view of another package
<<framework>>	Pattern package
<<stub>>	Proxy for the public content of another package

Architecture Analysis

- Purpose:
  - Minimizing coupling between parts of the system.
  - Architecture analysis partitions related classes into analysis packages.
  - The analysis packages are then divided into levels.
- Goals:
  - Minimizing dependencies between analysis packages
  - Minimizing public and protected elements in each analysis package
  - Maximizing private elements

Architecture - Example

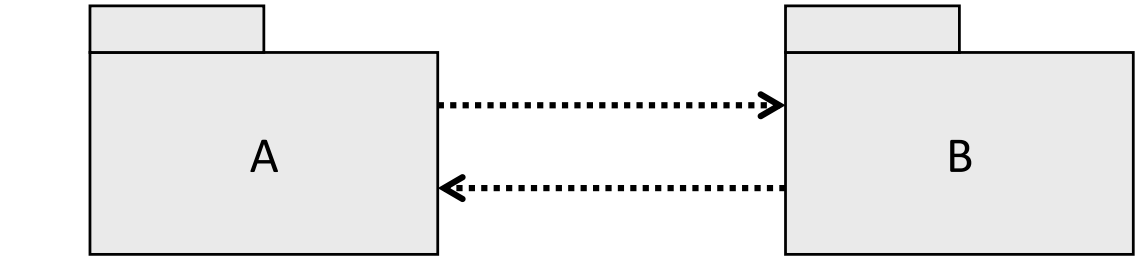


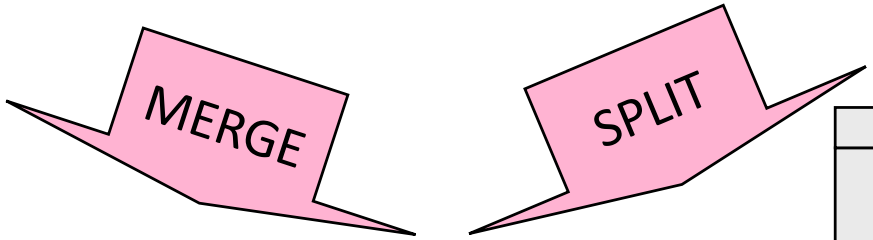
How do you find packages?

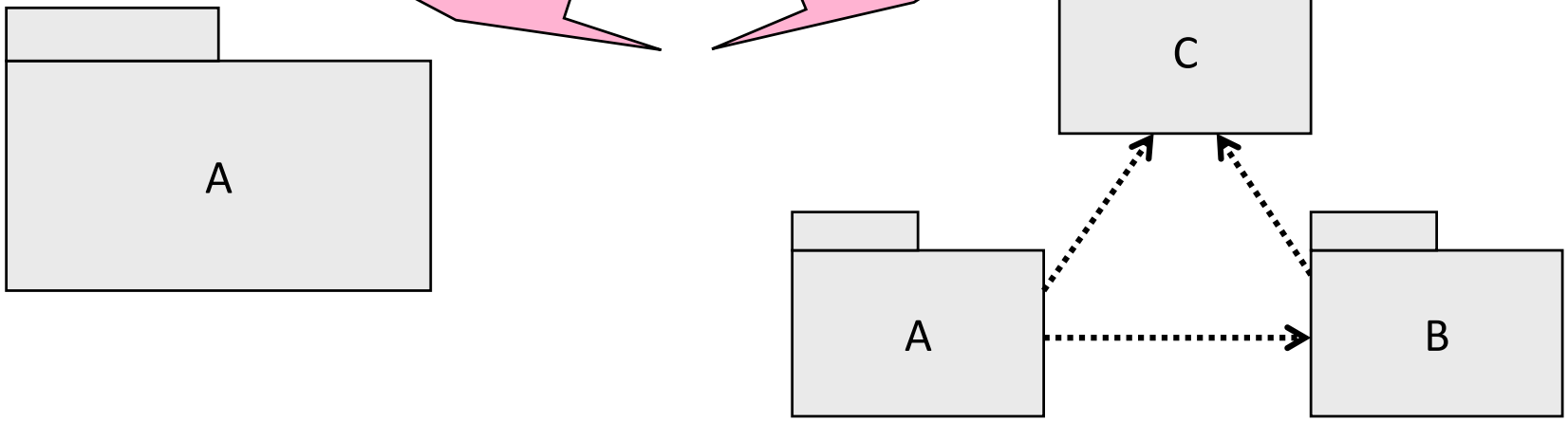
- Identifying model elements with a strong semantic connection
- Using the static model
  - Clusters of classes in the class diagram
  - Inheritance hierarchy
  - Use cases can/should also be used
- After identifying packages:
  - Minimization of public/protected elements and package dependencies by:
    - Moving classes between packages
    - Introducing new packages or removing old ones

Cyclic Package Dependencies

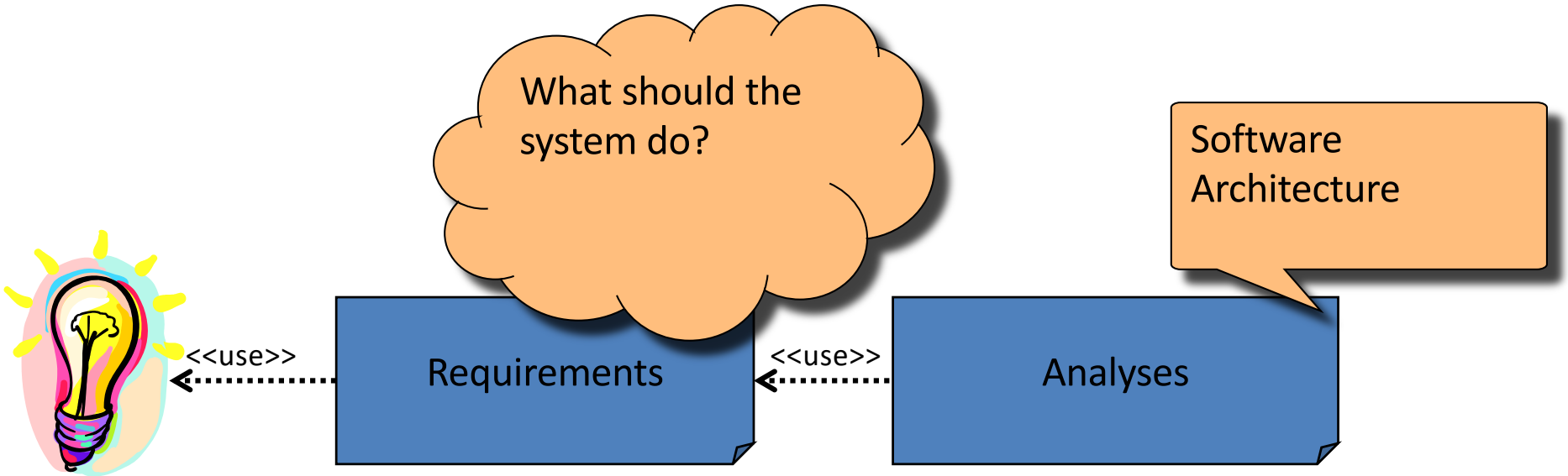
Cyclic dependencies should be avoided.







Summary Procedure



(Verbal) Description
Use Cases

Use Cases
Class Diagrams
Packages

USE CASE DIAGRAM

Use Cases

- An (itemized) story of possible use
- Created from a user interview
  - Tied to a user
- Use diagrams and text

- Use case prioritization

Use Cases Capture Functional Requirements

- Not captured:
  - Performance,
  - Reliability,
  - Hardware compatibility,
  - ...

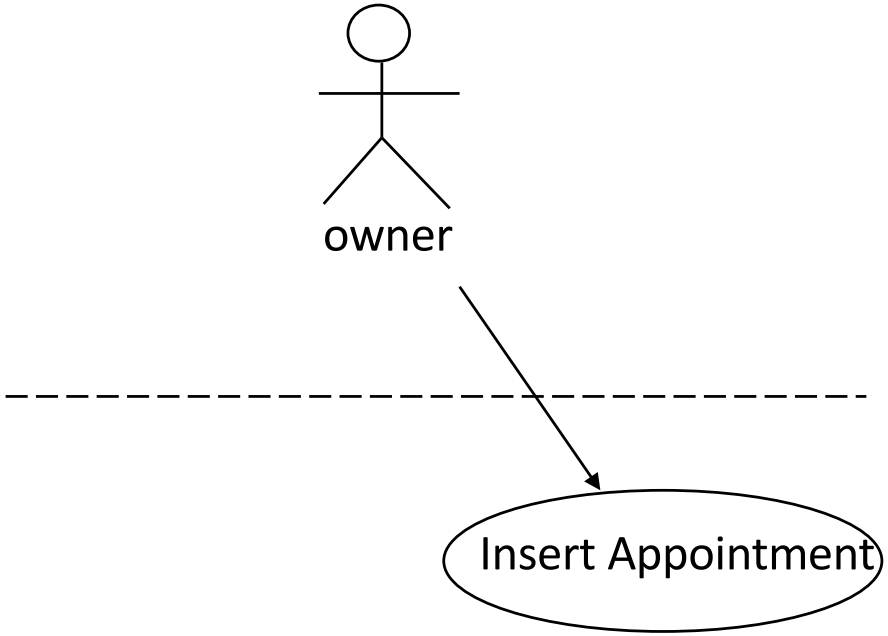
- Need to capture non-functional requirements separately

What to do

- Three tasks:
  1. Find system boundary
  2. Find actors
     - Actors are roles
  3. Find use cases

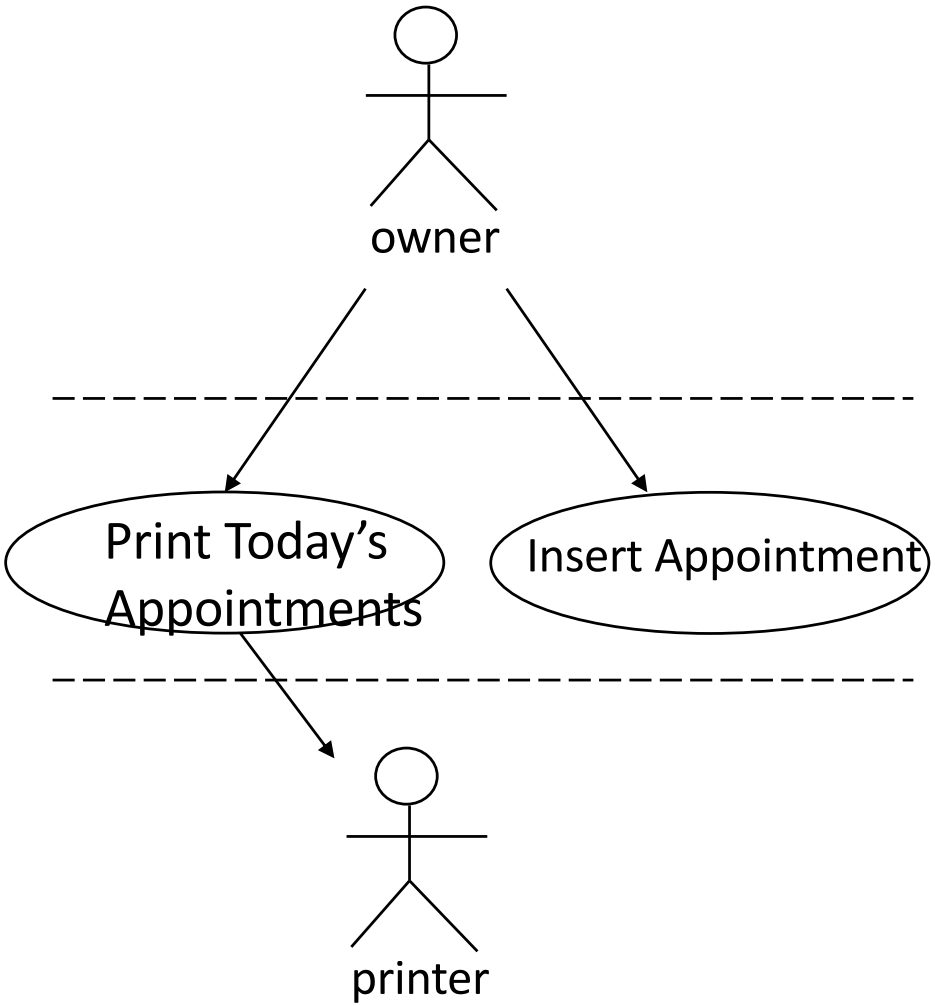
Use Case Diagrams

- Actors
  - Are outside the system
  - Interact with it
  - Are *roles*, not people
  - E.g.: calendar owner, secretary, MS Outlook, Time
- Use Cases
  - Describe one use of the system
- Relationships
- System Boundary
  - Actors are outside, use cases inside



Use Case Diagrams

- Actors
  - Are outside the system
  - Interact with it
  - Are *roles*, not people
  - E.g.: calendar owner, secretary, MS Outlook, Time
- Use Cases
  - Describe one use of the system
- Relationships
- System Boundary
  - Actors are outside, use cases inside



Identifying Actors

- Who or what uses the system?
- What roles do they play in interaction?
- Who maintains the system?
- What other systems interact with it?
- Who gets and provides information?
- Do things happen at fixed times?

Use Case

- For every role, describe typical interactions with the system
- We want to understand the requirements, not model the system in detail.

Example

Use Case: Insert Appointment

ID: UC1.1

Actors: owner

Preconditions:

Flow of events:

1. The user specifies the date, start time, end time, and description of the appointment.
2. The calendar warns about conflicting appointments
3. The appointment is entered in the calendar

Postconditions: The appointment is in the calendar

Requirements

- The uses are
  - Clear
  - Detailed
- Is anything missing? (Who, what, when, where?)
- Bad use case: The user enters the appointment data.
- Use cases describe one example!
- Details depend on the phase.

Example: a Later Iteration

Use Case: Insert Appointment
ID: UC1.2
Actors: owner
Preconditions:
Flow of events:

1. The owner selects the day of the appointment
2. The system shows the day, including all appointments already scheduled
3. The user clicks on the start time, drags to the end time, and releases the mouse button
4. The system displays a box for the appointment and a cursor inside the box. If there are conflicting appointments, the box is displayed in a different color.
5. The user types a description.
6. The appointment is entered in the database

Postconditions: The appointment is in the calendar

More Complicated Use Cases

- A use case describes one example of use!
- But: what to do with very similar cases?

Extra Keywords

- If, for, while
  - If there is a conflict, the system warns
  - For all appointments in the day, the calendar shows a box
- Alternative flows

Example: Alternative Flows

Use Case: Insert Appointment

ID: UC1.1.1

Actors: owner

Preconditions:

Flow of events:

1. The user specifies date, beginning time, end time, and description of appointment
2. If there is a conflicting appointment
    1. The calendar issues a warning
3. The appointment is entered in the calendar

Postconditions: The appointment is in the calendar

Alternative Flows: The user may cancel the creation of an appointment at any time before entering the description

Postconditions: The calendar is not changed

Alternative Flows: The user may leave the description blank

Postconditions: The calendar enters the description "appointment."

Scenarios

- A path through a use case (without branches, ifs, fors, and whiles) is called a *scenario.*

- Try to stick with scenarios as much as possible, especially initially.

- When are use cases good?

Benefits and Drawbacks

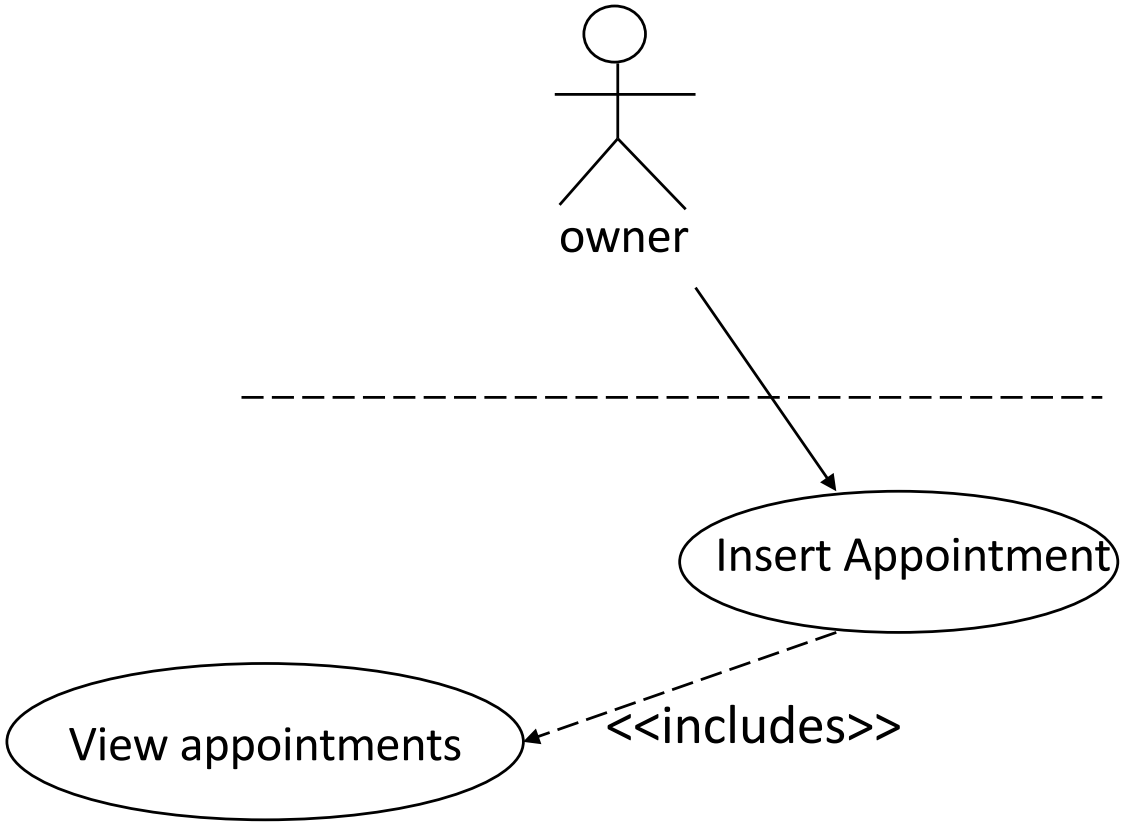
- Use cases are good when
  - There are many actors
  - There are many functional requirements
  - The functional requirements are not immediately clear
- Not as good for e.g.
  - A very fast sorting algorithm (too much)
  - A highly secure system (too little)

Advanced Concepts

- Actor generalization
- Use case generalization
- Inclusion
- Extension points

- Use extreme case with advanced concepts
  - They can easily be misunderstood
  - They can easily make use cases more complex, instead of simple.
  - Remember, use cases are also for clients

Inclusion



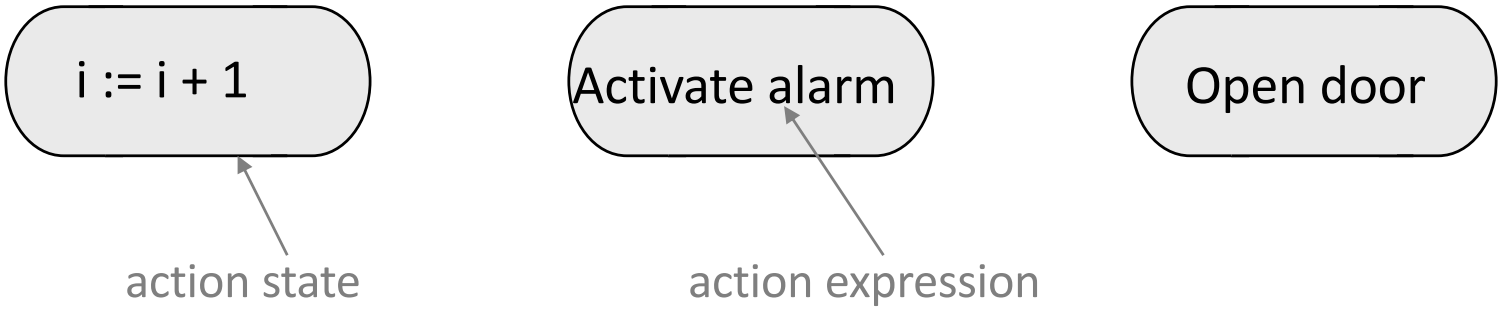
ACTIVITY DIAGRAM

What are activity diagrams?

- "OO Flow Charts"
- Process modeling through
  - activities and transitions
- Special case of state charts
  - Each state has an action that specifies a process or function and is executed when the state is entered.
- Activity diagrams can be attached to any modeling element.
  - Use cases, classes, ..., business processes

Action States

- Atomic Units of Activity Diagrams
- Represent a state
- Have an action expression
- The execution of an action expression cannot be interrupted
- The time required for execution is irrelevant.



Special Activity States

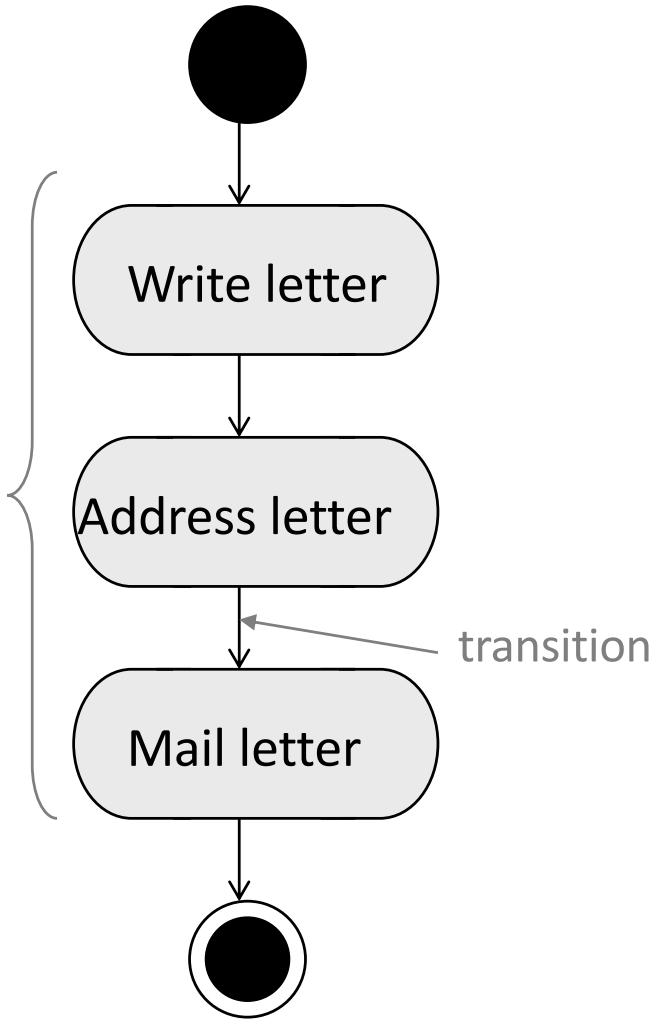
- Start and Stop States
- Every activity diagram begins with the start state and ends with the stop state.

Subactivity States

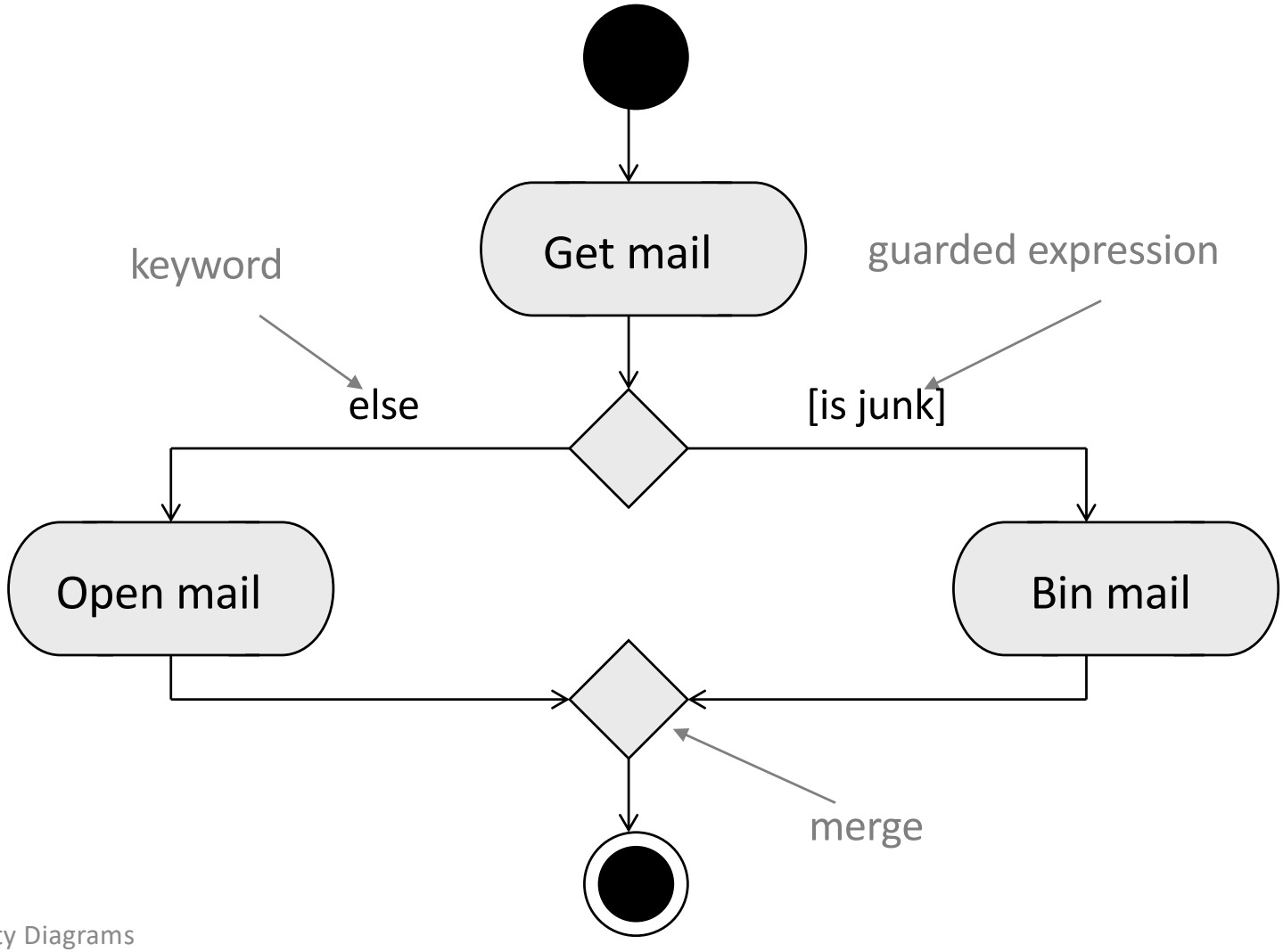
- Contain a nested activity graph
- Subactivity states can in turn consist of subactivity states or activity states.

Transitions

- Connection between states of an activity diagram
- When a state is complete, the next connected state is executed.

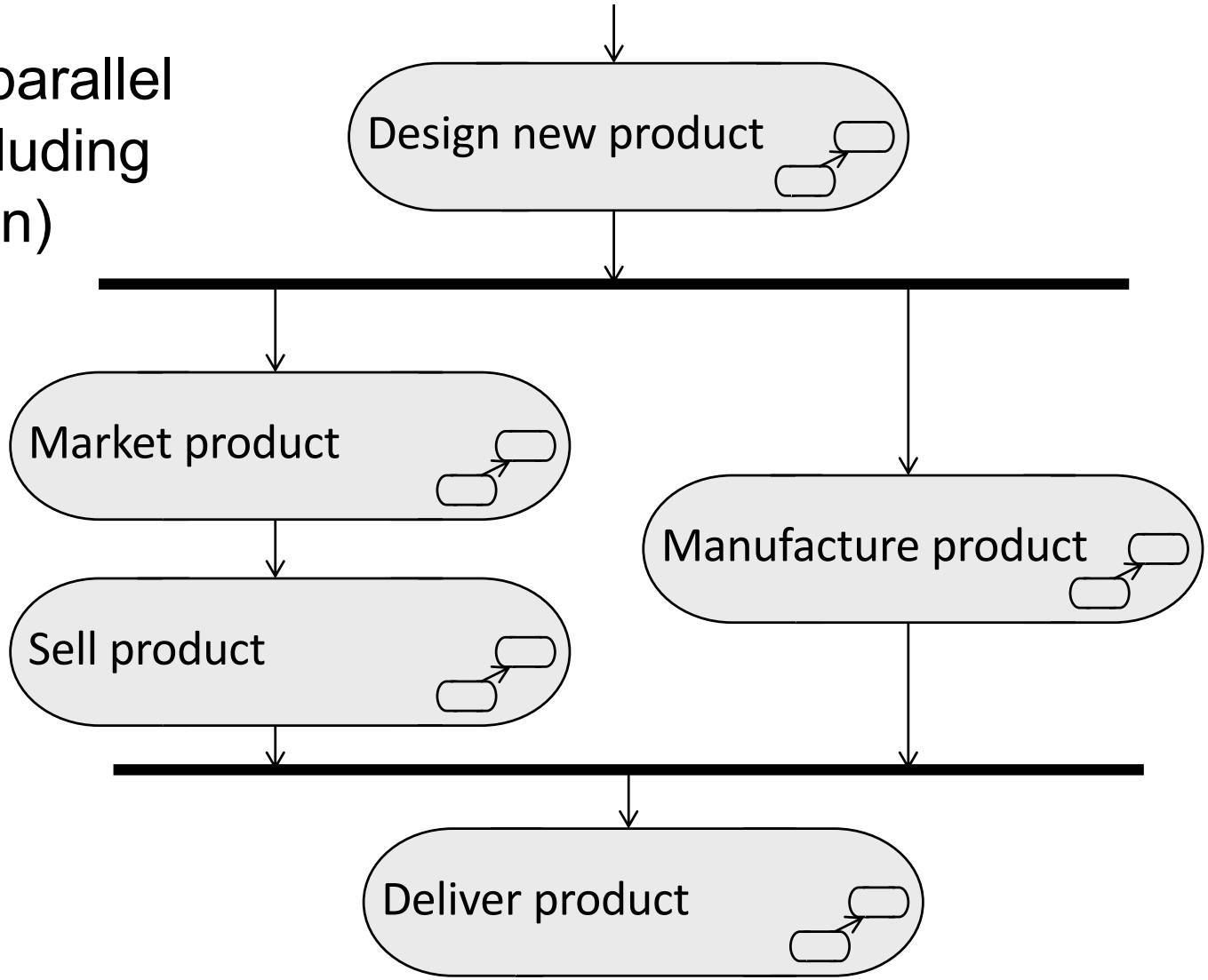


Decisions

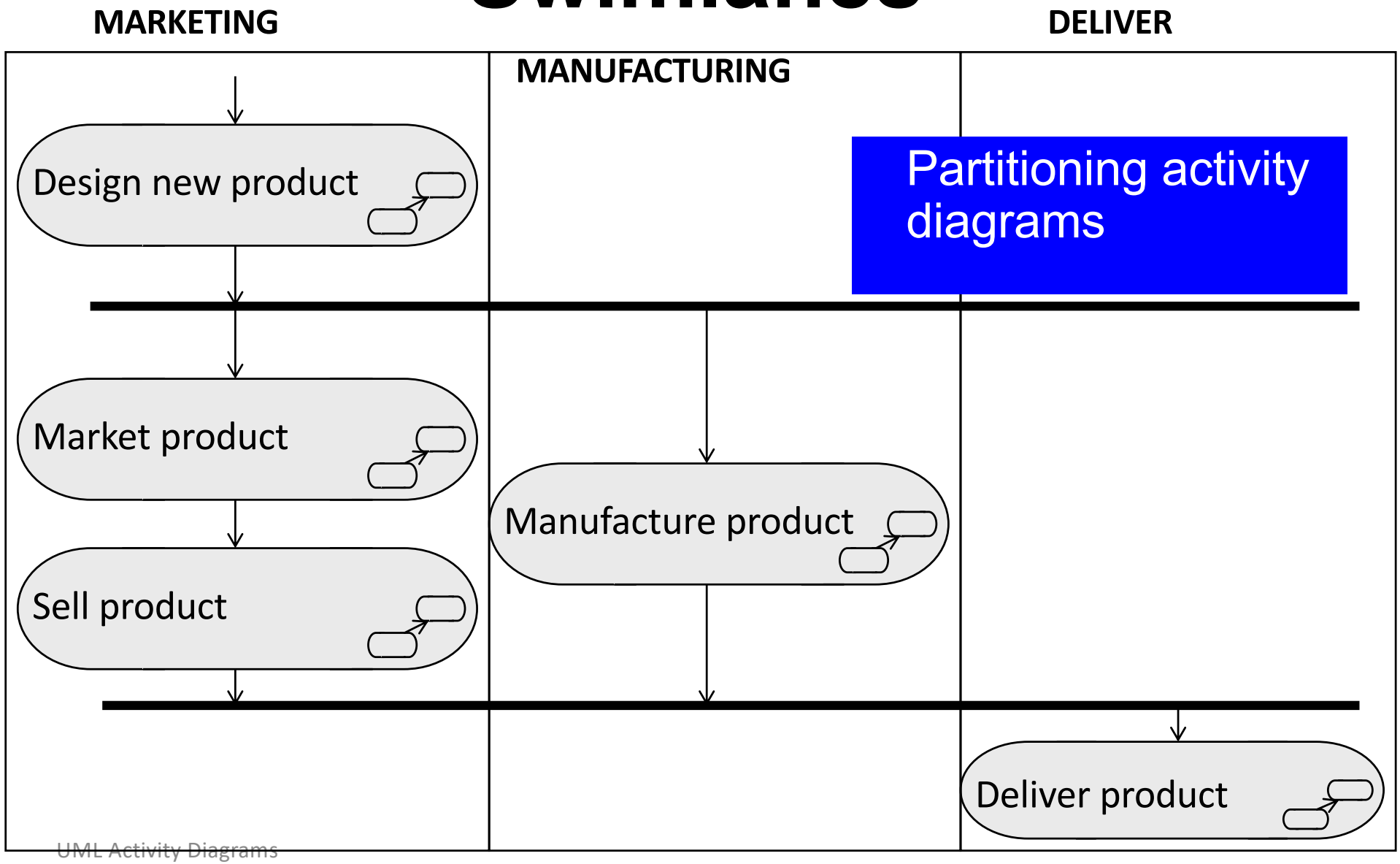

Get mail
keyword
guarded expression
else
[is junk]
Open mail
Bin mail
merge

Forks and Joins

- For modeling parallel workflows (including synchronization)



Swimlanes


MARKETING
MANUFACTURING
DELIVER
Design new product
Partitioning activity diagrams
Market product
Manufacture product
Sell product
Deliver product

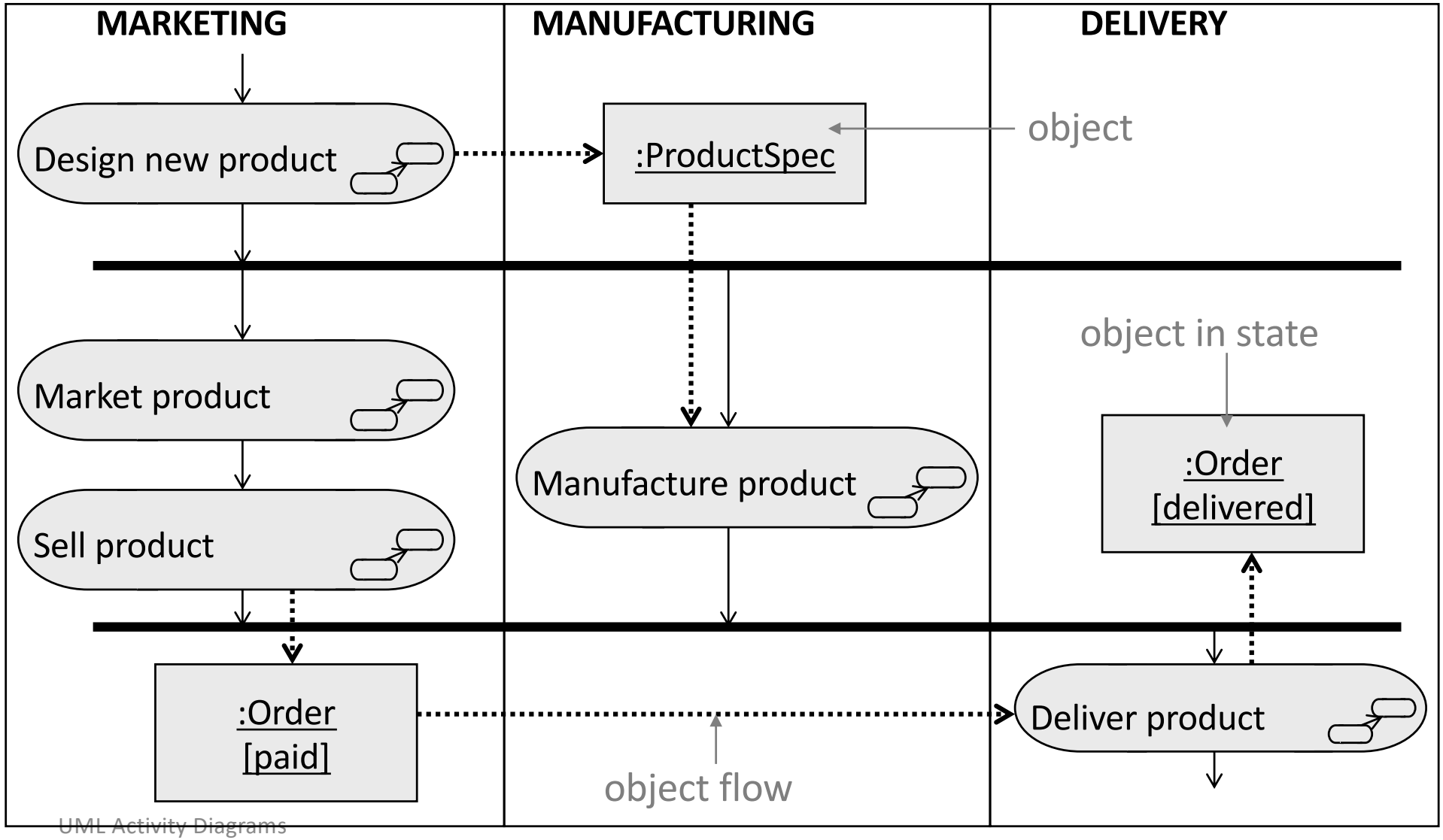
Interpretation of Swimlanes

- In UML, the semantics of swimlanes are not defined.
- Swimlanes represent:
  - Organizational units
  - Use cases
  - Classes
  - Processes
  - ...

Object Flow

- Activities can have objects as inputs or outputs. The state of objects can also be changed by activities.

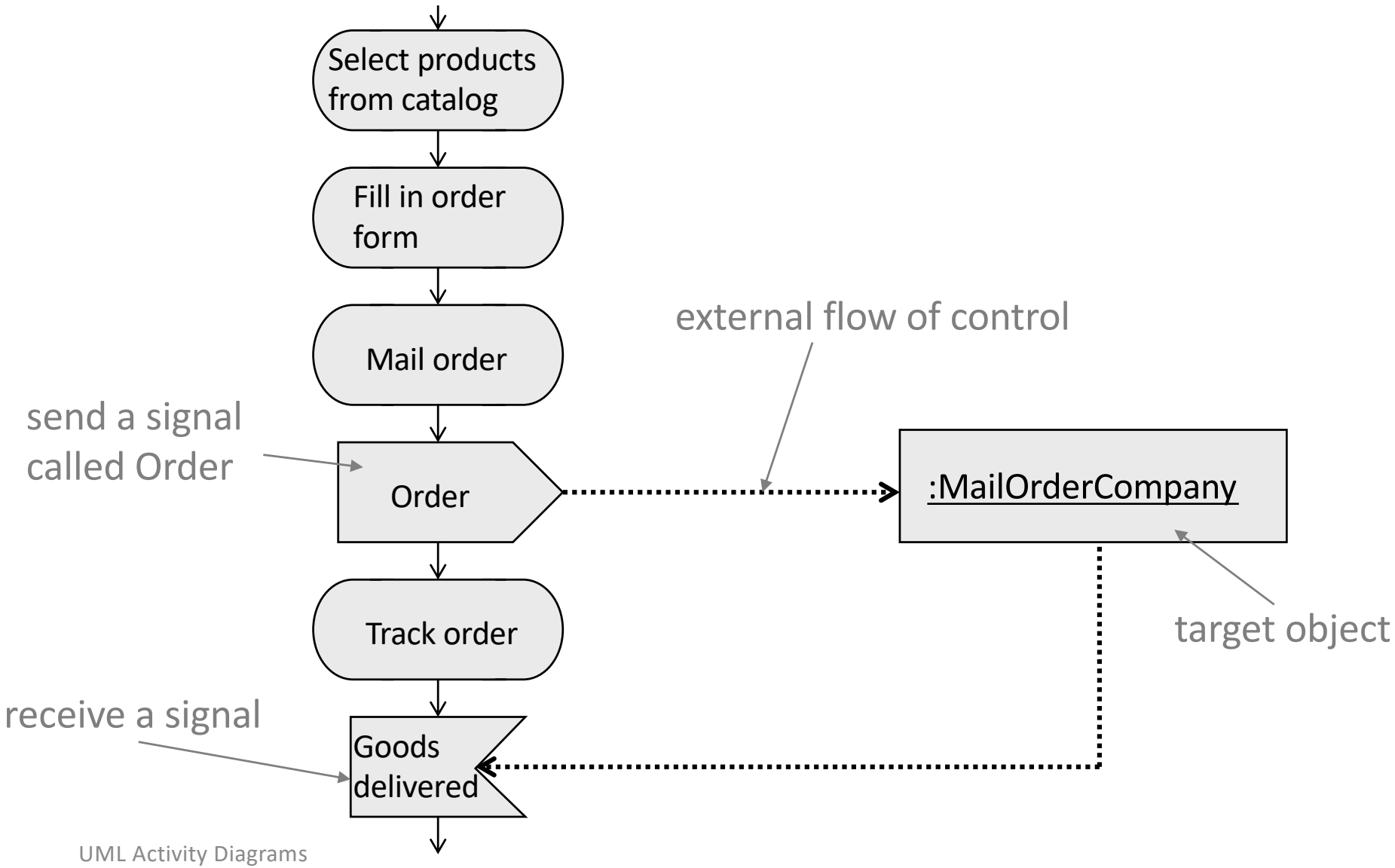
Object Flow - Example



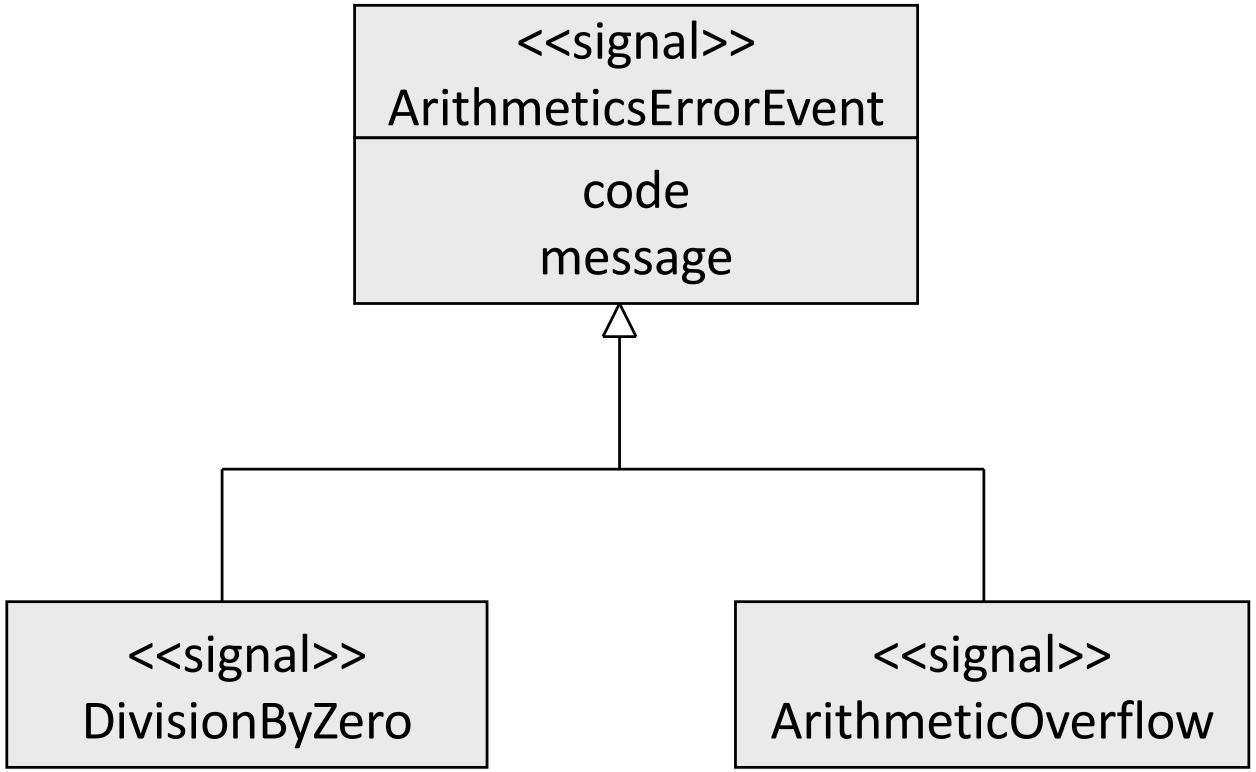
Signals

- A signal is used for asynchronous communication between objects
- Signals can also be used in activity diagrams
- Signals in UML can be modeled as classes with the stereotype `<<signal>>`.
- Signals have an implicit function `send(targetSet)` and otherwise only attributes.

Signals - Example

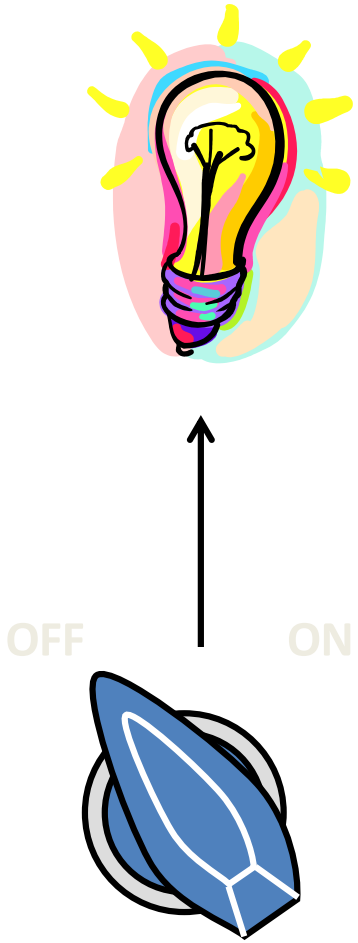


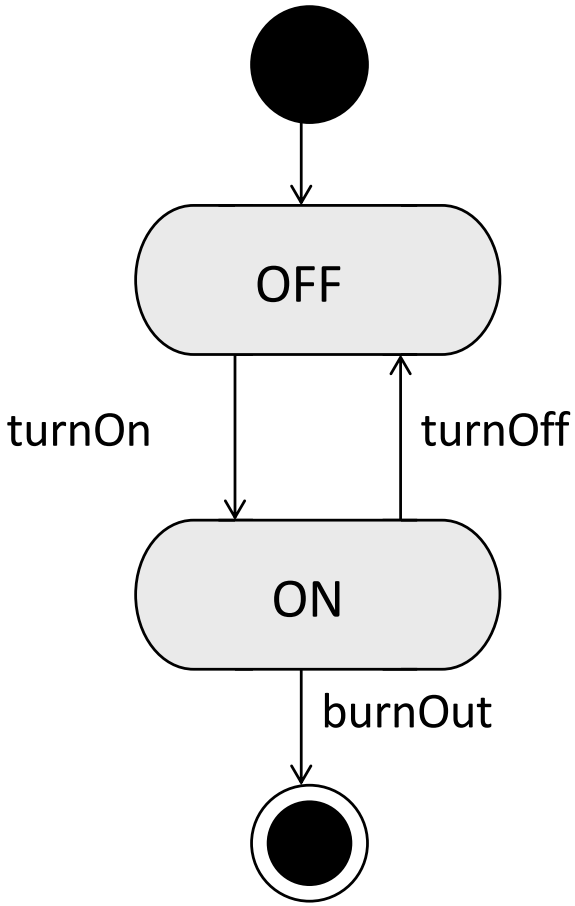
Signal Hierarchy - Example



STATE MACHINE DIAGRAM

Example





States

- *"..a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event."*
- The state of an object is given by:
  - Attribute values
  - Relationships to other objects
  - Activities performed by the object.

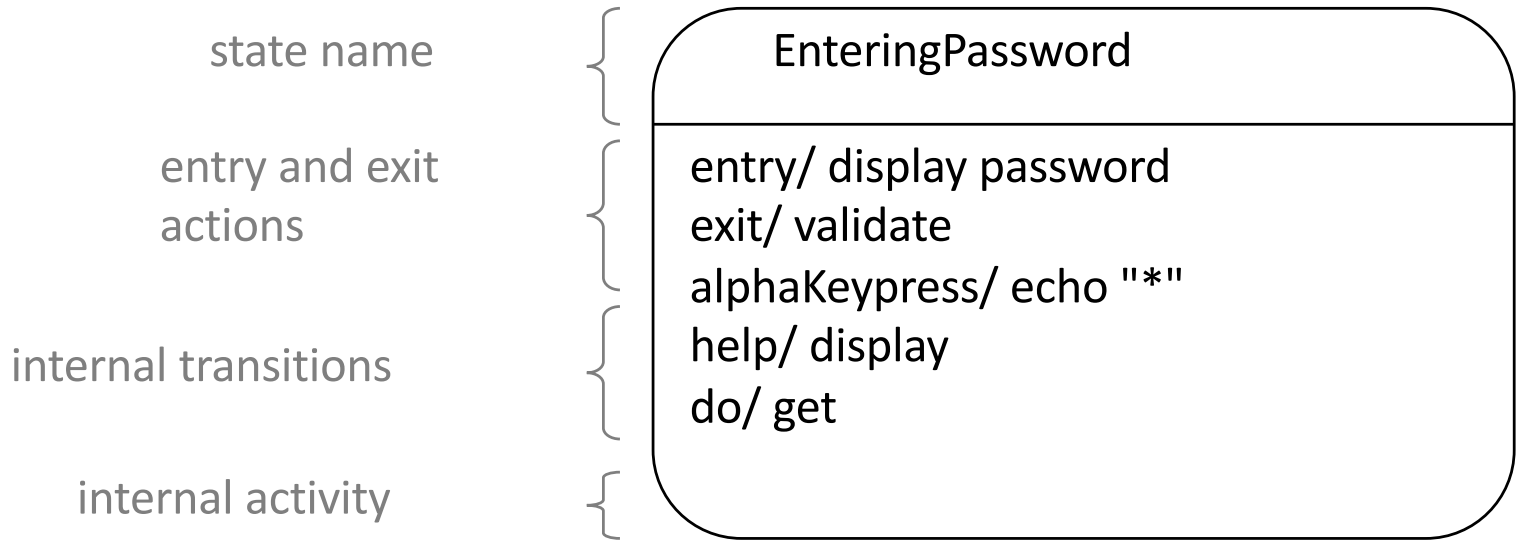
States of objects?

- State = Semantically significant property of an object
- Identify states of objects that really differ!
- Example:

```
class Color
{
    int red;
    int green;
    int blue;
}
```

What are the states?

State Syntax

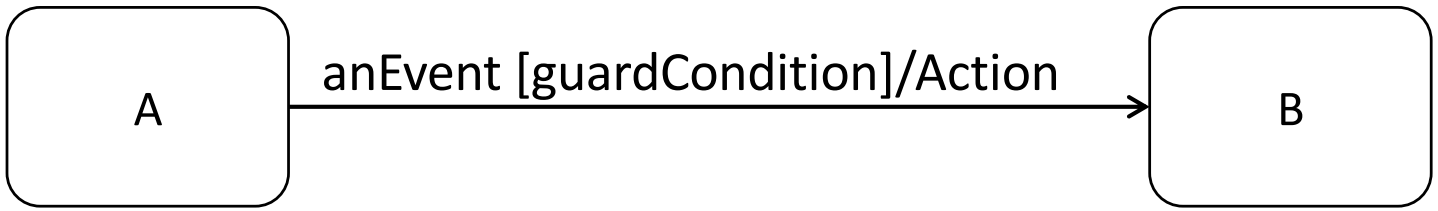


Actions cannot be interrupted and do not require any time.

Activities can be interrupted and require a finite amount of time.

Transitions

- Event: External or internal, trigger for transition
- Guard Condition: Boolean expression that must evaluate to TRUE
- Action: Executed when the transition is fired.

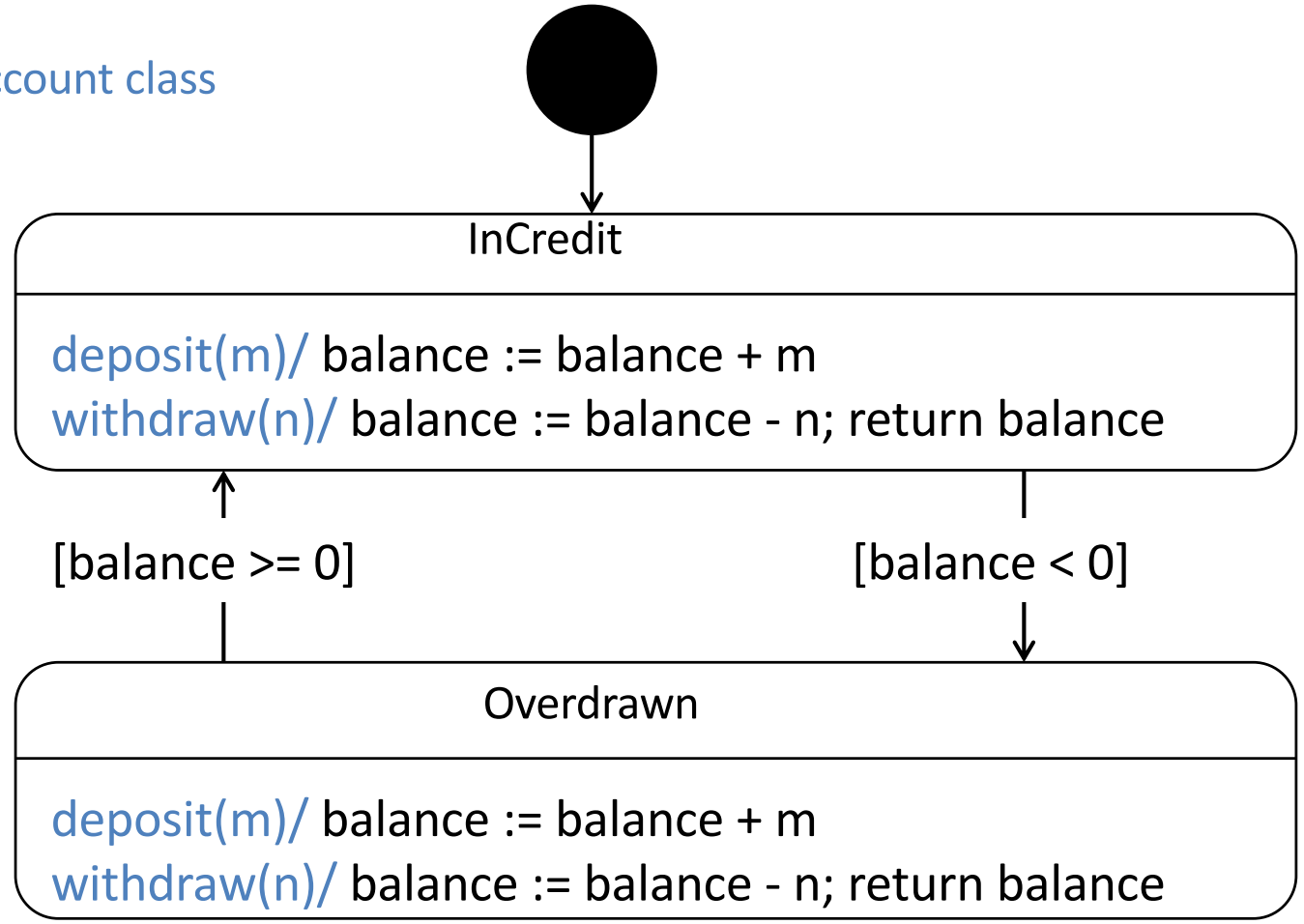


Events

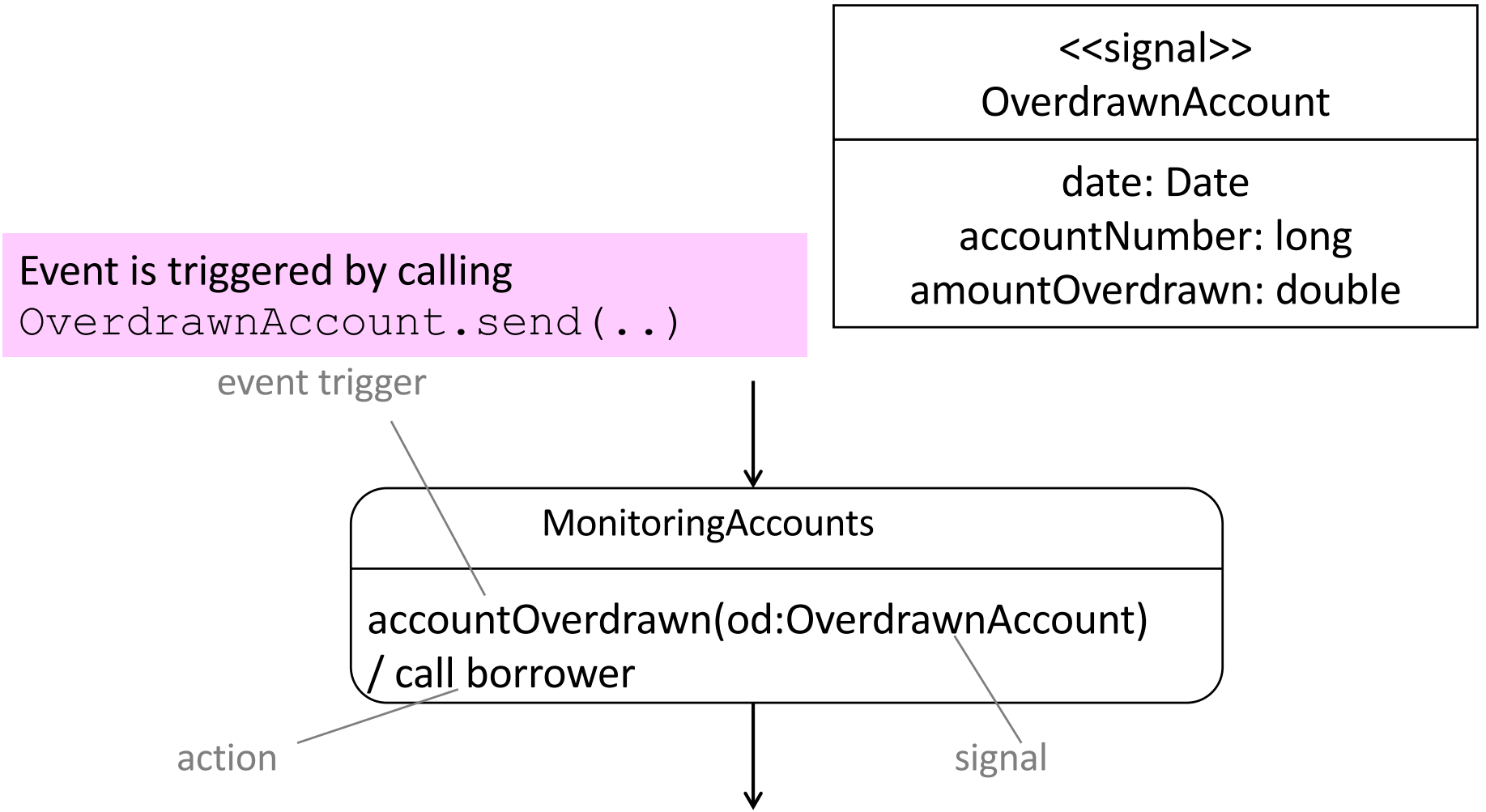
- 4 types:
  - Call events: Triggers the execution of a class method.
  - Signal events: One-way, asynchronous communication between objects.
  - Change events: Action associated with the event is executed when a condition evaluates to TRUE.
  - Time events

Example - Call Event

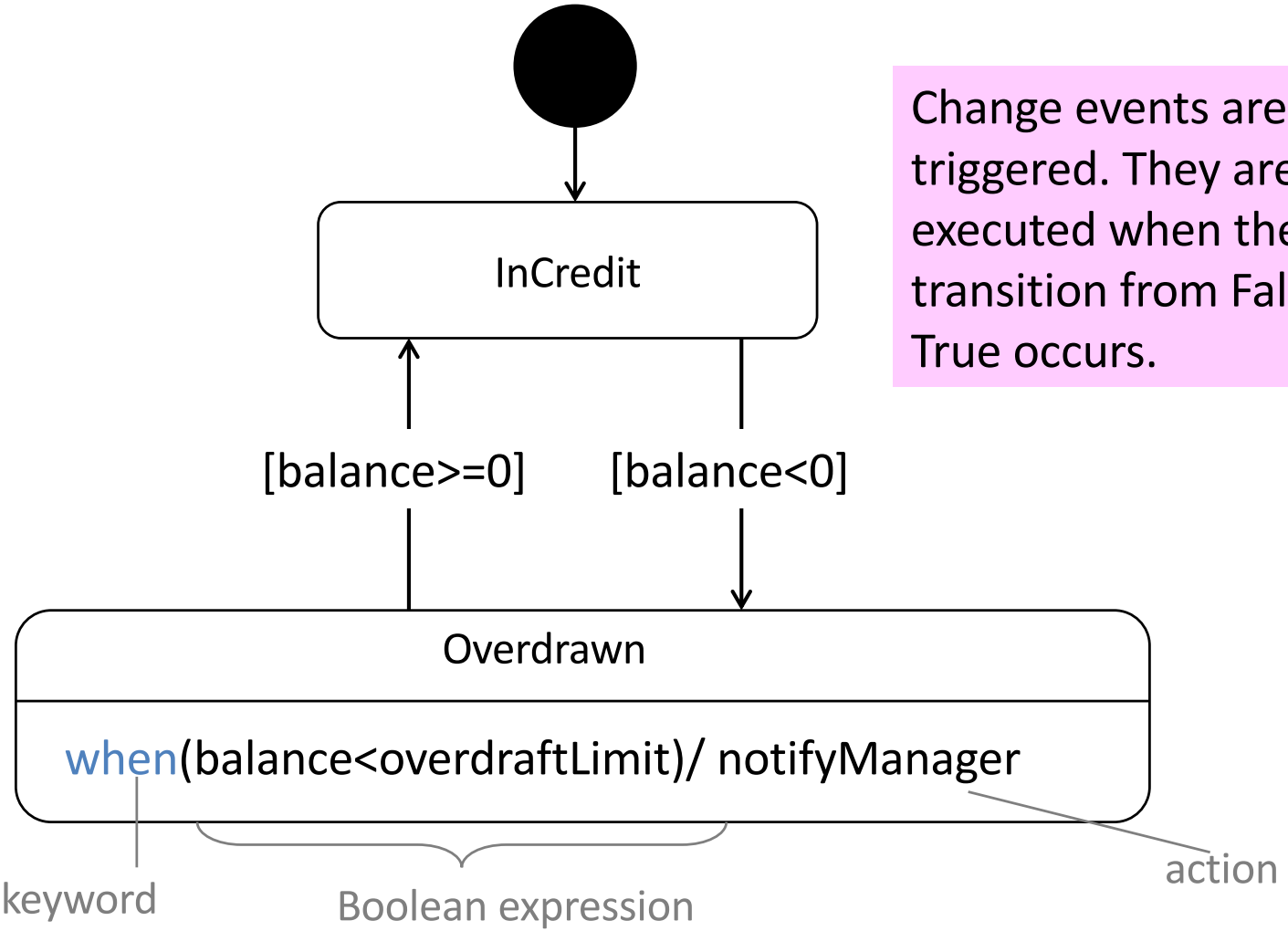
Context: BankAccount class



Example - Signal Event

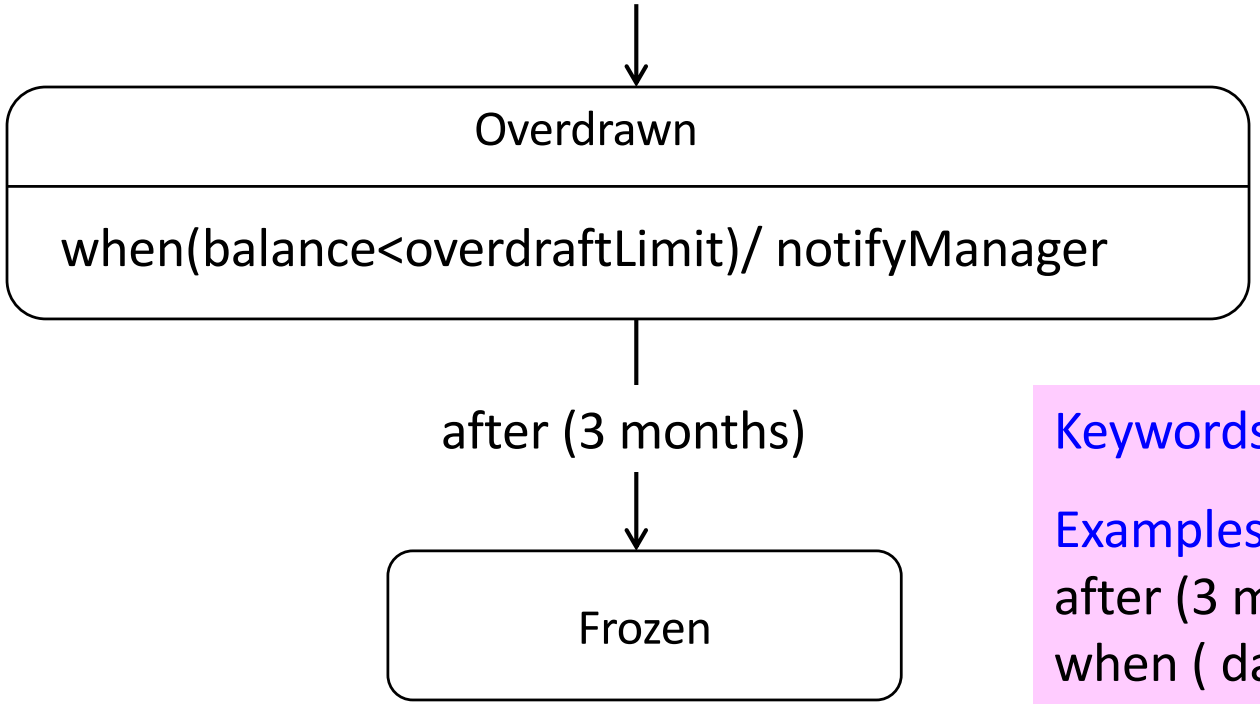


Example - Change Event



Change events are edge-triggered. They are only executed when the transition from False to True occurs.

Example - Time Events



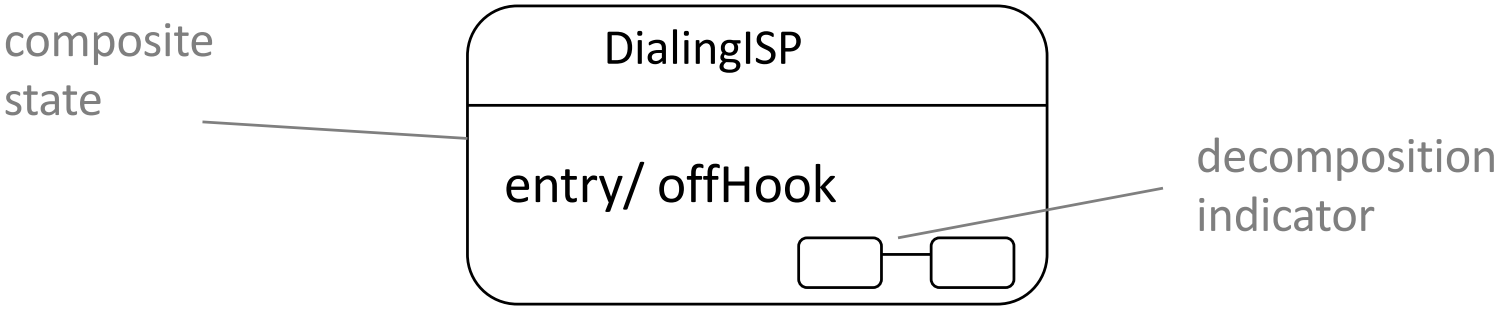
Keywords: after, when

Examples:
after (3 months)
when (date = 1.1.2004)

Symbols in expressions must correspond to attributes in the reactive object! (date!)

Composite States

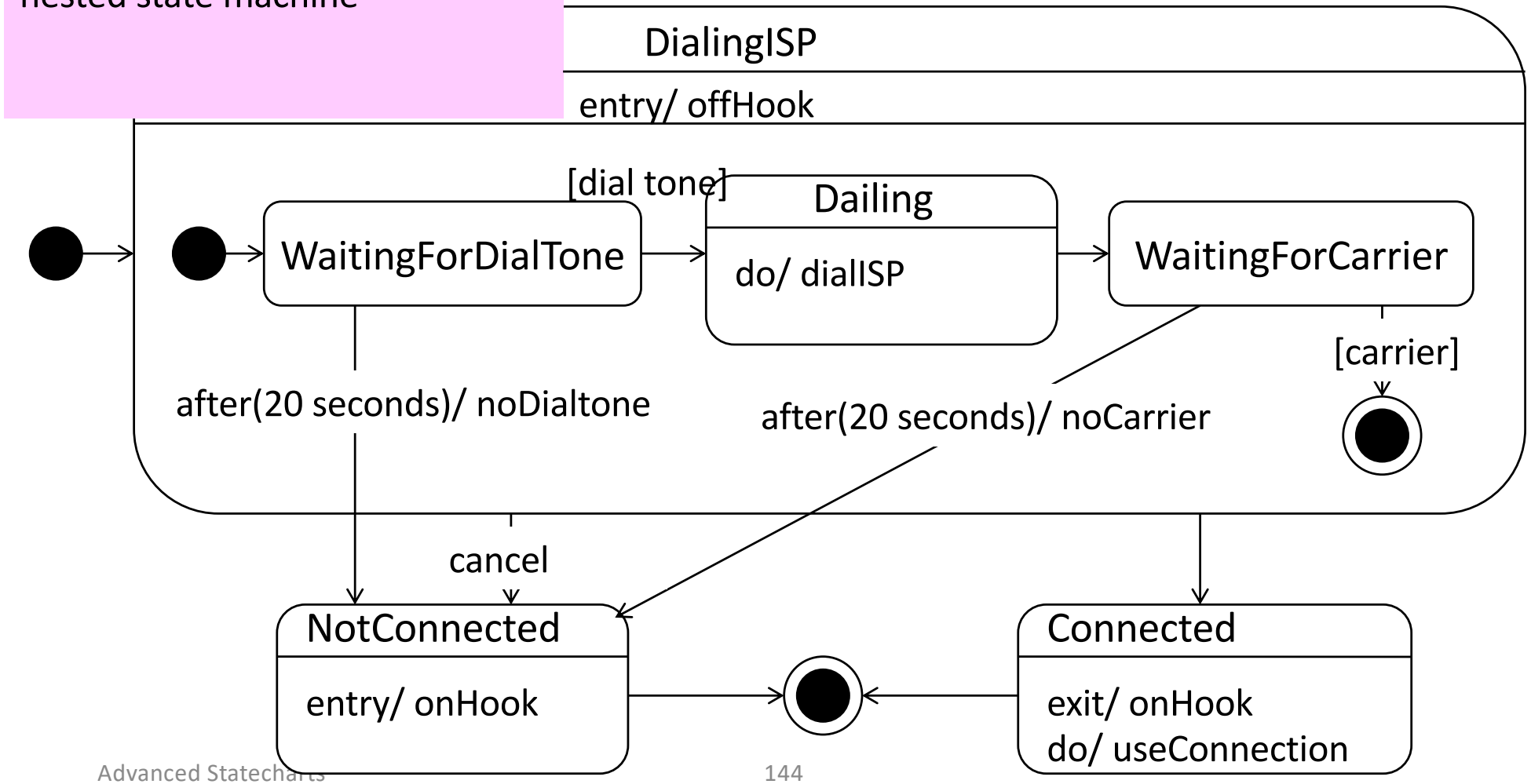
Contains one or more (nested) state machines (submachines)



Decomposition indicator not necessary if substates are explicitly represented!

Sequential Composite States

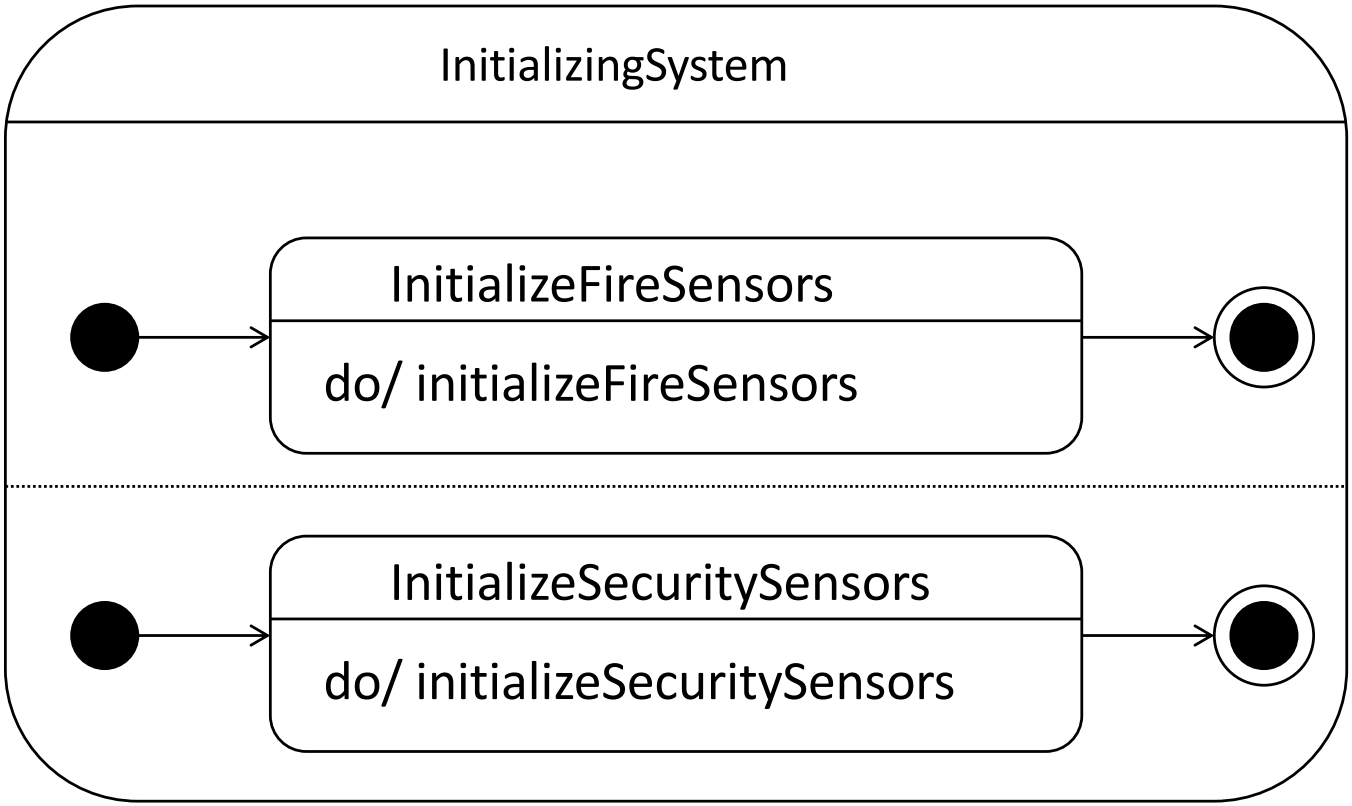
Composite state with exactly one nested state machine



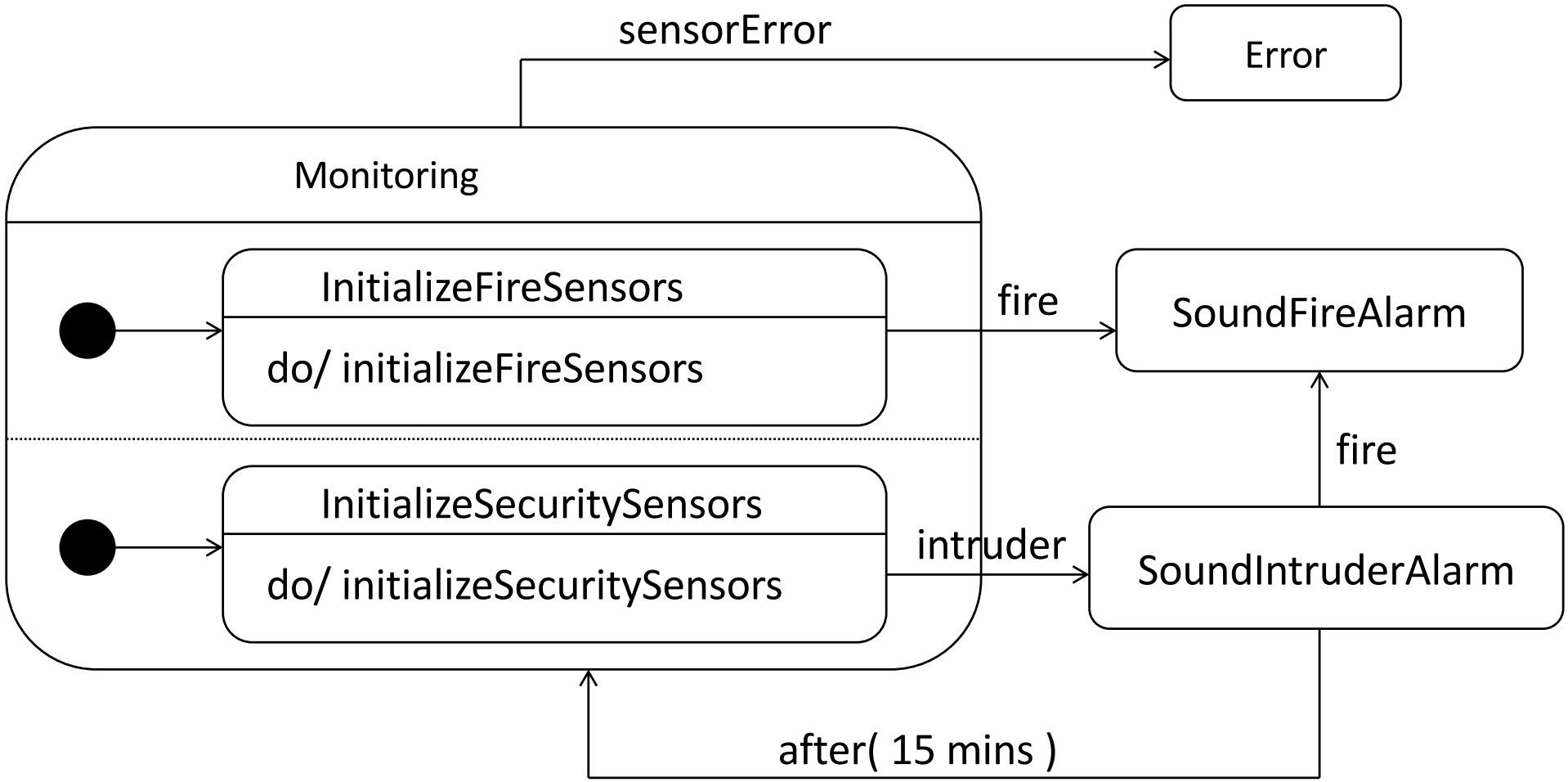
Concurrent Composite States

- 2 or more submachines that are executed in parallel (FORK).
- 2 types of termination:
  - All submachines are terminated (JOIN)
  - One submachine is terminated and goes to a state outside! No synchronization!

Example - Join



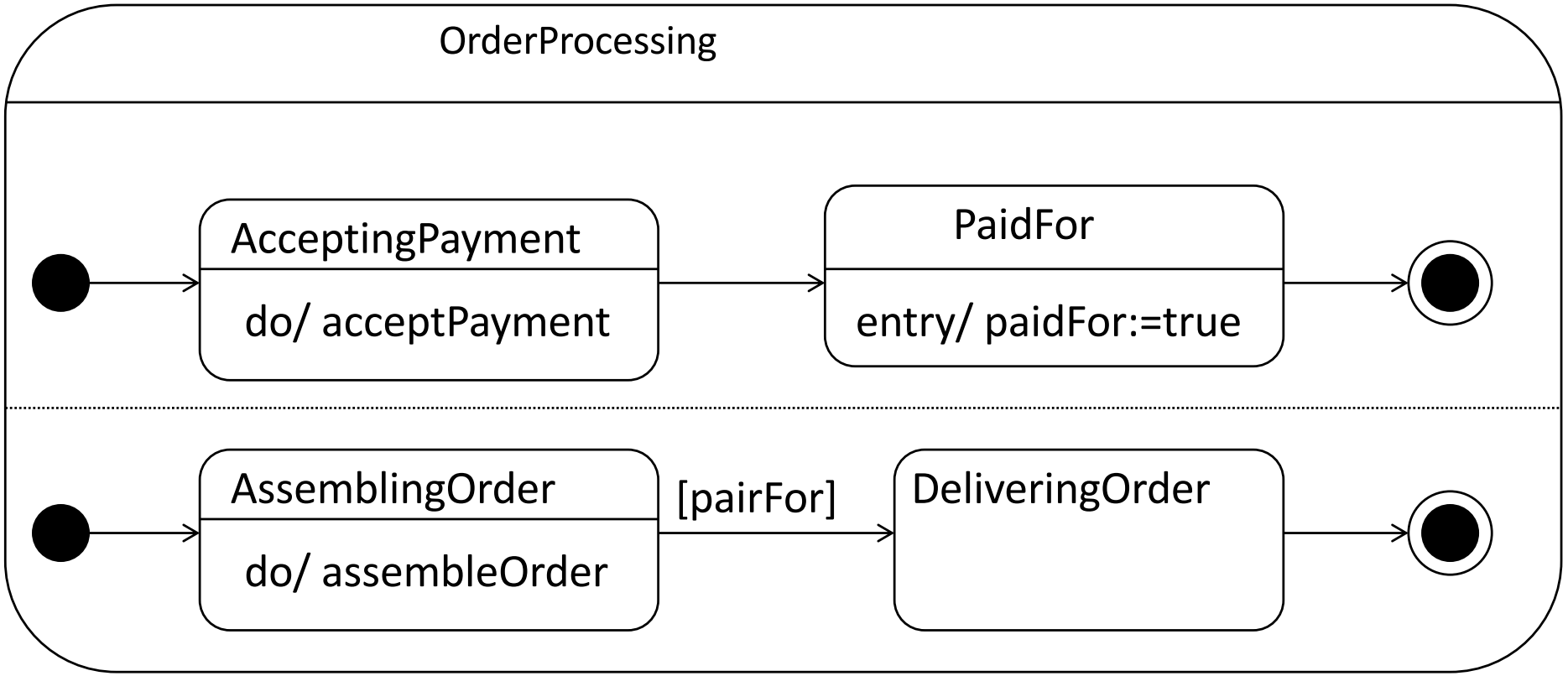
Example - Without synchronization



Asynchronous Communication

- Previously synchronous communication between submachines
- Asynchronous communication through:
  - Attributes
  - Sync states

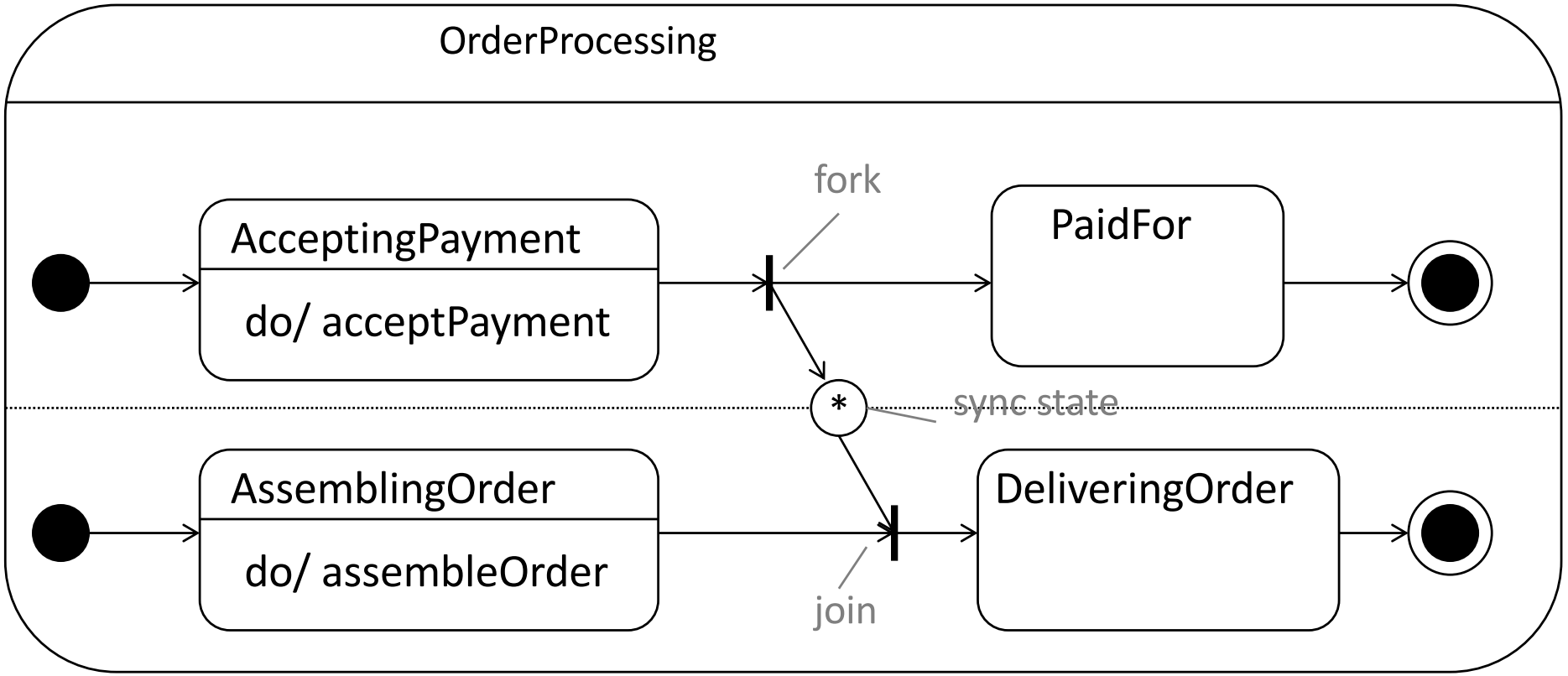
Communication - Attributes



Communication Sync States

- Sync state = state that stores messages like a queue.
- Whenever a transition to the sync state occurs, an entry is stored.
- When a transition from the sync state occurs, an entry is removed.
- The number of entries can be limited.
- No information about implementation details provided by sync states!

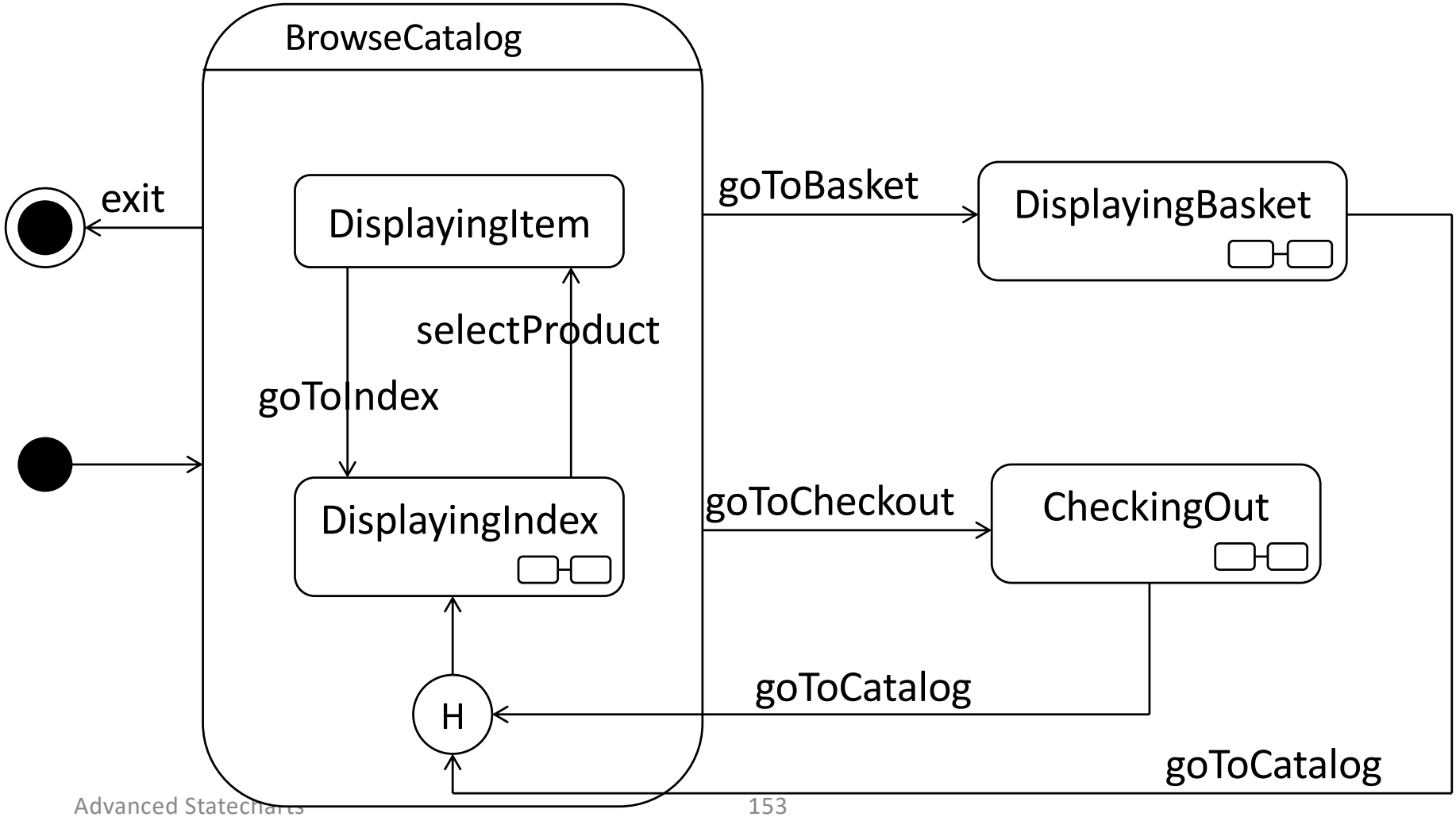
Example - Sync States



History

- Important in the following situation:
  1. You are in substate A of a composite state
  2. You leave the composite state
  3. You visit other states outside
  4. You want to return to the composite state (specifically to substate A)
- Shallow History (H): Only one level is saved.
- Deep History (H*): All levels are saved.

Example - History



OTHER BEHAVIOR DIAGRAMS

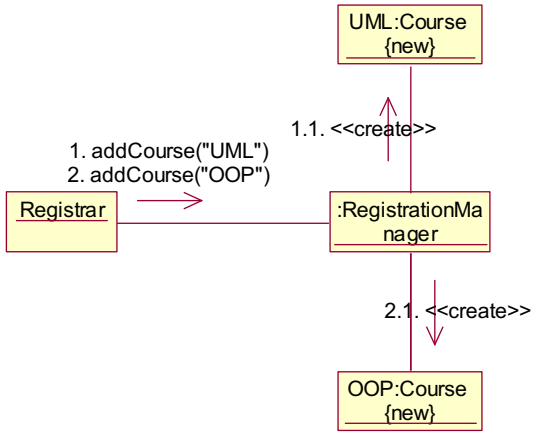
Interaction Diagrams

- Show how objects interact to realize a use case
- A diagram corresponds to one use case

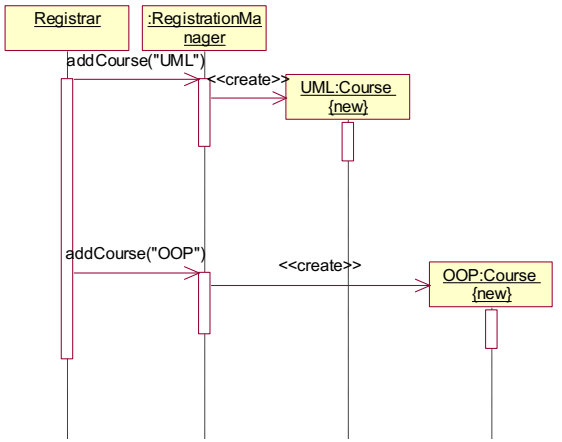
- Like CRC Cards: describe use-case realization in increasing detail
  - Can extend and document CRC cards

Communication & Sequence Diagrams

- Communication Diagrams
  - Emphasize structural relationships



- Sequence Diagrams
  - Emphasize temporal relationships

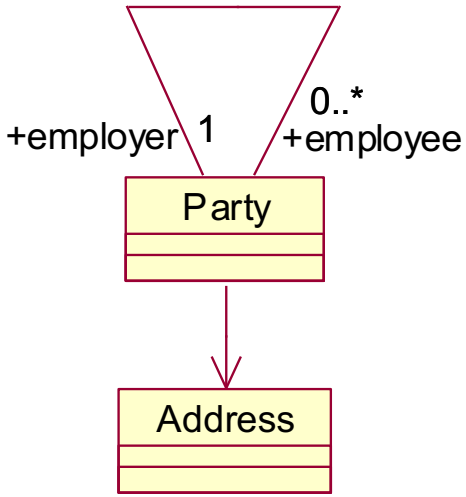


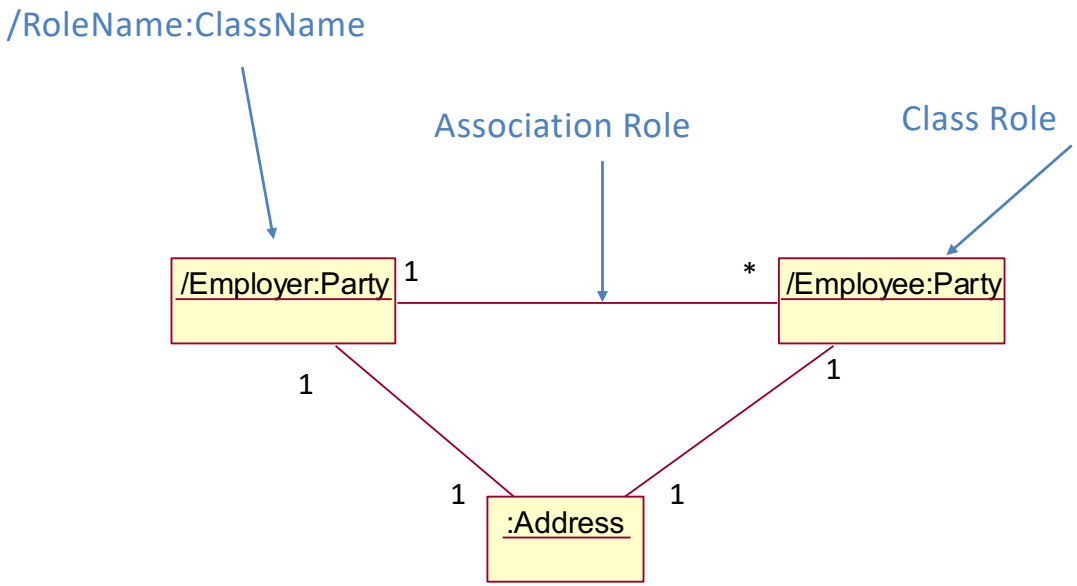
- We can convert from one to the other

Role and Instance Diagrams

- Role Diagrams:
  - Emphasize different roles of the same class
- Instance Diagrams:
  - Describe exact interaction between objects

Communication Diagrams: Roles





Communication Diagrams

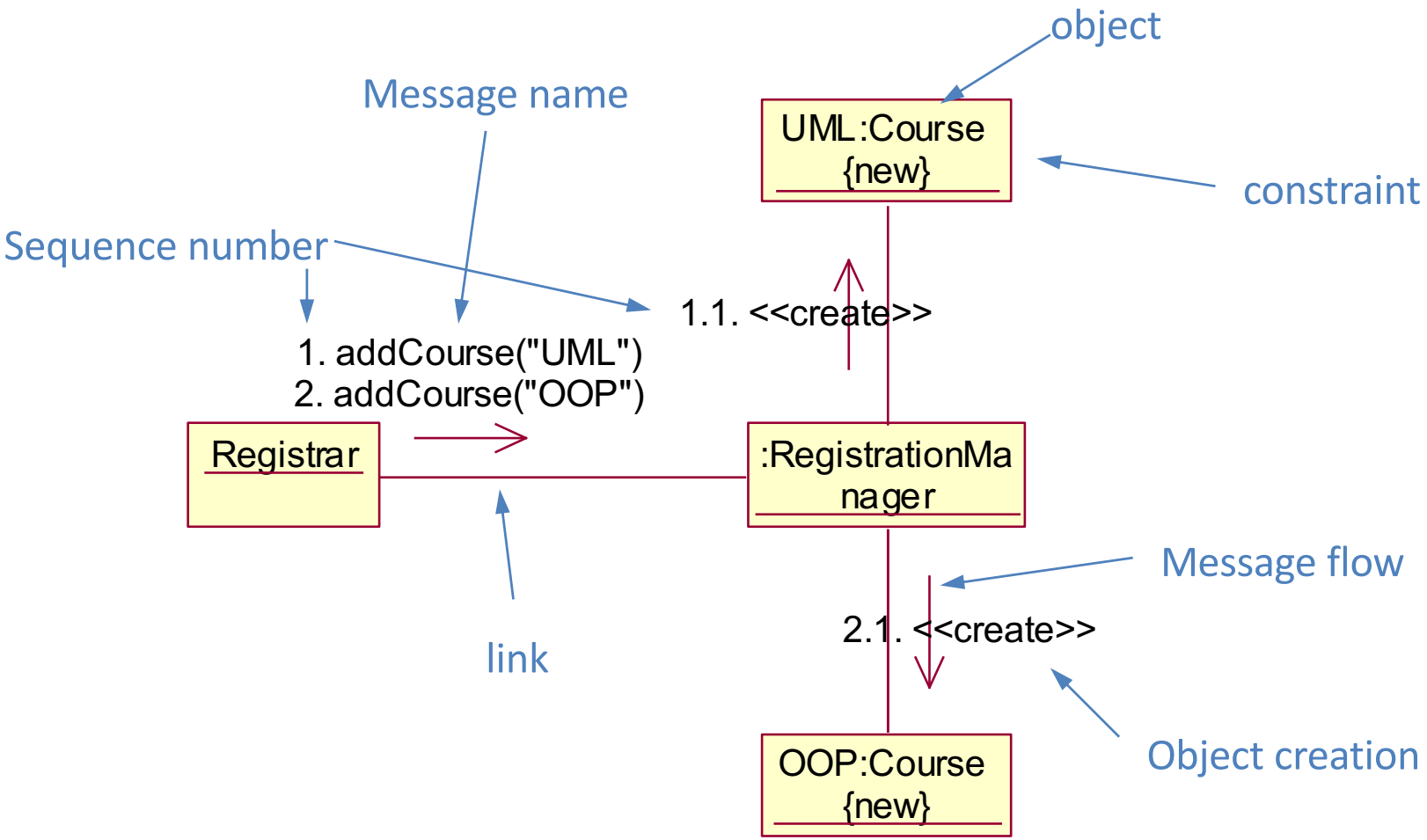
Use case: Add course

Preconditions: Registrar is logged in to system

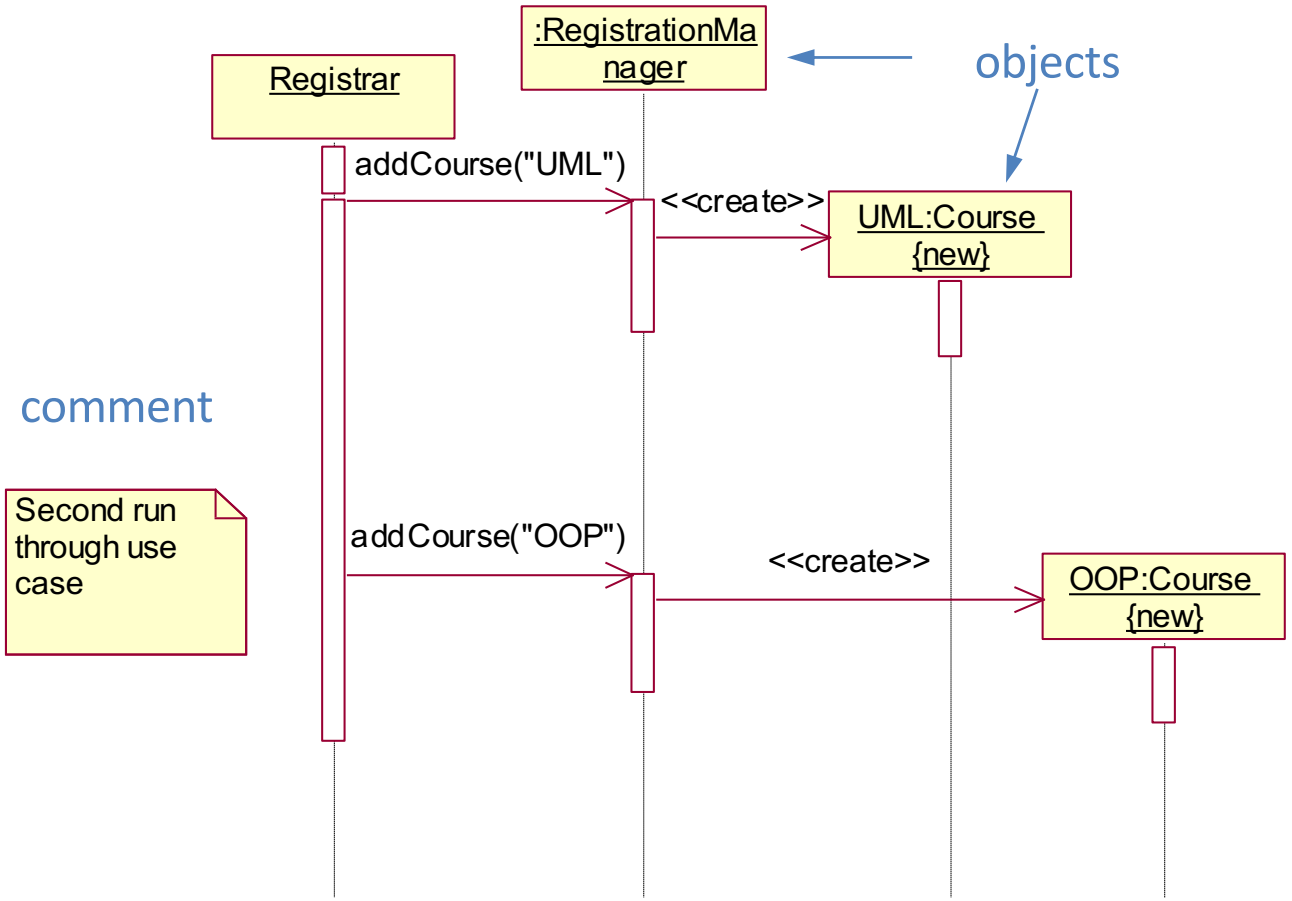
Flow of events:
1. Registrar selects "add course"
2. Registrar enters name of new course
3. System creates new course

Postcondition: A new course is present in the system

Communication Diagrams

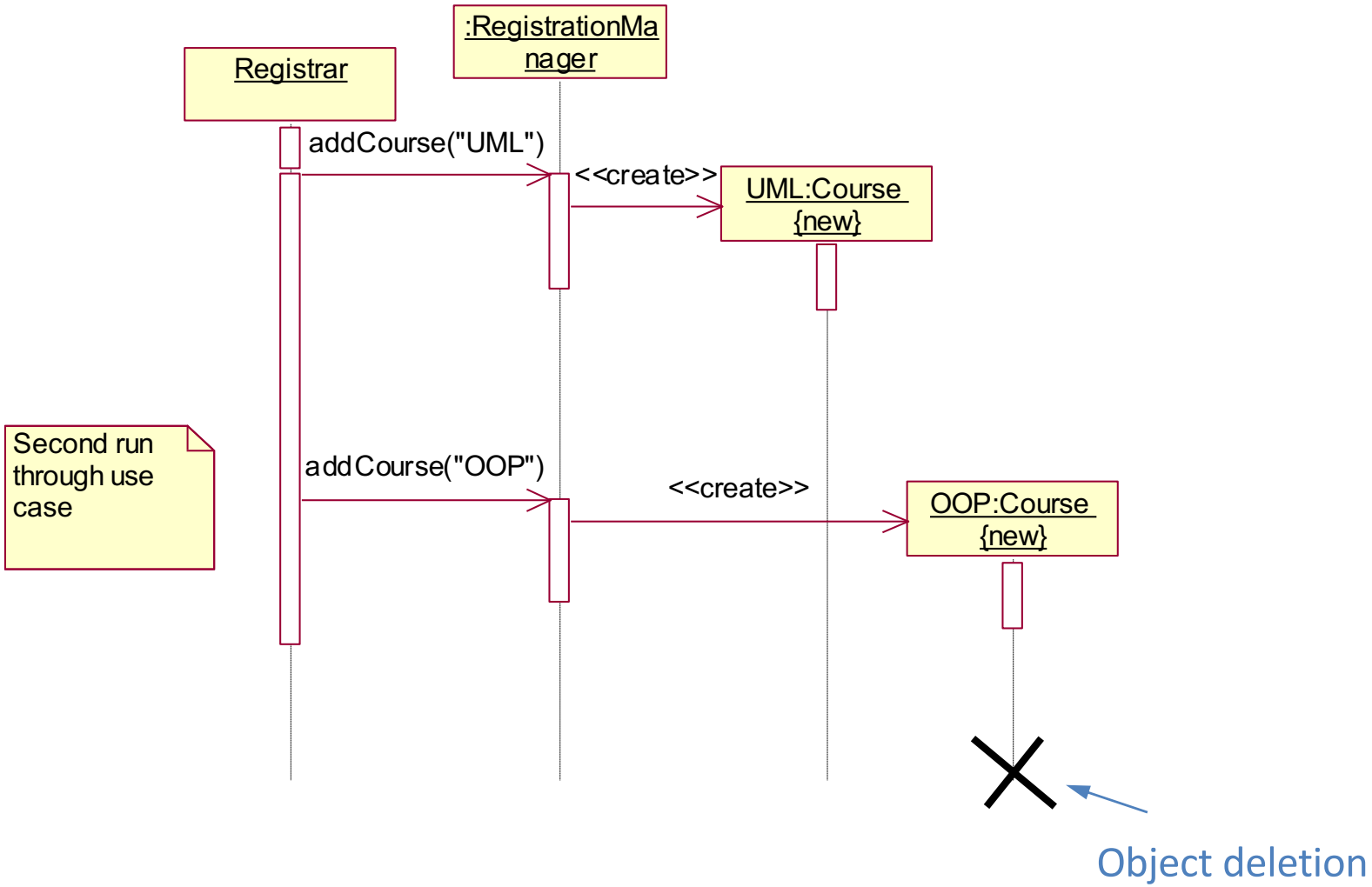


Sequence Diagrams



A different view of the same interaction!

Sequence Diagrams



Message Flow

Procedure call: caller waits for return

Asynchronous message: caller continues without waiting for reply

Return from call: only indicated when important

Unspecified

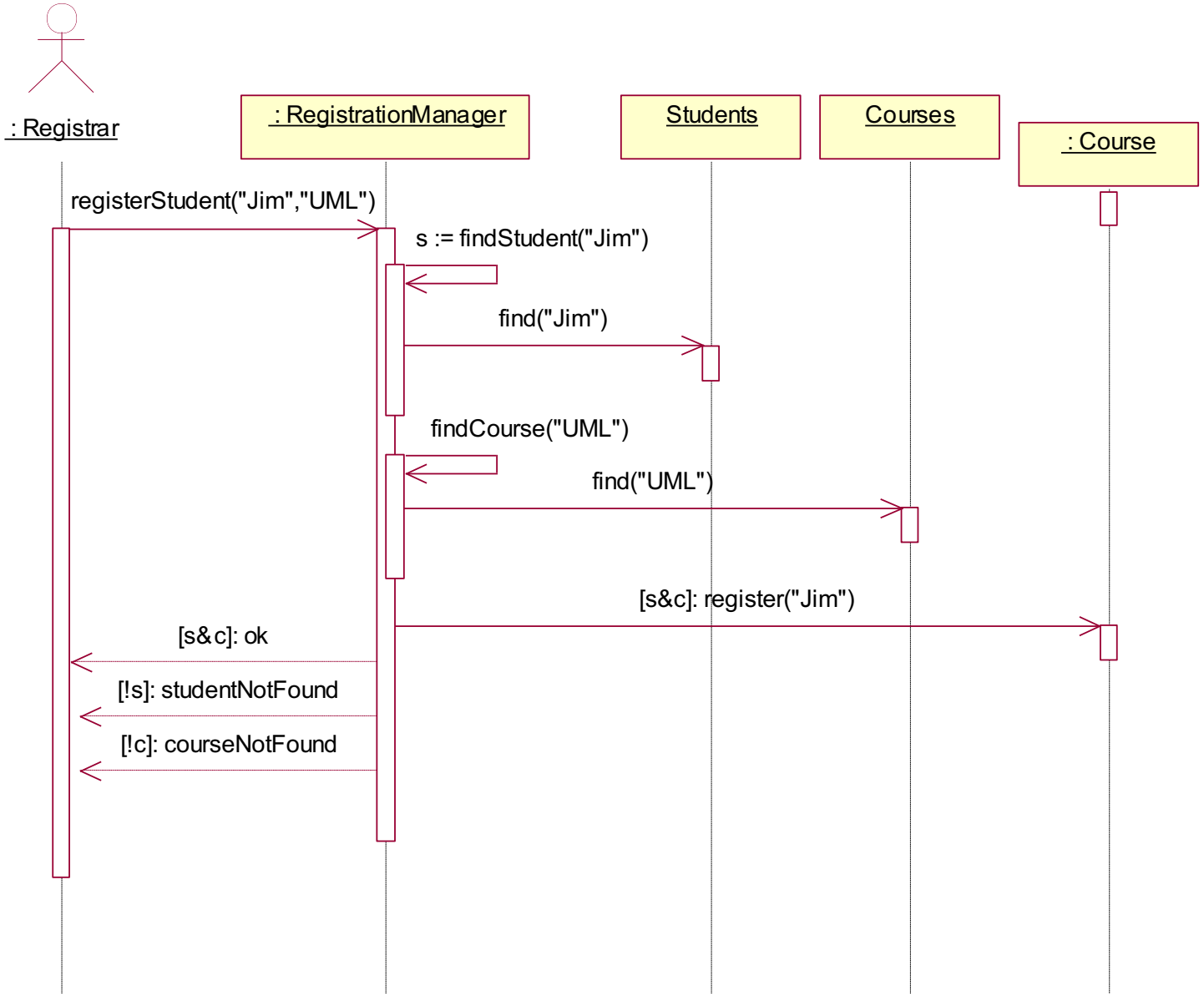
More Complicated Diagrams

Flow of events:

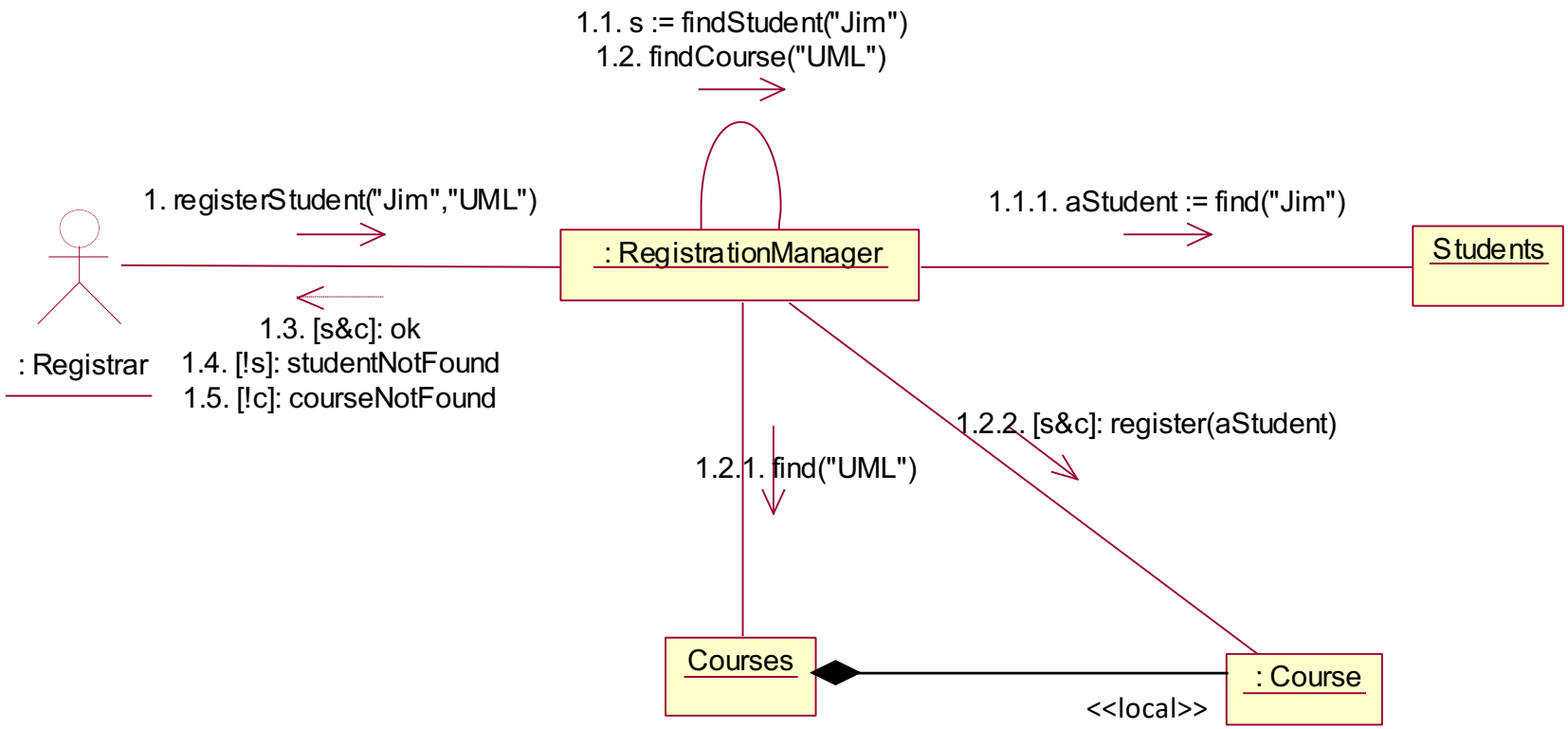
1. The registrar enters the name of the student (Jim) and the name of the course to register for (UML).
2. If the student exists and the course exists
    1. The system registers the student for the course
3. If the student does not exist
    1. The system displays an error message
4. If the course does not exist
    1. The system displays an error message

Self-calls and conditions

- Calls of own object shown as
  - Nested control flow (sequence diagrams)
  - Loop (collaboration)
- Conditions for an action shown in square brackets: [student exists]


: Registrar
: RegistrationManager
Students
Courses
: Course
registerStudent("Jim","UML")
s := findStudent("Jim")
find("Jim")
findCourse("UML")
find("UML")
[s&c]: register("Jim")
[s&c]: ok
[!s]: studentNotFound
[!c]: courseNotFound

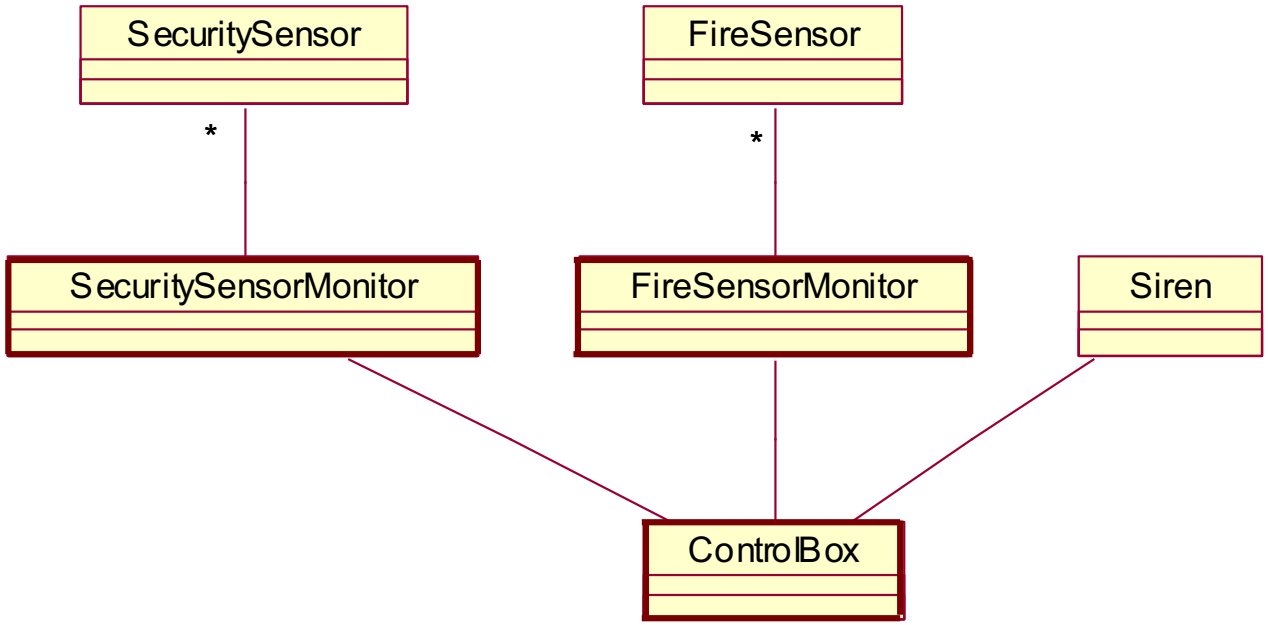
Corresponding Communication Diagram



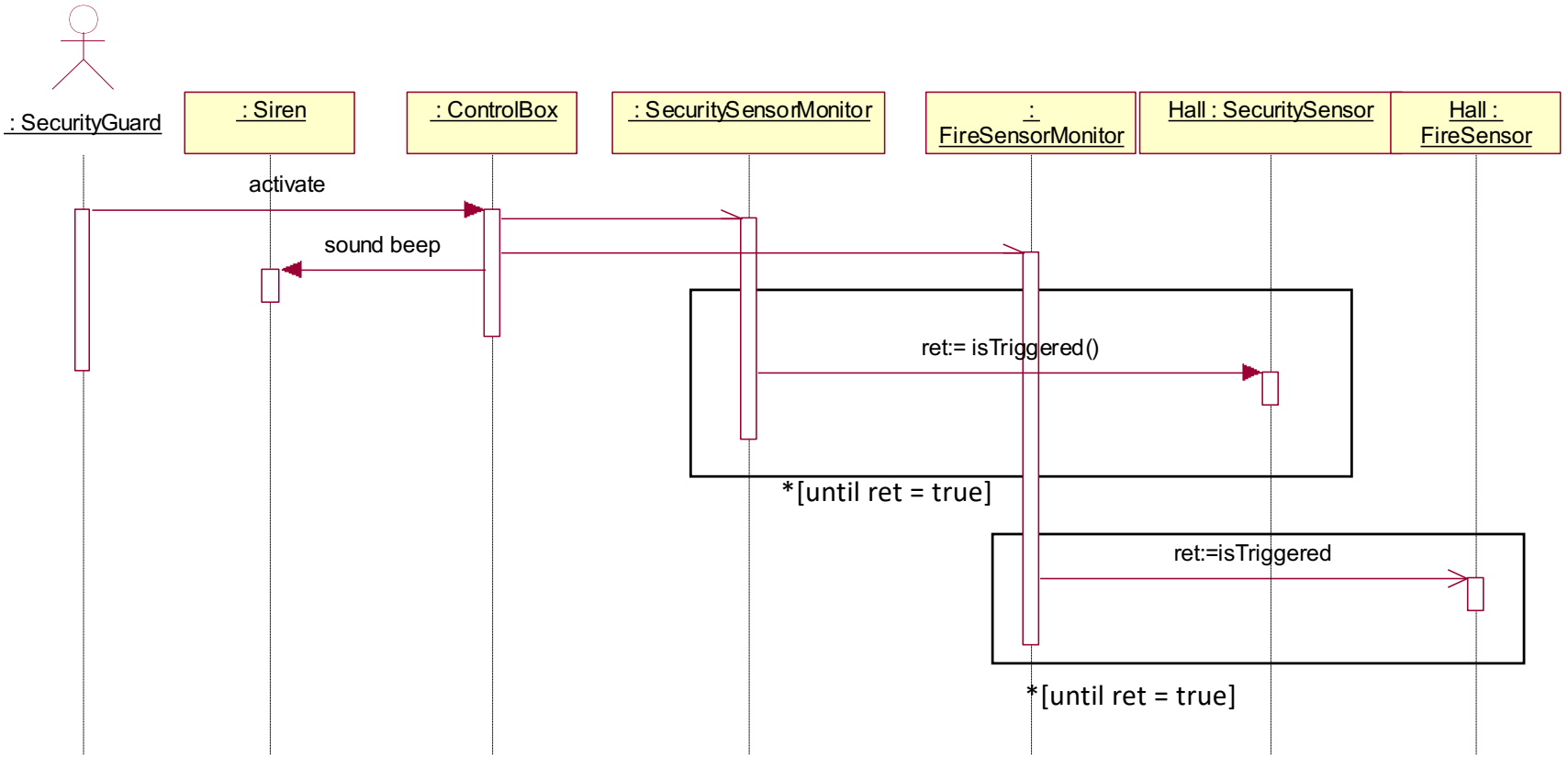
Iteration

- Use iteration expressions to repeat steps
- [i := 1…n] repeat n times
- [I := 1..7] repeat 7 times
- [while(bool)] repeat until expression is false
- [until (bool)]
- [for each (collection)]

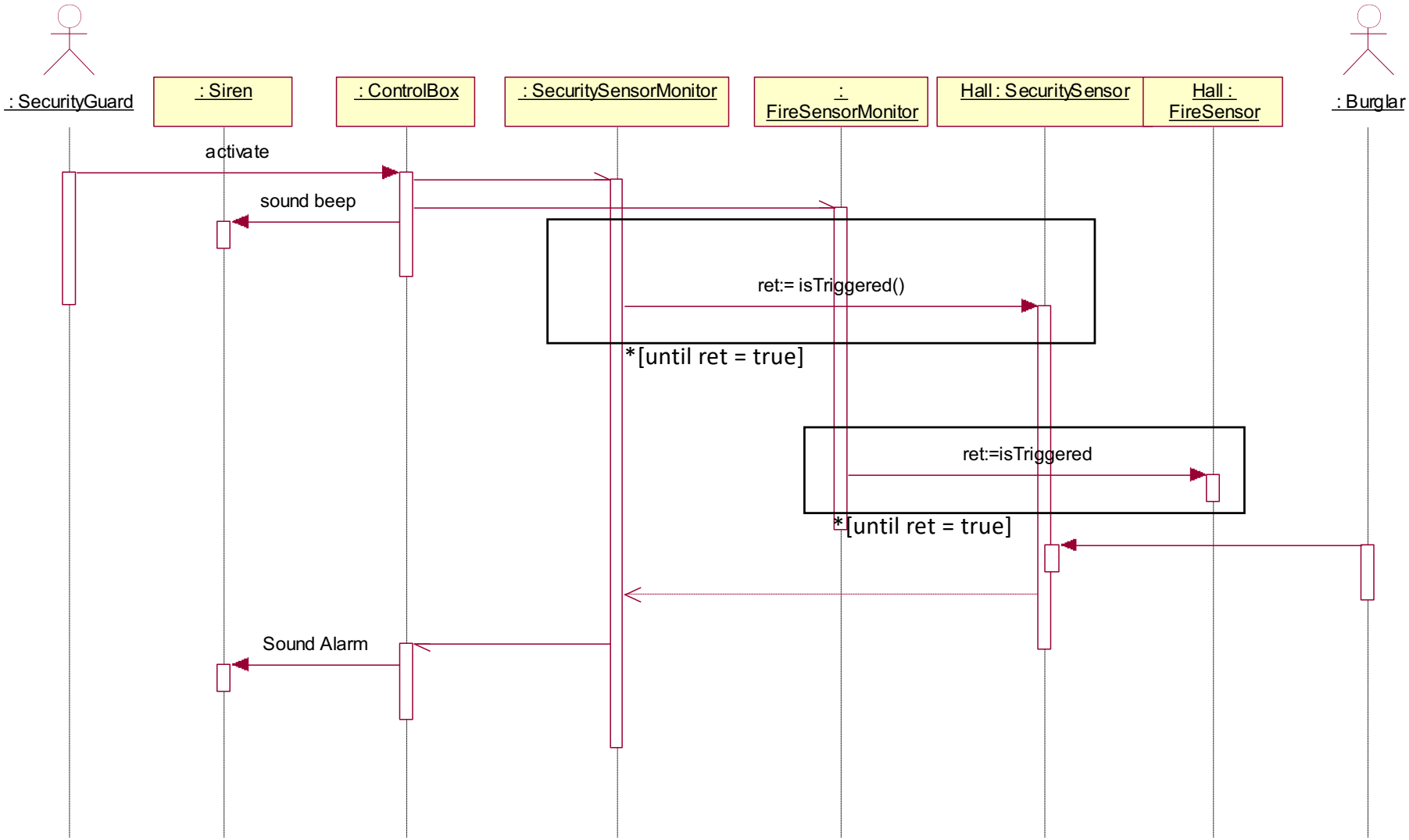
Concurrency



An Alarm System



Alarm System



Further Constructs

- Object State

Conclusions

- Communication and sequence diagrams are tools to explain how use cases are realized
- Communication diagrams emphasize relationships, sequence diagrams emphasize sequencing
- Can be used in combination with CRC cards