Chair of Software Engineering

Software Architecture

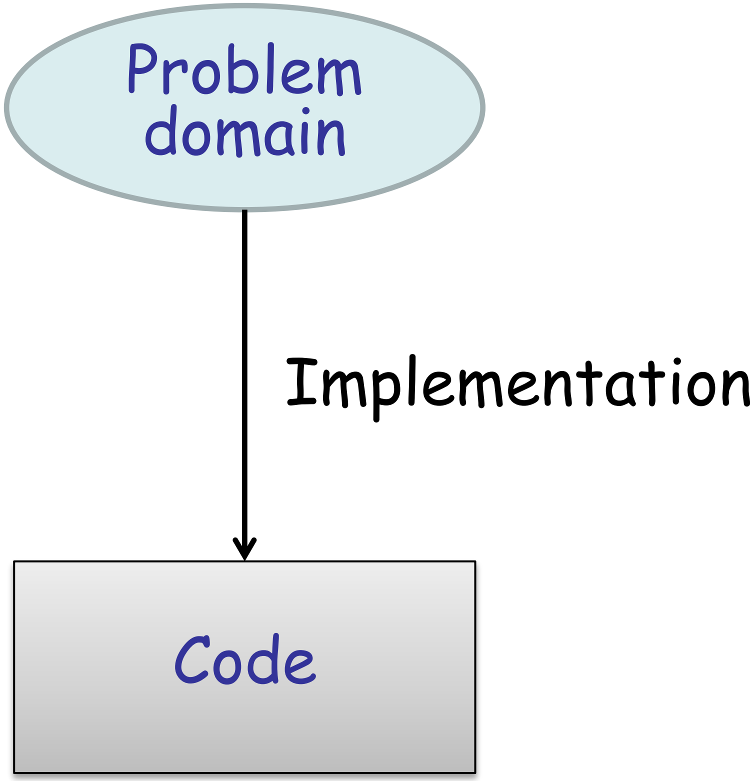
Bertrand Meyer, Carlo A. Furia, Martin Nordio
(Christian Estler)

ETH Zurich, February-May 2011

Lecture 16:
UML – Unified Modeling Language

Why do we need models?

Ideal world...

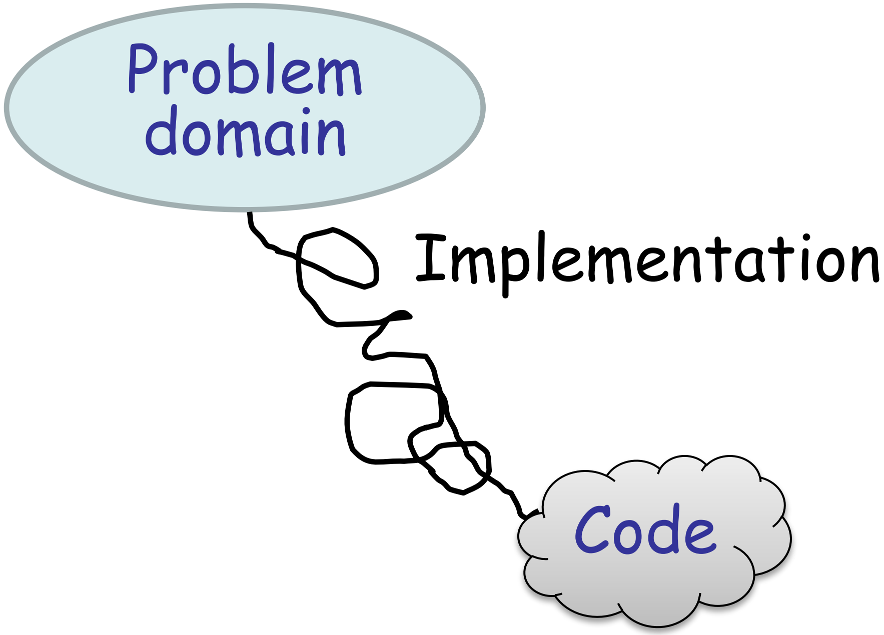


Problem domain

Implementation

Code

Reality



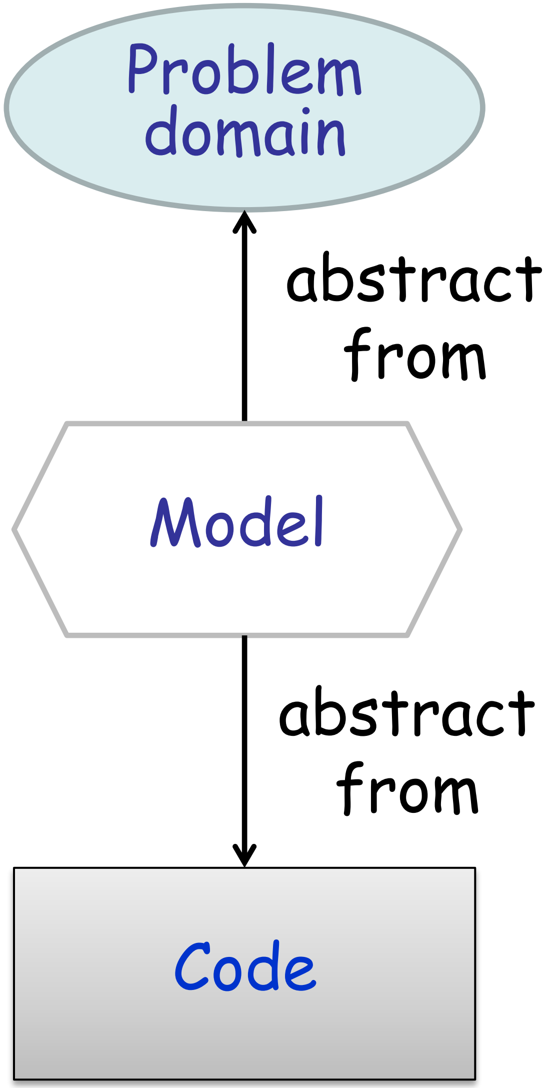
Problem domain

Implementation

Code

Why do we need models?

- Models are abstractions of „the real thing“
- They hide complexity by looking at a problem from a certain perspective
    - Focus on relevant parts
    - Ignoring irrelevant details
    - What is relevant depends on the model
- Example: to model the main components of a car, we do not need internal details of the engine.



Why model software?

- Why is code itself not a good model?

- Software is getting increasingly more complex
  - Windows XP: ~40 millions lines of code
  - A single programmer **cannot manage** this amount of code in its entirety

- Code is **not easily understandable** by developers who did not write it

- We need **simpler representations** for complex systems

- Modeling is a means for **dealing with complexity**

UML - Unified Modeling Language

- Unified Modeling Language (UML)
  - General purpose modeling language (for [OO software] systems)
  - Today's de-facto standard in Industry
- Sine '97, UML is defined/evolved by the Object Management Group (OMG)
  - Founded 1989 by IBM, Apple, Sun, …
  - Microsoft joined 2008
  - Today more than 800 members

UML - Unified Modeling Language

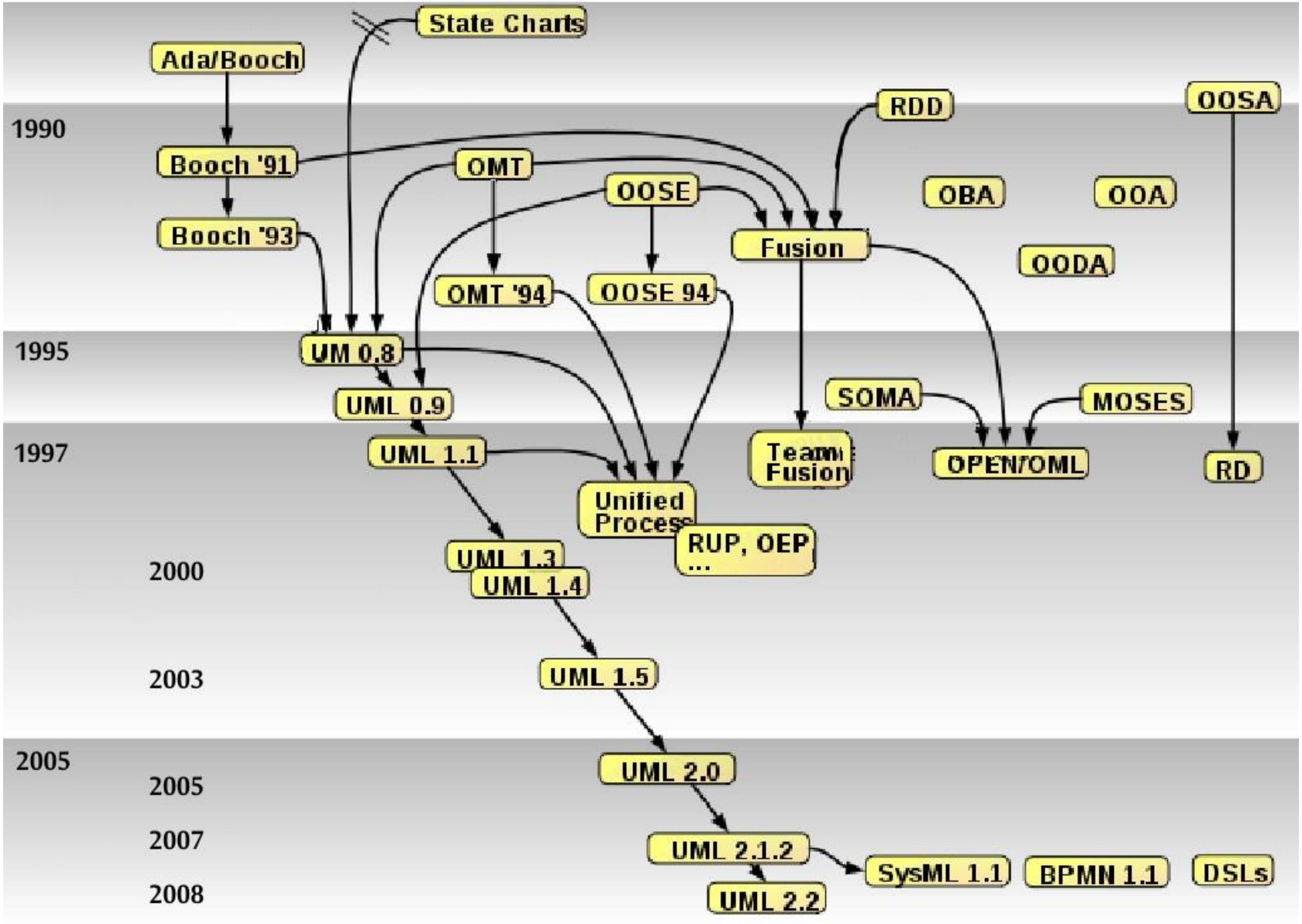
Authors: The Three Amigos

Grady Booch

James Rumbaugh

Ivar Jacobson

Why "Unified" Modeling Language?



What is UML?

UML is a standardized language for specifying, visualizing, constructing and documenting (software) systems

- Specification: the language is supposed to be simple enough to be understood by the clients

- Visualization: models can be represented graphically

- Construction: the language is supposed to be precise enough to make code generation possible

- Documentation: the language is supposed to be widespread enough to make your models understandable by other developers

What is UML?

- UML defines
  - Entities of models and their (possible) relations
  - Different graphical notations to visualize structure and behavior

- A model in UML consist of
  - Diagrams
  - Documentation which complements the diagrams

What UML is *not*!

- Programming language
  - this would bound the language to a specific computing architecture
  - however code generation is encouraged
- Software development process
  - Choose your own process, (e.g. Waterfall-model, V-model, …)
  - Use UML to model & document
- CASE tool specification
  - however tools do exist: Sun, IBM Rose, Microsoft Visio, Borland Together etc.

Diagrams in UML

- UML currently defines 14 types of diagrams
  - 7 types of Structure Diagrams
  - 7 types of Behavior Diagrams

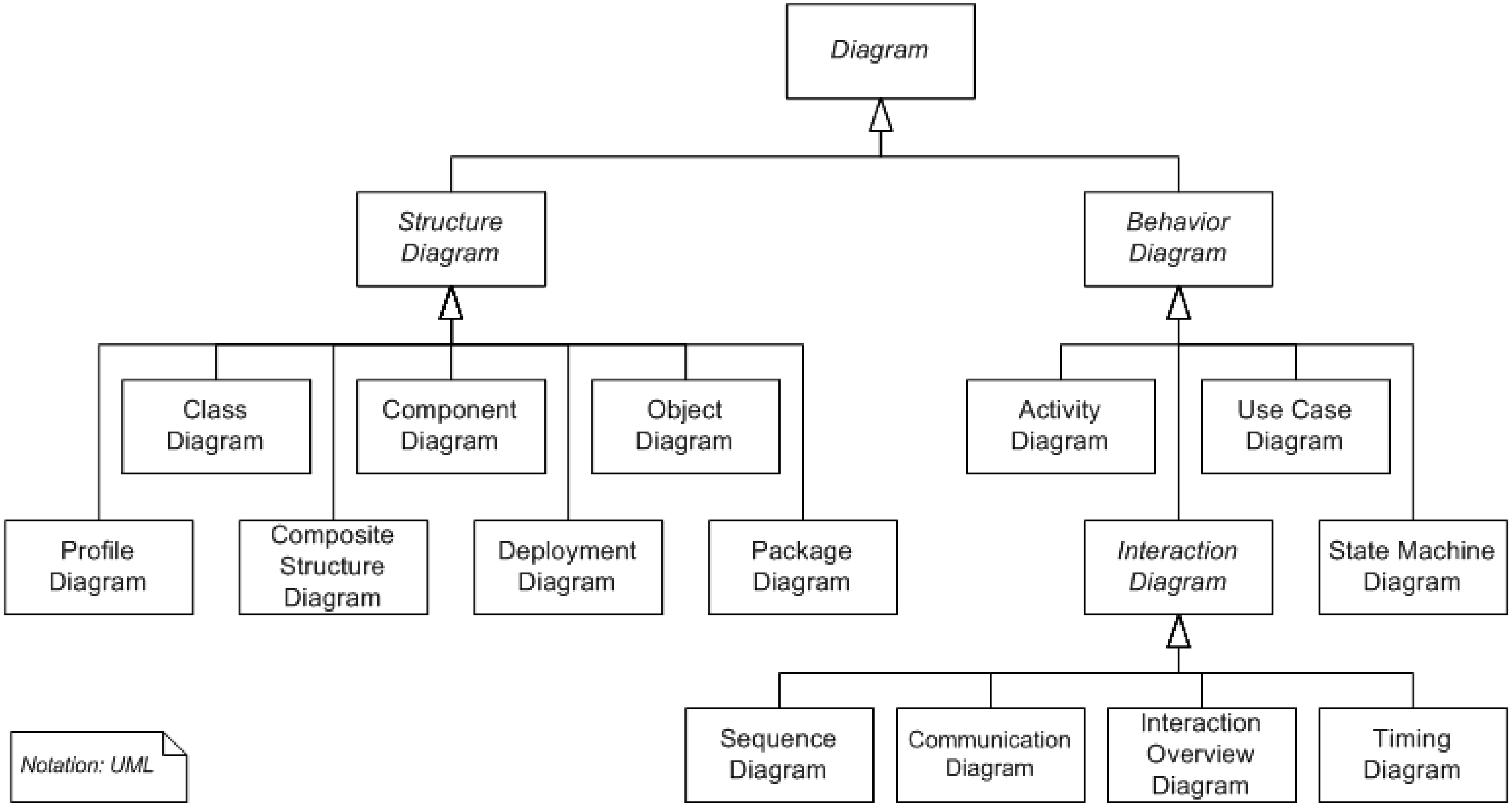
- Different diagrams provide different levels of abstraction
  - High-level structure vs. low-level structure

    Example: *components* vs. *objects*

  - High-level behavior vs. low-level behavior

    Example: *use-case* vs. *feature-call sequence*

Diagrams in UML



Case study*

- ETH cafeteria wants to introduce a card-based payment system
- Students upload money to their card using a special automaton (similar to an ATM)
- Using their cards, students pay cashless in the cafeteria

* inspired by: http://www.fbi.h-da.de/labore/case/uml.html

Use Case diagram

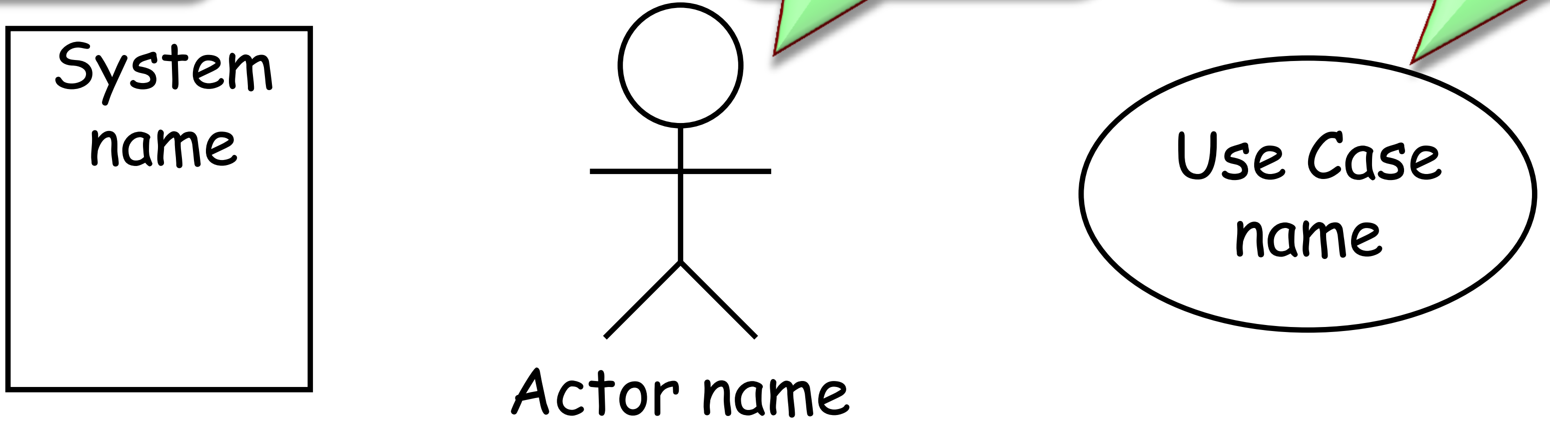
- Use Case diagrams
  - High-level abstraction of the system's external behavior
  - From the client's perspective
  - What the client plans to do with the system

ents of a Use Ca

Boundaries of the planned system

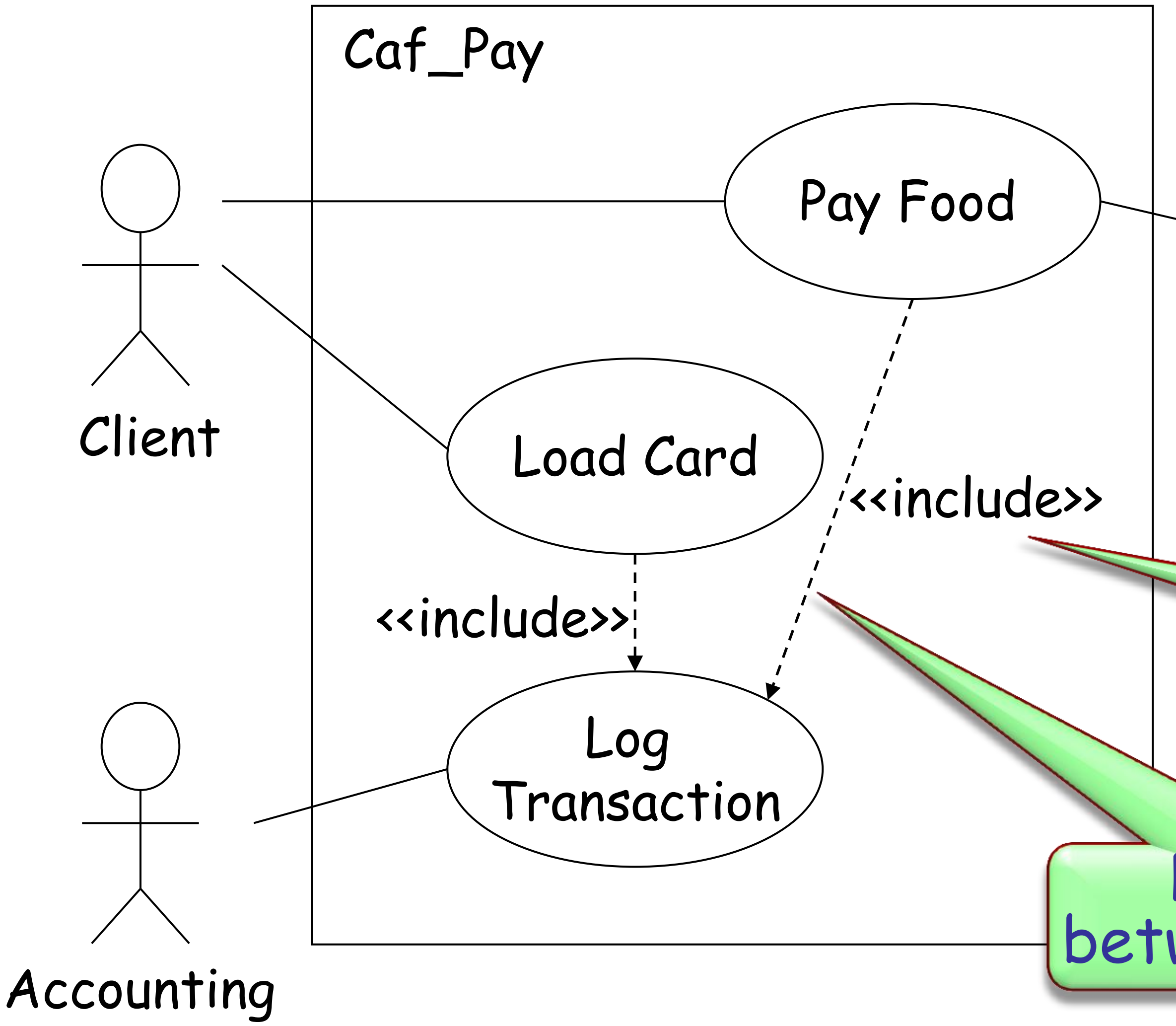
Person or System that interacts with the planned system

Task that should be executed by the planned system



Use Case diagram - Example

- Use Case diagram for the payment system



Association between Actor and Use Case

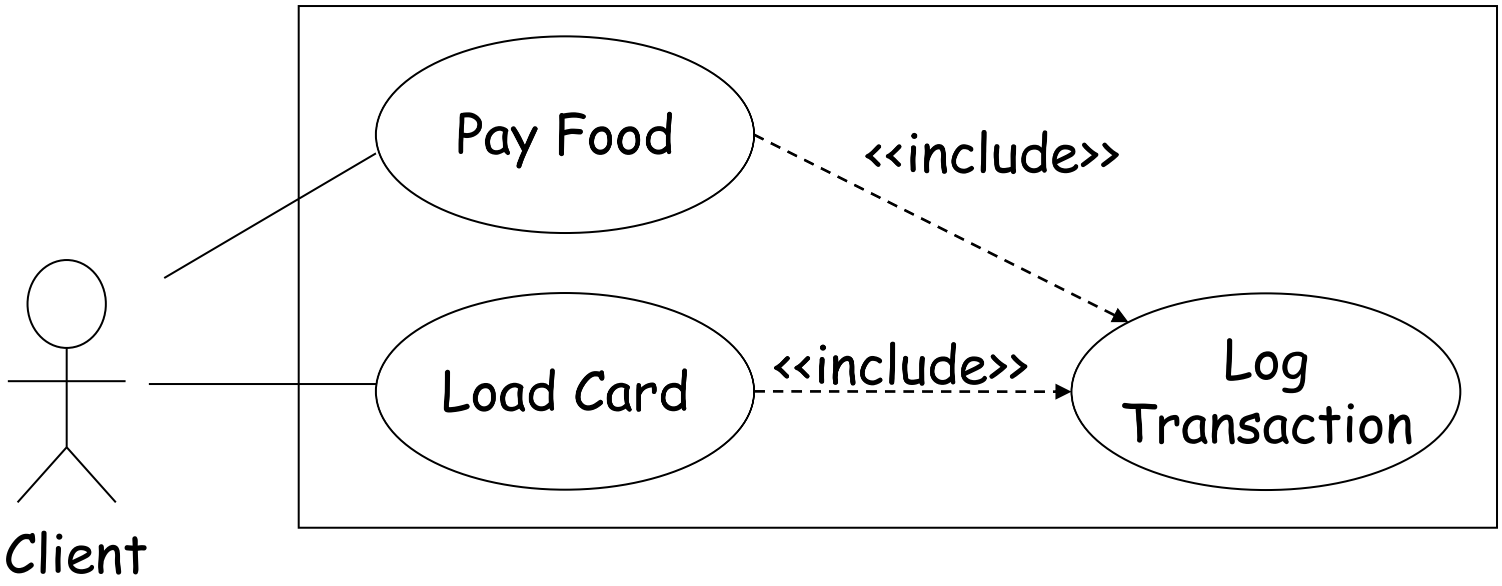
Stereotype

Dependency between Use Cases

Use Case diagrams: include-association

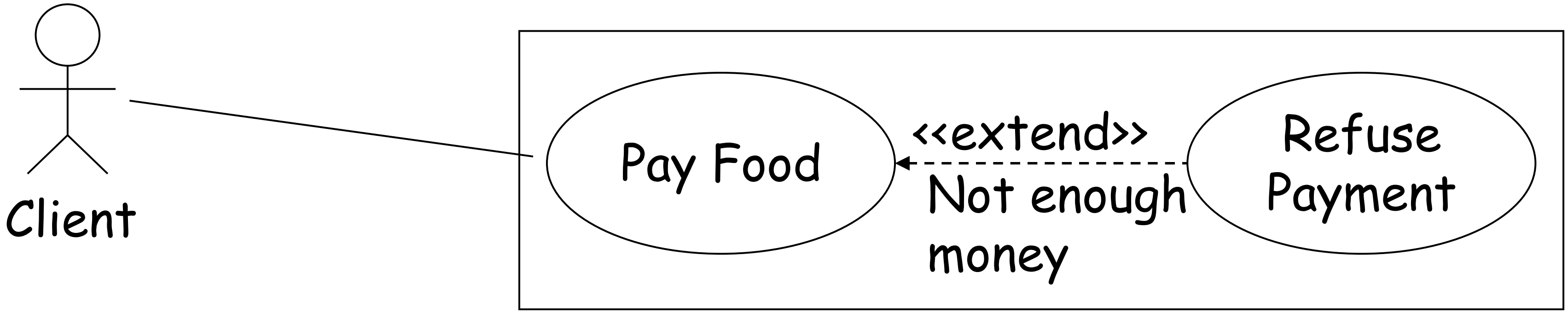
<<include>> stereotype to include use cases:

reusing common functionality, no duplicates



"Pay Food" and "Load Card" use the functionality provided by "Log Transaction"

Use Case diagrams: extend-association



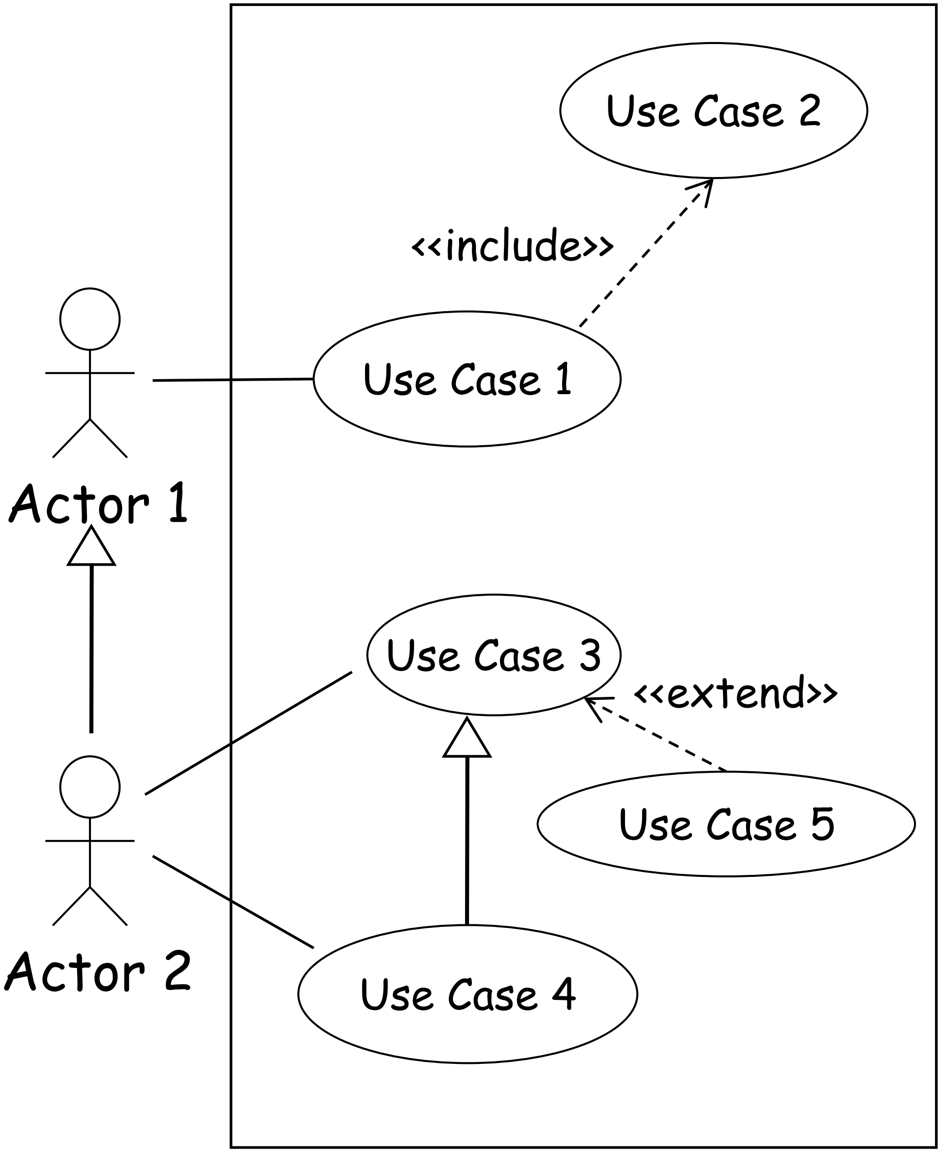
<<extend>> stereotype to provide special case

Normal case specifies point at which the behavior may diverge (**extension point**)

Extending case specifies condition under which the special case applies (as **entry condition**)

Elements of Use Case diagrams

- Entities:
  - actors
  - use cases
- Relations:
  - association between an actor and a use case
  - generalization between actors
  - generalization between use cases
  - dependencies between use cases
- Comments:
  - system boundaries



Use Case specification

- Each Use Case shown in a diagram should be accompanied by a textual specification

- The specification should follow the scheme:
  - Use Case name
  - Actors
  - Entry Condition
  - Normal behavior
  - Exceptions
  - Exit Condition
  - Special Requirements (e.g. non-functional requirements)

Use Case specification

- Example for „ Pay Food“ Use Case

Name: Pay Food

Actors: Client, Teller

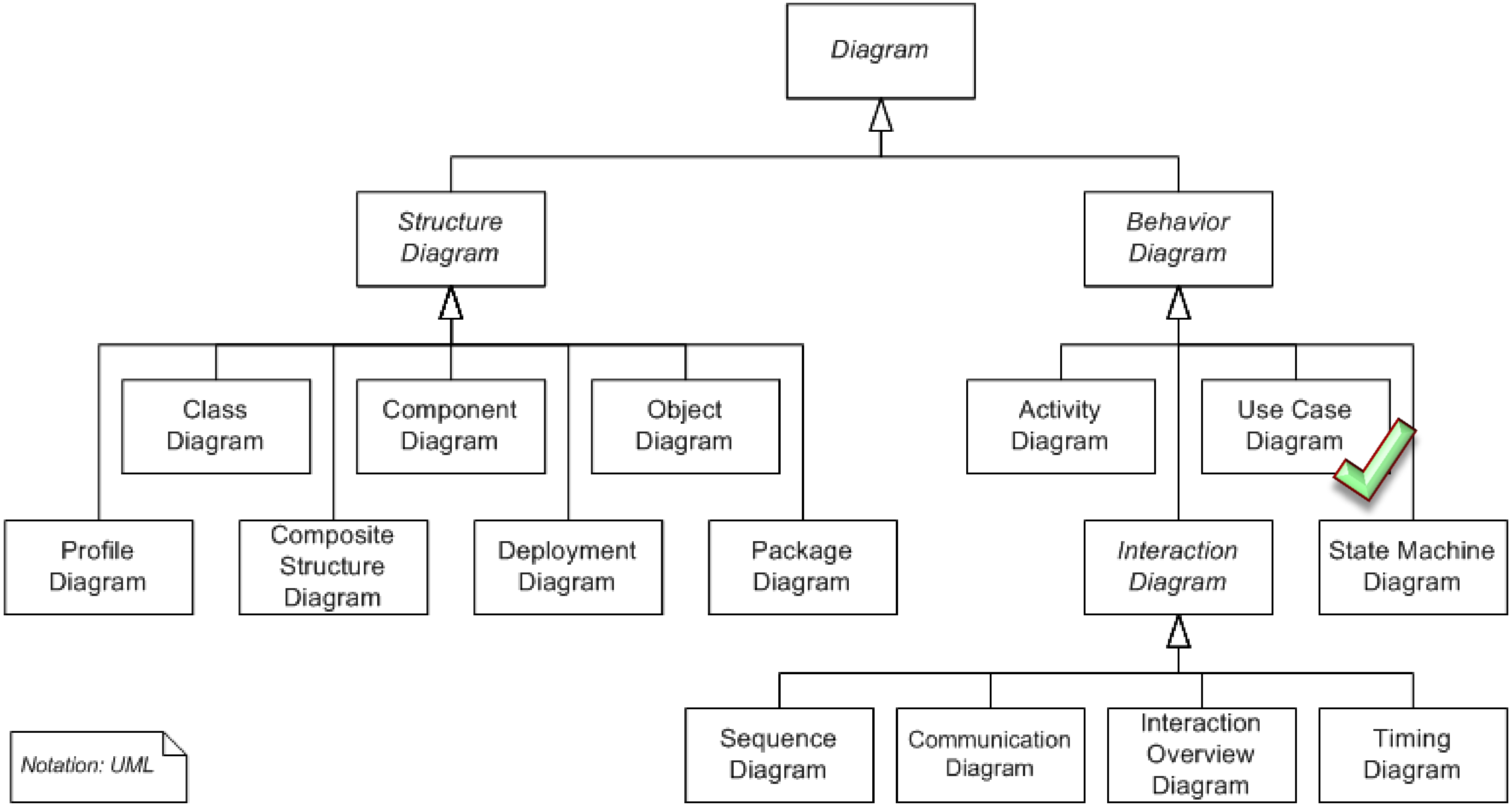
Entry Condition: Client has food and wants to pay it

Normal behavior: Teller types in food; Total amount is shown on display; Client puts card into reading device; Amount gets withdrawn; If not enough money on card, then an error message is shown; Return card to client

Exceptions: If card is not readable, then show error message and return card; If power failure while card in reading device, wait until power is back and return card – payment needs to be redone

Exit Condition: Client has paid the food and gets the card back

Diagrams in UML



Activity diagrams

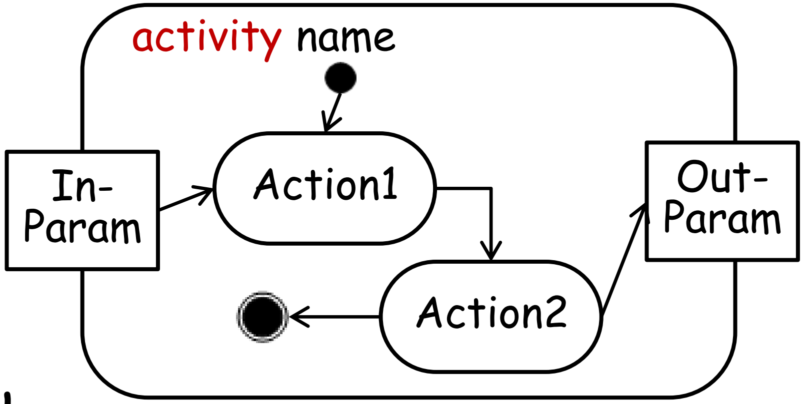
- Activity diagrams are used to model (work)flows
- They are used visualize complex behavior, e.g.
  - Business process
  - Algorithms (though less common)
- Tokens are used to determine the flow, similar to Petri-nets
- A common usage: detailed modeling of Use Cases

Elements of Activity diagrams

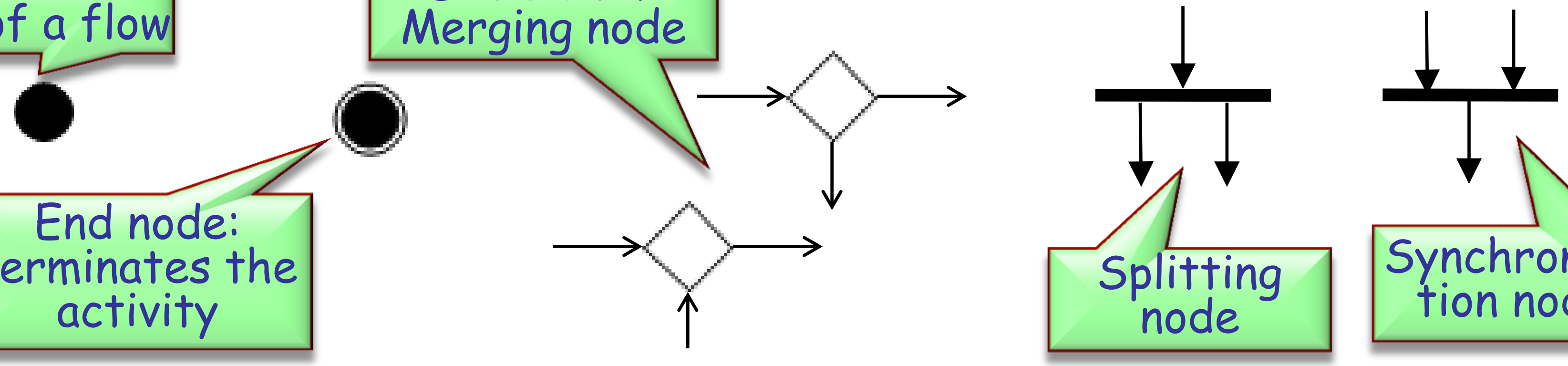
- **Action**: atomic element, no further splitting possible

- **Activity**: can contain activities, actions, control nodes

Allows refinement



- **Control nodes**: used to denote control struture in the flowgraph



Initial node: start of a flow

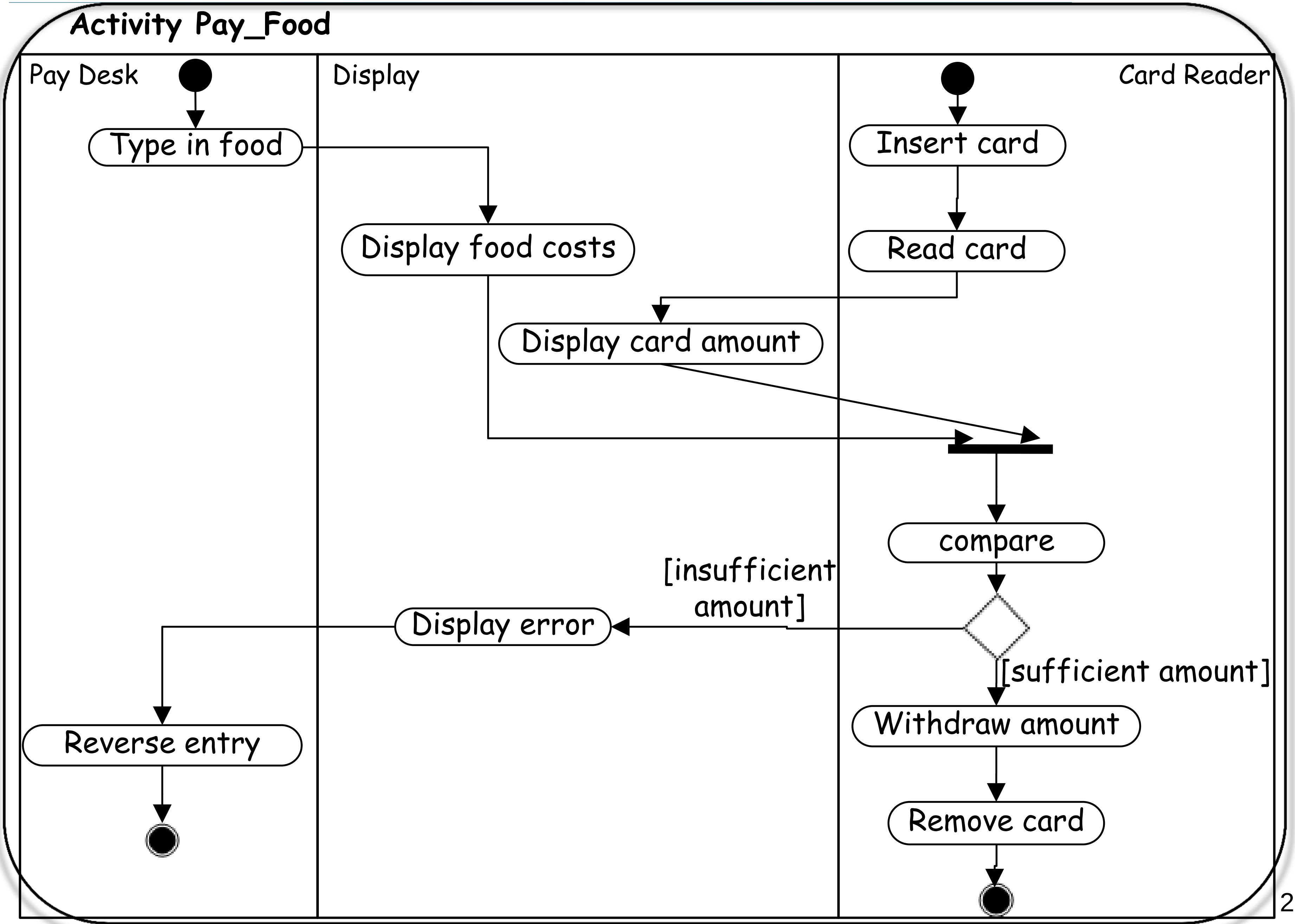
End node: terminates the activity

Decision- / Merging node

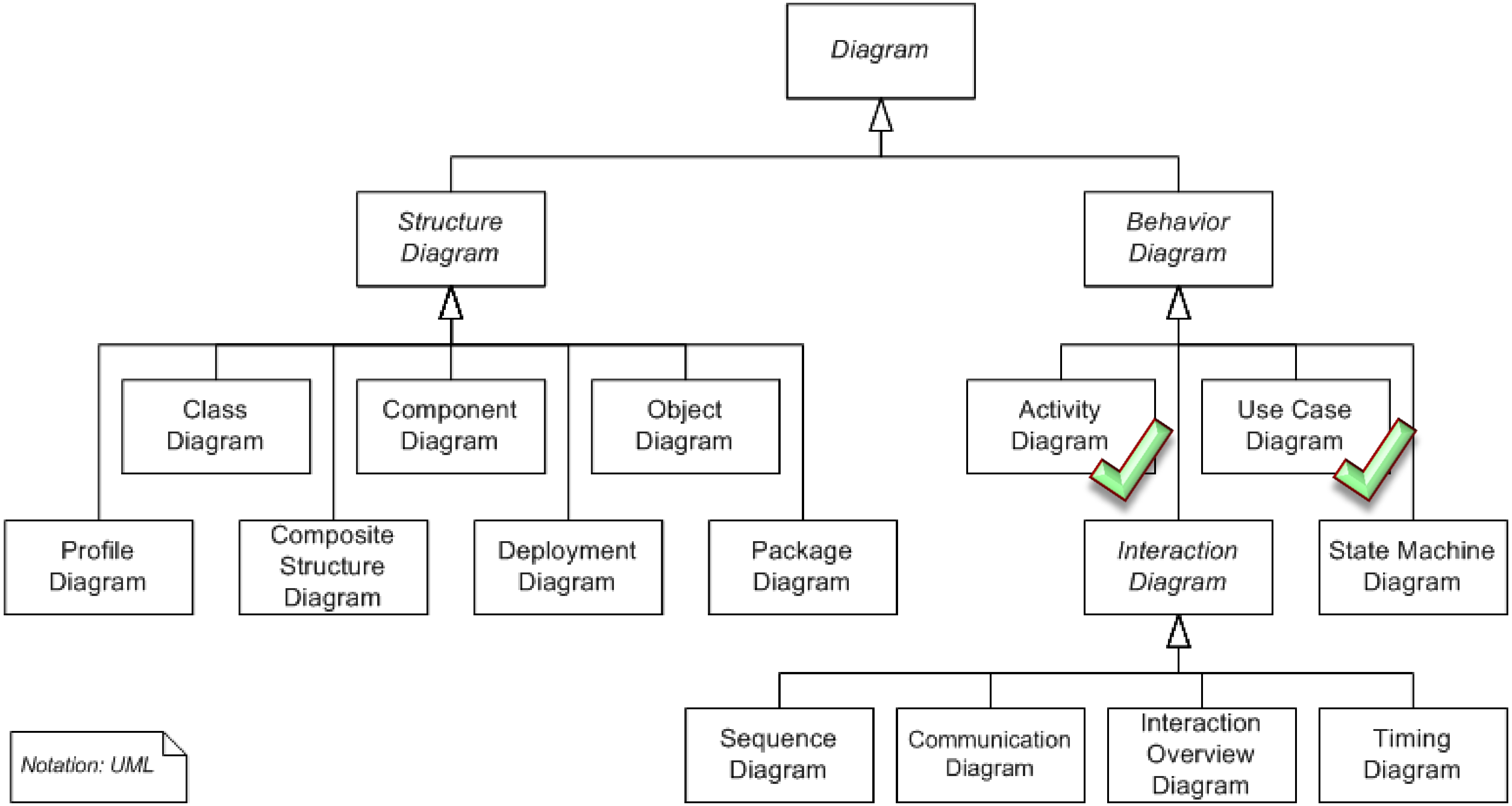
Splitting node

Synchronization node

An activity diagram for the case study



Diagrams in UML



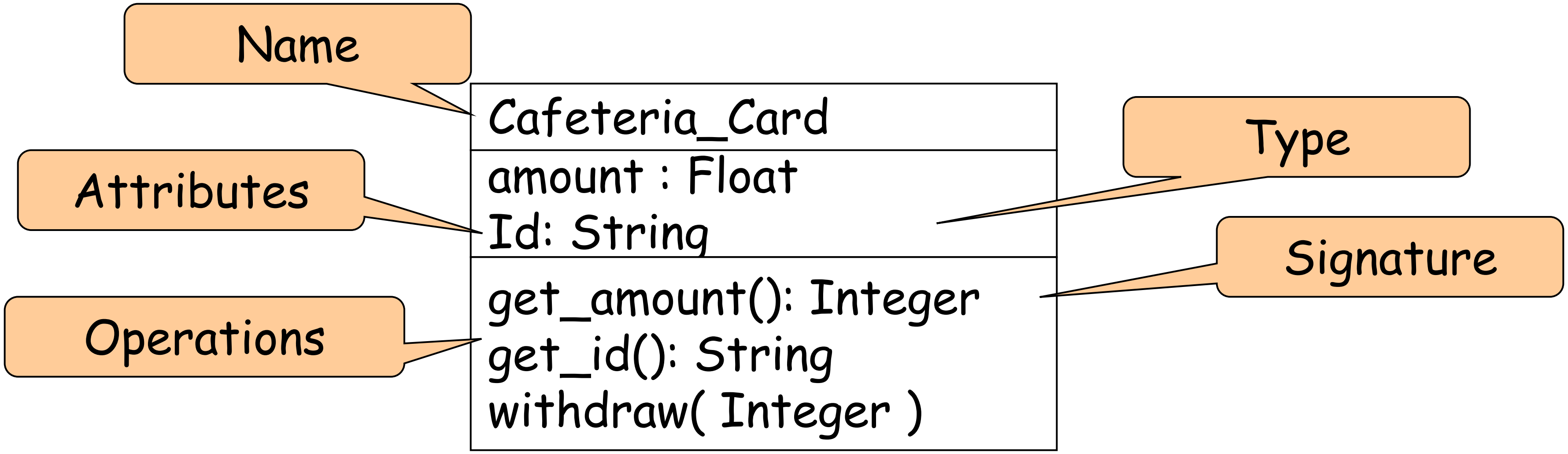
UML Class diagrams

- Keep in mind:
  - Use Cases represent an external view of the system's behavior
  - Classes represent inner structure of the system

  → No correlation between use cases and classes

- Class diagrams are used at different levels of abstraction with different levels of details
  - Early phase: identifying classes and their relations in the problem domain (high-level, no implementation details)
  - Implementation phase: high level of detail (attributes, visibility, …), all classes relevant to implement the system

Classes



Name

Attributes

Operations

Type

Signature

Cafeteria_Card

amount : Float
Id: String

get_amount(): Integer
get_id(): String
withdraw(Integer)

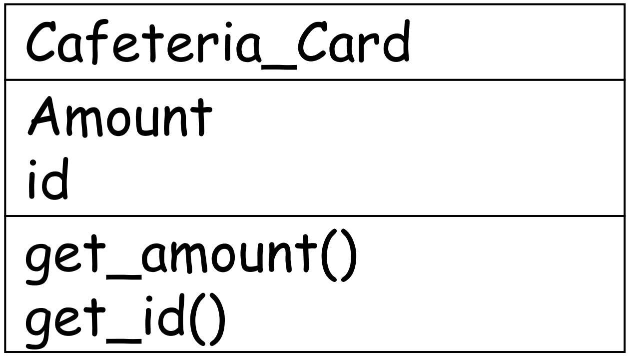
A class encapsulates **state** (attributes) and **behavior** (operations)

- Each attribute has a type
- Each operation has a signature

The class name is the only mandatory information

More on classes

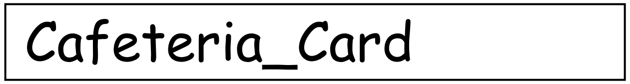
Valid UML class diagrams



Cafeteria_Card

Amount
id

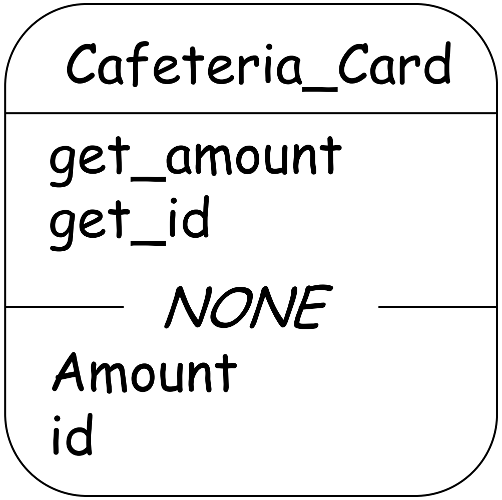
get_amount()
get_id()



Cafeteria_Card

Corresponding BON diagram

- No distinction between attributes and operations
  (uniform access principle)



Cafeteria_Card

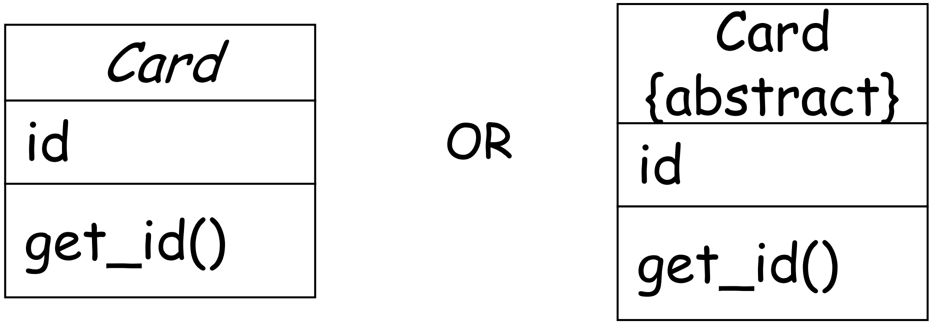
get_amount
get_id

NONE

Amount
id

More on classes

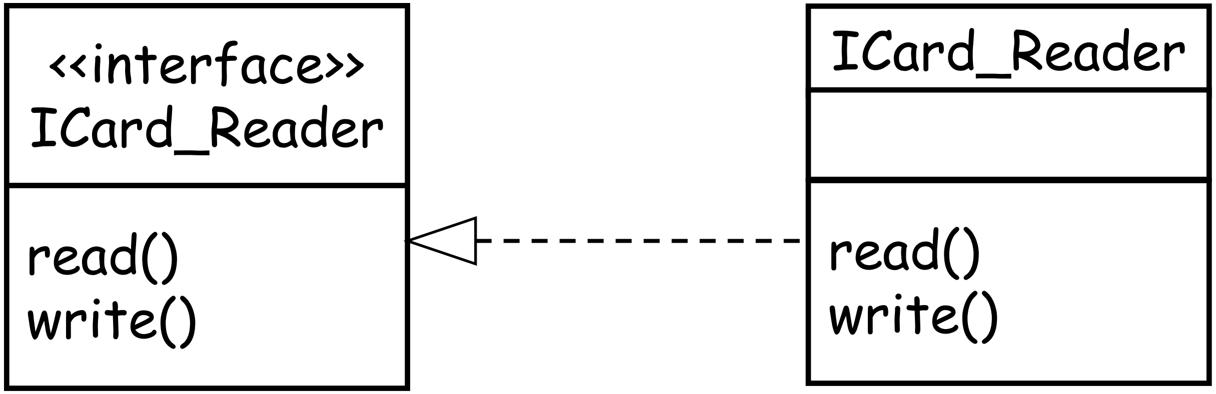
- Abstract classes have a italicized class name or {abstract} property (also applicable to operations)



- Parameterized classes

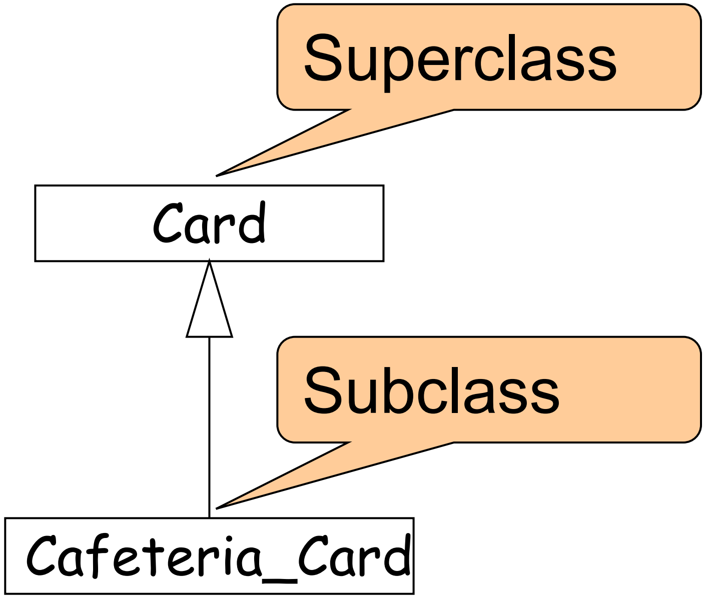
Interface classes

- Interface classes have a keyword <<interface>>
- Interfaces have no attributes
- Classes implement an interface using an implementation relation



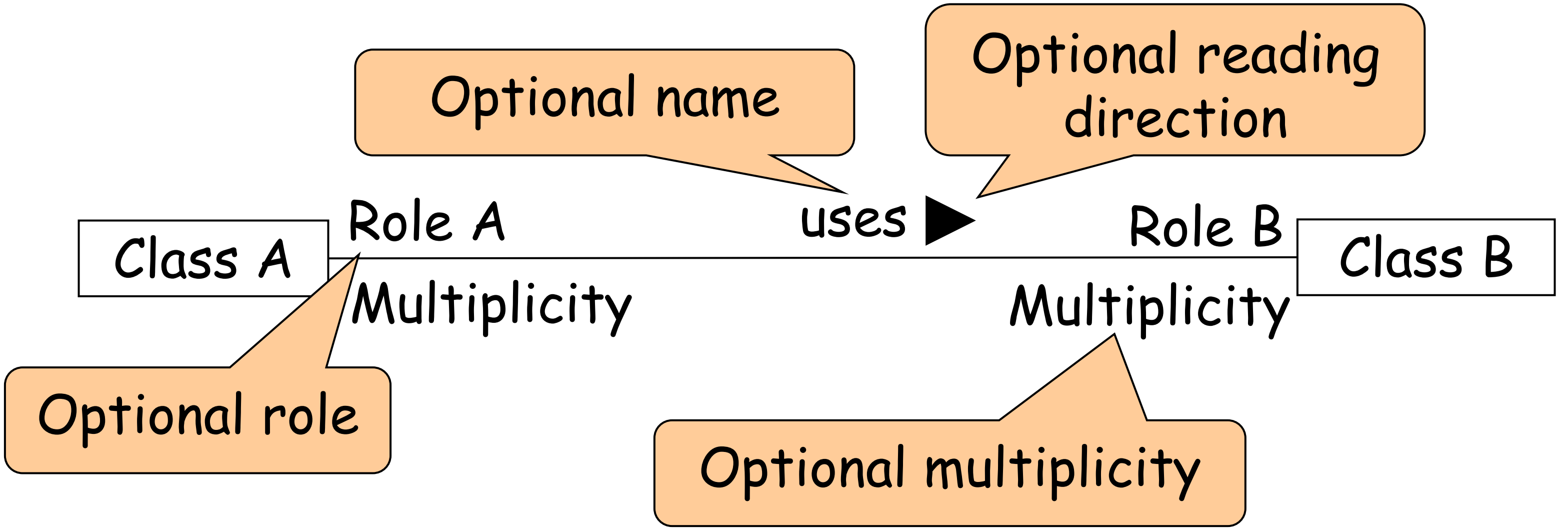
Generalization and specialization

- Generalization expresses a **kind-of** ("is-a") **relationship**
- Generalization is implemented by **inheritance**
  - The child classes inherit the attributes and operations of the parent class
- Generalization simplifies the model by **eliminating redundancy**



Associations

- A line between two classes denotes an association
- An association is a type of relation between classes
- Objects of the classes can communicate using the association, e.g.
  - Class A has an attribute of type B
  - Class A creates instances of B
  - Class A receives a message with argument of type B

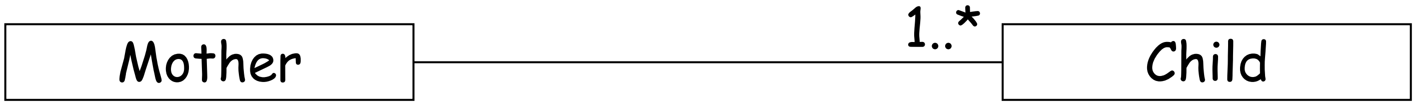


Association multiplicity

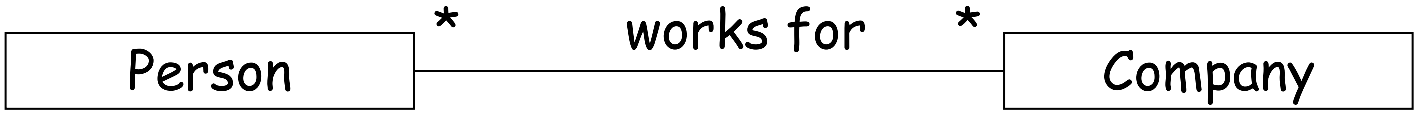
- Multiplicity denotes how many objects of the class take part in the relation
  - 1-to-1


City
1
is capital of
1
Country

  - 1-to-many

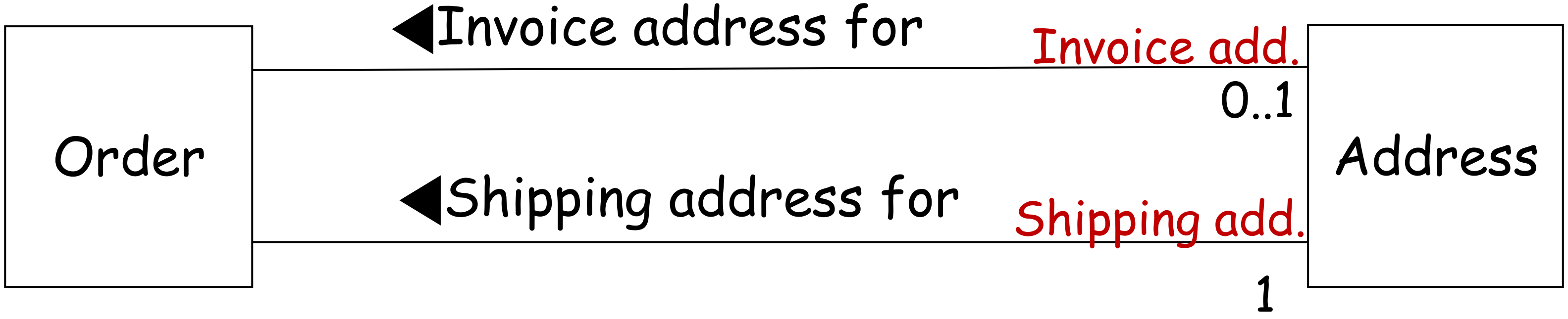

Mother
1..*
Child

  - many-to-many


Person
*
works for
*
Company

Association roles

- Different instances of an class can be differentiate using roles
- Example: Invoice and shipping address are both addresses



Order

◀Invoice address for

Invoice add.

0..1

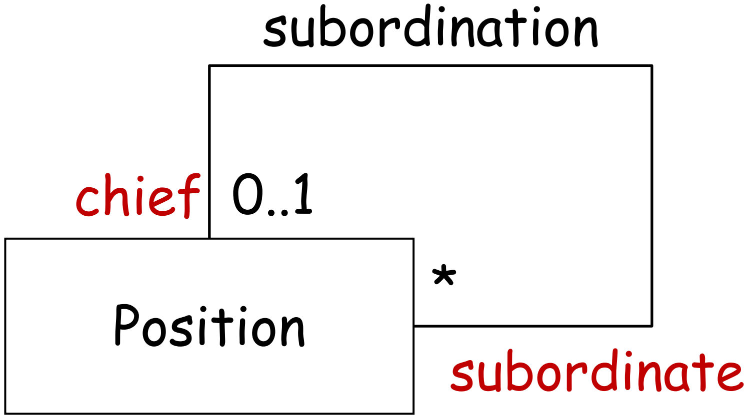
◀Shipping address for

Shipping add.

1

Address

- Example: Position hierarchy



subordination

chief

0..1

Position

*

subordinate

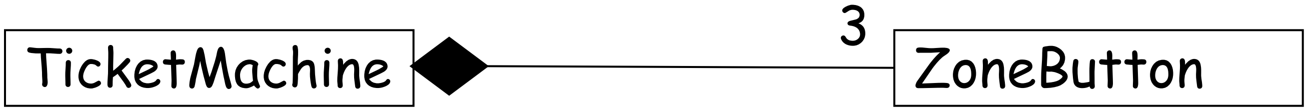
Special associations

- **Aggregation** – "part-of" relation between objects
  - Component can be part of multiple aggregates
  - Component can be created and destroyed independently of the aggregate

Aggregate

Component

- **Composition** – strong aggregation
  - A component can only be part of a single aggregate
  - Exists only together with the aggregate



More on associations

- **Ordering** of an end – whether the objects at this end are ordered
- **Changeability** of an end – whether the set of objects at this end can be changed after creation

Polygon

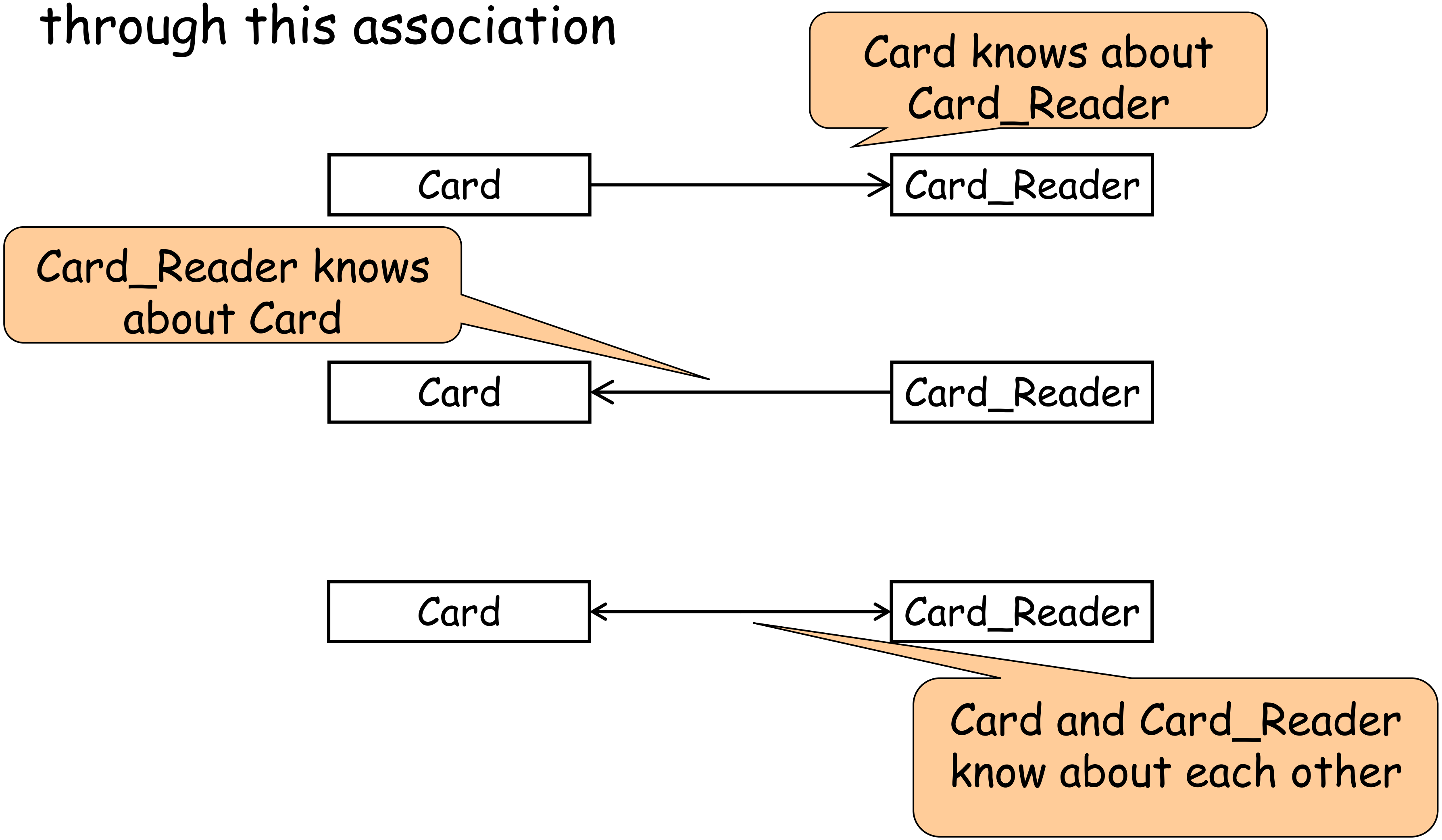
3..*

{frozen,{ordered}

Point

Navigability of association

- Associations can be directed
- Direction denotes whether objects can be accessed through this association



Card knows about Card_Reader

Card → Card_Reader

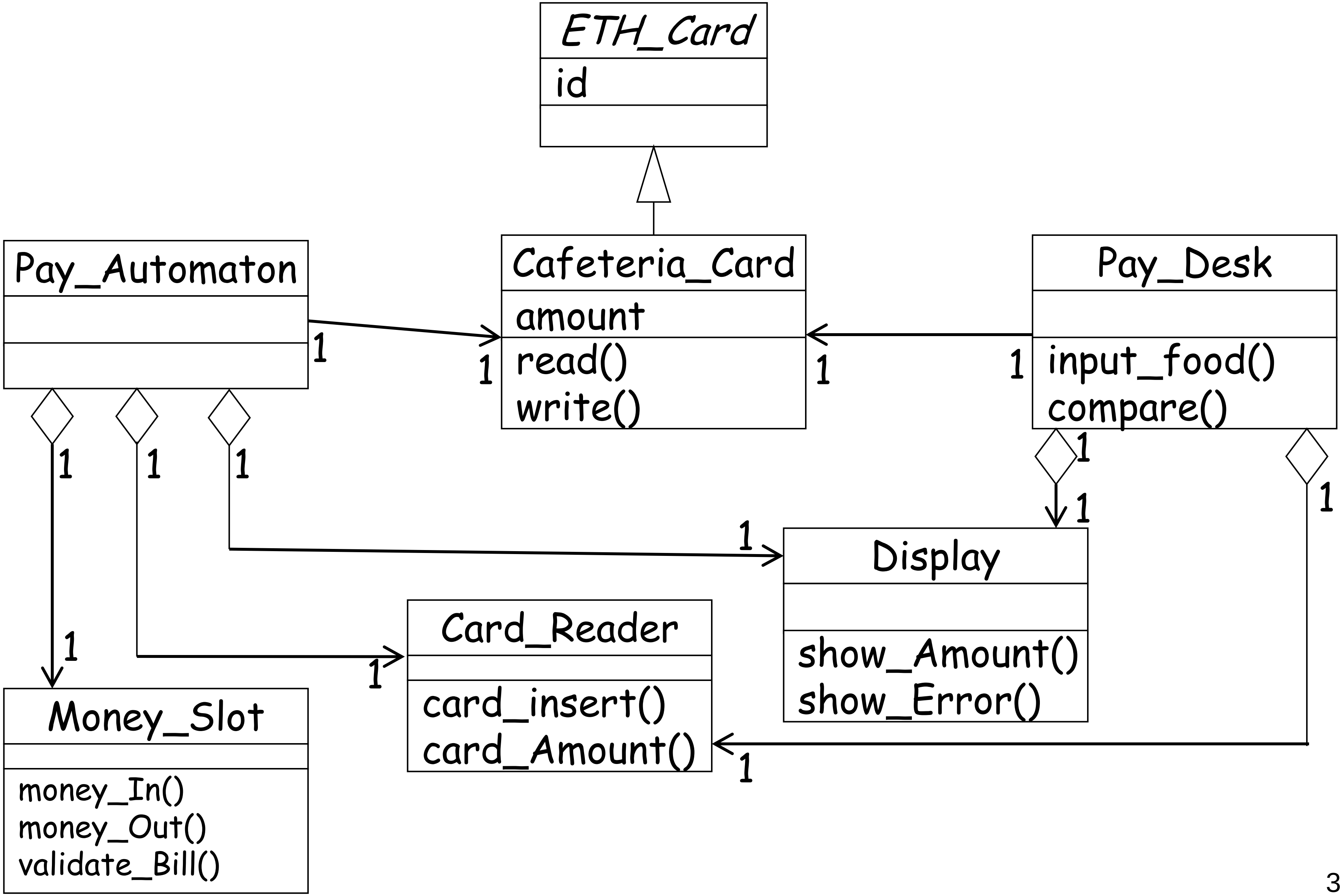
Card_Reader knows about Card

Card ← Card_Reader

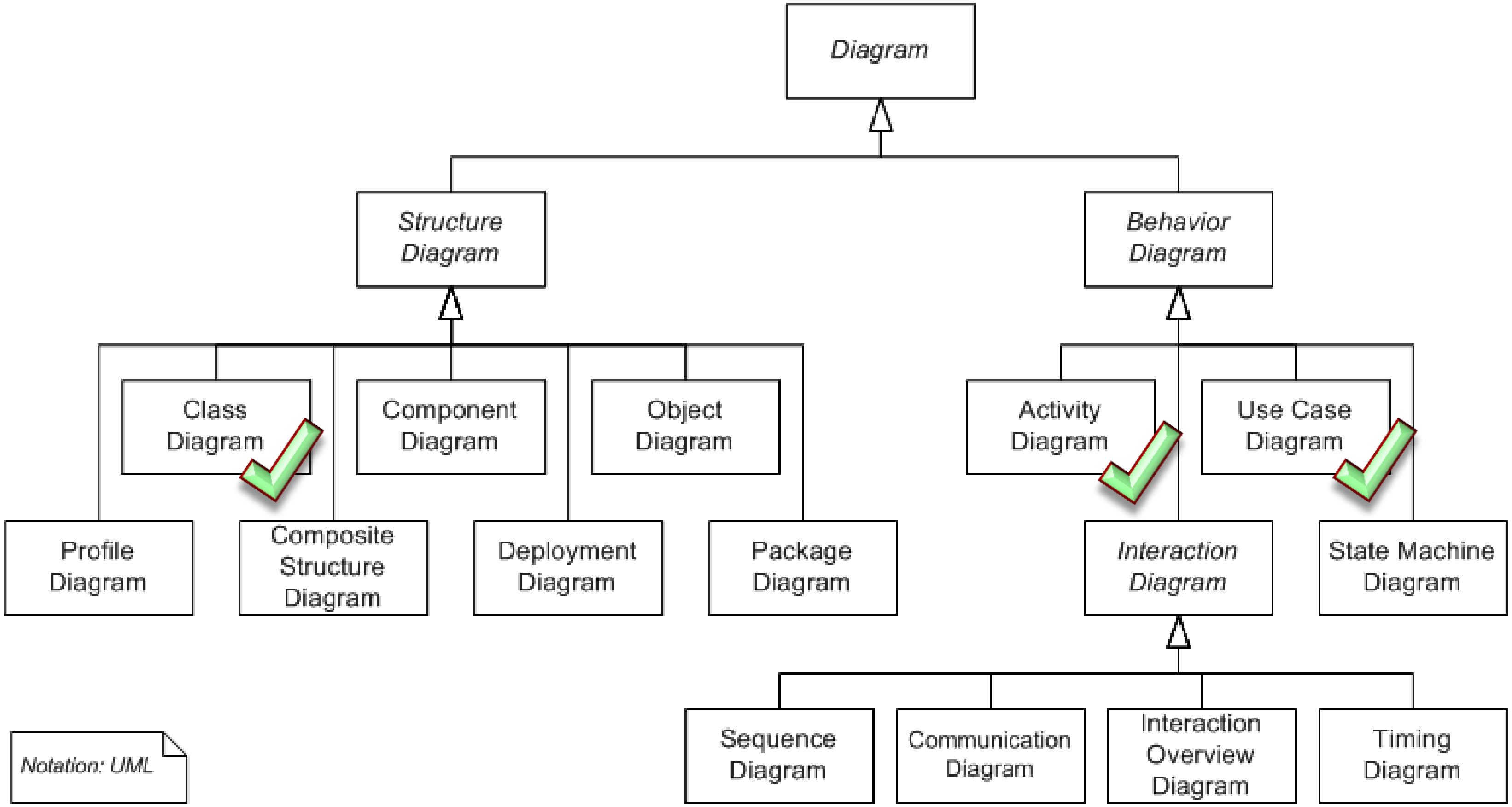
Card ↔ Card_Reader

Card and Card_Reader know about each other

Class diagram for the case study

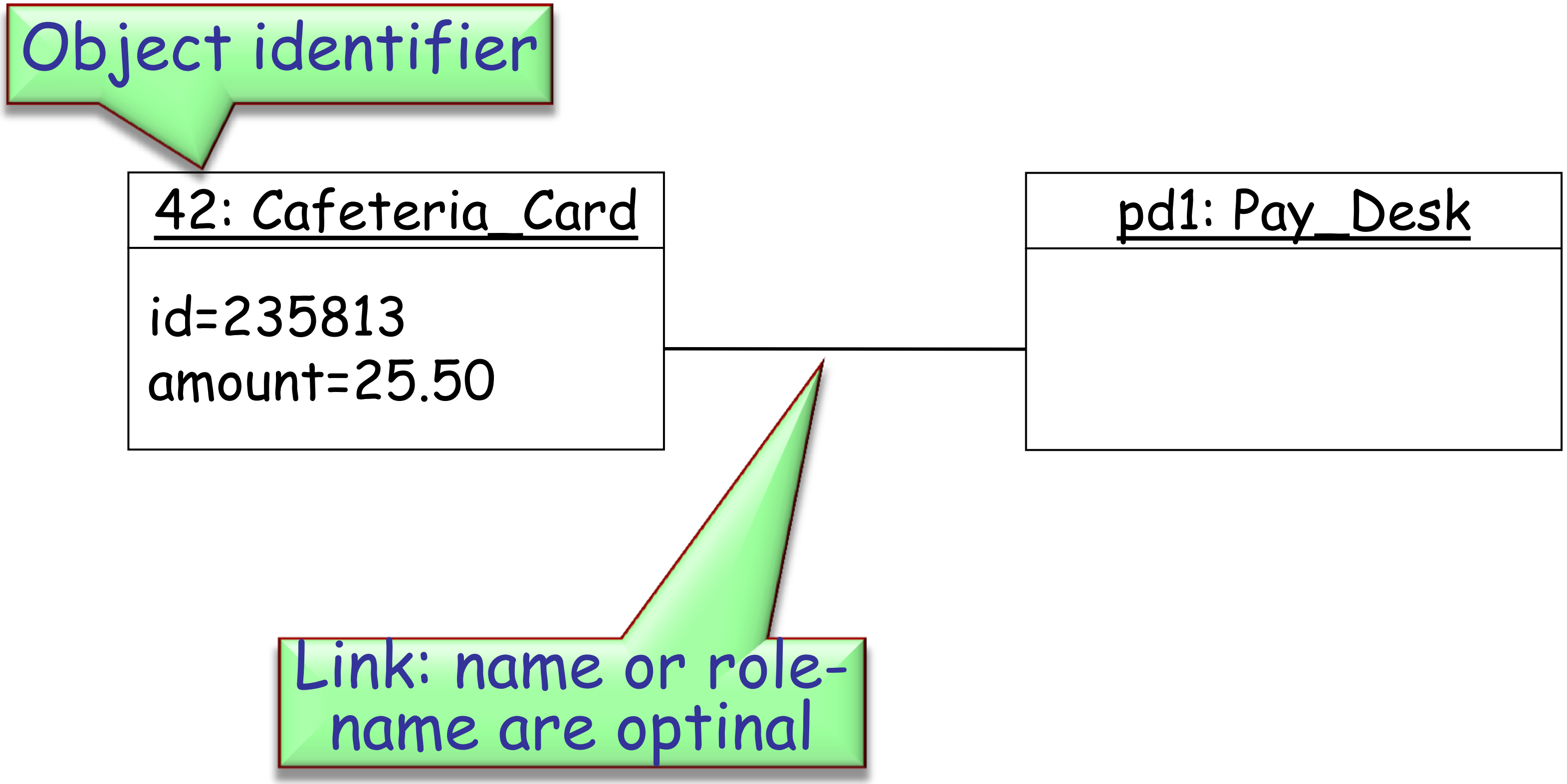


Diagrams in UML

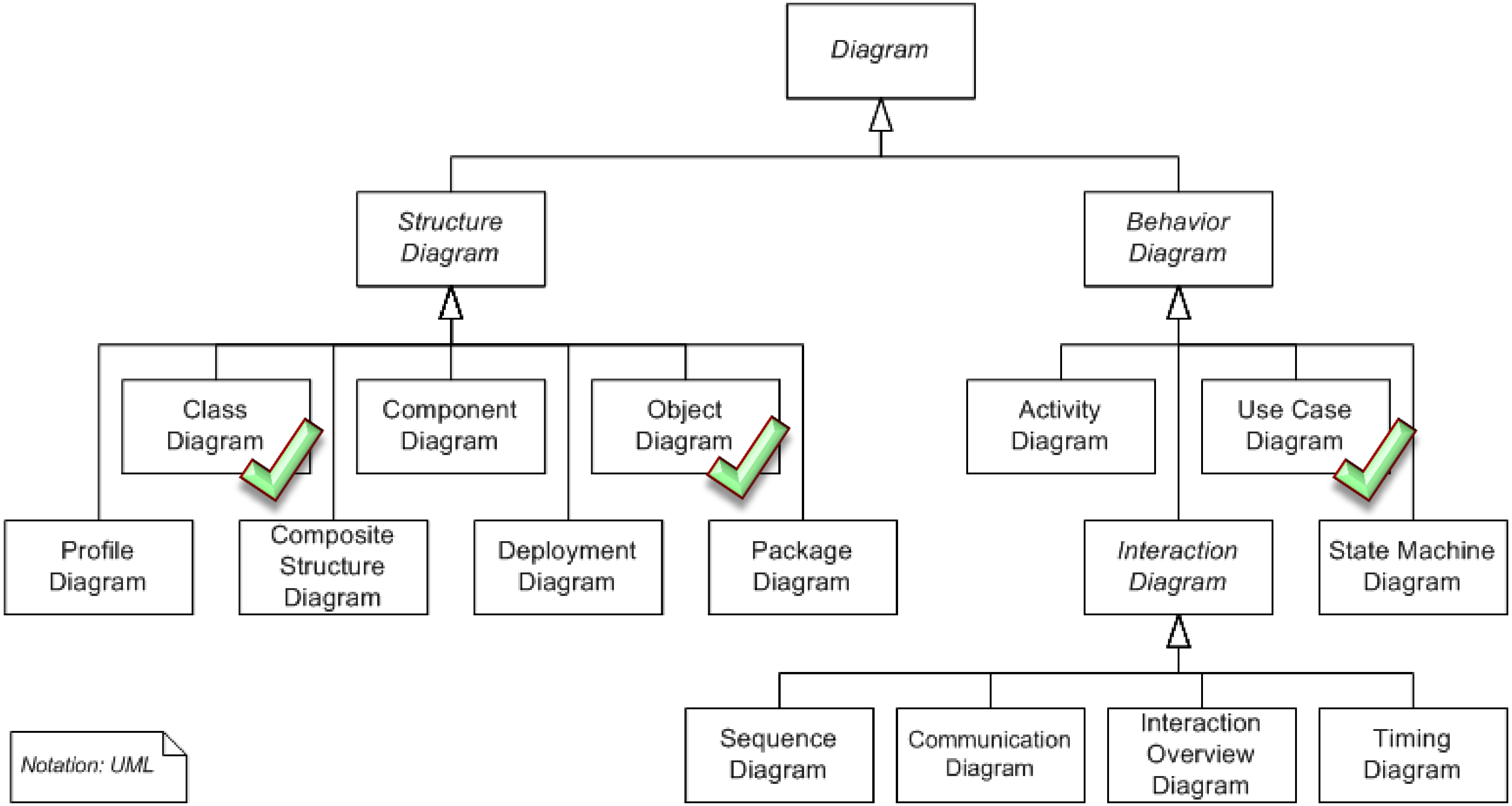


UML Object diagrams

- An Object diagram is used to denote a snapshot of the system at runtime
- It shows the existing objects, their attribute values and relations at that particular point of time



Diagrams in UML



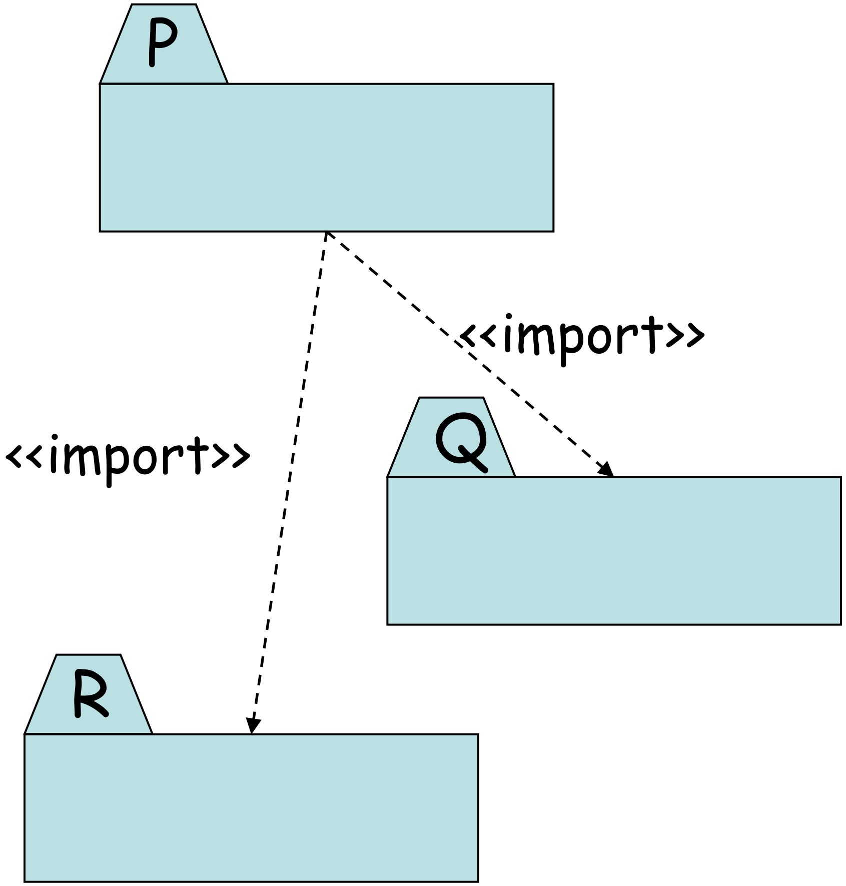
UML packages

A package is a UML mechanism for **organizing elements** into groups

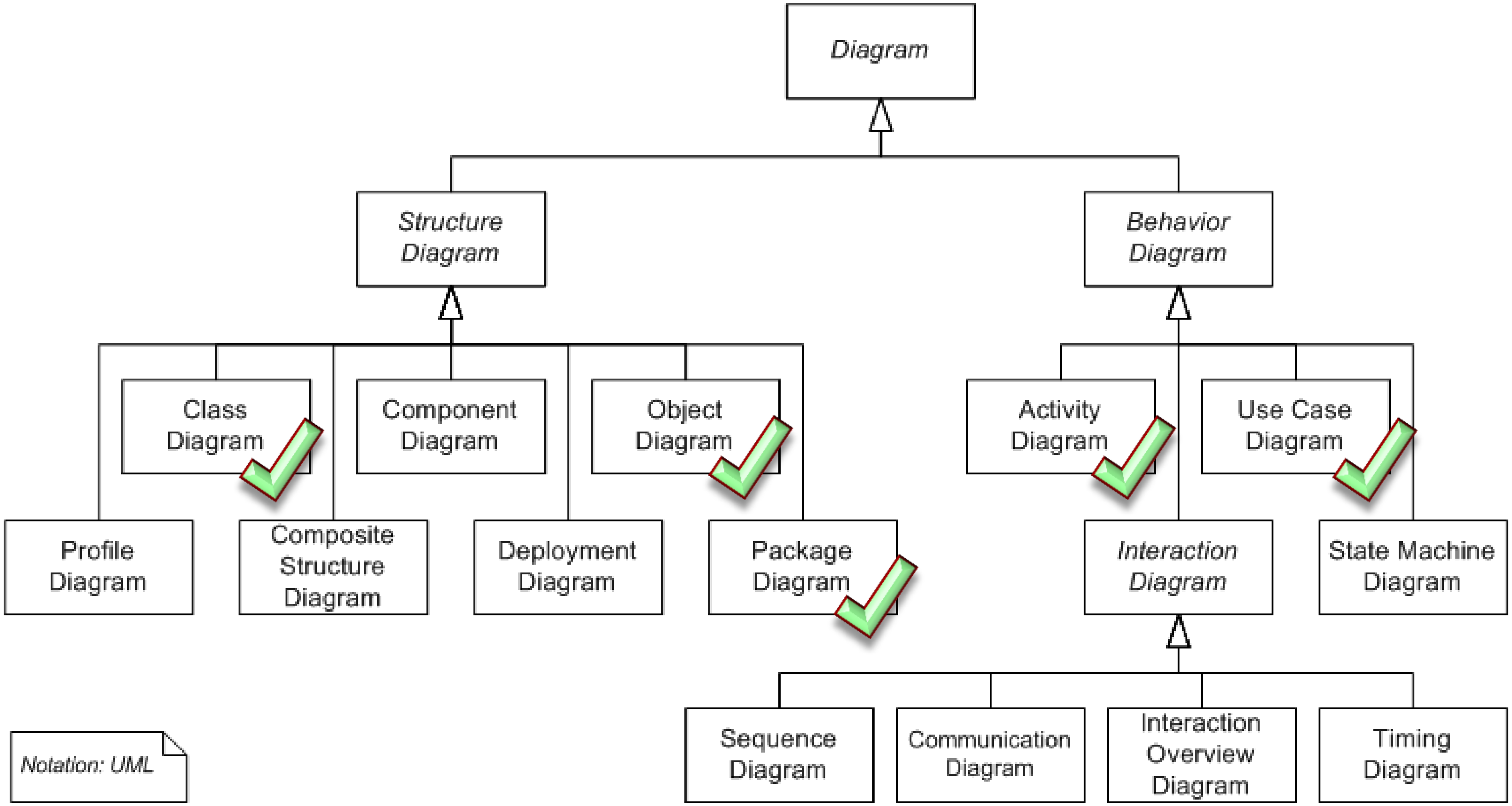
- Usually not an application domain concept
- Increase readability of UML models

Decompose complex systems into subsystems

- Each subsystem is modeled as a package

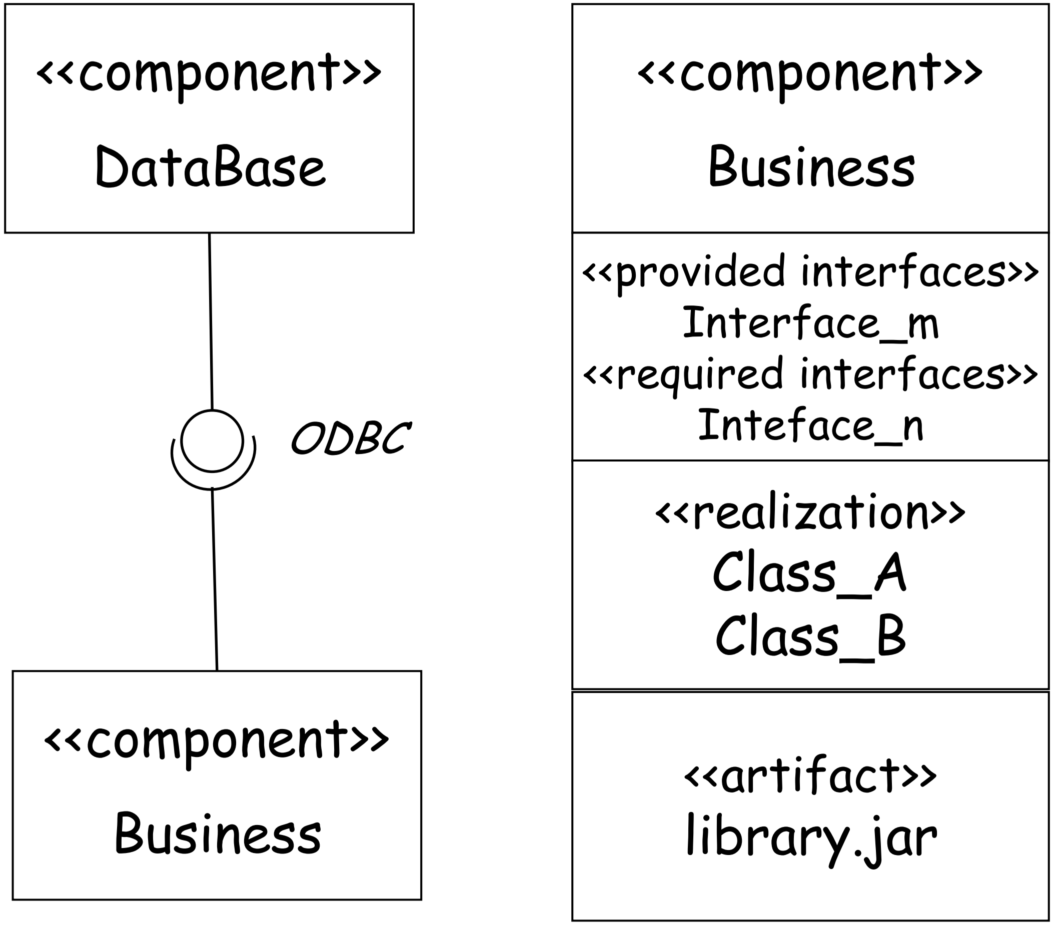


Diagrams in UML



Component diagrams

- Entities:
  - components
    - programs
    - documents
    - files
    - libraries
    - DB tables
  - interfaces
  - classes
  - objects
- Relations:
  - dependency
  - association (composition)
  - implementation



<<component>>
DataBase

ODBC

<<component>>
Business

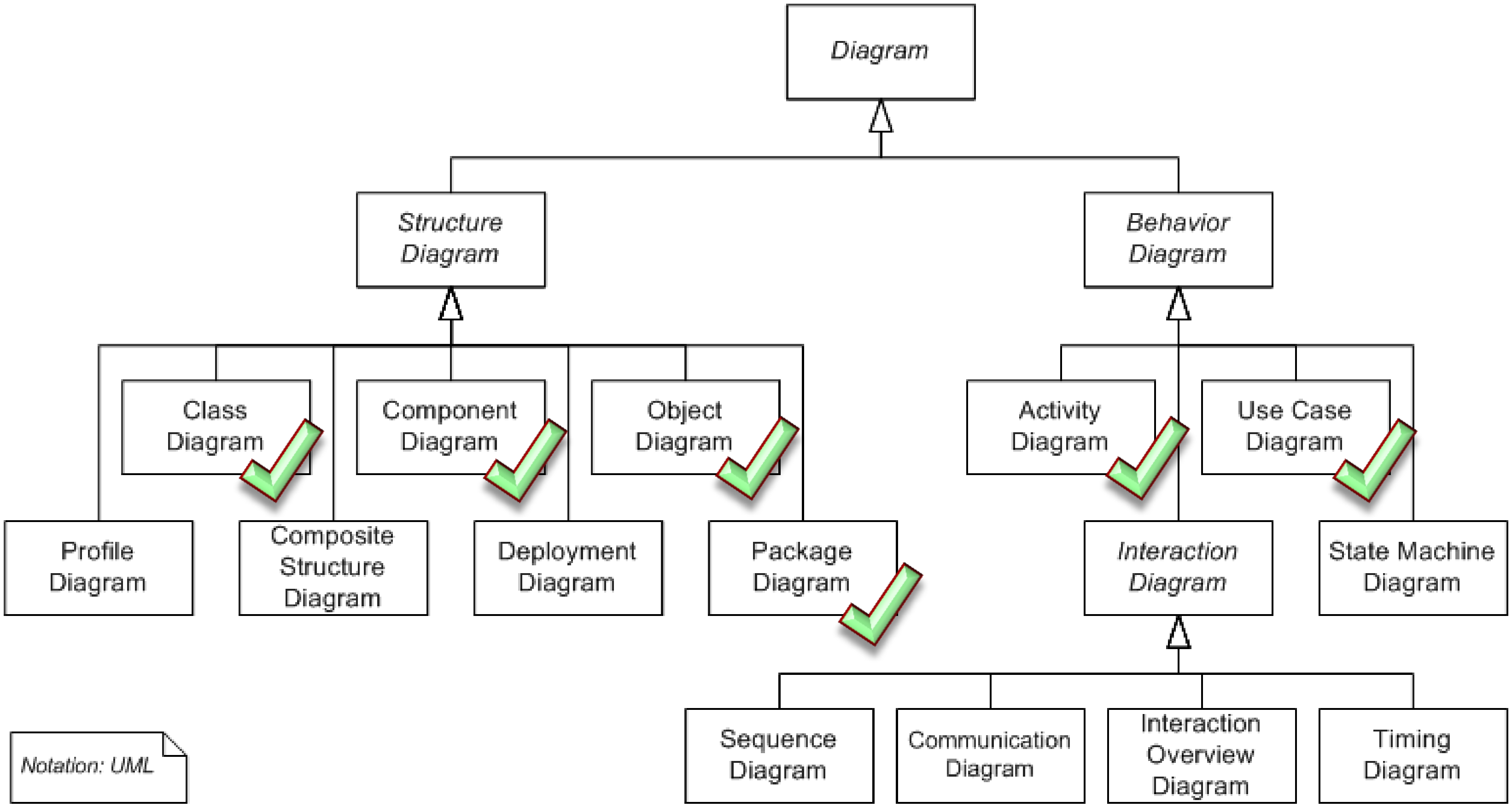
<<component>>
Business

<<provided interfaces>>
Interface_m
<<required interfaces>>
Inteface_n

<<realization>>
Class_A
Class_B

<<artifact>>
library.jar

Diagrams in UML



Overview

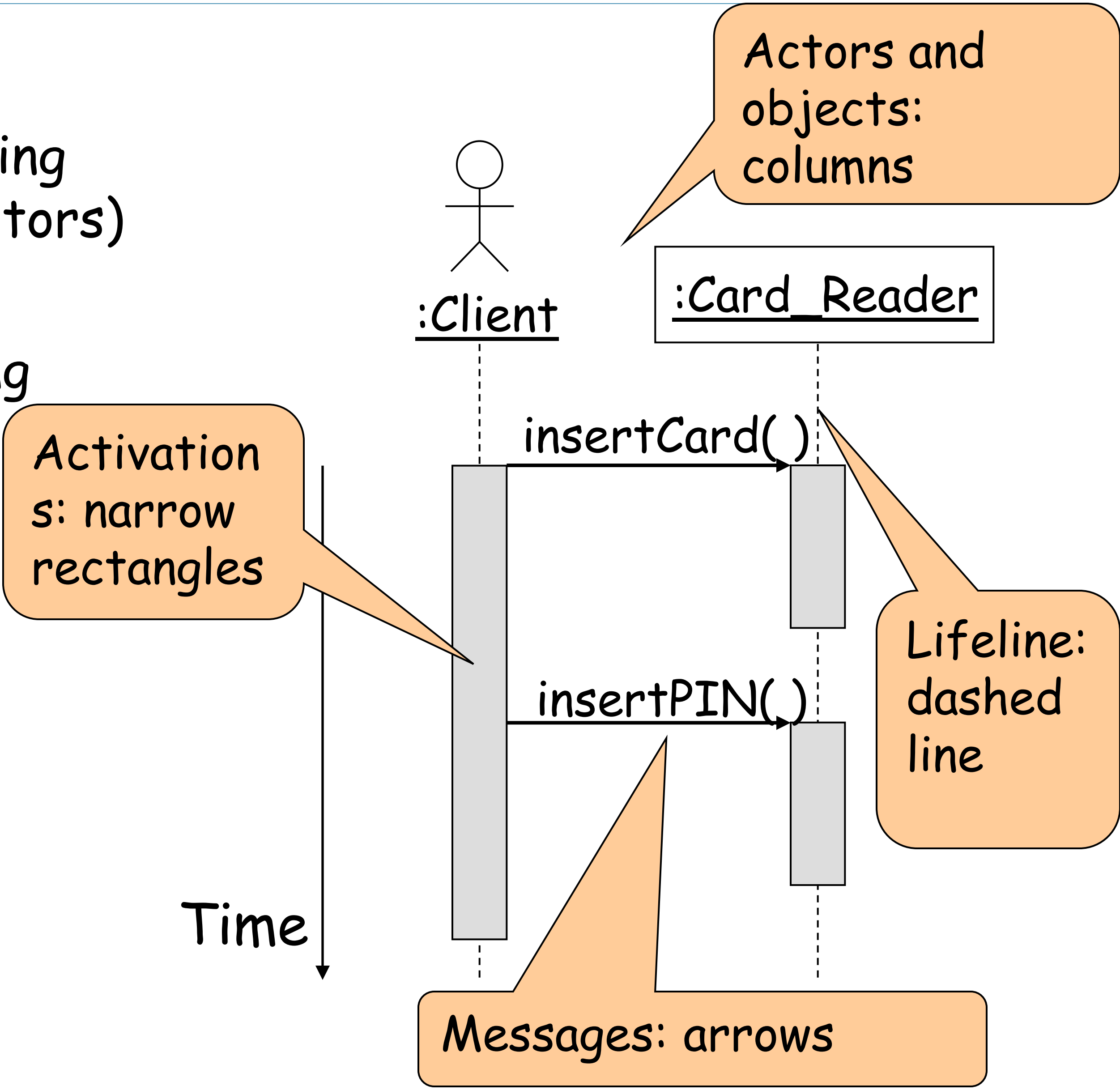
- We will now look at two more diagrams which are used to model the behavior of a system.

- Sequence diagrams: used to describe the interaction of objects and show their “communication protocol”

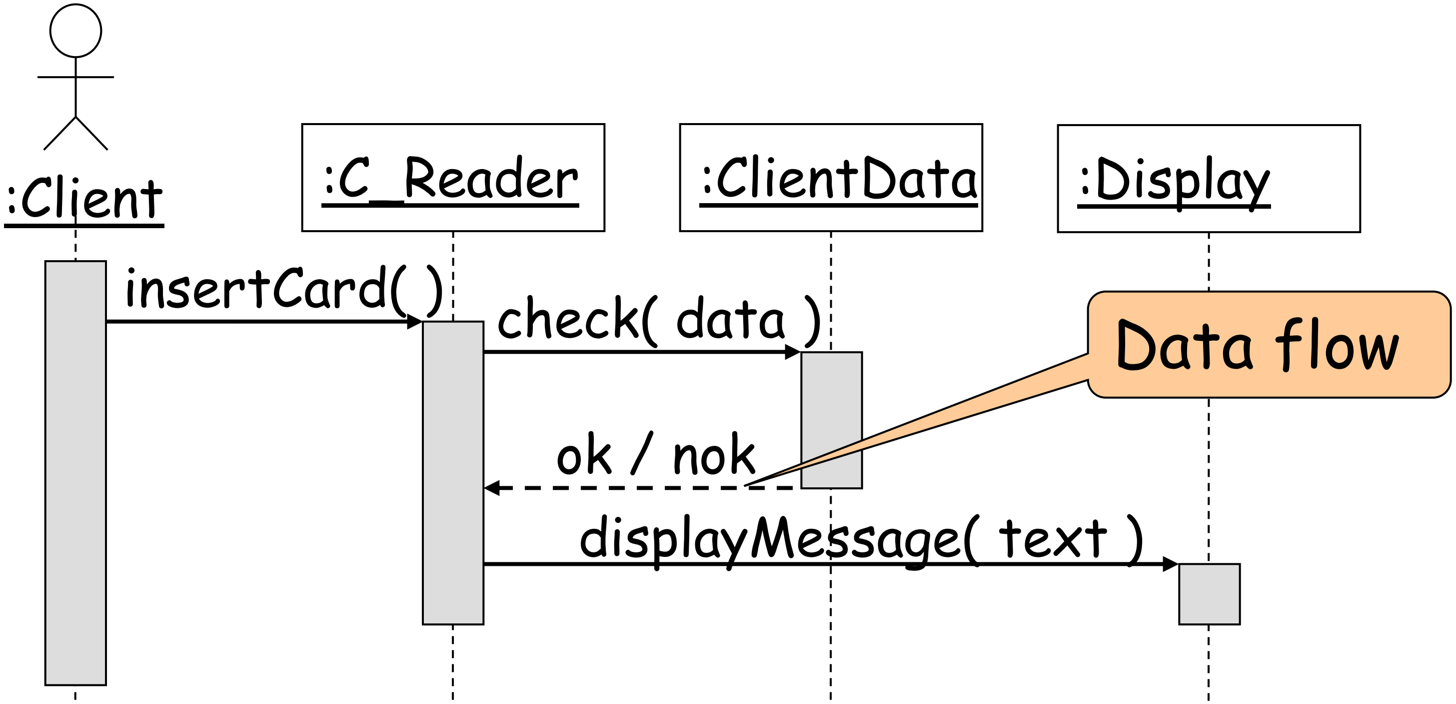
- State diagrams: focus on the state of an object (or system) an how it changes due to events

Sequence diagrams

- Entities:
  - objects (including instances of actors)
- Relations:
  - message passing
- Sugar:
  - lifelines
  - activations
  - creations
  - destructions
  - frames



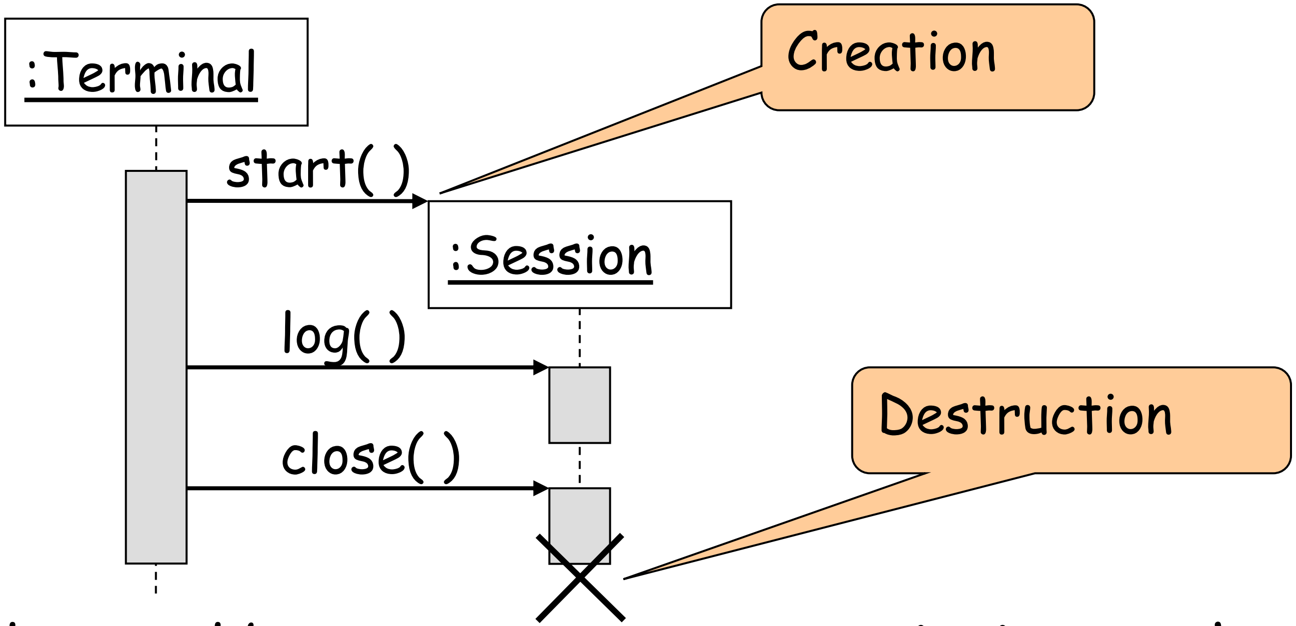
Nested messages



The source of an arrow indicates the activation which sent the message

An activation is as long as all nested activations

Creation and destruction



Creation is denoted by a message arrow pointing to the object

In garbage collection environments, destruction can be used to denote the end of the useful life of an object

From Use Cases to Sequence diagrams

Sequence diagrams are **derived from flows of events** of use cases

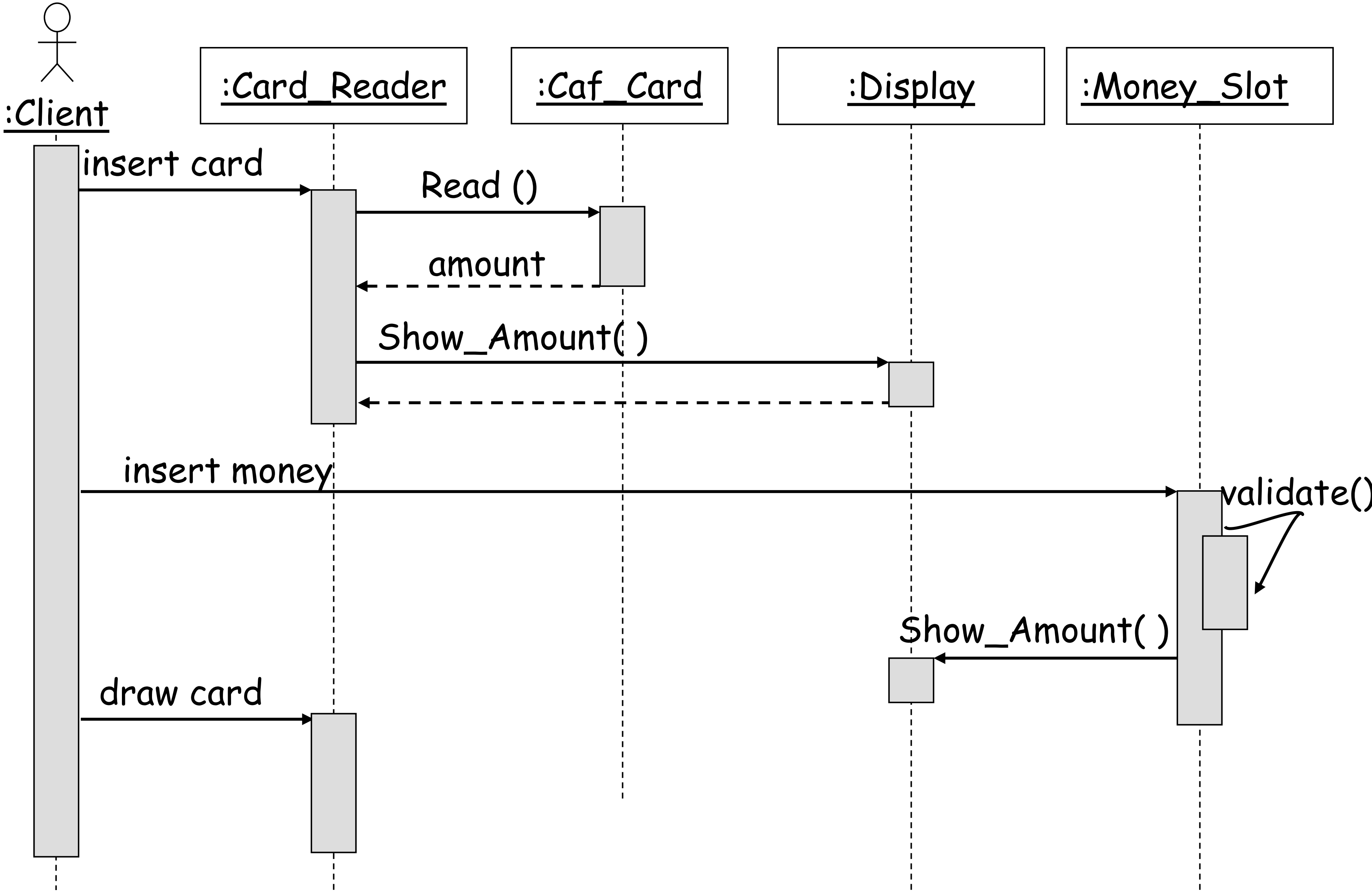
An event always has a **sender** and a **receiver**

- Find the objects for each event

Relation to object identification

- Objects/classes have already been identified during object modeling
- Additional objects are identified as a result of dynamic modeling

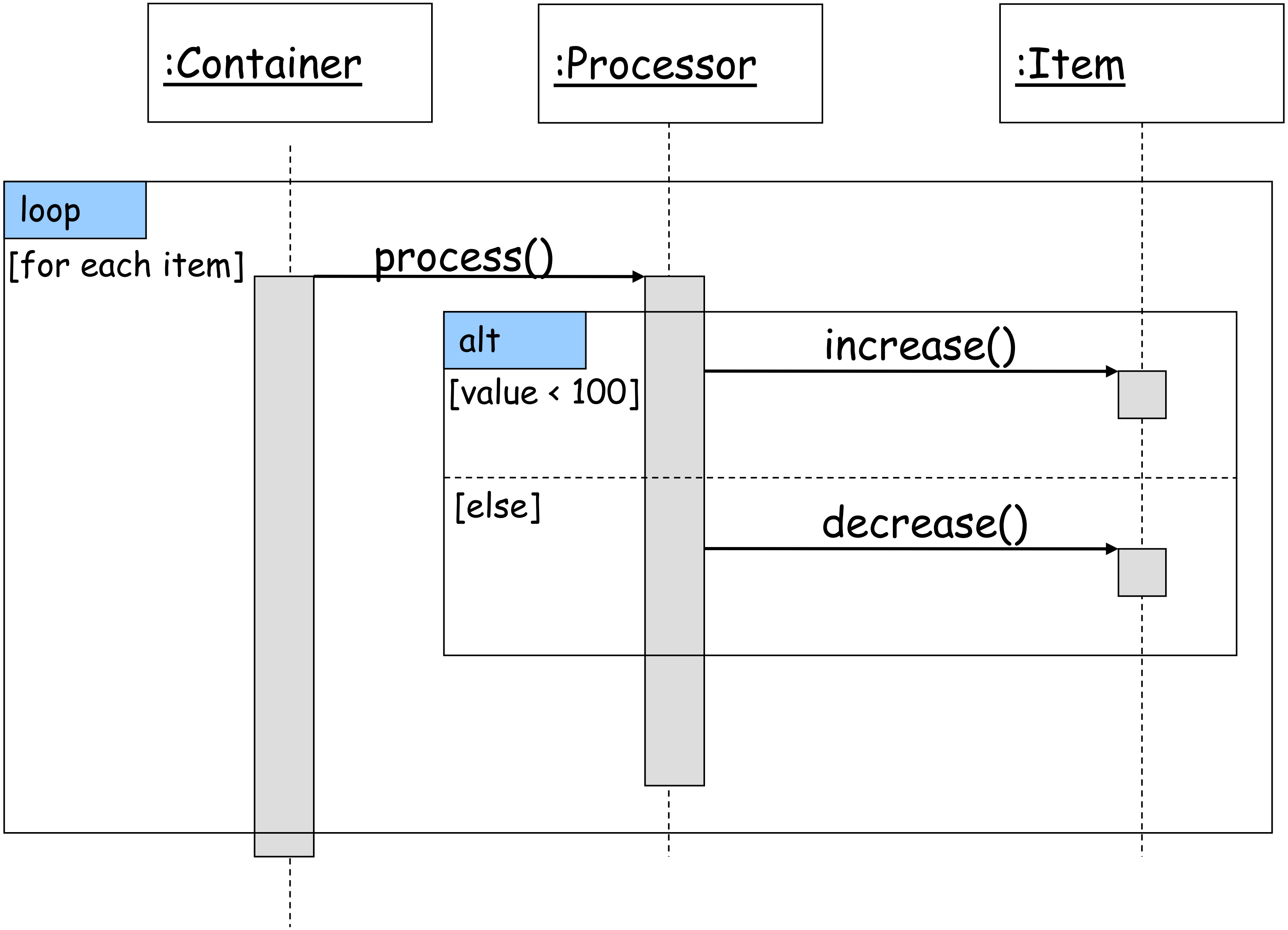
Example Sequence diagram



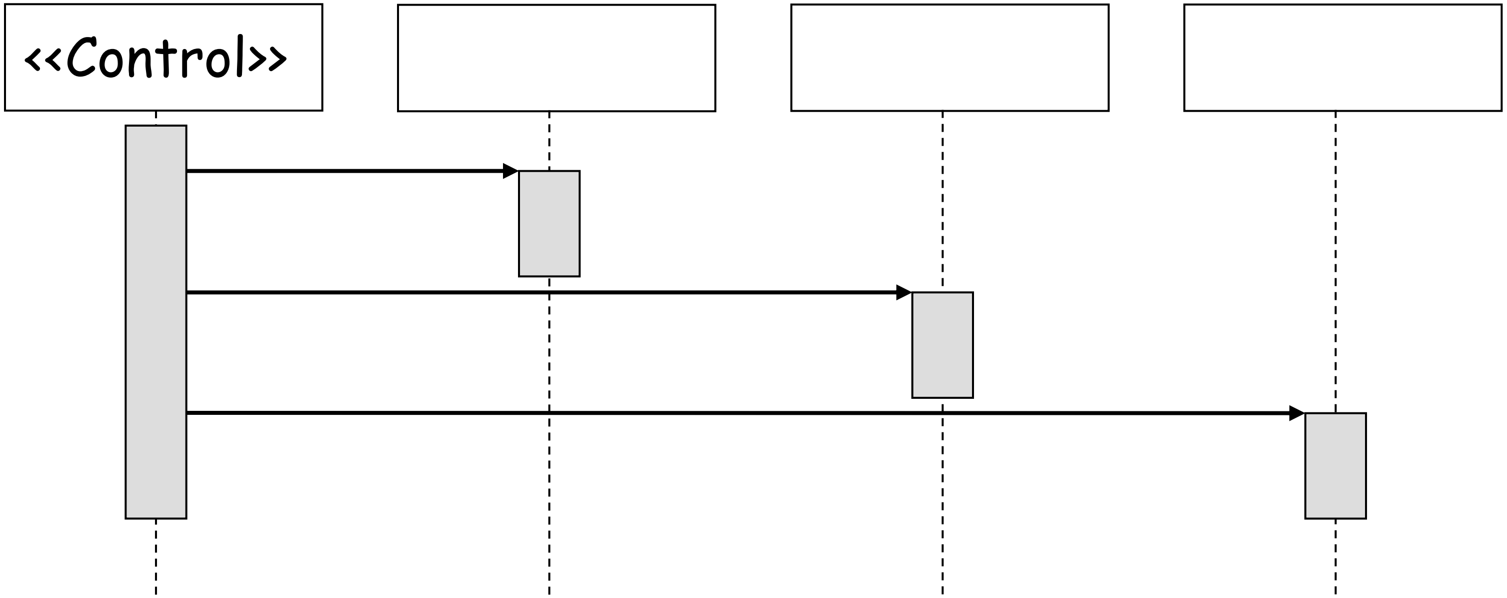
Example Sequence diagram

- The diagram shows only the successful case
- Exceptional case could go either on another diagram or could be incorporated to this one
- Sequence diagrams show main scenario and "interesting" cases
    - interesting: exceptional or important variant behavior
- Need not draw diagram for every possible case
- would lead to too many diagrams

Interaction frames



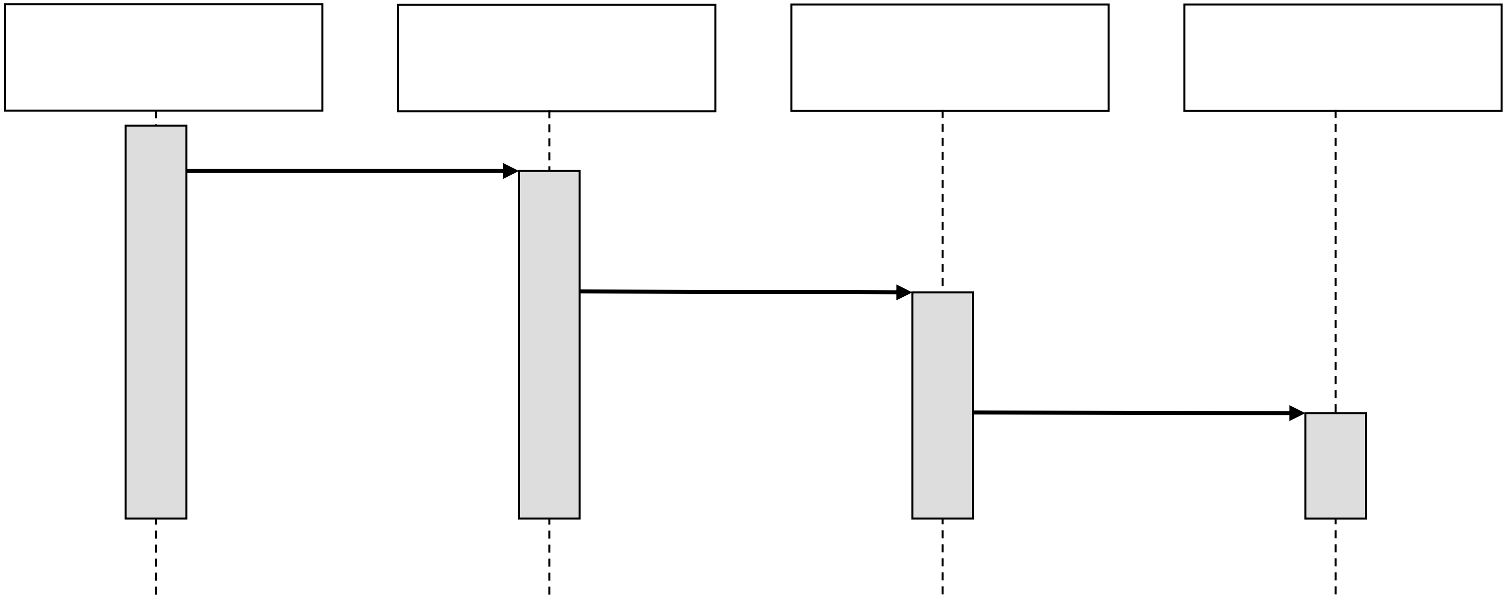
Fork structure



The **dynamic behavior is placed in a single object**, usually a control object

It knows all the other objects and often uses them for direct queries and commands

Stair structure



The **dynamic behavior is distributed**

- Each object delegates some responsibility to other objects
- Each object knows only a few of the other objects and knows which objects can help with a specific behavior

Fork or stair?

Object-oriented supporters claim that the stair structure is better

- The more the responsibility is spread out, the better

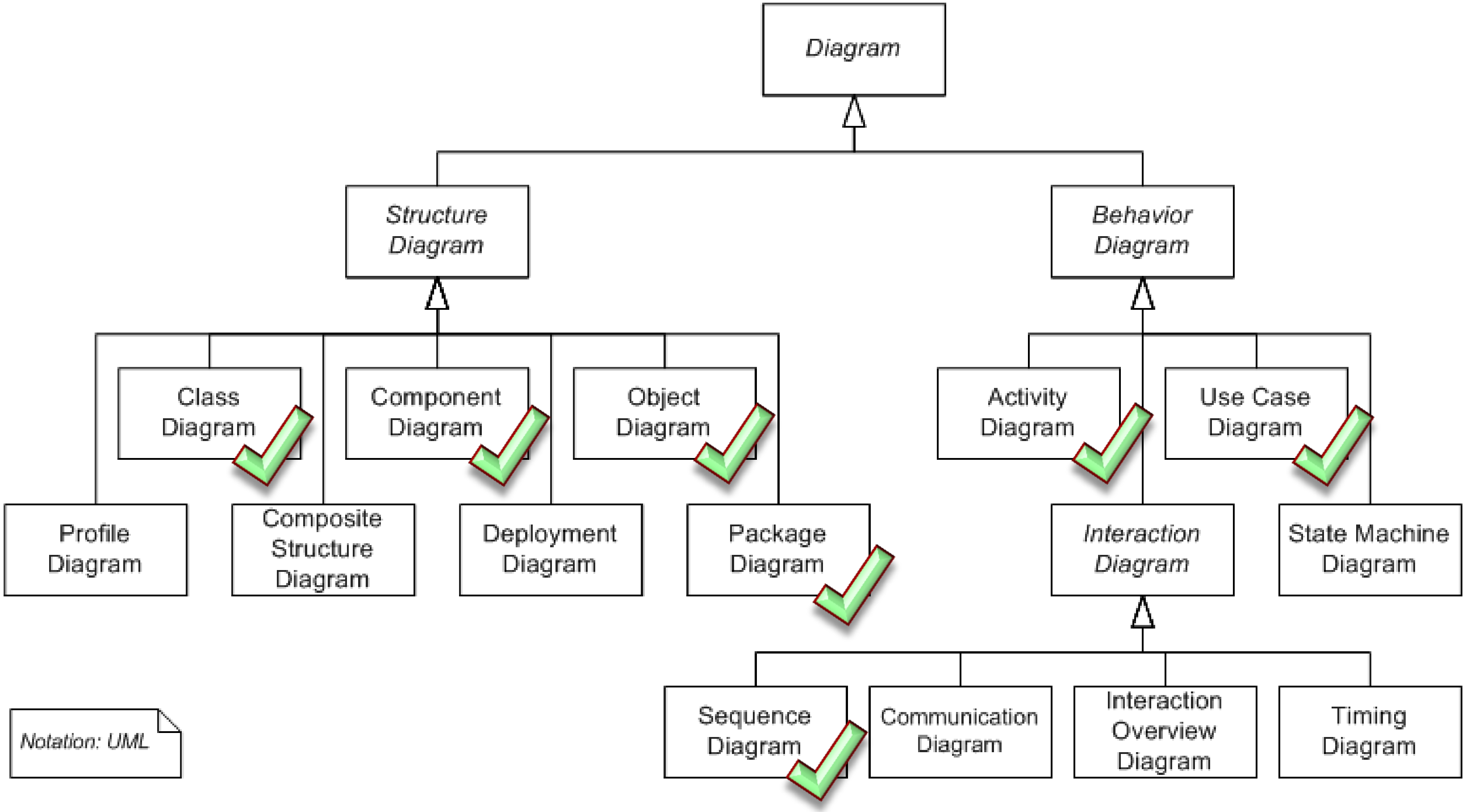
Choose the **stair** (decentralized control) if

- The operations have a **strong connection**
- The operations will **always** be performed in the **same order**

Choose the **fork** (centralized control) if

- The operations can **change order**
- **New operations** are expected to be added as a result of new requirements

Diagrams in UML



State Machine Diagrams

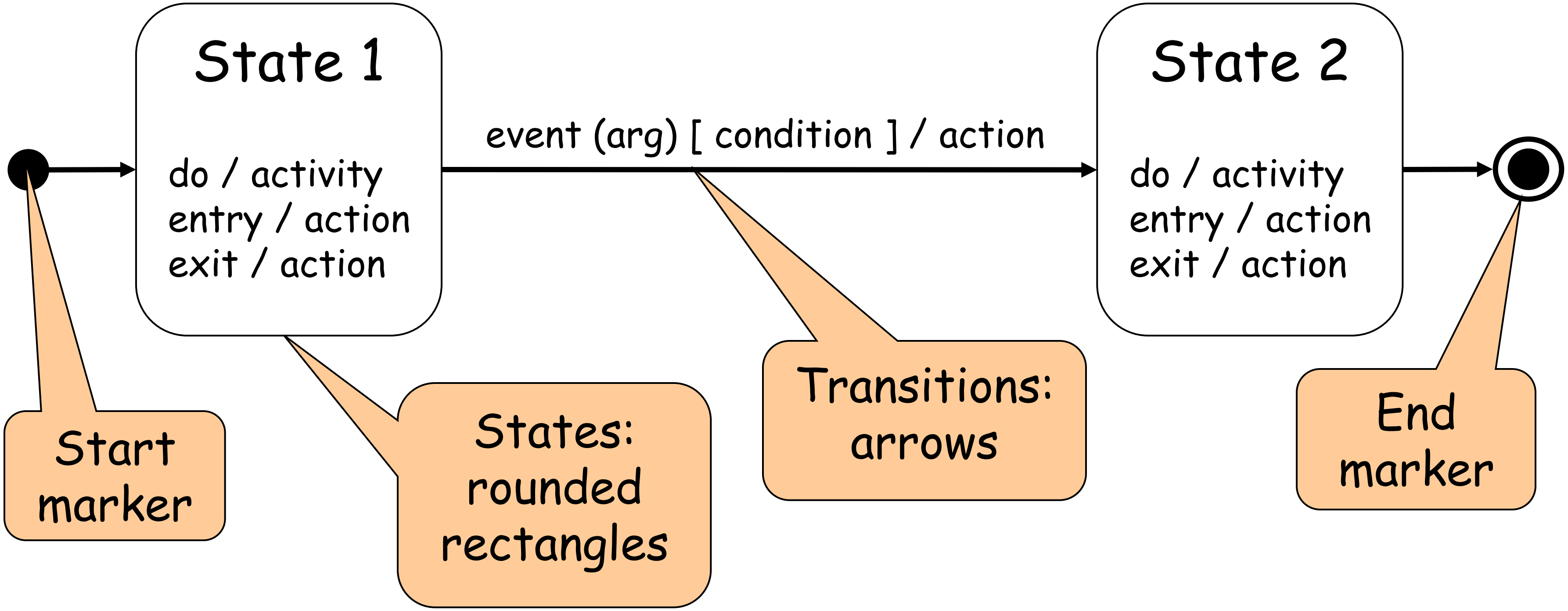
- UML State Machine Diagrams are a powerful notation to model finite automata

- It shows the states which an object or a (sub)system – depending on the level of abstraction – can have at runtime

- It also shows the events which trigger a change of state

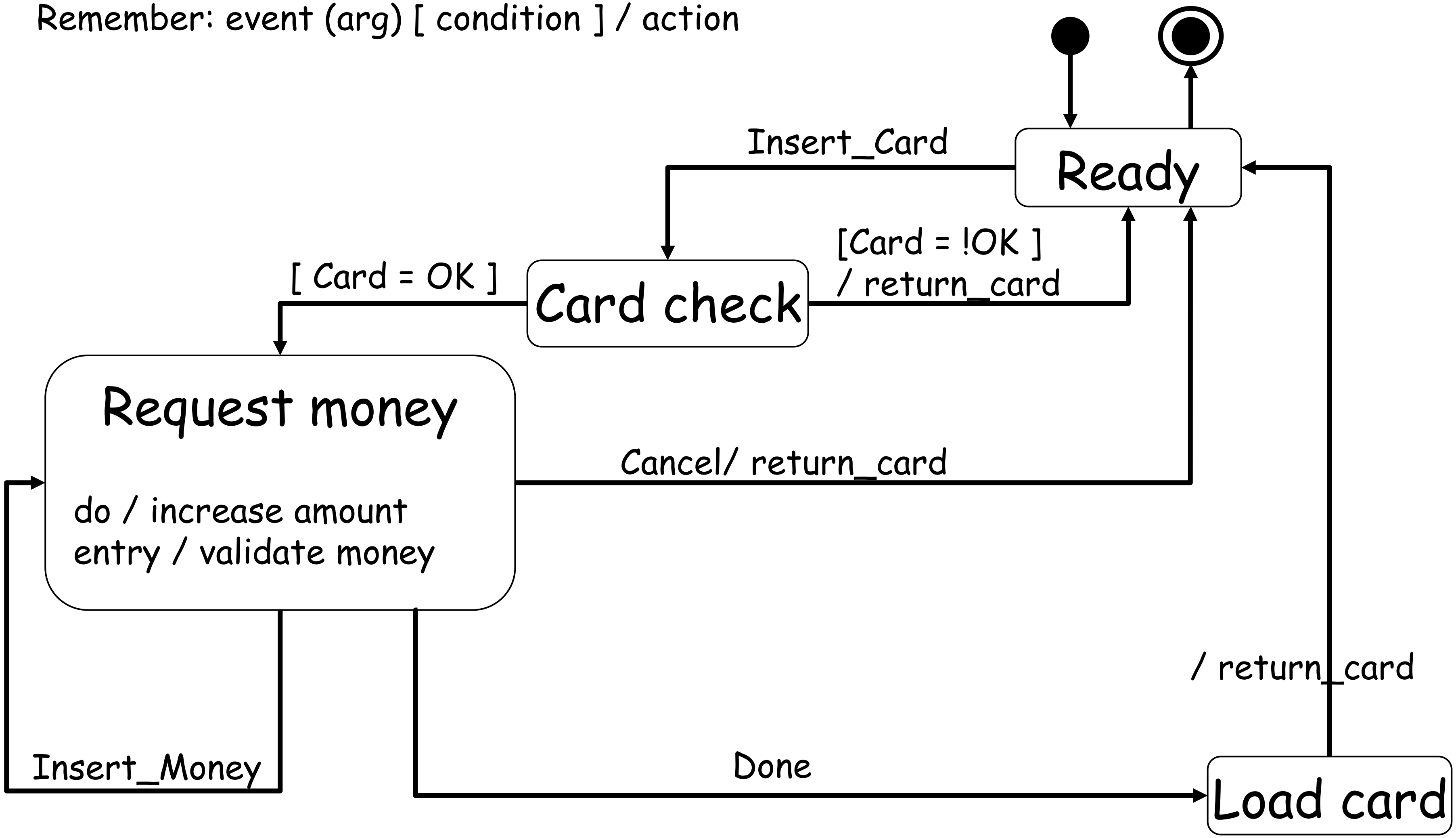
State Machine diagrams

- Entities:
  - states: name, activity, entry/exit action
- Relations:
  - transitions between states: event, condition, action



State Machine diagram for the case study

Remember: event (arg) [condition] / action



Composite/nested State Machine diagrams

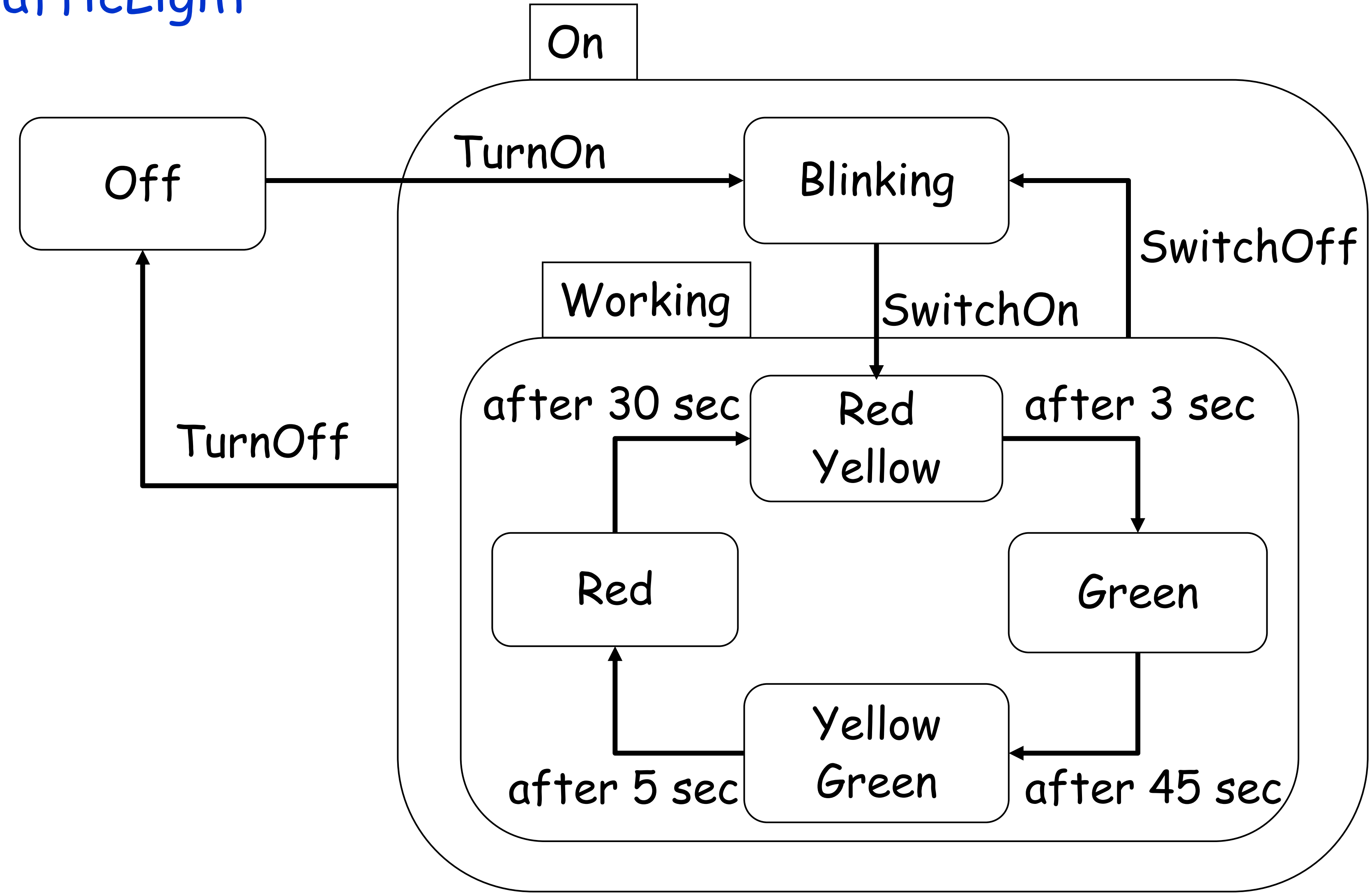
Activities in states can be **composite items** that denote other state diagrams

Sets of substates in a nested state diagram can be denoted with a superstate

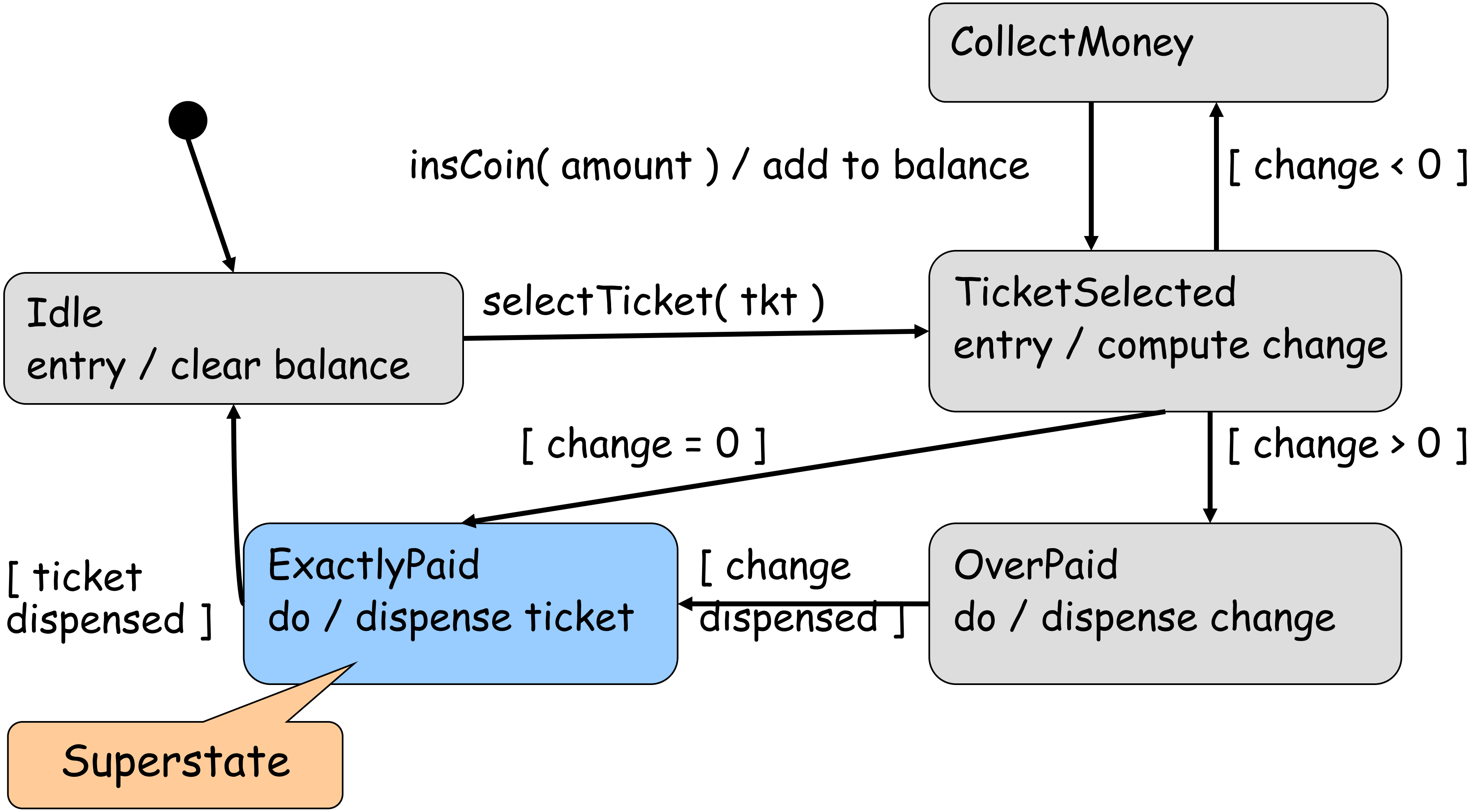
- Avoid spaghetti models
- Reduce the number of lines in a state diagram

State diagrams: example composite state

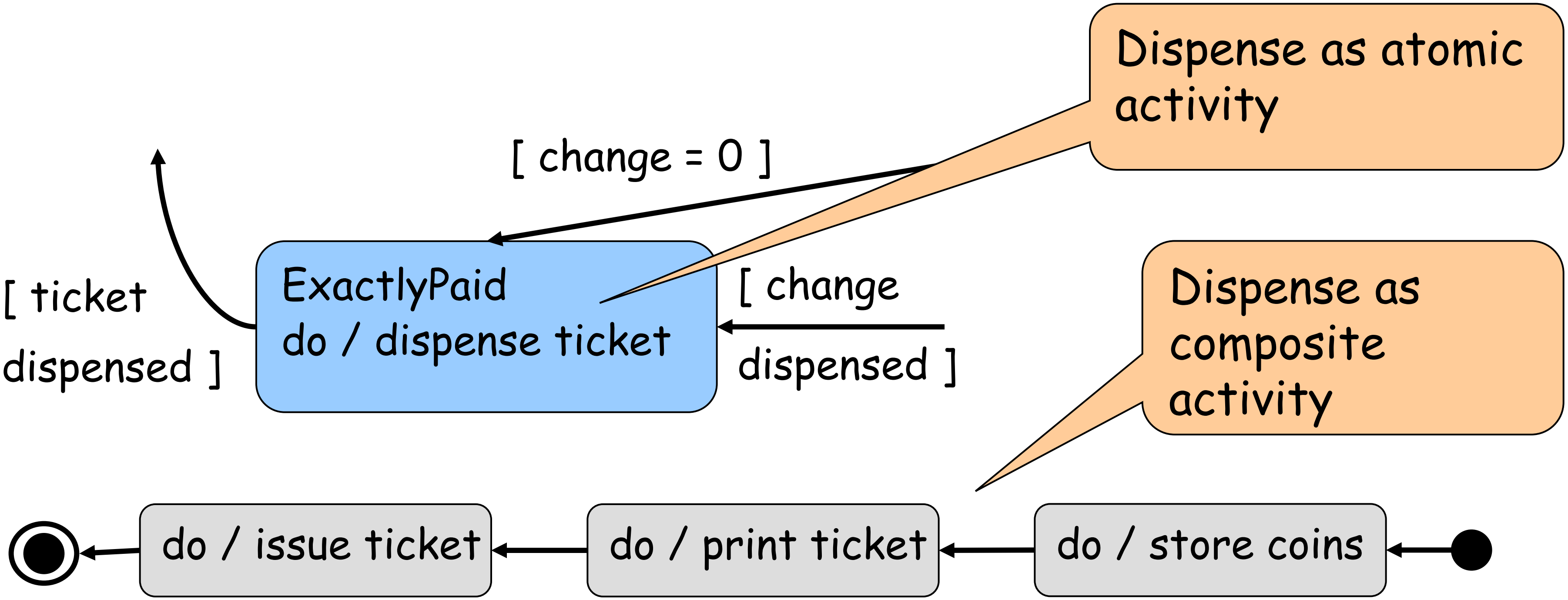
TrafficLight



Example: superstate



Expanding the superstate



Dispense as atomic activity

Dispense as composite activity

Transitions from other states to the superstate **enter the first substate** of the superstate

Transitions to other states from a superstate are **inherited by all the substates** (state inheritance)

State diagram vs. Sequence diagram

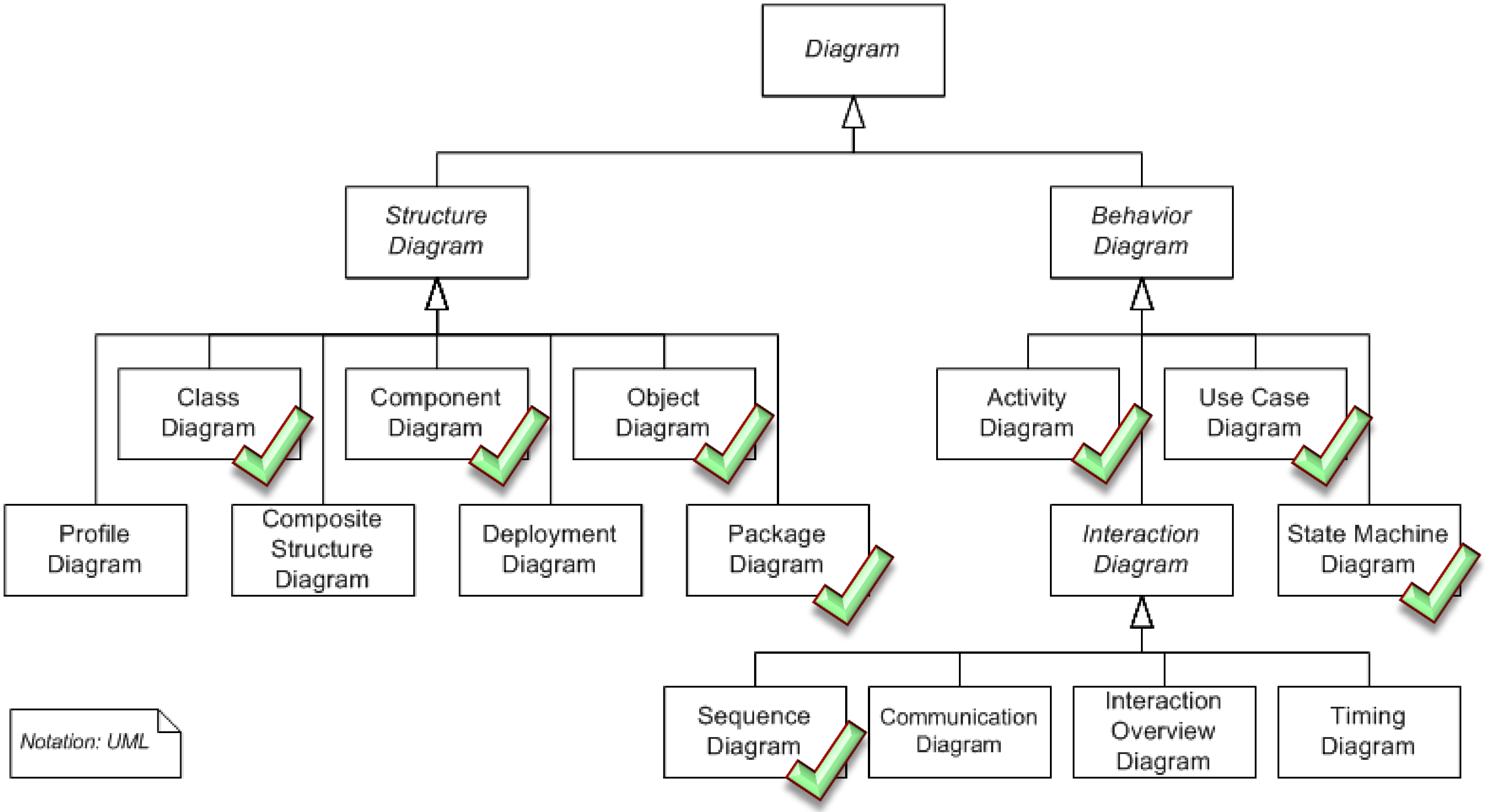
State diagrams help to identify

- Changes to an individual object over time

Sequence diagrams help to identify

- The temporal relationship between objects
- Sequence of operations as a response to one or more events

Diagrams in UML



Practical tips

- Create component diagrams only for large, distributed systems
- Create state diagrams only for classes with complex, interesting behavior (usually classes representing entities from the problem domain or performing control)
- Create activity diagrams for complex algorithms and business processes (not for every operation)
- Create sequence diagrams for nontrivial collaborations and protocols (not for every scenario)
- Don't put too much information on a diagram
- Choose the level of abstraction and maintain it