

# UML Diagramming and Notation



**Teknik Informatika – Universitas Komputer Indonesia**

# List of Material

- Introduction of UML
- Use Case Modelling
- Activity Diagram Modelling
- Sequence Diagram Modelling
- Class and Object Diagram Modelling
- State Diagram Modelling
- Deployment Diagram

# **Introduction of UML**

# UML History (1)

1. OO languages muncul pada pertengahan tahun 70 sampai 80.
2. Antara tahun 89 sampai 94, metode OO meningkat dari 10% sampai 50%.
3. Dicituskan oleh Three Amigos:
  - a. Grady Booch - Fusion
  - b. James Rumbaugh – Object Modeling Technique (OMT)
  - c. Ivar Jacobson – Object-oriented Software Engineering: A Use Case Approach (Objectory)
  - d. ( And David Harel - StateChart)

# UML History

Unification of ideas began in mid 90's.

Rumbaugh joins Booch at Rational '94

v0.8 draft Unified Method '95

Jacobson joins Rational '95

UML v0.9 in June '96

UML 1.0 offered to OMG in January '97

UML 1.1 offered to OMG in July '97

Maintenance through OMG RTF

UML 1.2 in June '98

UML 1.3 in fall '99

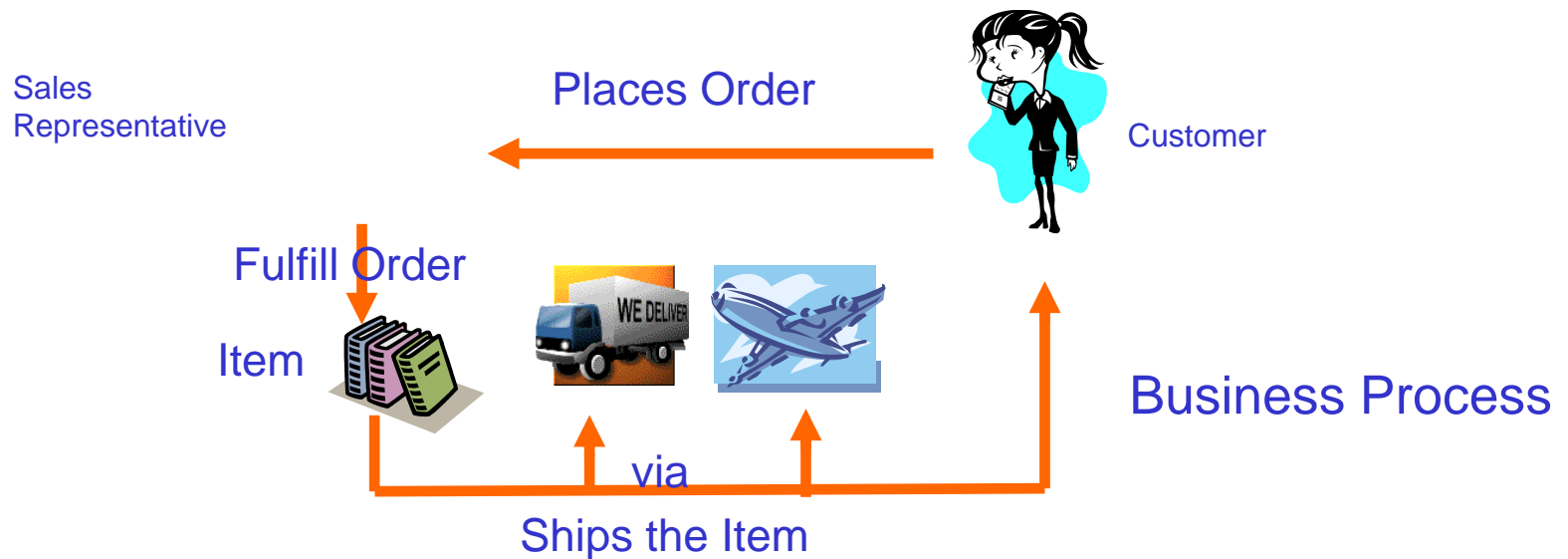
UML 1.5 <http://www.omg.org/technology/documents/formal/uml.htm>

UML 2.0 underway <http://www.uml.org/>

# UML (Unified Modelling Language)

- An effort by IBM (Rational) – OMG to standardize OOA&D notation
- Combine the best of the best from
  - Data Modeling (Entity Relationship Diagrams);
  - Business Modeling (work flow); Object Modeling
  - Component Modeling (development and reuse - middleware, COTS/GOTS/OSS/...:)
- Offers vocabulary and rules for communication
- Not a process but a language

# UML for Visual Modelling



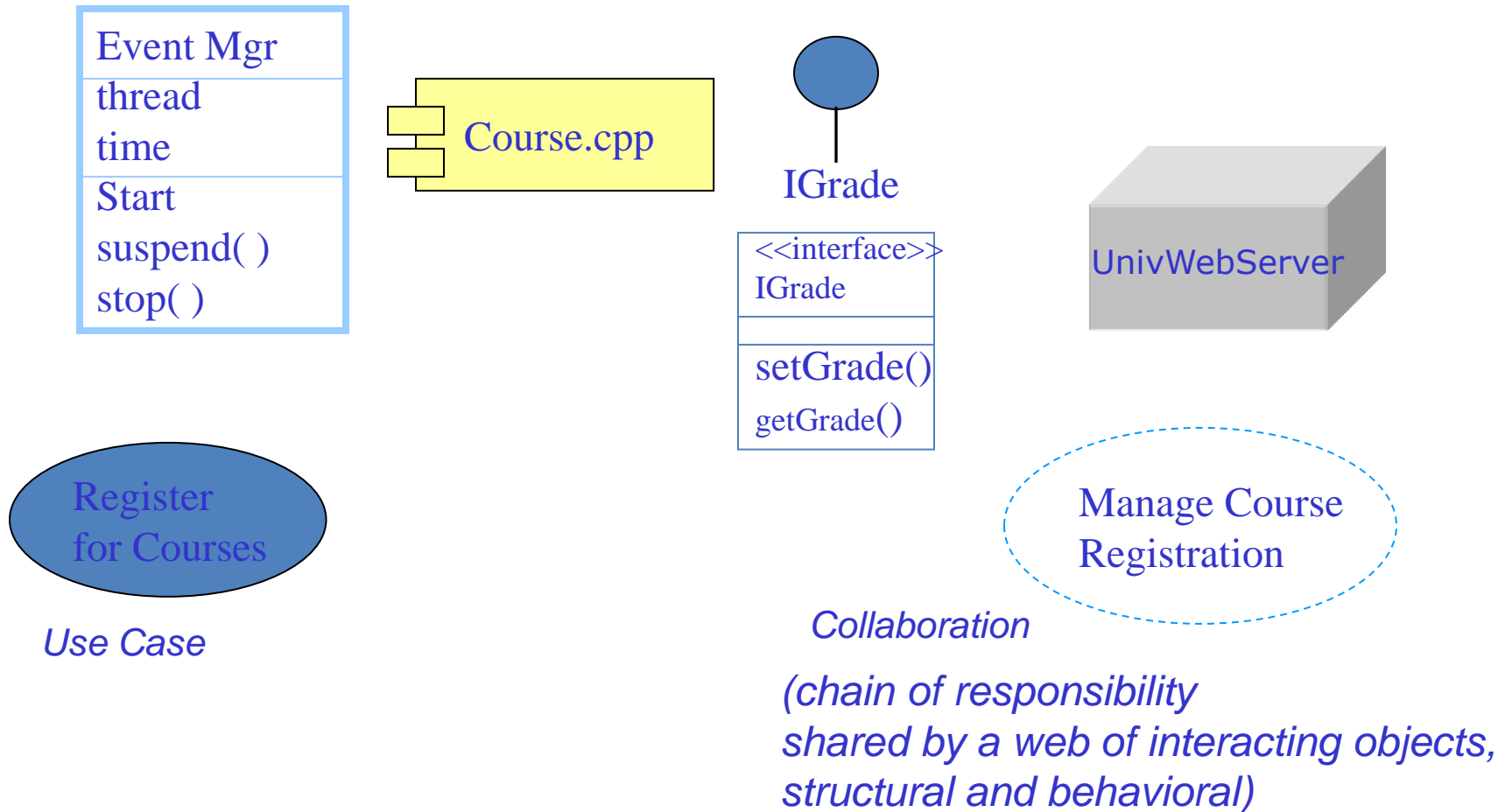
*A picture is worth a thousand words!*

# Building Blocks of UML

1. **Things** - important modelling concepts
2. **Relationships** – tying individual things
3. **Diagram** – grouping interrelated collections  
of things and relationships



# Structural Thing in UML



# Behavioral Thing in UML

- ❑ Verbs.
- ❑ Dynamic parts of UML models: “behavior over time”
- ❑ Usually connected to structural things.
- ❑ Two primary kinds of behavioral things:

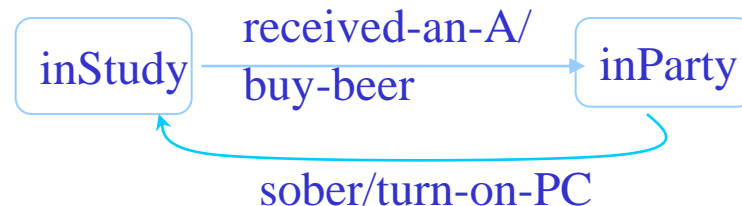
- ❑ *Interaction*

a set of objects exchanging messages, to accomplish a specific purpose.



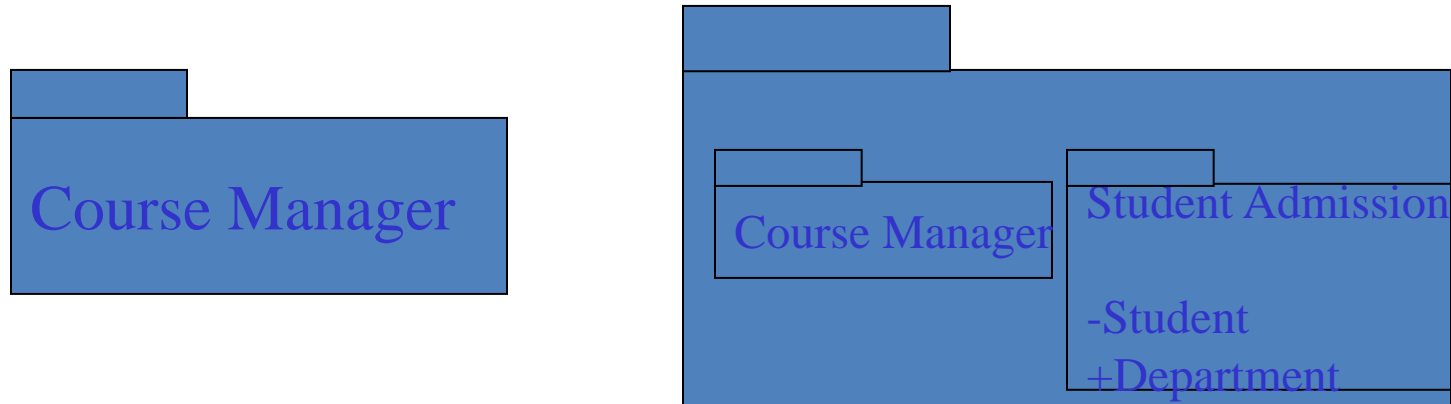
- ❑ *State Machine*

specifies the sequence of states an object or an interaction goes through during its lifetime in response to events.



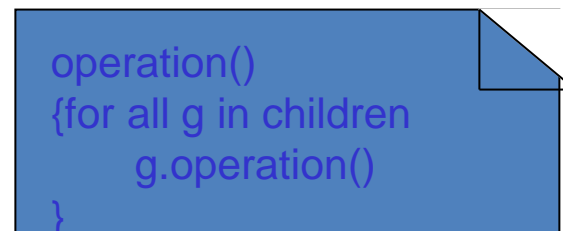
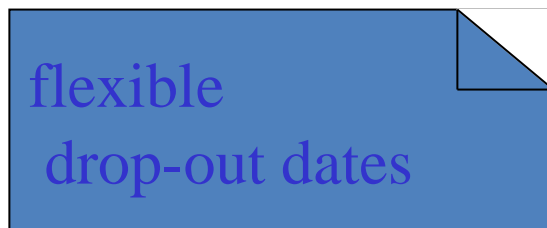
# Group Thing in UML

- For organizing elements (structural/behavioral) into groups.
- Purely conceptual; only exists at development time.
- Can be nested.
- Variations of packages are: Frameworks, models, & subsystems.



## Annotational Things in UML: *Note*

- Explanatory/Comment parts of UML models - usually called adornments
- Expressed in informal or formal text.



# Relationship in UML

## 1. Associations

Structural relationship that describes a set of links, a link being a connection between objects. *variants: aggregation & composition*

## 2. Generalization

a specialized element (the child) is more specific the generalized element.

## 3. Realization

one element guarantees to carry out what is expected by the other element.  
(e.g, interfaces and classes/components; use cases and collaborations)

## 4. Dependency

a change to one thing (independent) may affect the semantics of the other thing (dependent).

# UML 1.x VS UML 2.0

## UML 1.x: 9 diagram types.

### Structural Diagrams

Represent the *static* aspects of a system.

- ☐ Class;  
Object
- ☐ Component
- ☐ Deployment

### Behavioral Diagrams

Represent the *dynamic* aspects.

- ☐ Use case
- ☐ Sequence;  
Collaboration
- ☐ Statechart
- ☐ Activity

## UML 2.0: 12 diagram types

### Structural Diagrams

- ☐ Class;  
Object
- ☐ Component
- ☐ Deployment
- ☐ Composite Structure
- ☐ Package

### Behavioral Diagrams

- ☐ Use case
- ☐ Statechart
- ☐ Activity

### Interaction Diagrams

- ☐ Sequence;  
*Communication*
- ☐ Interaction  
Overview
- ☐ Timing

# UML Diagram

- Structural diagrams
  - Used to describe the relation between classes
- Behavior diagrams
  - Used to describe the interaction between people (actors) and a use case (how the actors use the system)

# Structural Diagram

1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagrams

# Behavioral Diagram

1. Use case diagrams
2. Sequence diagrams
3. Collaboration diagrams
4. Statechart diagrams
5. Activity diagrams



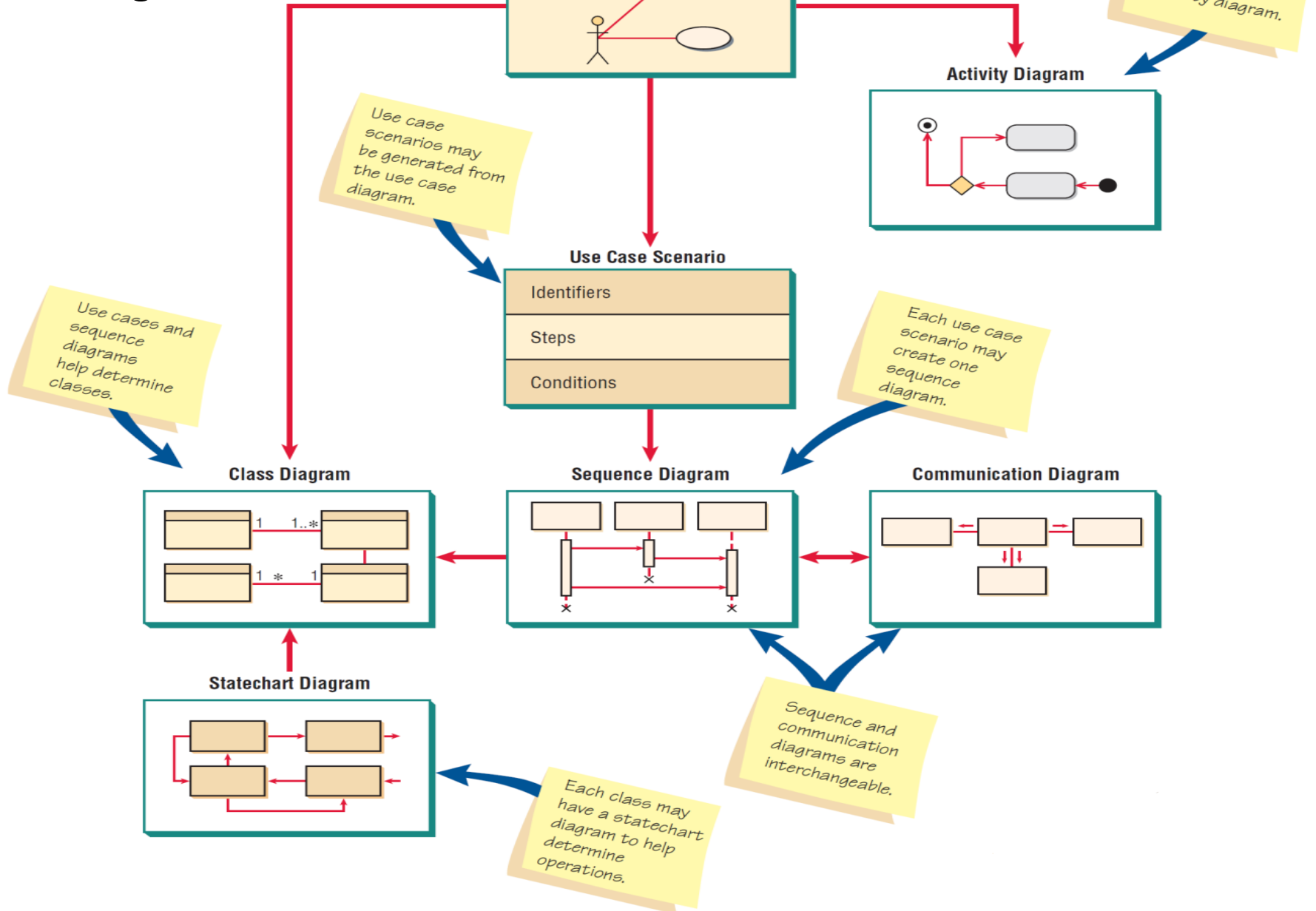
# Commonly Used Diagram (1)

- Use case diagram
  - Describing how the system is used
  - The starting point for UML modeling
- Use case scenario
  - A verbal articulation of exceptions to the main behavior described by the primary use case
- Activity diagram
  - Illustrates the overall flow of activities

# Commonly Used Diagram (2)

- Sequence diagrams
  - Show the sequence of activities and class relationships
- Class diagrams
  - Show classes and relationships
- Statechart diagrams
  - Show the state transitions

# Overview of UML Diagram

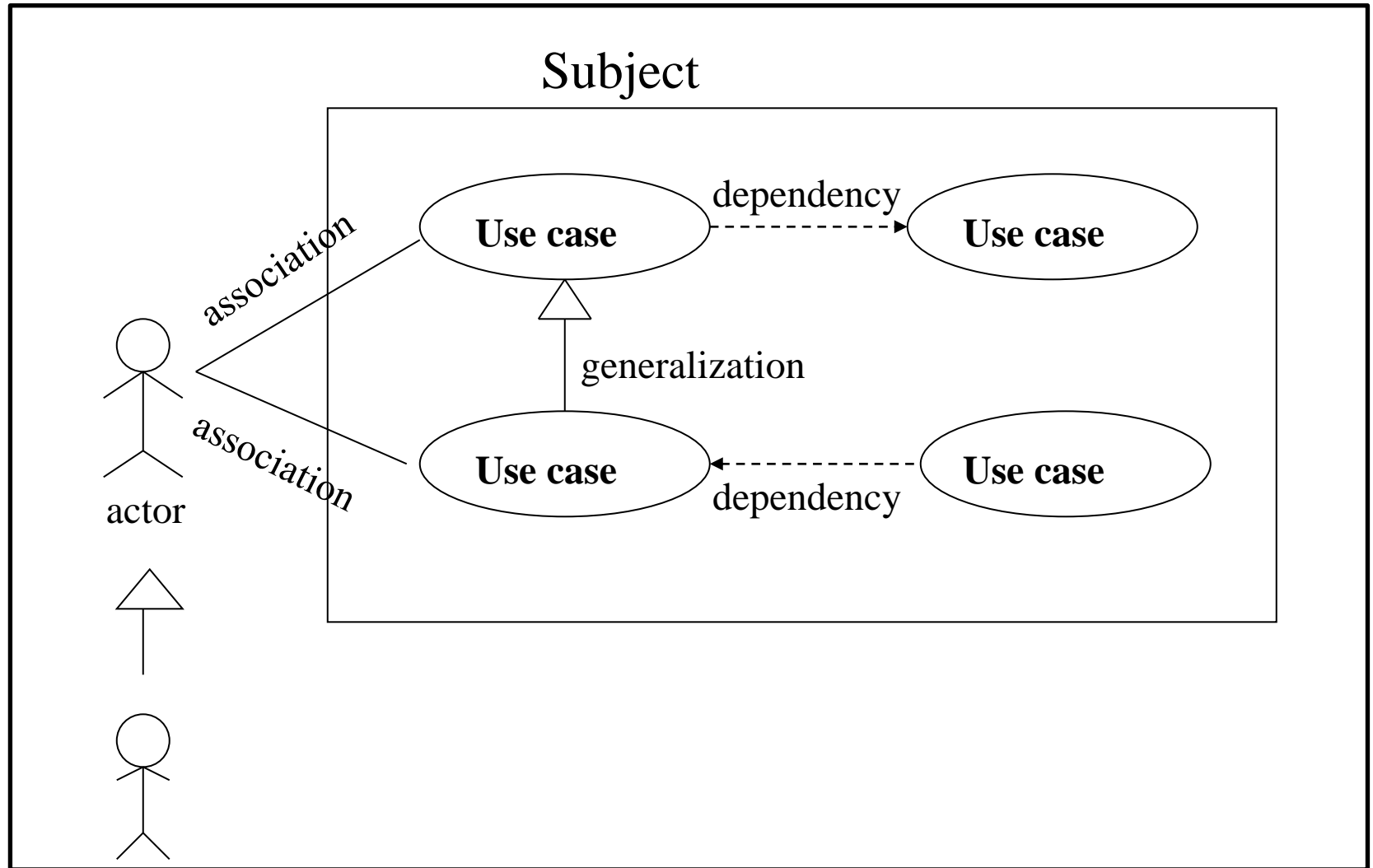


# **Use Case Modelling**


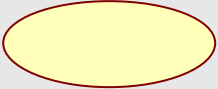

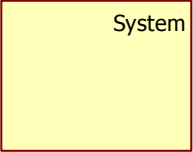
# Use Case Model

- Describes what a user expects the system to do
  - functional requirements
- May describe only the functionalities that are visible to the user
  - requirements view
- May include additional functionalities to elaborate those in the previous step
  - design view
- Consists of use case diagrams and textual descriptions

# Element of Use Case Diagram



# Use Case Symbol

SYMBOL	NAME OF SYMBOL	EXPLANATION
	Actor	Accessing use case
	Use Case	Show what the system do
	Association	Relate the actor with use case
	System Boundary	Show boundary between system and its environment

# Use Case Definition (1)

- Subject

- A black box that describes the system or subsystem that is modeled
- Example: ATM system, login subsystem
- Represented **optionally** as a rectangle in the use case diagram, but generally not shown

- Actor

- A role played by an external entity that interacts with the subject
- One object may play multiple roles in a context in which case there will be multiple actors

example: bank manager playing the role of a teller or that of a customer



# Use Case Definition (2)

- Primary actor
  - An actor who initiates the major, main or important use cases in the system
  - Example : a customer in a banking system
- Secondary actor
  - An actor who is involved with one or more use cases but does not initiate any use case
  - Example : database
- There is no syntactic difference between a primary actor and a secondary actor

# Use Case Definition (3)

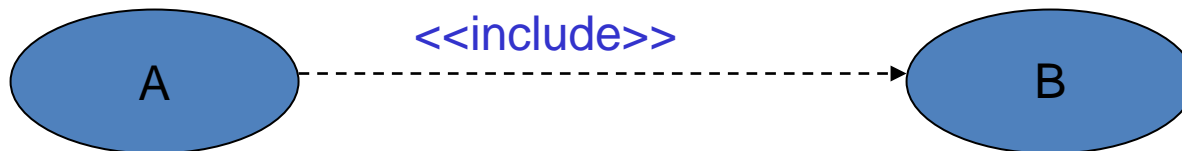
- Generalization between actors
  - One actor can be a specialization of another actor
  - Based on the same concept as the specialization relationship between classes
  - Example : preferred customer in a bank is a specialization of a customer
- Use case
  - An important functionality to be implemented and is visible to the actors
  - An interacting behavior between an actor and the subject
    - Must yield an observable result to the actor
  - Example: “deposit” in a banking system

# Use Case Definition (4)

- Association
  - An interaction between **an actor and a use case**
  - **Unidirectional** associations must be represented by arrows
    - Direction of arrow indicates information flow
  - **Bi-directional** associations can be represented by double-sided arrows or straight lines

# Use Case Definition (5)

- “include” dependency
  - One use case **may include** another use case
  - If use case A includes use case B, B **must be implemented** in order to implement A
  - Represented as a dashed arrow from A to B with a label “<<include>>”
  - Example : use case “withdraw” includes use case “update account”



# Use Case Definition (6)

- “extend” dependency
  - One use case may be extended by another use case
  - If use case A is extended by use case B, then both A and B can be independently implemented and used
    - A will occasionally use B depending on some constraints

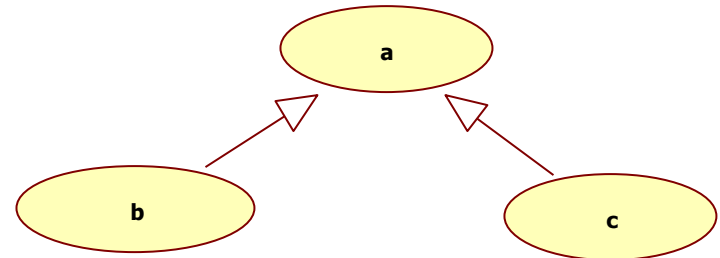
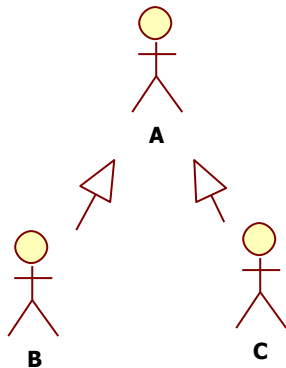
# Use Case Definition (6)

- “extend” dependency (continued)
  - Represented as a dashed arrow from B to A with a label “<<extend>>”
    - Notice that the arrow is reversed
  - Example :
    - Use case “withdraw” is extended by use case “compute penalty” when the user withdraws an amount more than the balance in the account; the use case “compute penalty” is therefore occasionally used by “withdraw”.

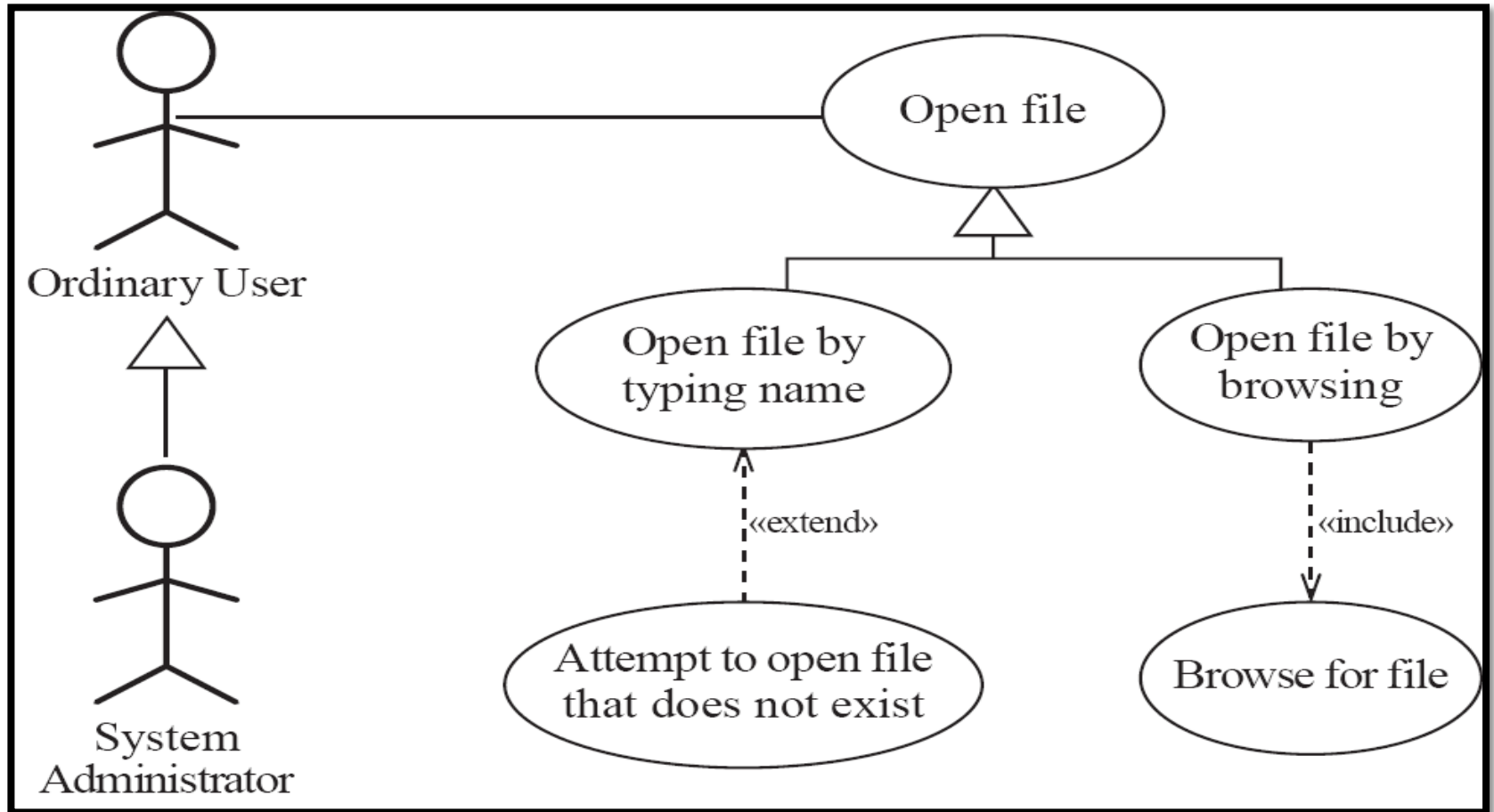


# Use Case Definition (7)

- “**generalization**” dependency
  - Aktor dan use case bisa digeneralisasi
  - Generalisasi digunakan untuk membuat aktor atau use case yang lebih spesifik dari suatu aktor dan use case.



# Example of Generalization, Extension, and Inclusion





# Constraints in Use Case Model

- Every use case must be connected to an actor or be included in another use case or extends another use case
- Every use case connected to an actor must return an observable result to the actor
  - The result may be data, confirmation or termination of an action

# How To Find Use Case?

- Every requirement is a use case
- Every functionality that supports the implementation of a requirement is a use case
  - Design issue
  - Found when the first (abstract) use case model is refined to express a design
- Do not confuse a “use case” with a “method” in implementation
  - Generally, there is a one-to-many relationship between a use case and a method

# How To Find Use Case Relationship?

- Extracted from the application domain
- Must be justifiable from the application domain or from the designer's choice
- Examples
  - Use case “withdraw” includes use case “update account” is justifiable from application domain
  - Use case “withdraw” is extended by “compute penalty” is a designer's choice
    - Designer can decide to implement two different versions of withdrawals or just only one with no extension

# Use Case Narrative/Scenario

- Important for **design** and **implementation** of use cases
- **Different types**
  - Textual (informal) descriptions
  - Algorithmic descriptions
  - Diagrammatic descriptions (activity diagram)

# Describe An Use Case

- A. **Name**: Give a short, descriptive name to the use case
- B. **Actors**: List the actors who can perform this use case
- C. **Goals**: Explain what the actor or actors are trying to achieve
- D. **Preconditions**: State of the system before the use case
- E. **Summary**: Give a short informal description
- F. **Related use cases**
- G. **Steps**: Describe each step using a 2-column format
- H. **Postconditions**: State of the system in following completion

**A and G are the most important**

# Example of Use Case Narrative/Scenario

<b>Use case name</b>	<b>Accessing the HSVO tools and devices</b>	
<b>Related requirements</b>	Email with information on how to log into the system, web address, username and password	
<b>Goal in Context</b>	Log into the HSVO system, and access the "Tools and Devices" section the system	
<b>Preconditions</b>	The user has the appropriate web address, username and password	
<b>Successful End Condition</b>	The user successfully logs in and is re-directed to the "Dashboard"	
<b>Fail end condition</b>	User is not able to log in	
<b>Primary Actors</b>	Student (or Tutor)	
<b>Secondary Actors</b>	None	
<b>Trigger</b>	The user's credentials are provided to the system for verification	
<b>Main Flow</b>	<b>Step</b>	<b>Action</b>
	1	User goes to the HSVO login page
	2	User enters details to login to the system: username, password, and selects current location
	3	The credentials are verified by the system
	4	The details are returned as verified and the user is redirected to personal dashboard. If there are any scheduled sessions coming up they are listed first; second, a list of recent activities or actions within the account is presented, and third, a list of news and/or recommended tools is provided.
<b>Extensions</b>	<b>Step</b>	<b>Branching action: Non verified credentials</b>

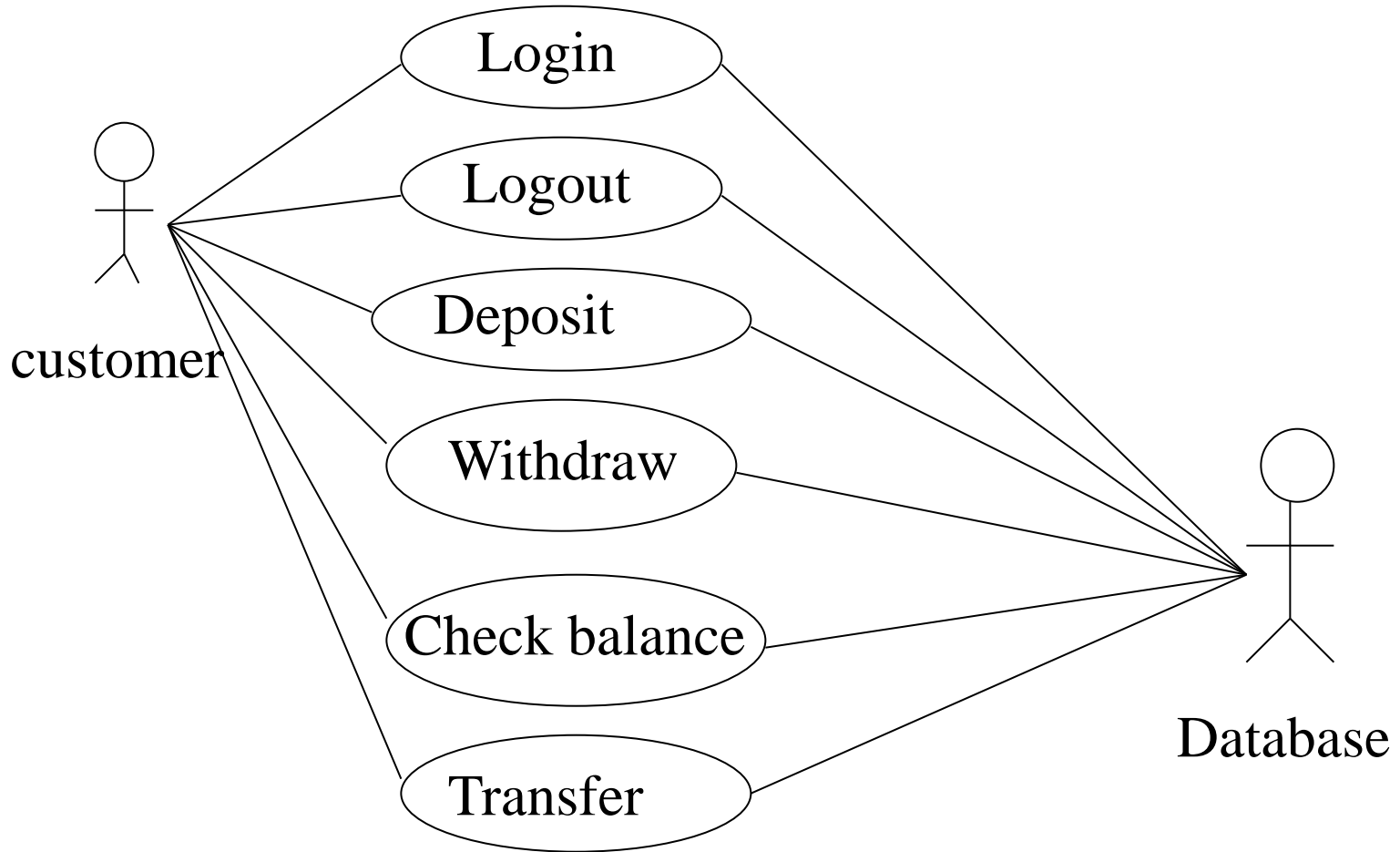
# **Case Study for Use Case Modelling**

# Case Study - ATM

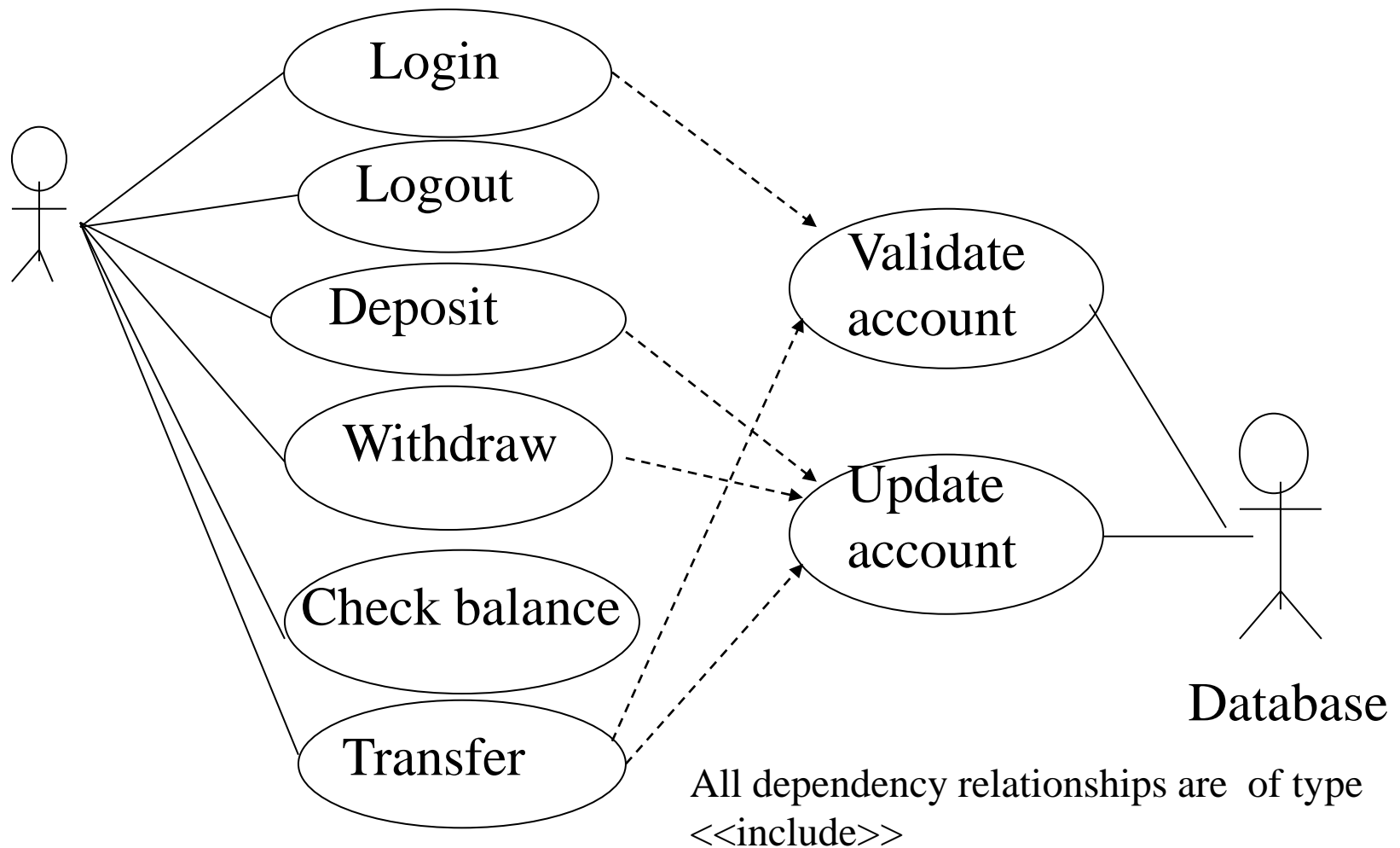
- Model only the transactions
- Customer accounts assumed to exist
  - Opening and closing of accounts are handled by another portion of the system
- Include operations “deposit”, “withdraw”, “check balance”, “transfer”
- If balance is zero or less than the amount to be withdrawn, then withdrawal should fail



# Case Study - ATM



# Case Study – ATM (Revised)



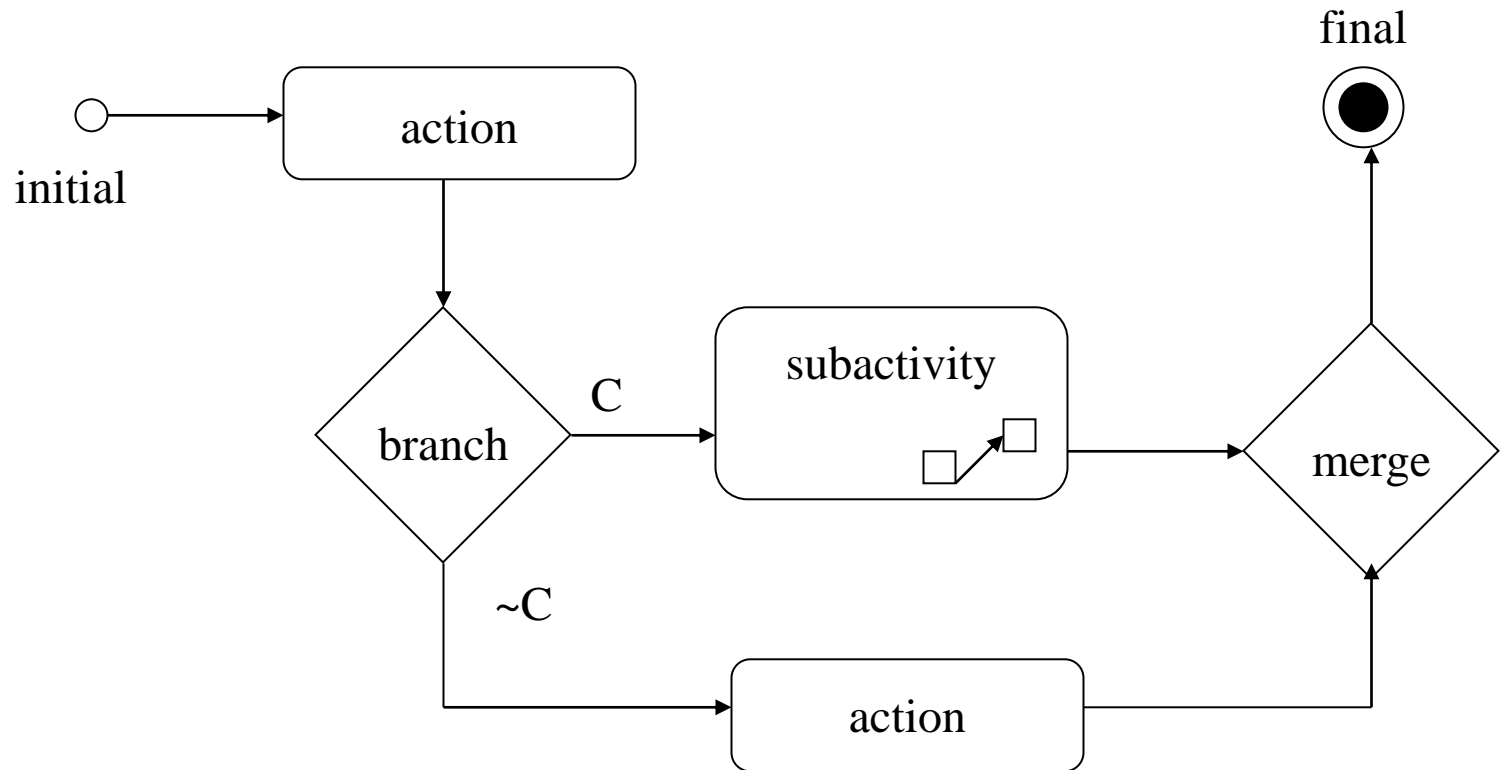
**Use case diagram for ATM – revised to show design issues**

# **Activity Diagram Modelling**

# Activity Diagram

- Represents a sequence of activities
- An **activity** is a group of atomic actions
- An **action** is indivisible (atomic) task
  - Example: change the value of a variable/field
- An activity may consist of sub-activities or actions or both
- In general, activities may be decomposable but actions are not.
  - Activities can be interrupted by events, but actions are not.

# Activity Diagram



C – Boolean expression

# Semantic of Activity Diagram (1)

- An activity diagram consists of a collection of action states, sub-activities and transitions
- Every activity diagram must have only one initial state and one or more final states
  - The initial state represents the beginning of the activity and a final state represents the termination of an activity
- Actions are represented by action states (rounded rectangles)
- A sub-activity is also represented by a rounded rectangle but with an icon inside the rectangle

# Semantic of Activity Diagram (2)

- When expanded, each sub-activity is **diagrammatically** substituted with the **incoming** and **outgoing** transitions matched
- Sequence of actions is represented by **transitions between actions**
  - Transitions are simple straight arrows with no labels or parameters
  - Transitions may have guards/conditions, send-clause and actions (very similar to those in state transition diagrams)
  - Transitions are augmented with conditions at branching

# Semantic of Activity Diagram (3)

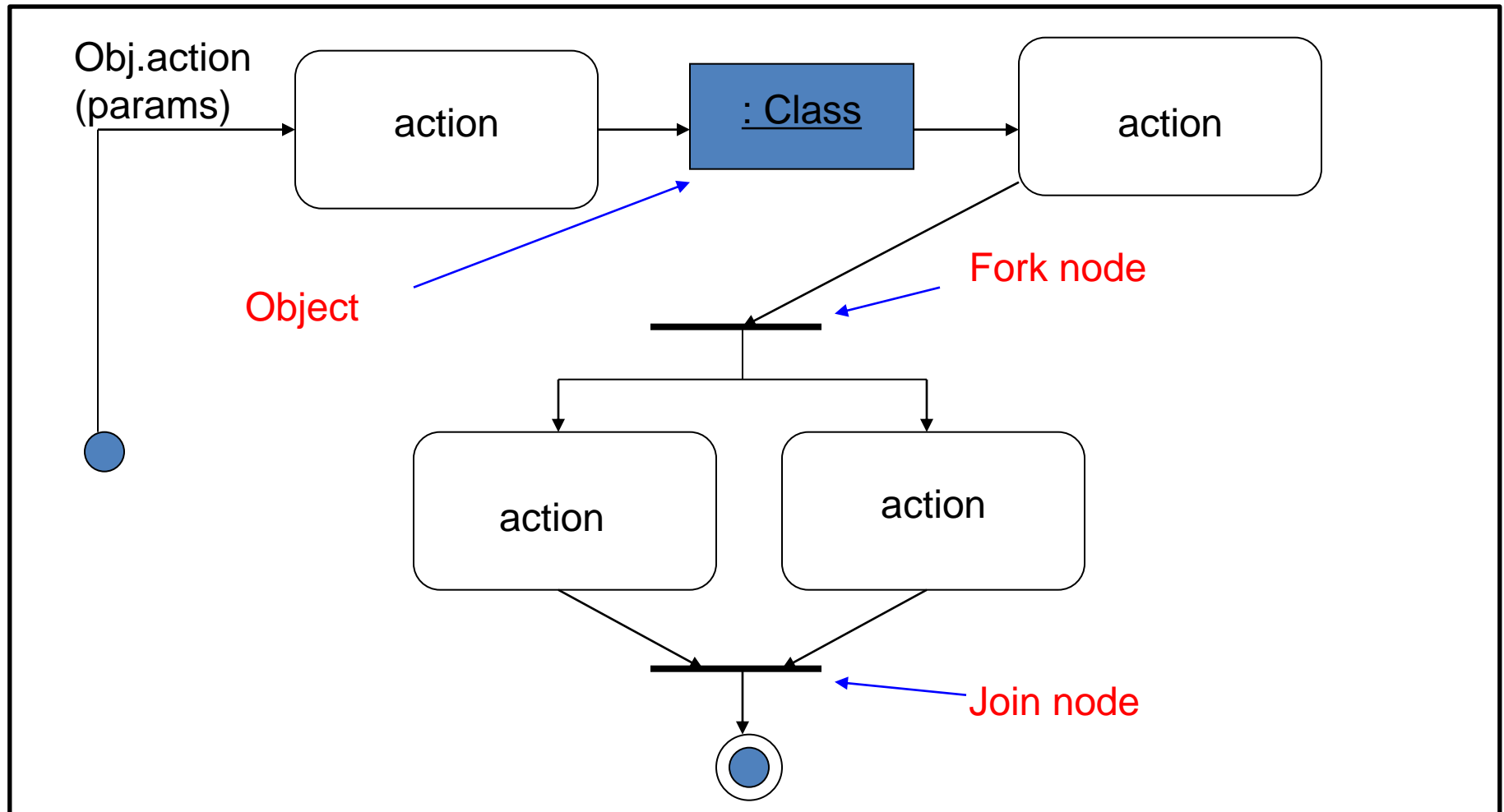
- A branch is represented by a diamond
  - Has one incoming transition to enter the branch
  - Two or more outgoing transitions augmented with mutually exclusive conditions
- A merger is also represented by a diamond
  - Two or more incoming transitions and one outgoing transition
  - The outgoing transition will be fired only when all the incoming transitions are fired



# When to Use Activity Diagram

- An activity diagram can be used to
  - describe a use case
  - describe a method in a sequence or communication diagram
  - describe an action associated with a transition in a state diagram, or the entry action or the exit action of a state diagram
- Caution: the word “action” in state diagram represents a higher level task while the same word in an activity diagram represents an atomic non-divisible computation

# More Syntax in Activity Diagram



# Example of Activity Diagram

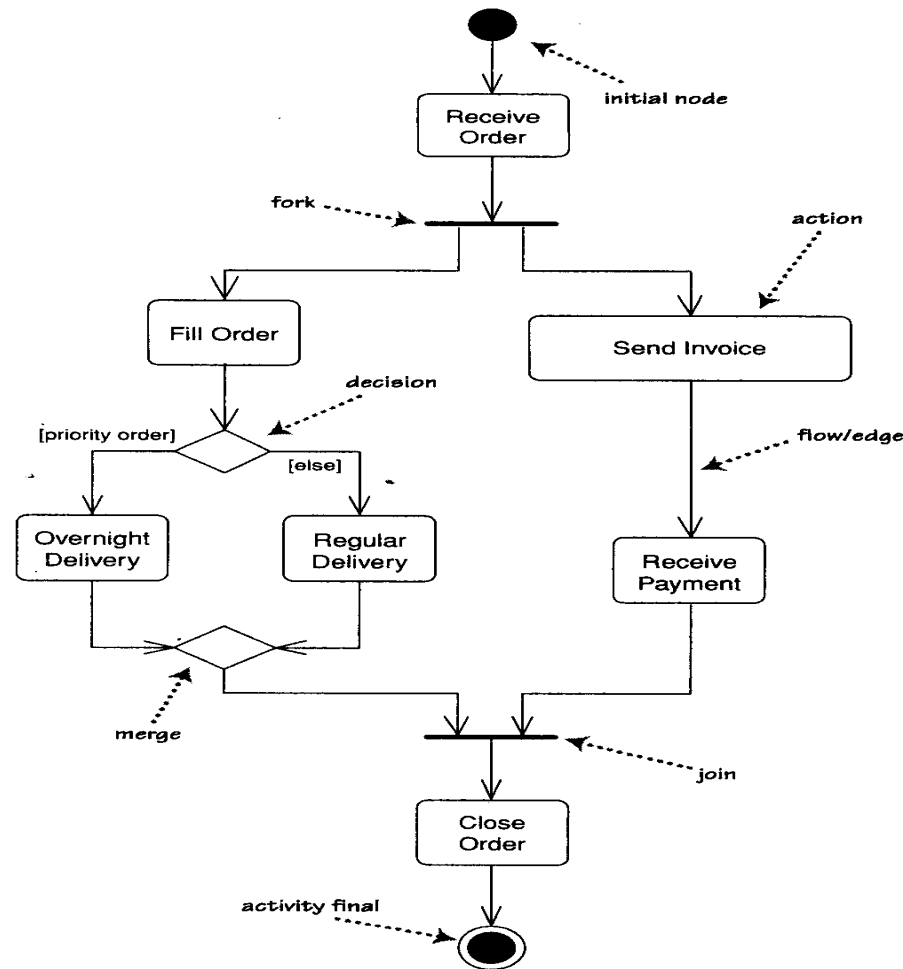
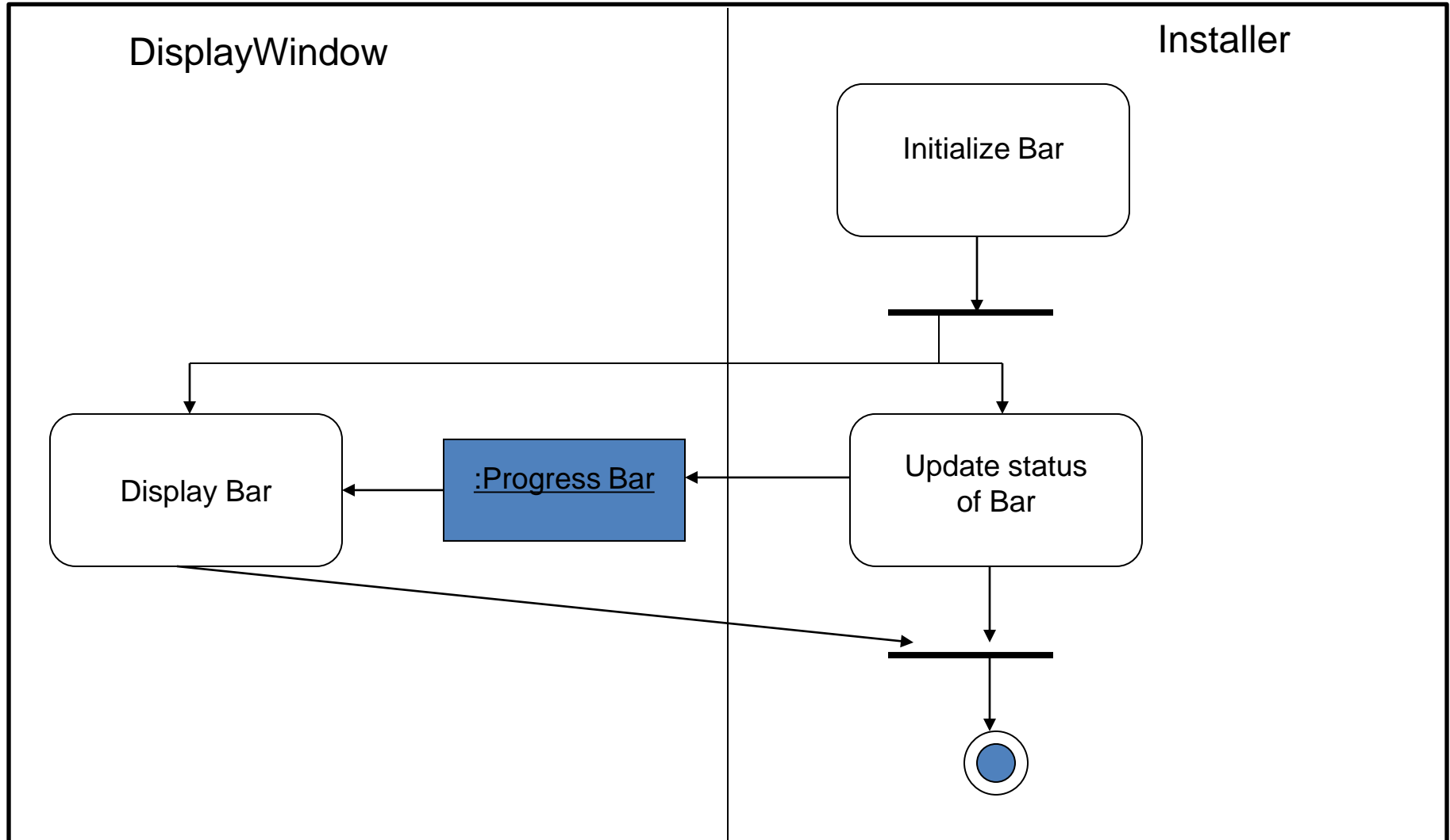


Figure 11.1 A simple activity diagram

# Activity Partition

- An activity partition represents a group of actions
  - Actions grouped based on who performs the actions
  - Actions grouped based on the functionality achieved by the actions
  - Actions grouped based on timing events
- Previous versions of UML call this as “swim lane”.

# More Syntax in Activity Diagram



# Sub Activity

An action state in an activity diagram can be represented by a sub activity as shown below



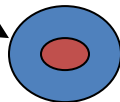
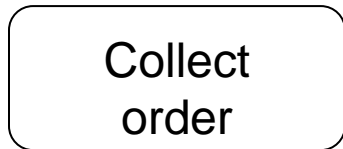
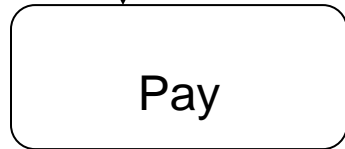
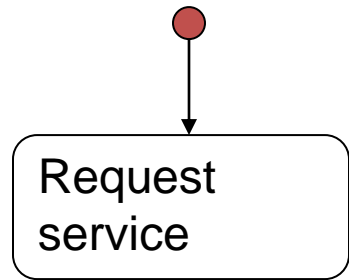
A sub activity represents a simplification of another activity diagram

It reduces the space for an activity diagram

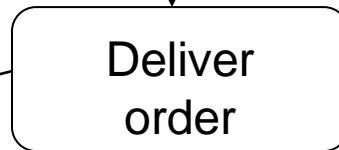
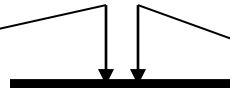
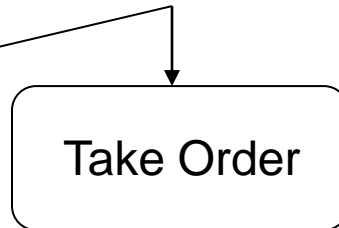
# Sub Activity Semantics

- A subactivity is a representation of **another activity diagram**
- **The incoming arrow** to a subactivity matches with the initial state of the activity diagram represented by the subactivity
- **The outgoing arrow** from a subactivity matches with the final state of the activity diagram represented by the subactivity

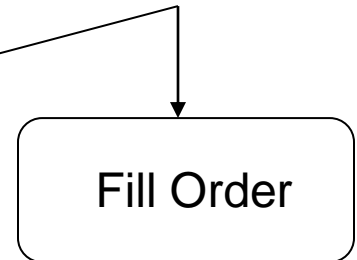
## Customer



## Sales



## Stockroom

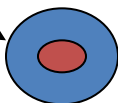
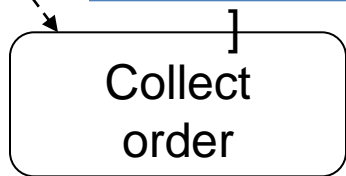
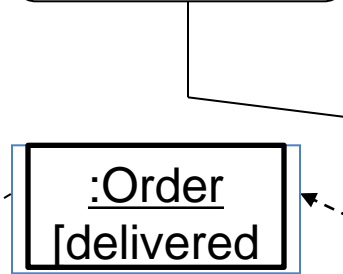
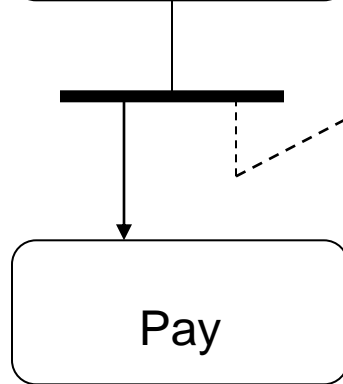
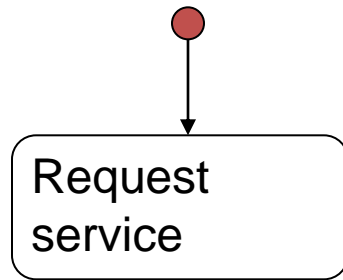




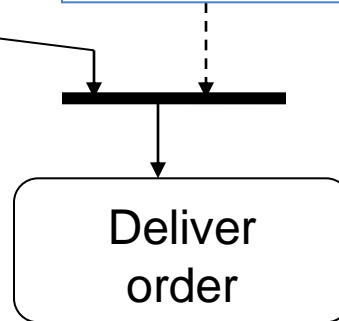
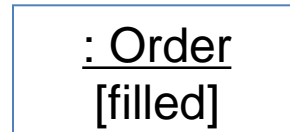
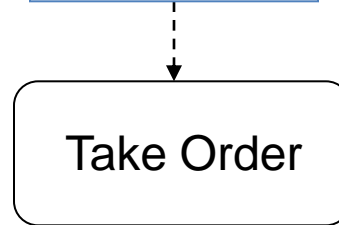
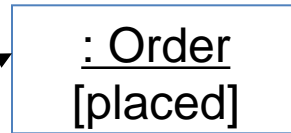
# Object as Parameters

- Objects passed as parameters between action states can be represented in the activity diagram (and in swimlane diagram) using the same syntax for objects
- The transition between an object parameters and an action state is represented with a dashed line, instead of a solid line

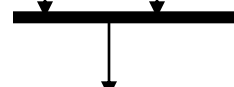
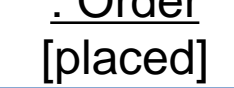
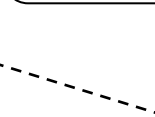
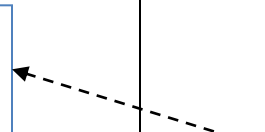
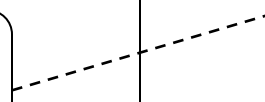
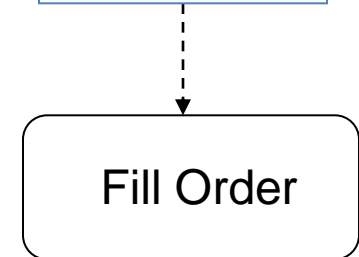
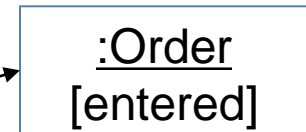
## Customer



## Sales



## Stockroom

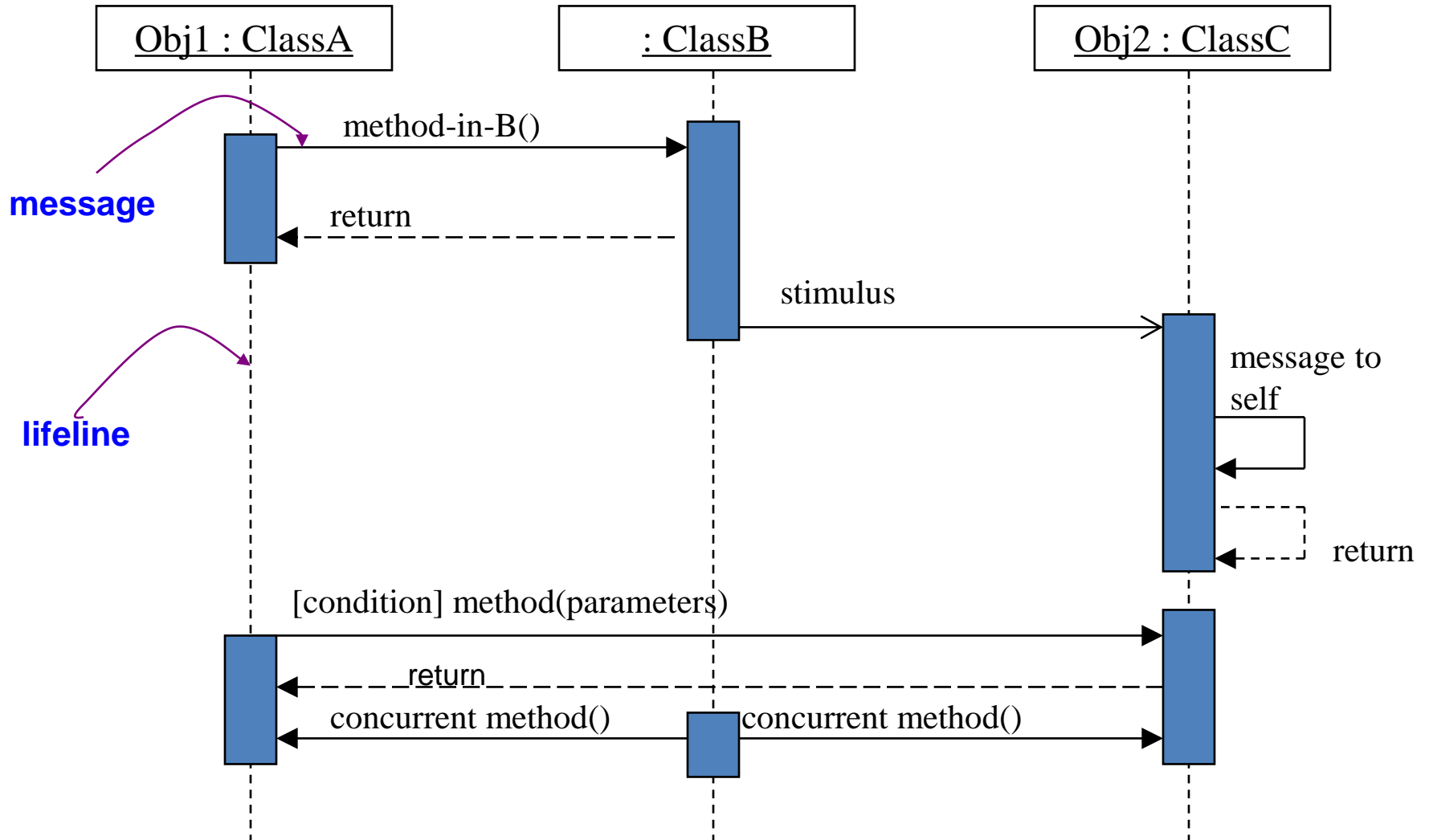


# **Sequence Diagram Modelling**

# Sequence diagram

- A two-dimensional chart that describes collaborations
- Each sequence diagram describes a particular scenario
  - E.g., a sequence diagram can describe successful withdrawal operation
  - Another sequence diagram can describe failure of withdrawal operation due to insufficient funds
  - It is also possible to describe both scenarios in one sequence diagram

# Sequence diagram: basic syntax



# Sequence diagram – semantics (1)

- Rectangular boxes on the top indicate objects
- The dotted vertical line indicates the life line of the corresponding object
- Rectangular boxes on the dotted vertical line indicate the duration in which the corresponding object is active; the object is idle otherwise
- A solid arrow with solid arrowhead indicates a message
- A solid arrow with thin arrowhead indicates a stimulus
- A dashed arrow indicates return of control

# Sequence diagram – semantics (2)

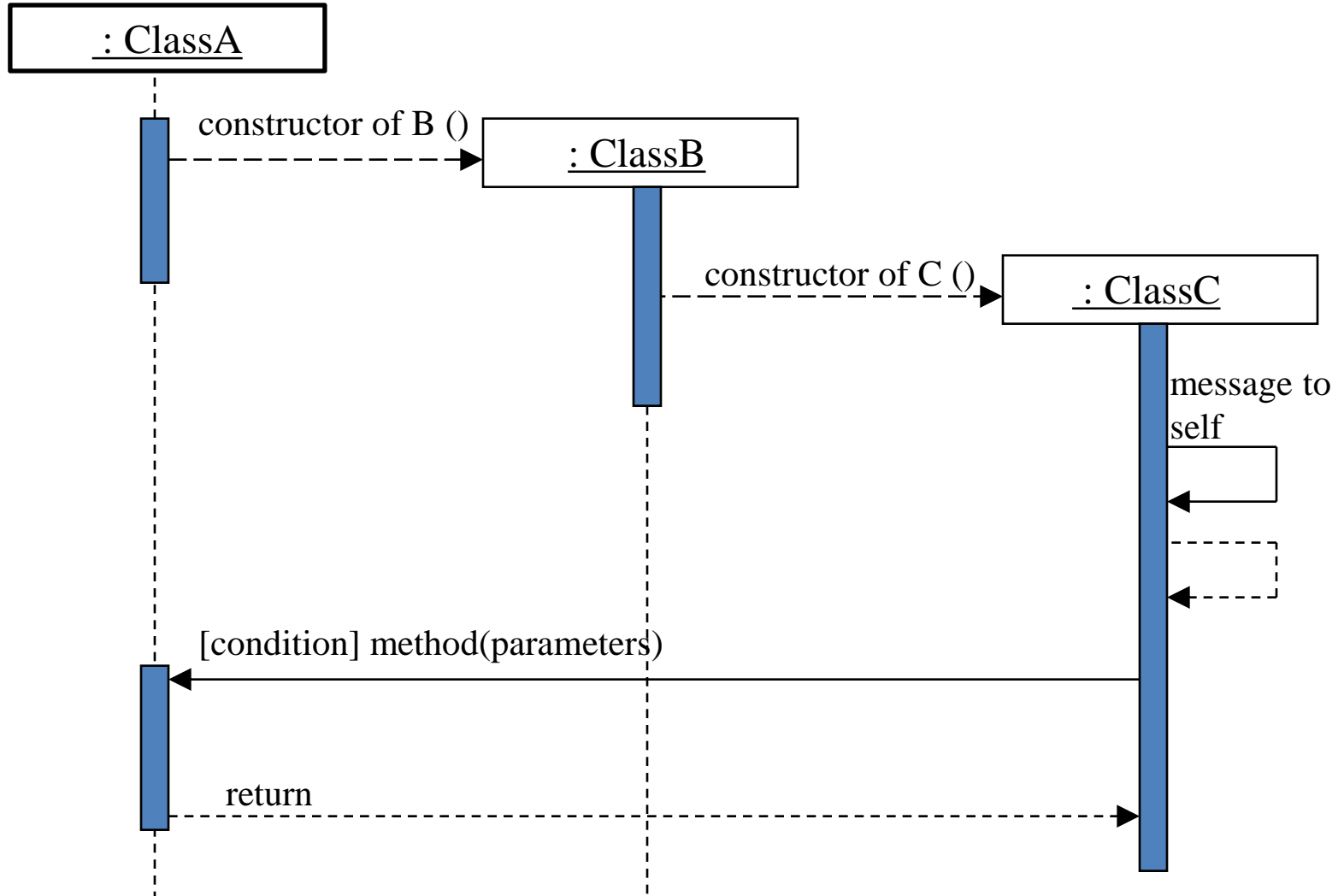
- Messages/stimuli may be augmented with conditions
- Messages/stimuli can be concurrent
  - start from one object and send messages or signals to more than one object at the same time
- The vertical dimension indicates time axis
  - A message 'Y' placed below a message 'X' in a sequence diagram indicates that message 'X' is sent before message 'Y'

# Sequence diagram – semantics (3)

- Objects with dashed vertical line for the entire diagram have lifeline for the entire scenario
  - These objects are assumed to be already created before this scenario starts and assumed to exist even after the scenario ends
- Objects with short life span within a scenario can be shown differently (see the next diagram)
- There is no ordering required among the placement of objects on the horizontal line
  - A designer may choose the ordering for the convenience of drawing the diagram



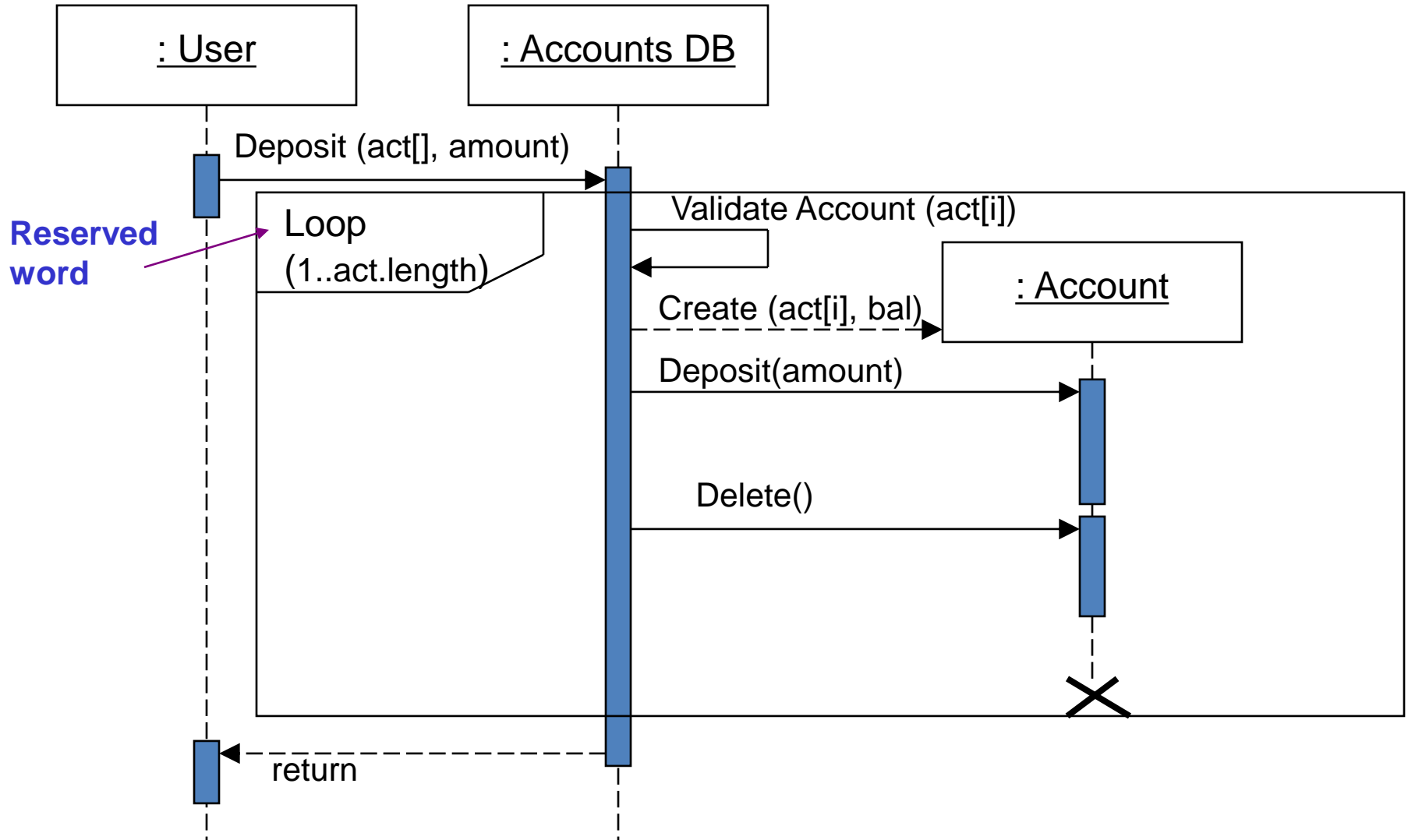
# Sequence diagram – extended syntax



# Sequence diagram – semantics (4)

- The creation of an object is shown by the vertical displacement of the object from the top of the diagram. The object is placed at the point of creation
  - See the creation of objects from class A and from class B
- The termination of an object can be shown by placing an “X” at the bottom of its life line
  - See the destruction of the object from class C

# Sequence diagram – Specifying Loops



Depositing the same amount into several accounts

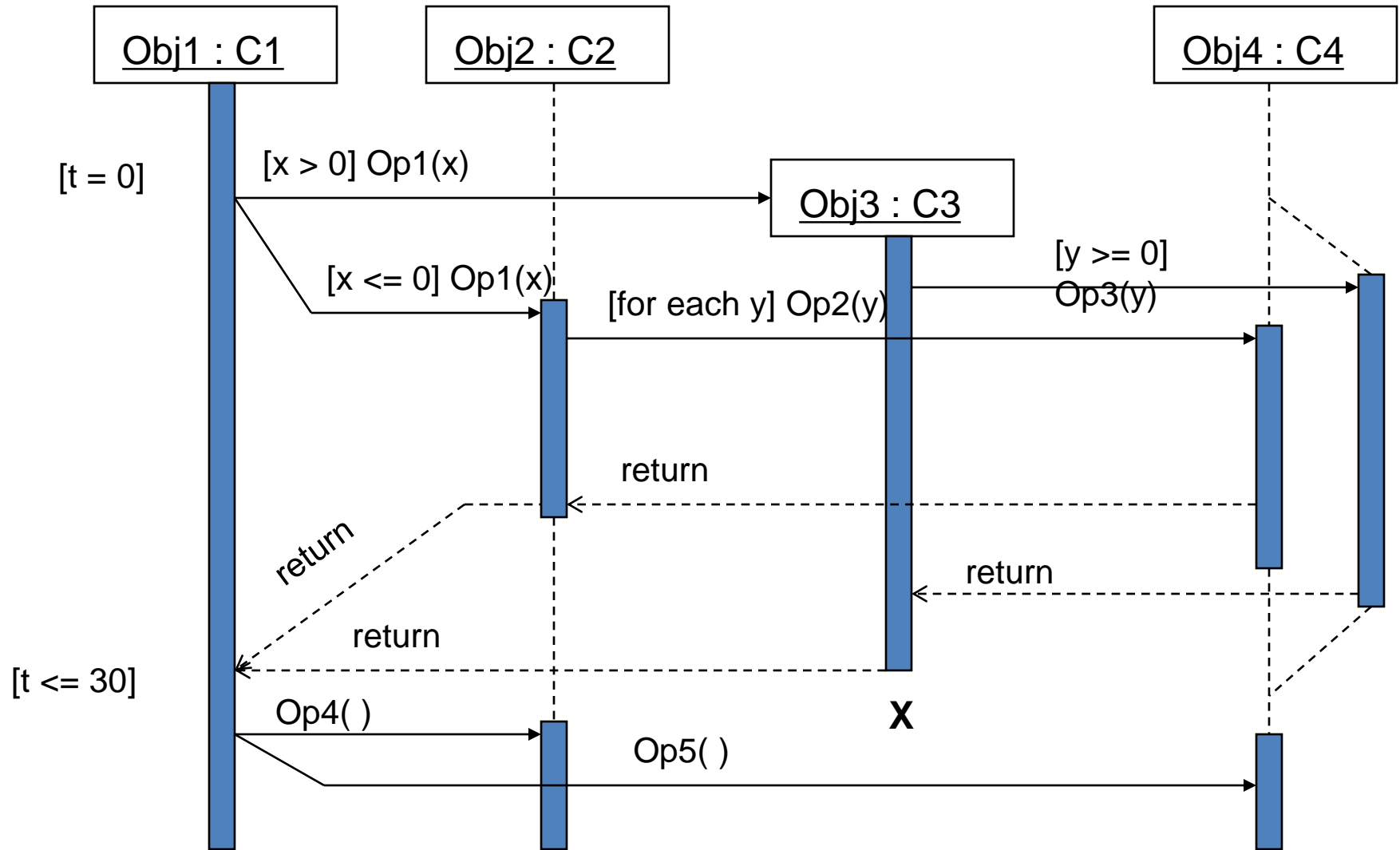
# Concurrent messages / stimuli - semantics

- an object can send two messages / stimuli to two different objects at the same time
  - concurrent execution
  - there may be a condition on each of these messages / stimuli but they can be totally different
  - the tail end of these two messages / stimuli coincide at the originating object
  - the arrow heads may physically be on different horizontal levels but logically they are at the same time line (horizontal level)
  - see “Op4() and Op5()” in the diagram (next page)

# Time-based messages / stimuli- semantics

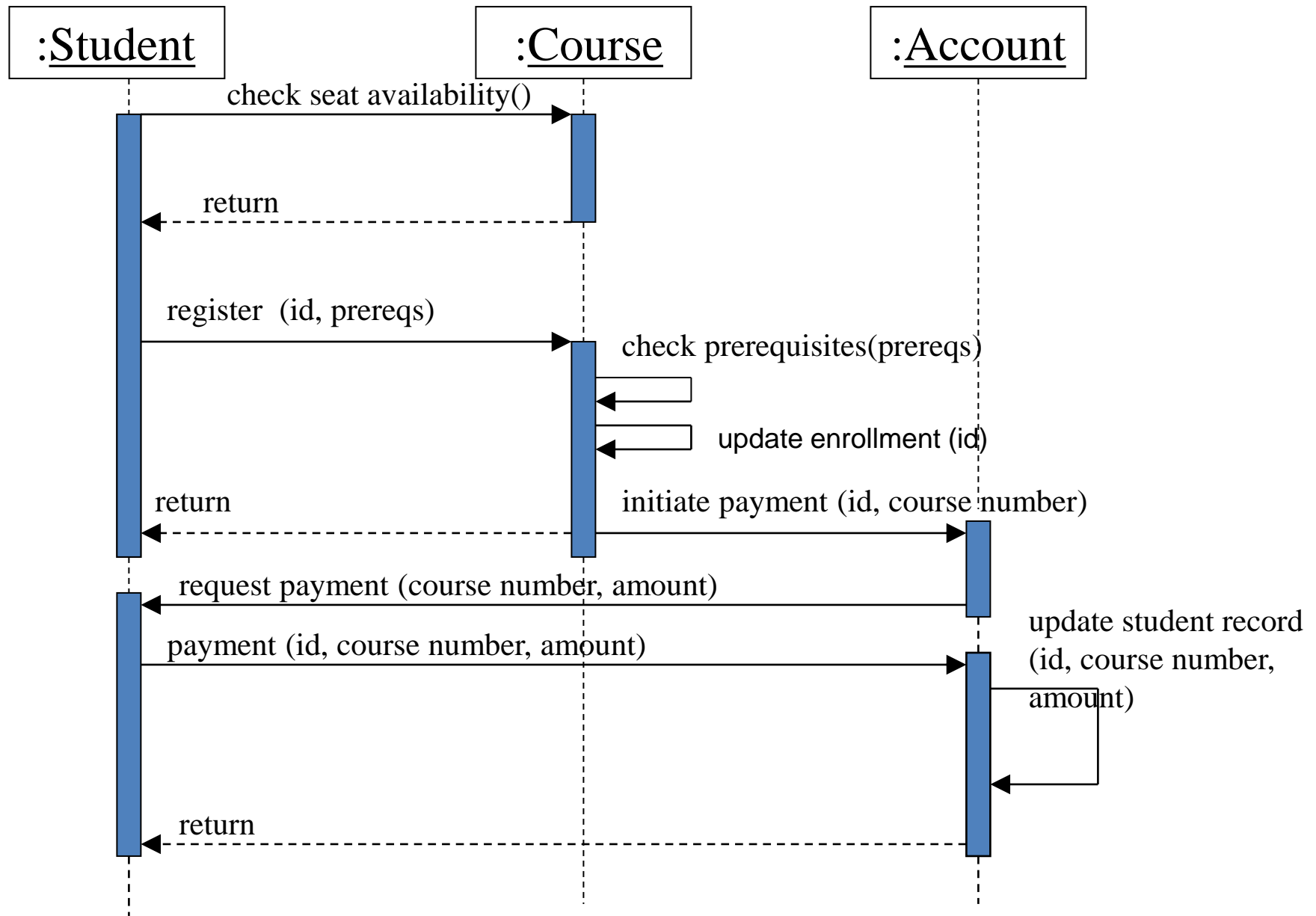
- a message can be split into two, leading to two destination objects with mutually exclusive conditions
  - see “Op1(x)” in the diagram
- a lifeline can be split into a side track for a specific duration to indicate mutually exclusive situations
  - see “Obj4” in the diagram
- iterations can be specified using a condition at the beginning of a message / stimulus
  - see “Op2(y)” in the diagram
- timing constraints can be added onto the messages / stimuli or on the vertical time axis

# Concurrent and time-based messages and stimuli



# Example – Course registration system

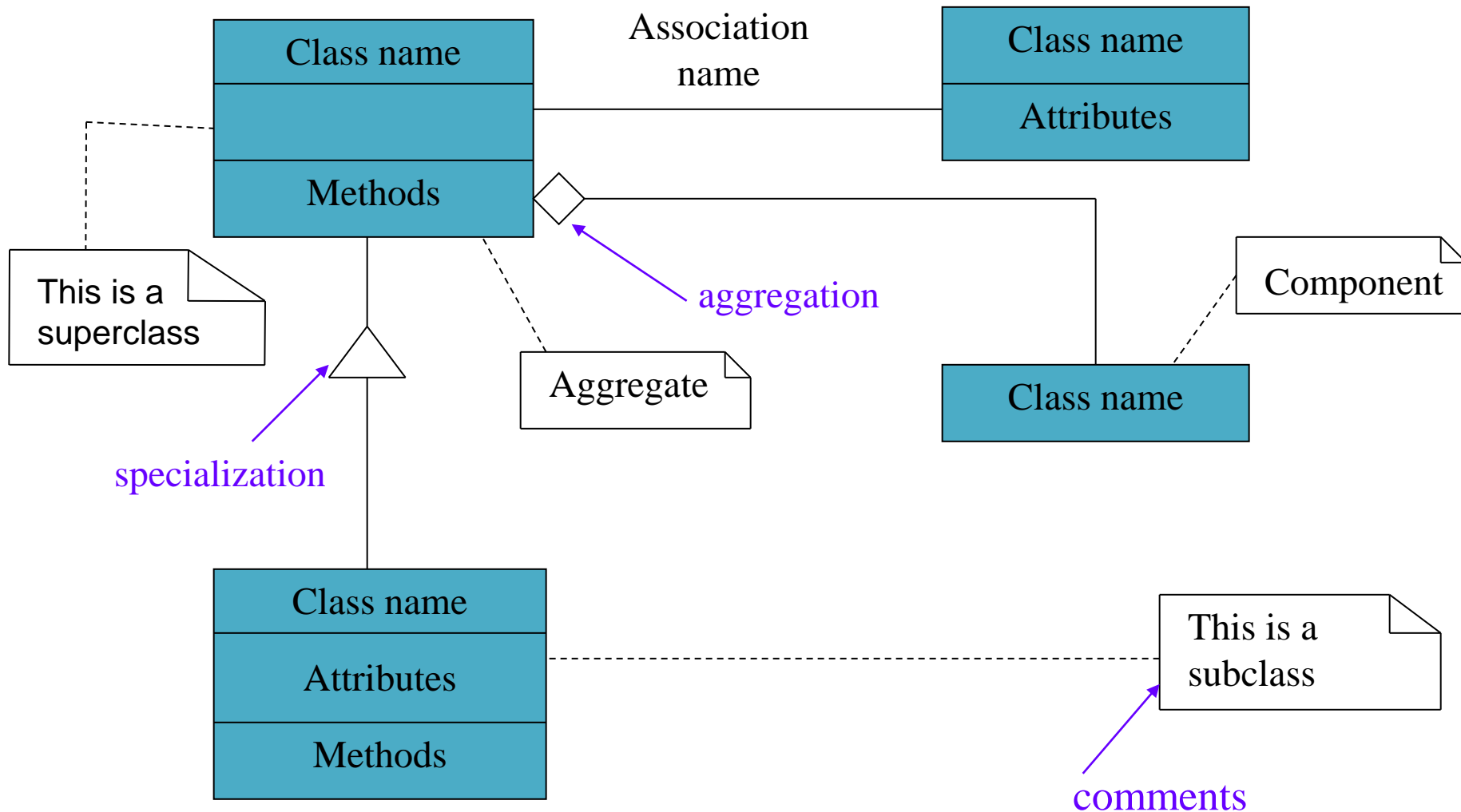
- Scenario
  - A student object checks the availability of seats in a course. If available, it sends a message to register. The course object checks for the prerequisites first. Upon acceptance, the course object returns the message back to the student object and at the same time informs the account object to bill the student. The account object then communicates with the student to get the payment for the course.
  - The following diagram shows a successful course registration process





# **Class and Object Diagram Modelling**

# Class diagram – basic syntax



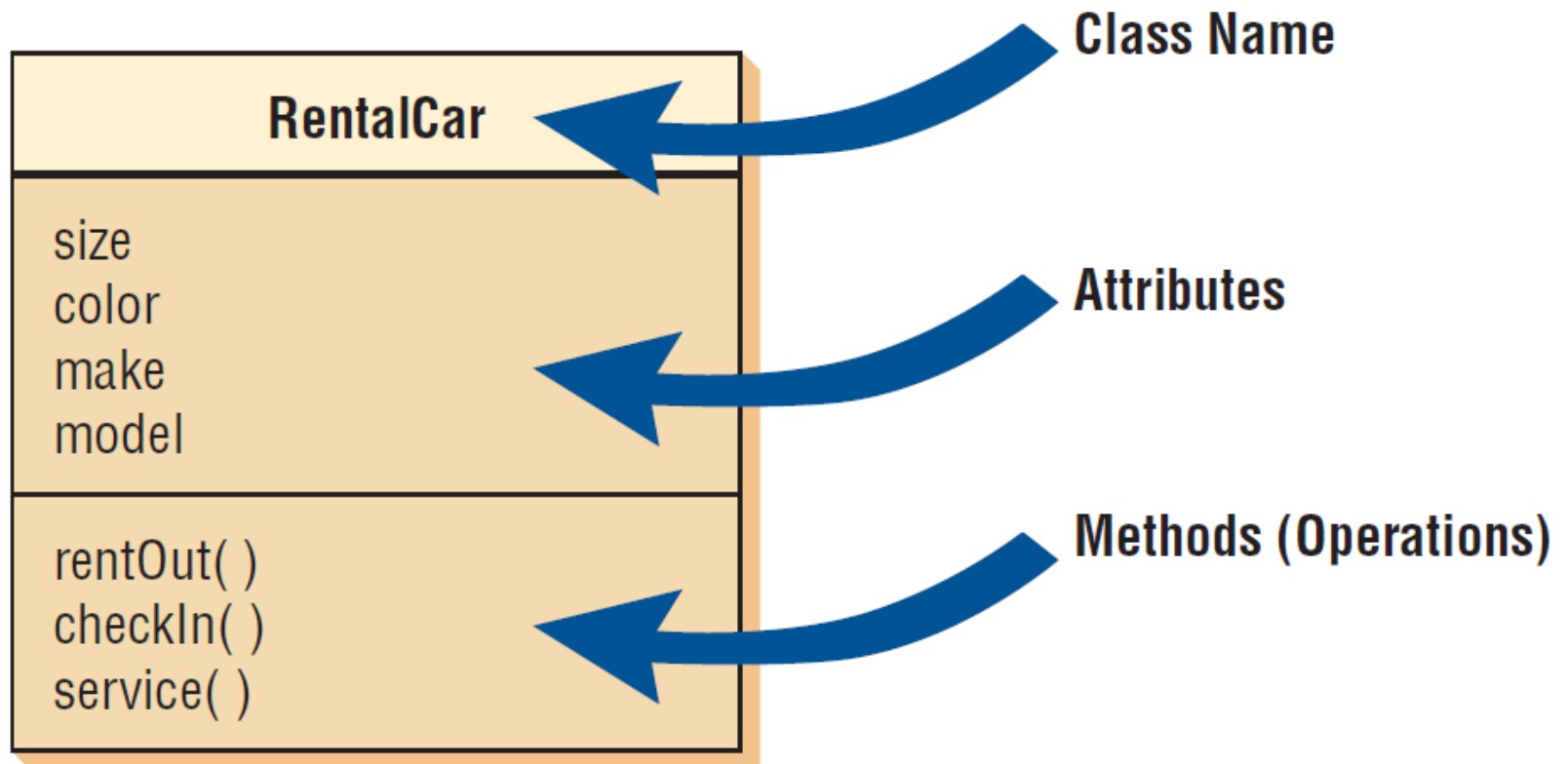
# Class Diagram: semantics (1)

- A class icon must have a class name. Optionally, it can have attributes and/or methods.
- Attributes and methods are strings and will not be validated by the modeling tools.
- Attributes can be specified by their names or by <name : type> pairs.
- Methods can be specified by their names or with partial signatures or with complete signatures.

# Class Diagram: semantics (2)

- Comments can be included in any diagram with a rectangle folded at right top corner
  - The dotted line from the comment is important to indicate which portion of the diagram is explained by the comment
- Suggestion
  - For validation purposes, when showing aggregation relationship, the aggregate (the one near the diamond edge) must include an attribute whose type is the component class

# Class



# Details of a class icon

+ public

- private

# protected

~ visible within  
the package

## Account

- Account number : Integer

- Balance : Real

- Overdraft : Boolean = true

Initial value

+ GetAccountNumber () : Integer

# UpdateBalance (sign :Sign, amt : Integer)

~ ReturnBalance () : Real

- ChangeOverdraft ()

# An abstract class

## *Polygon*

{ abstract, author = Kasi, last  
modified = Oct 2002 }

<<constructor>>

+ Polygon(List of Vertex vertices)

<<query>>

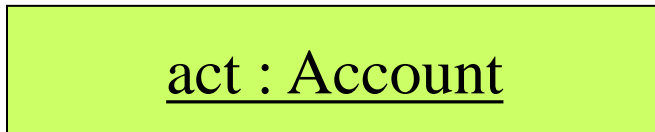
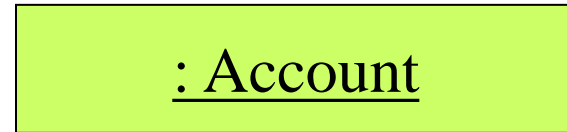
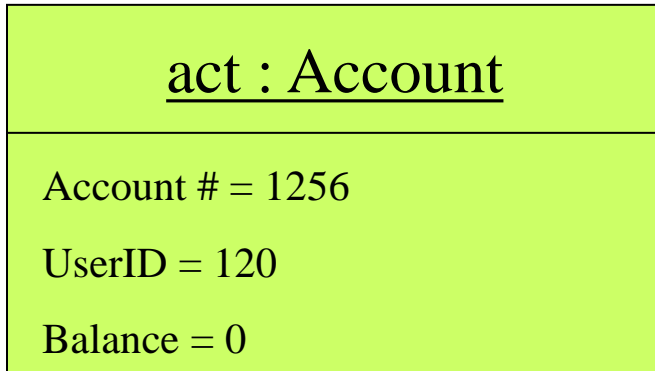
#area () : Real

# Multiplicity

Multiplicity	Explanation
0..1	0 or 1
1	Mandatory 1
0..*	0 or Many
1..*	1 or Many
*	Many

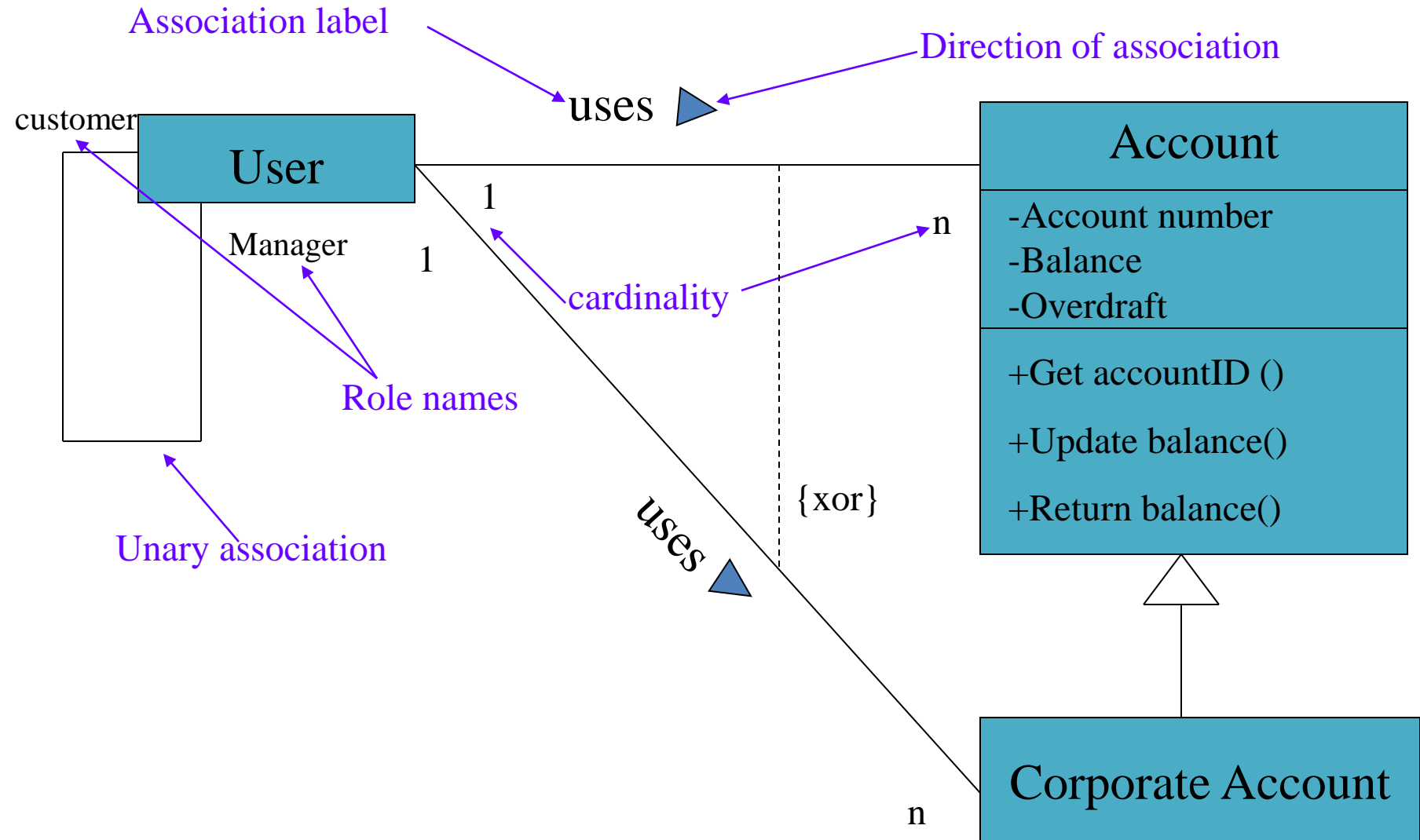


# Object Diagram



**Various representations of an account object**

# Association –syntax



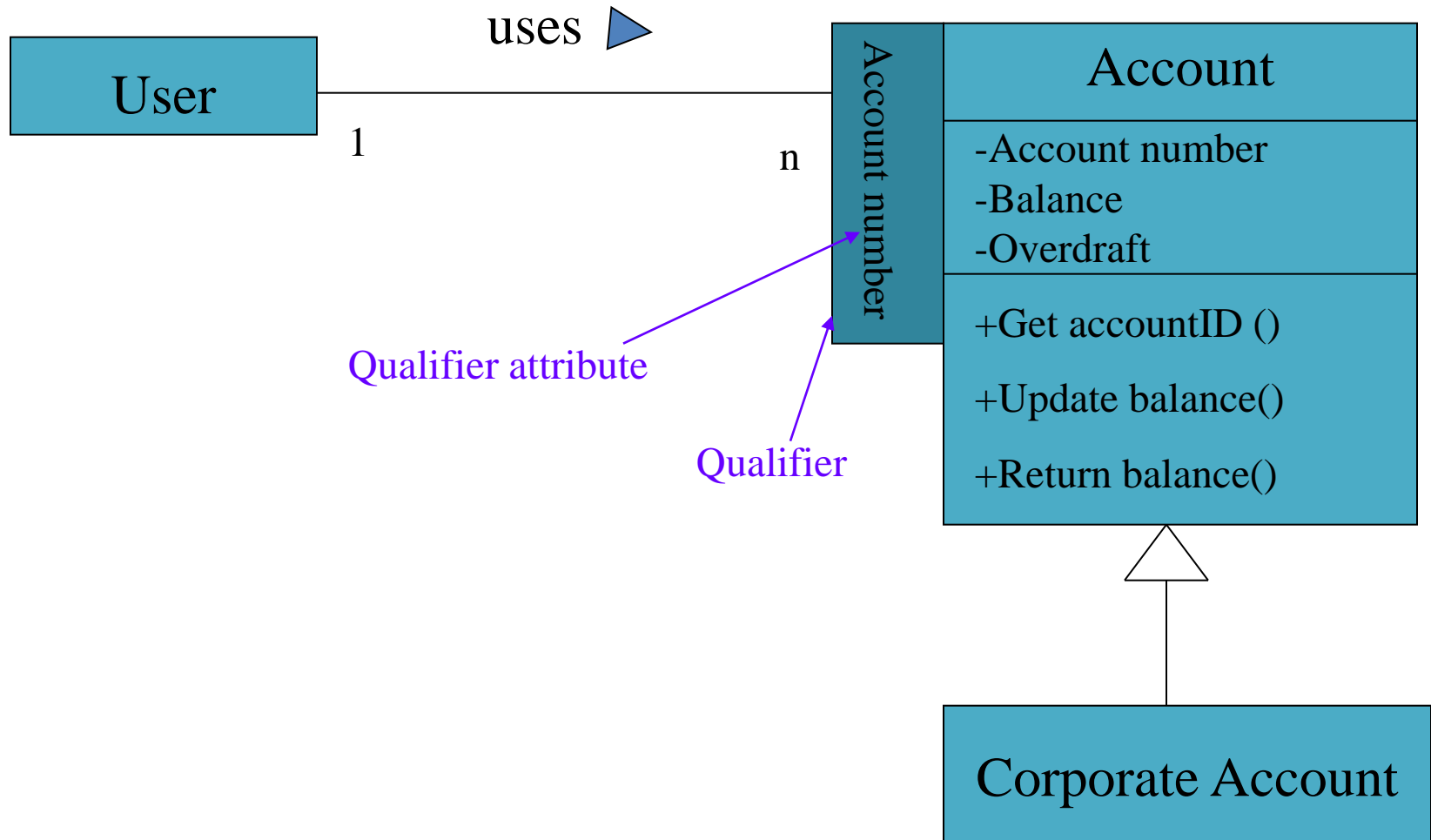
# Association – Semantics (1)

- Every association is expected to be labeled
  - UML does not require a name for an association
- Direction of an association, cardinality, role name are all optional
  - For unary associations, it is better to include role names
- Representations of cardinality
  - 0, 1, \* (zero or more), n..m (values in the range between n and m both inclusive)

# Association – Semantics (2)

- A constraint may be [optionally] placed between two associations
  - See the example in the previous slide that asserts an Exclusive OR relationship between the associations
- When a subclass specializes a superclass, it also inherits all associations between the superclass and other classes
- In the previous example, the association “uses” between “User” and “Account” is also inherited by the pair “User” and “Corporate Account”

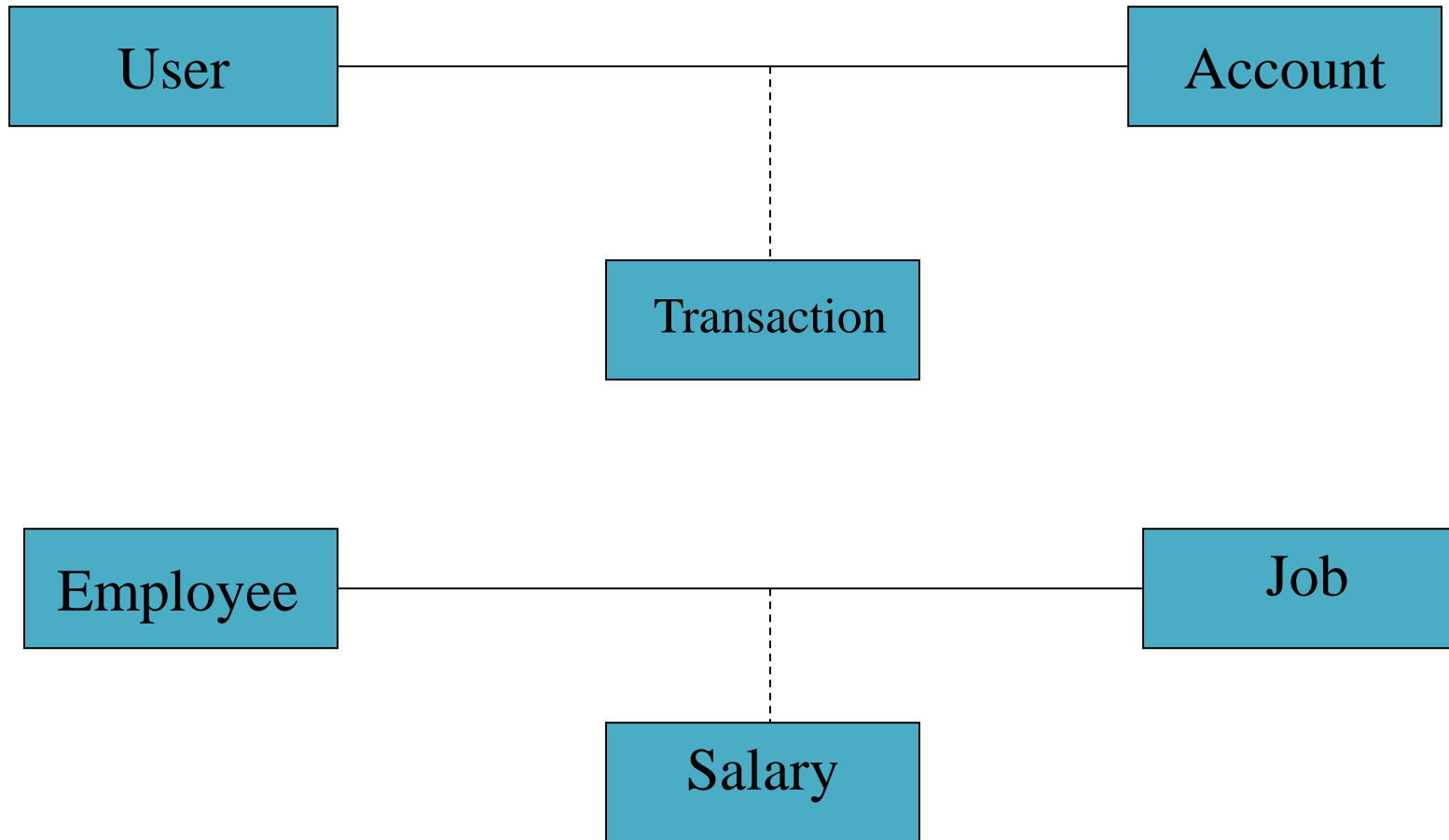
# Association with qualifiers



# Association - Qualifiers

- Qualifiers can be attached to a “one-to-many” association
  - It is rectangle attached to the “many” end of the association
- A qualifier is a collection of variables whose values uniquely identify an instance at the “many” end of the association
  - In the example, an account number uniquely identifies an account in a collection of accounts
- Qualifier is part of the association

# Association Class



# Association Class - semantics

- A piece of information that belongs to both classes in an association is put into a separate class called “association class”
  - Association class is a dependent class that depends on the other two classes in the association
  - An association class cannot exist independently
  - An object of an association class must refer to objects of the other two classes in the association
    - Example: A “Transaction” object depends on a “User” object and on an “Account” object.



# Shared Aggregation

An aggregation relationship in which the component can be shared by classes/objects outside the aggregation



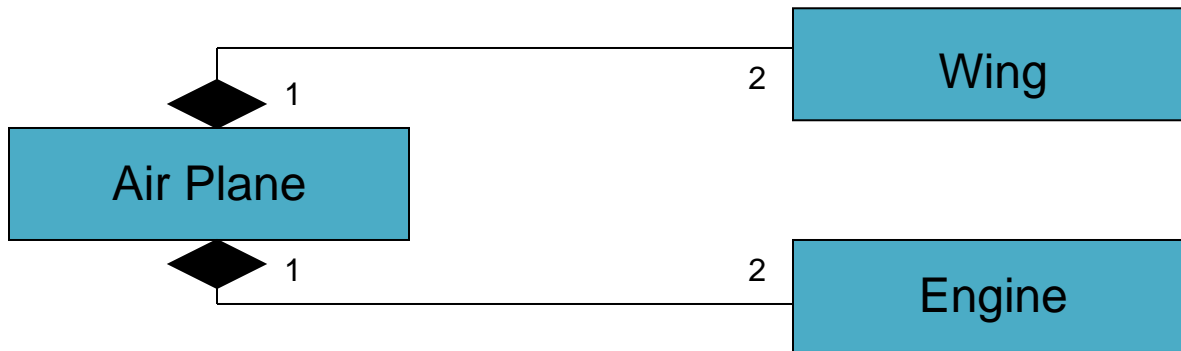
Person object is shared by both Team and Family objects

Shared aggregation is indicated by a hollow diamond

**Caution:** Changes made to a component object will affect all the aggregates that include the component.

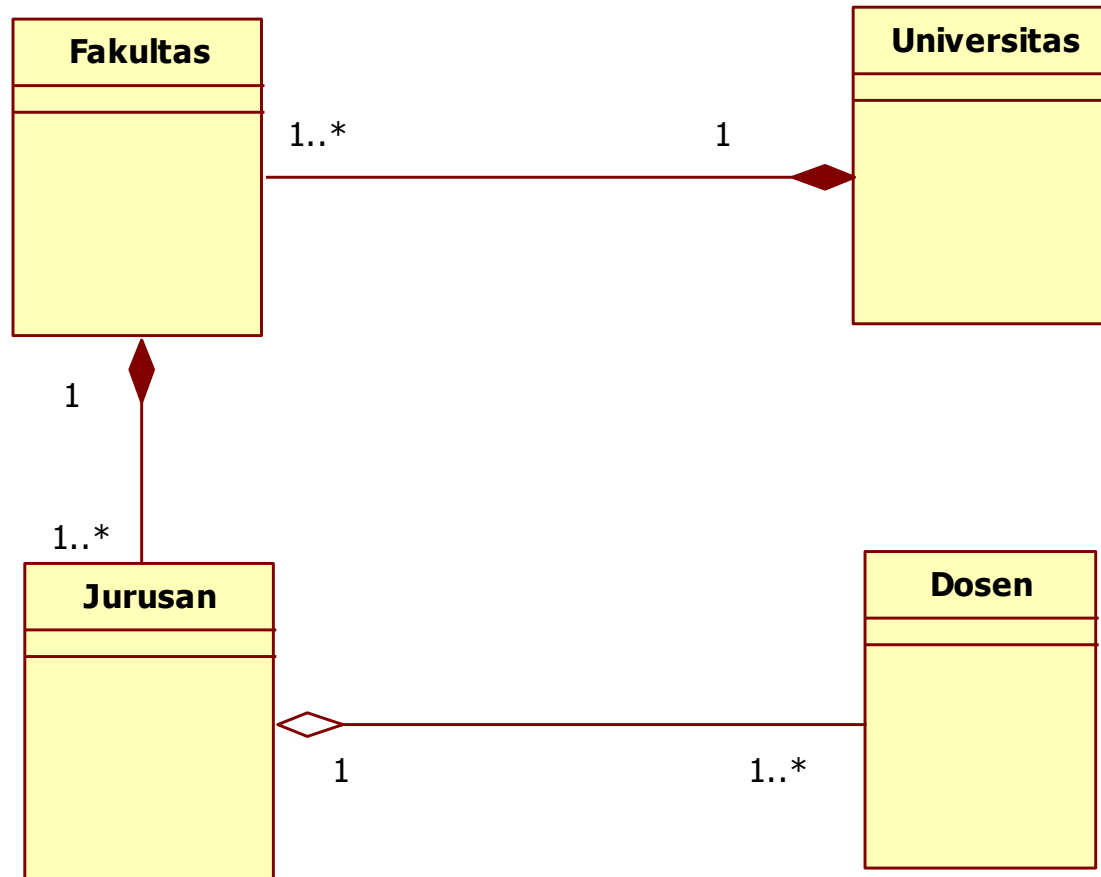
# Composite Aggregation

An aggregation relationship in which the component is an exclusive part of the aggregate; hence, not shared.



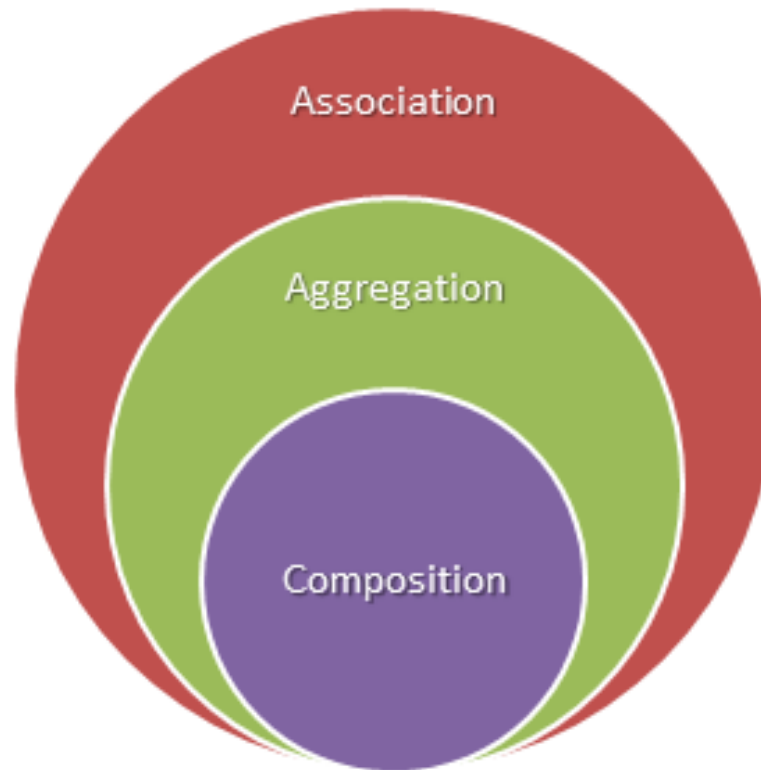
Composite aggregation is indicated by a filled diamond

# Composition VS Aggregation

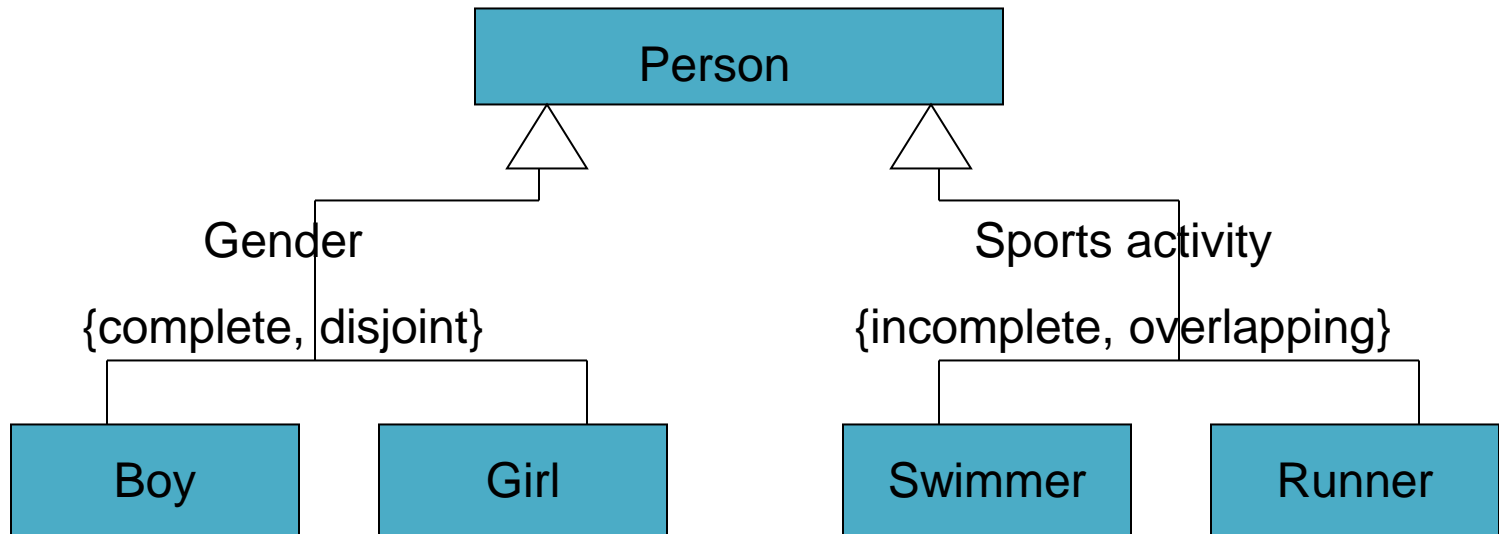


Bila Universitas ditutup maka Fakultas dan Jurusan akan hilang akan tetapi Dosen tetap akan ada. Begitupun relasi antar Fakultas dengan Jurusan

# Association VS Composition VS Aggregation



# Advanced Specialization



These optional domain words make the relationships easier to understand.

# How to find classes? (1)

- Nouns in requirements document or use case descriptions may provide a good starting point, but often are inadequate
- Each class should contain a distinct set of operations relevant to the system under consideration
  - Think of a class as an ADT
- Remove vague classes
  - Classes that do not adequately describe themselves
    - A class that represents the internet

# How to find classes? (2)

- Try not to include implementation-oriented classes in the analysis model
  - May be introduced later during design and/or implementation
  - Examples: array, tree, list
  - These classes will not only occupy so much space in the diagram but also tend to divert the focus of analysis

# How to identify associations? (1)

- An association corresponds to a semantic dependency between classes
  - Class A uses a service from class B (client-server)
  - Class A has a structural component whose type is class B (aggregation)
  - Class A sends data to or receives data from class B (communication)



# How to identify associations? (2)

- Include only those associations that are relevant to the current model
  - Constrained by assumptions, simplifications, system boundary (what is expected to be provided by the system)
  - Three different associations between “Faculty member” and “Course”
    - “Faculty member” teaches “Course” in a course registration system
    - “Faculty member” creates “Course” in a curriculum development system
    - “Faculty member” evaluates “Course” in a course evaluation/inspection system

# How to identify associations? (3)

- Eliminate redundant associations
  - “Faculty member” teaches “Course”
  - “Course” is taught between “Time” to “Time”
  - Therefore, “Faculty member” teaches between “Time” to “Time”
    - use transitivity between associations
- Remember that subclasses inherit the associations of a superclass

# How to identify aggregations?

- Aggregations are also associations
- Identify as Association if it is not clear whether it is Association or Aggregation
  - “Mail” has “Address” (aggregation)
  - “Mail” uses “Address” for delivery (association)
  - “Customer” has “Address” (aggregation)
  - “Customer” resides at “Address” (association)
  - “TV” includes “Screen” (aggregation)
  - “TV” sends information to “Screen” (association)

# How to identify specialization?

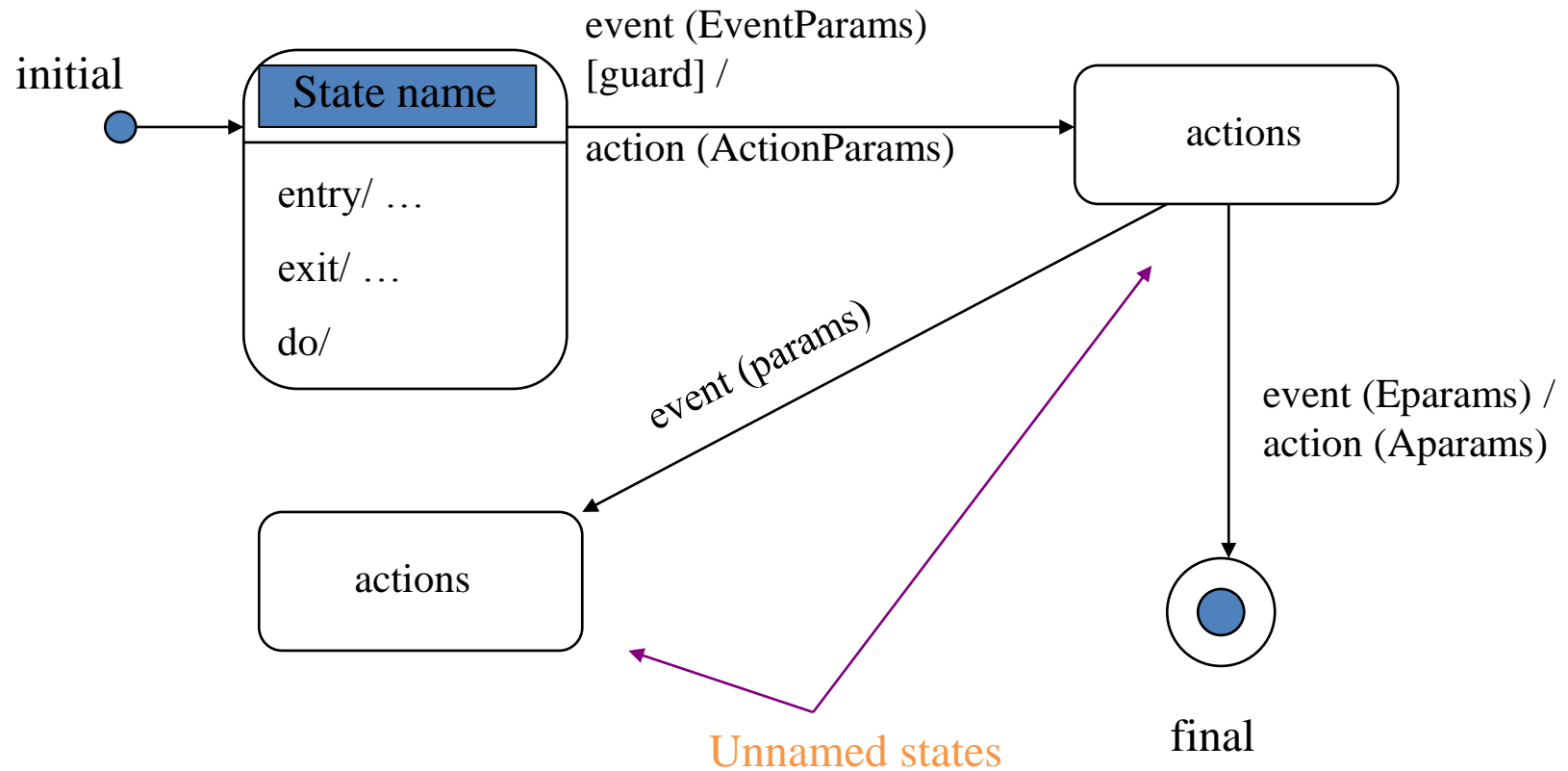
- Generally, specialization relationships are noticeable in the application domain
- Top-down approach
  - “Student”, “Full-time Student”, “Part-time Student”
  - “TV”, “Plasma TV”, “Flat Panel TV”
  - “Customer”, “Bank manager”, “Teller”

# How to identify specialization (2)

- Some of them are discovered during analysis
- Bottom-up approach
  - “Part-time Instructor” derived from “Instructor” and “Student” while modeling a department
  - “User” derived from “Customer”, “Bank Manager” and “Teller” while modeling an ATM system
  - “Material” derived from “Book”, “Journal” and “Magazine” while modeling a library catalog system

# **Statechart Diagram Modelling**

# State diagram – basic syntax



# State Diagram

- A transition between states is represented by the event that triggers the transition
- Transitions may have guards or conditions under which the transitions fire
- A state may optionally have a label
- Every state may have
  - An entry action – executed as soon as the state is entered
  - An exit action – executed just before leaving the state
  - A “do” action – executed while the object is in this state; it may be ongoing action until the object leaves the state



# State Formal Definition (1)

- A state is a condition in the life of an object during which the object performs an action or waits for some event
- A state is represented by the collection of attributes and their corresponding values
- An object after being created must be in at one particular state at any instant
  - Unless otherwise mentioned, an object remains in a state for a **finite time**
  - UML allows modeling of transient states (states that exist only for a very short and insignificant duration)

# State Formal Definition (2)

- A state (directly or indirectly) includes links (instances of associations) connected with the object at that instant
- A state may be decomposed into concurrent sub-states (AND relationship)
- A state may be composed using mutually exclusive disjoint sub-states (OR relationship)

# Event Formal Definition

- A noteworthy occurrence
  - UML manual version 1.5
- Something that happens within the system or interacting with the system at an instant
- Something that has a significant impact on the system
- Examples
  - sending a signal or data
  - receiving a signal or data
  - making a request for execution
  - a Boolean condition becoming true
  - a timeout condition becoming true

# Four types of events in UML

- Signal event
  - occurs when an object sends a signal to another object
- Call event
  - occurs when a method or operation in an object is invoked
- Change event
  - occurs when a Boolean condition is changed
- Time event
  - occurs when a time limit has reached

# Transition (1)

- Represents the change of states of an object
  - switch from “Empty balance” to “Positive balance”
- A transition is an abstraction of an operation
  - The above transition is an abstraction of deposit operation
- A transition has finite and significant duration
  - Observable time taken to complete deposit operation

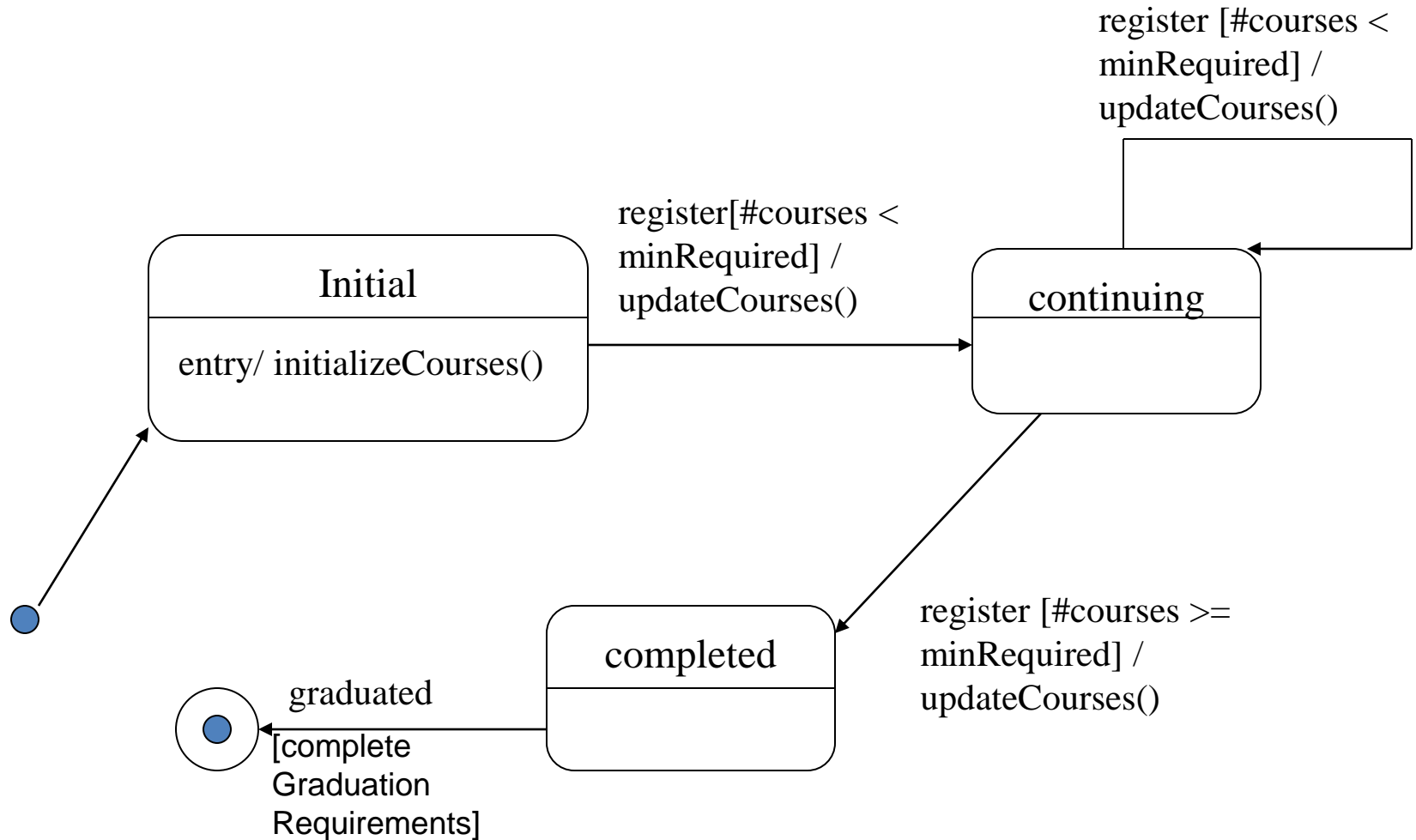
# Transition (2)

- A transition may have parameters
  - A transition corresponding to the deposit operation will have the “amount” as a parameter
- A transition is triggered/invoked/fired by the occurrence of an event
  - The transition corresponding to deposit will occur by the event “request for deposit”
  - The transition from “Positive balance” to “Empty balance” occurs by the completion of the operation “withdrawal”

# Transition (3)

- A transition may have a guard/condition
  - The transition corresponding to the withdrawal operation will occur only if the balance is greater than or equal to the amount to be withdrawn
  - The condition associated with a transition is always the precondition for the transition and hence must be checked before the transition occurs
- An event may cause several transitions to fire
  - The event that triggers the transition to move to “Empty balance” state after withdrawal may also cause a message to be sent to the account holder and at the same time may also cause a note to be recorded in the account log

# Example of State Diagram: Student Class



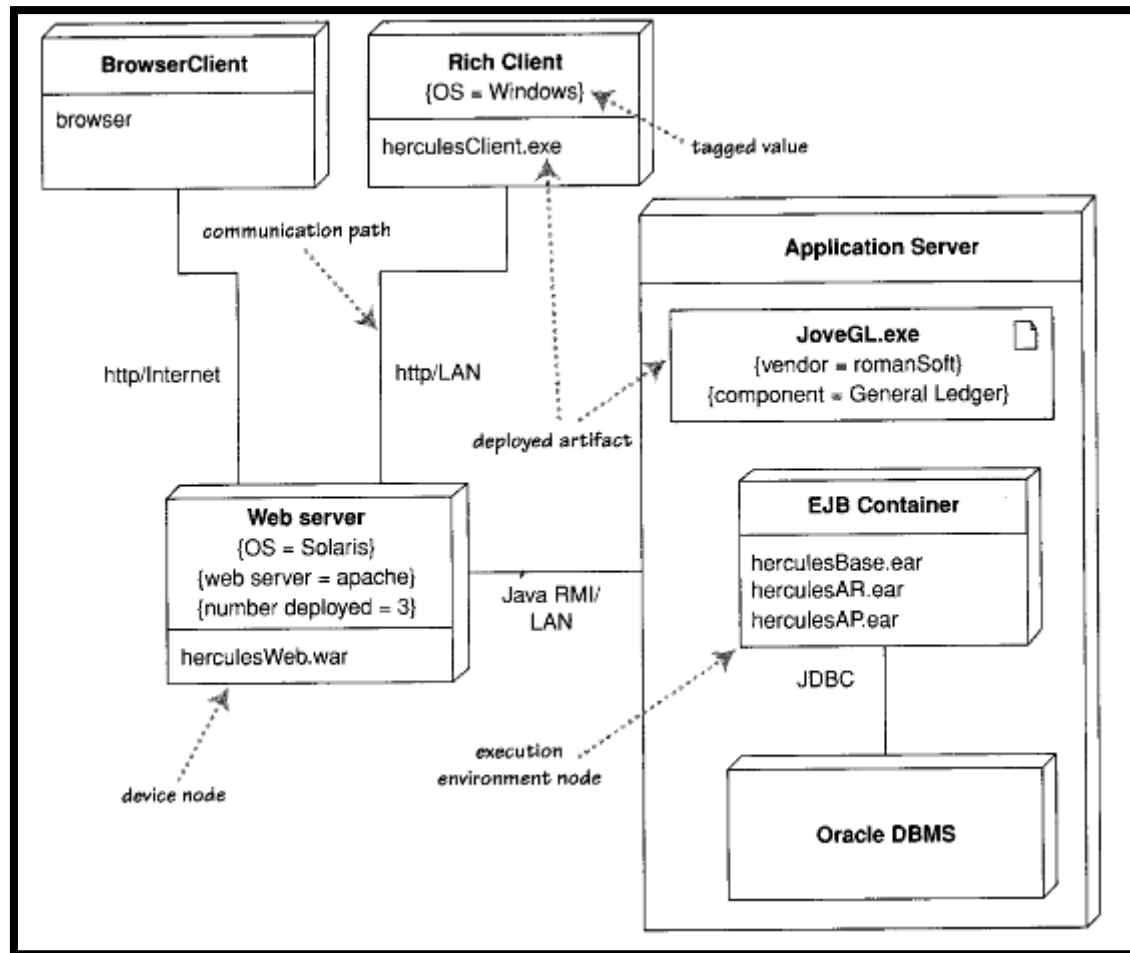


# **Deployment Diagram Modelling**

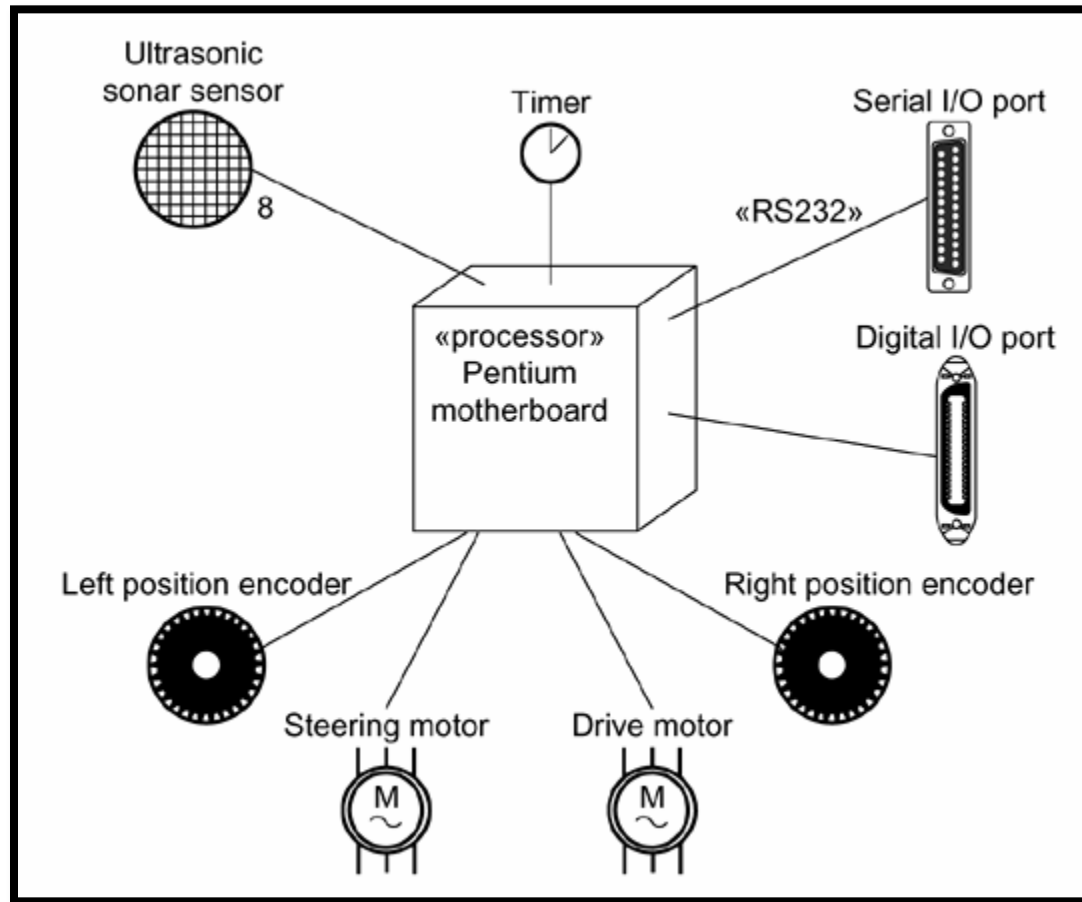
# Deployment Diagram

- Deployment diagrams show a system's physical layout, revealing which pieces of software run on what pieces of hardware.
- embedded system:
  - Device, node, and hardware
  - Client/server System
  - Pure Distributed System
  - Re-engineering Application

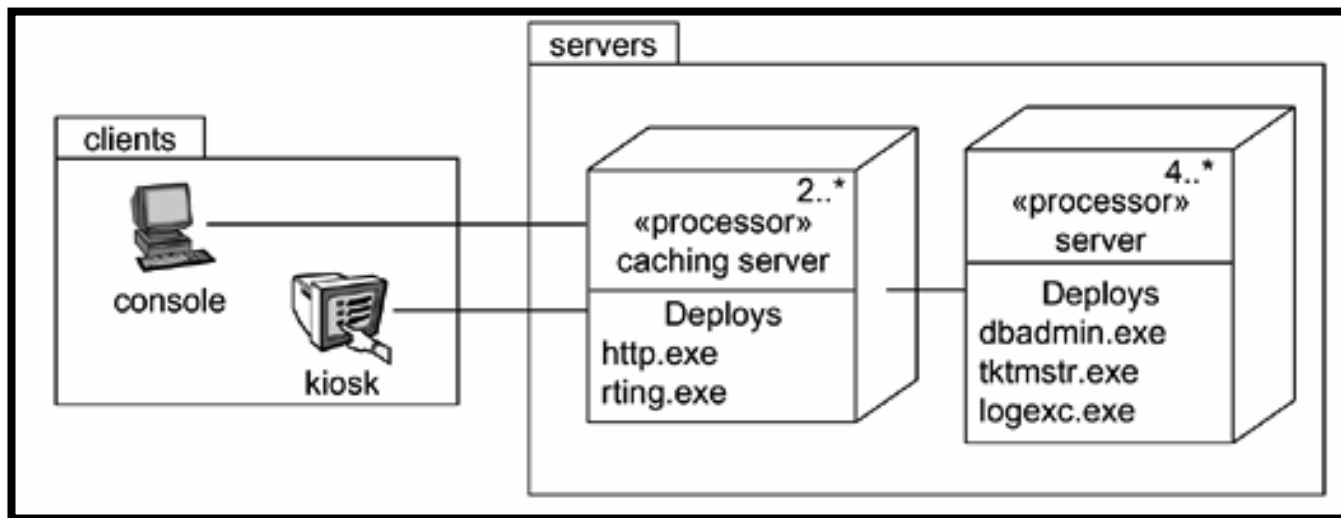
# Example Deployment Diagram



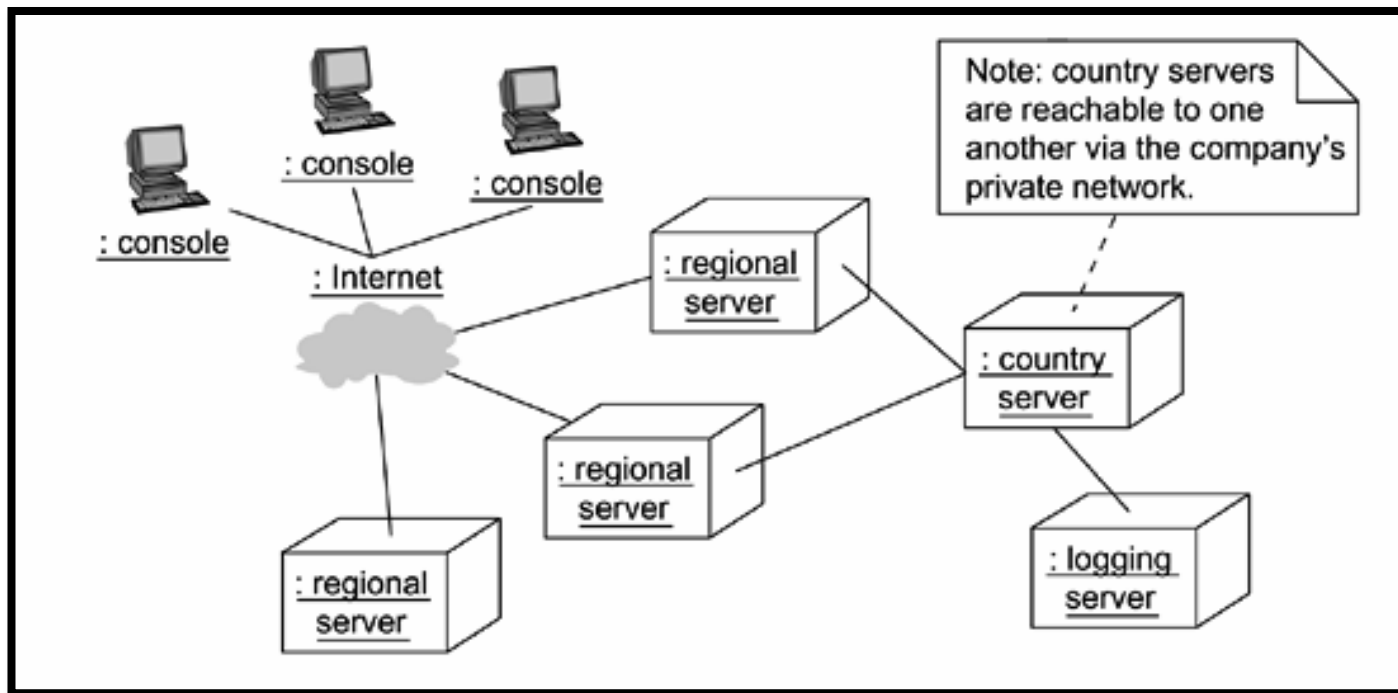
# Example Deployment Diagram: Embedded System



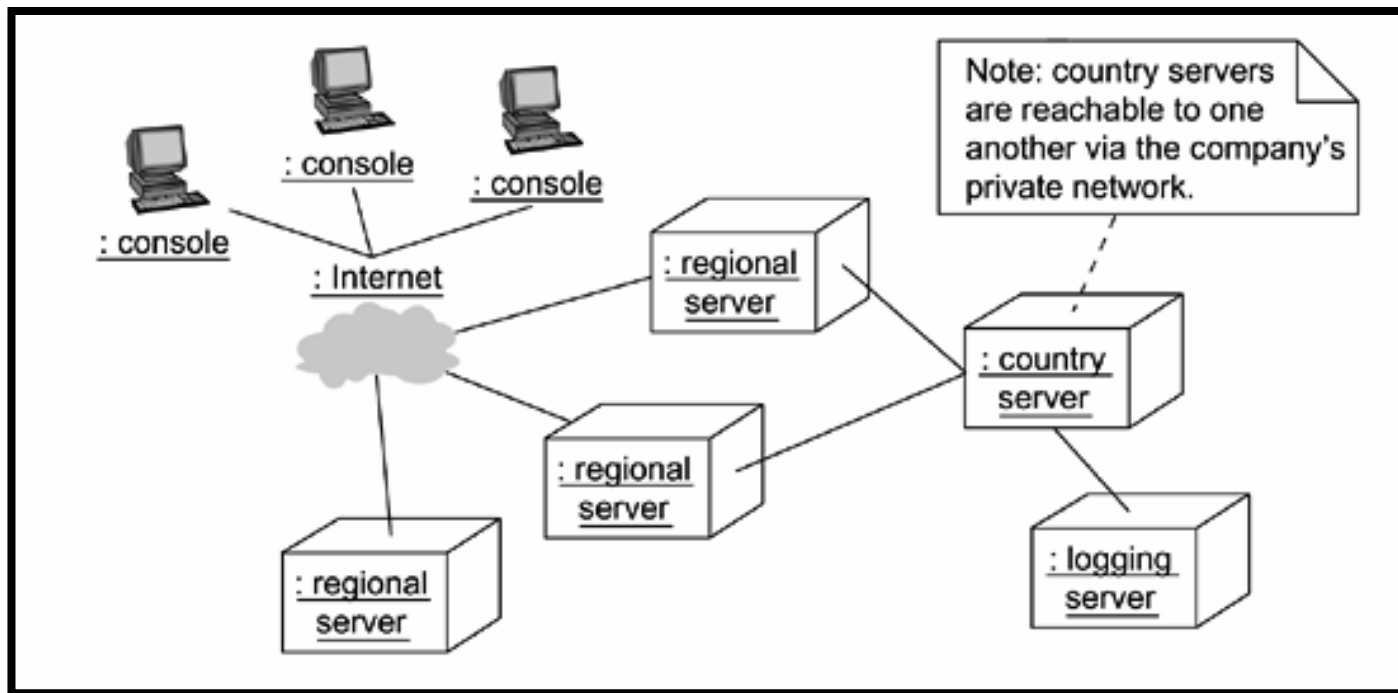
# Example Deployment Diagram: Client/Server System



# Example Deployment Diagram: Distributed System



# Example Deployment Diagram: Distributed System



# When to Use Deployment Diagram?

Don't let the brevity of this part make you think that deployment diagrams shouldn't be used . They are very handy in showing what is deployed where, so any nontrivial deployment can make good use of them.