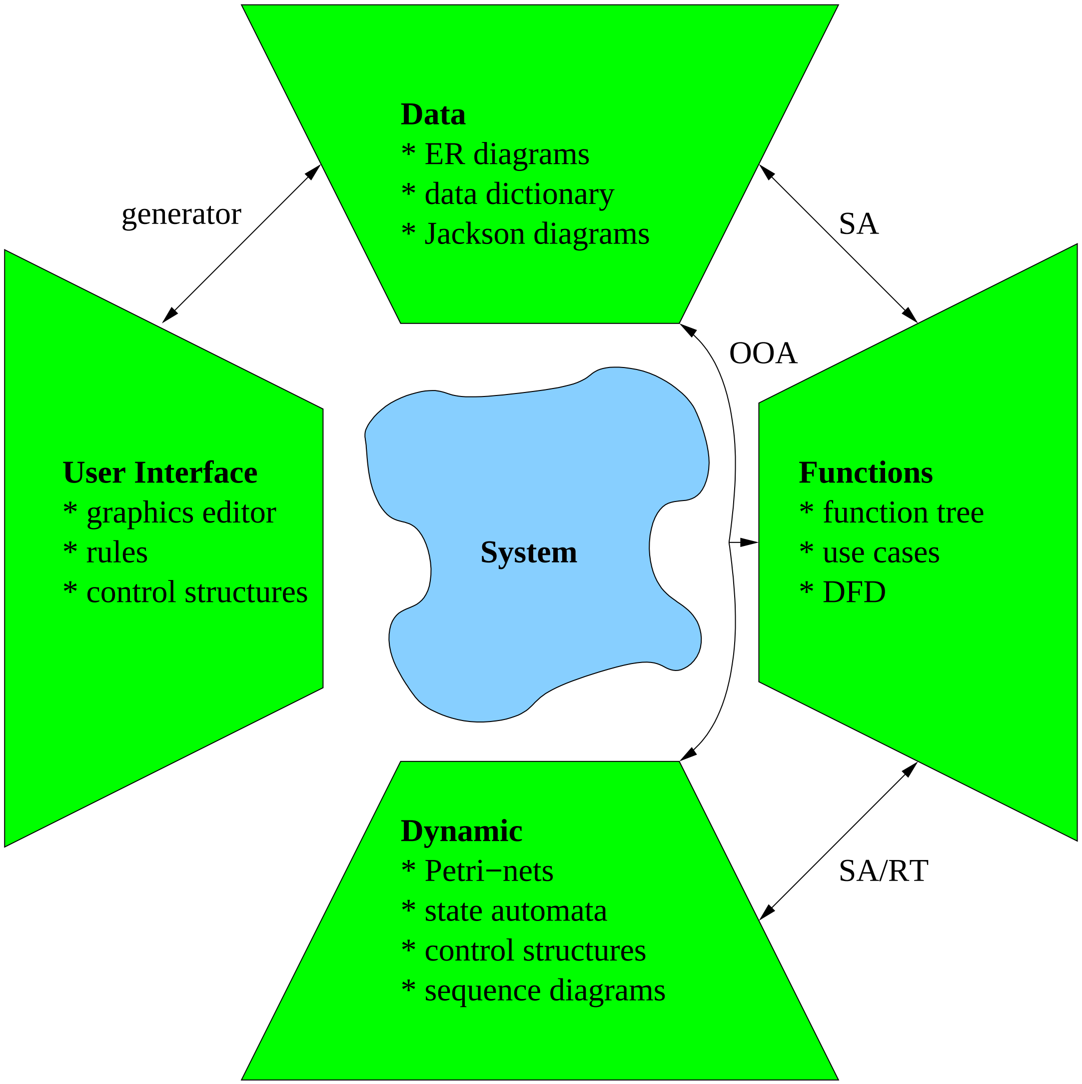
Modeling with UML

- UML = Unified Modeling Language
- semi-formal standard diagramatic notation
- each diagram supports one or more development phases
  - analysis,
  - design, and
  - implementation
- each diagram combines several fundamental techniques
- each fundamental technique offers a particular **view** of the system
  - **data** / **functions**
  - **dynamic** / **user interface**

Overview Fundamental Techniques



Fundamental Techniques $\leftrightarrow$ Views

functional view

- hierarchy $\rightarrow$ **function tree**
- process $\rightarrow$ **use case diagram** (UML)
- information flow $\rightarrow$ **data flow diagram** (DFD)

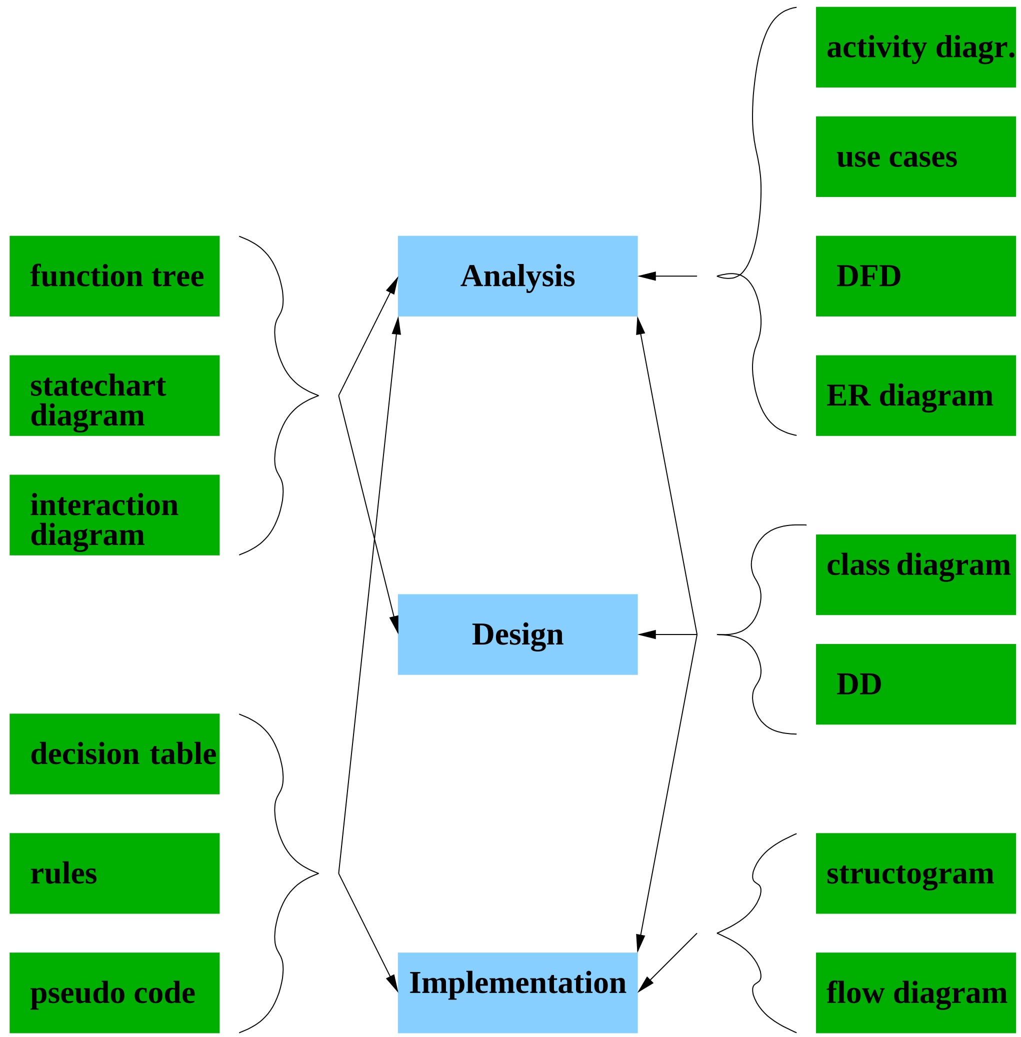
data oriented view

- data structures $\rightarrow$ **data dictionary** (DD)
- class structure and relations $\rightarrow$ **class diagram** (UML)

state-oriented view

- **state chart diagram** (UML)
- **activity diagram** (UML)
- **interaction diagram** (UML)

Mapping Fundamental Techniques to Phases



Use Cases (Jacobson, UML), Template

Use case: name

Goal: achieved by successful execution

Category: primary, secondary, optional

Precondition:

Postcondition/success:

Postcondition/failure:

Actors:

Trigger:

Description: numbered tasks

Extensions: wrt previous tasks

Alternatives: wrt tasks

Example: MUA

Use case: **compose message**

Goal: mail message sent to outgoing server

Category: primary

Postcondition/success: acknowledgement stored

Actors: end-user

Description:

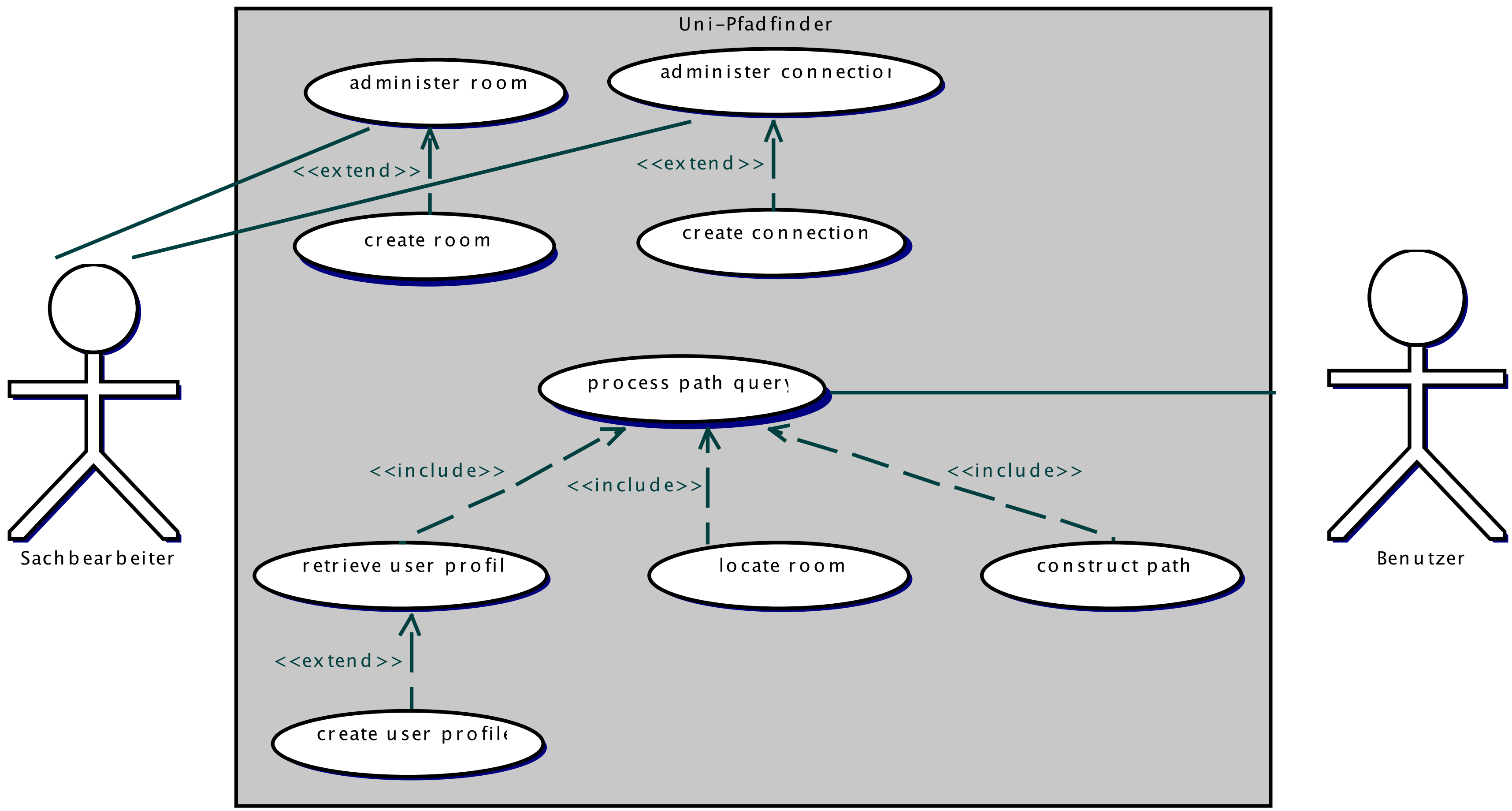
1. enter recipients
2. enter text
3. select sending options

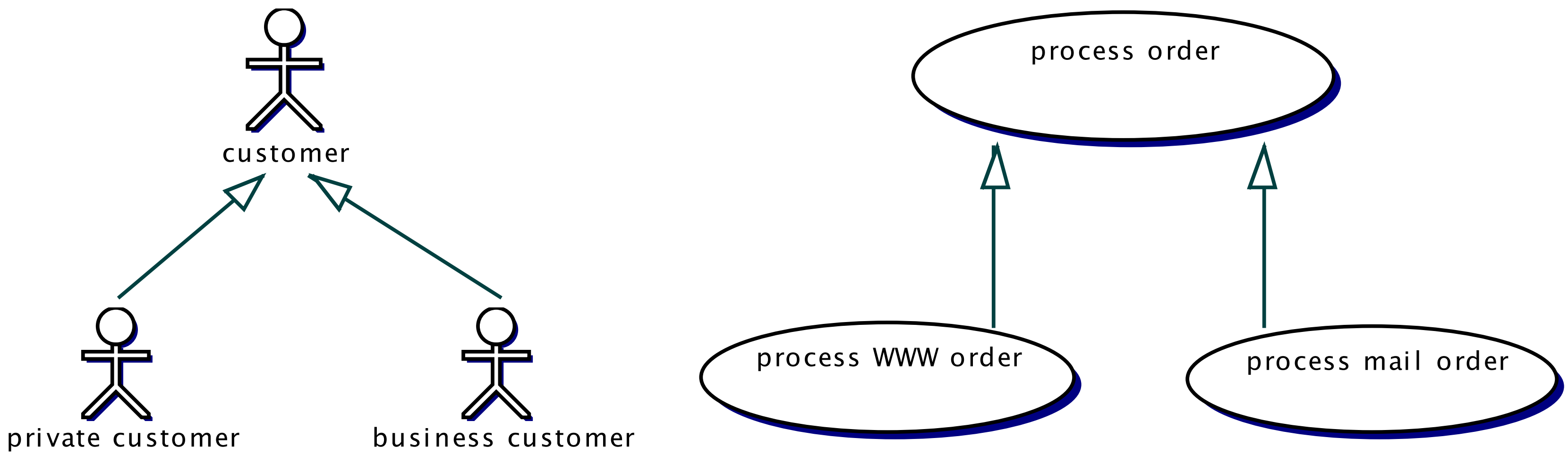
Extensions:

- 1a. select recipients from address book
- 2a. enter formatting hints

Alternatives:

- 1b. extract recipients from message (reply)
- 2b. edit and compose multi-media fragments


Uni-Pfadfinder
administer room
administer connectioı
<<extend>>
<<extend>>
create room
create connection
process path query
<<include>>
<<include>>
<<include>>
retrieve user profil
locate room
construct path
<<extend>>
create user profile
Sachbearbeiter
Benutzer

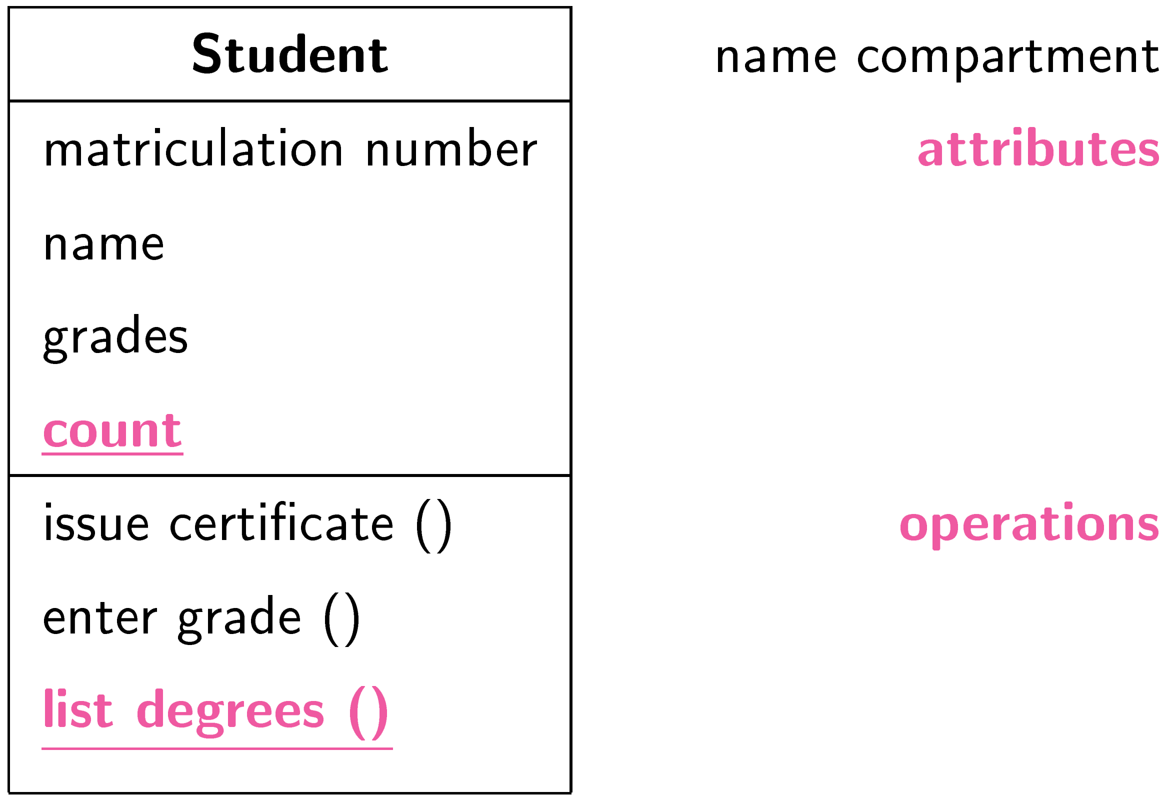


- generalization
- concrete and abstract use cases
- concrete and abstract actors

Class Diagrams (UML)

- representation of **classes** and their **static relationships**
- no information on dynamic behavior
- UML notation is graph with
  - **nodes**: classes (rectangles)
  - **edges**: various relationships between classes
- may contain interface, packages, relationships, as well as instances (objects, links)

Classes



- only name compartment obligatory
- additional compartments may be defined
- class attributes / operations underlined

Contents of name compartment

1. optional stereotype
   ≪control≫, ≪boundary≫, ≪entity≫ (defined by designer)
2. class name
   abstract classes indicated by *italics*
3. optional property list (tagged value)
   {abstract}, {leaf, author="John Doe"}

Attributes compartment

Syntax of an attribute

visibility name : *type* [*multiplicity ordering*] = *initial-value* { *properties* }

visibility	+, #, -, ~	Design, Implementation
name		all phases
type	classifier name / PL type	(Analysis), Design, Implementation
multiplicity	sequence of intervals	Design, Implementation
ordering	ordered / unordered	Design, Implementation
initial-value	language dependent	(Design), Implementation
properties	*e.g.*, {frozen}	(Design), Implementation

Visibility

- +, `public`
- #, `protected`
- -, `private`
- ~, `package`
- alternatively: notation of the implementation language

Multiplicity

Defines set of non-negative integers

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only n (where n ¿ 1)
0..n	Zero to n (where n ¿ 1)
1..n	One to n (where n ¿ 1)

Operations compartment

Syntax of an operation

visibility name (*parameter-list*) : *return-type* { properties }

visibility	+, #, -, ~	Design, Implementation
name		all phases
parameter-list	*kind name* : *type*	Design, Implementation
	kind $\in$ **in**, **out**, **inout**	
return-type	classifier name / PL type	(Analysis), Design, Implementation
properties	*e.g.*, {query}	(Analysis), Design, Implementation
	{concurrency=...}	
	{abstract}	

- class operations underlined

Relations in Class Diagrams

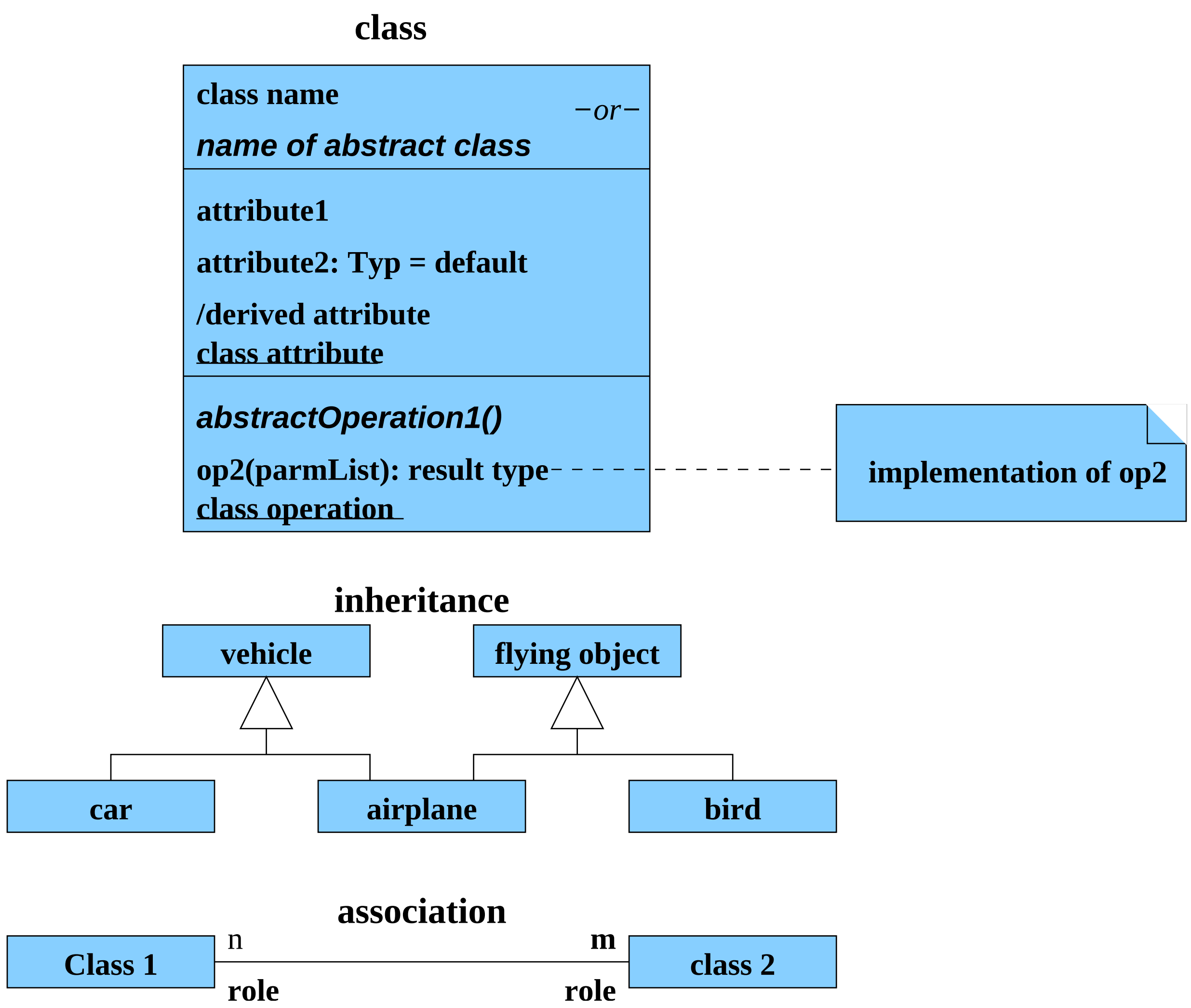
Binary Association

- indicates “collaboration” between two classes (possibly reflexive)
- solid line between two classes
- optional:
  - association name
  - decoration with role names
  - navigation (Design)
  - multiplicities (Design)

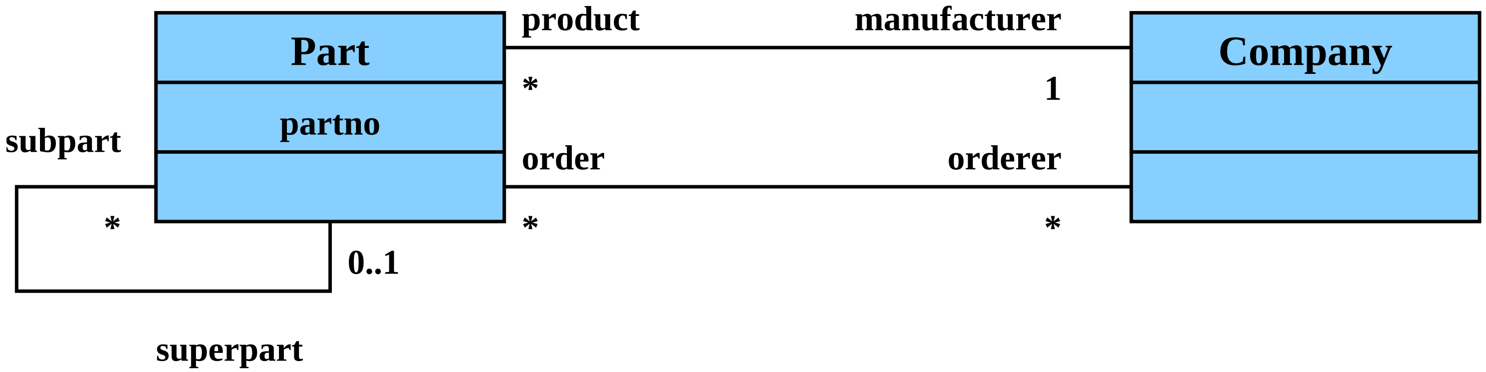
Generalization

- indicates subclass relation
- solid line with open arrow towards super class

Example: Class Diagram



Example: class diagram with associations



Constraints

- Constraints (*Restriktionen*) wrt object state or association
- Notation: {**constraint**}
- Example constraints on associations:
  {**sorted**}, {**immutable**}, {**read-only**}, {**subset**}, {**xor**}
- natural language, pseudo code, predicate logic, . . . , **OCL**

→ Design by Contract (Bertrand Meyer, Eiffel)

Constraints for Design by Contract

- pre- and postconditions of operations

  Ex: operation **int sqrt()**

  precondition: {**this.value** >= **0**}

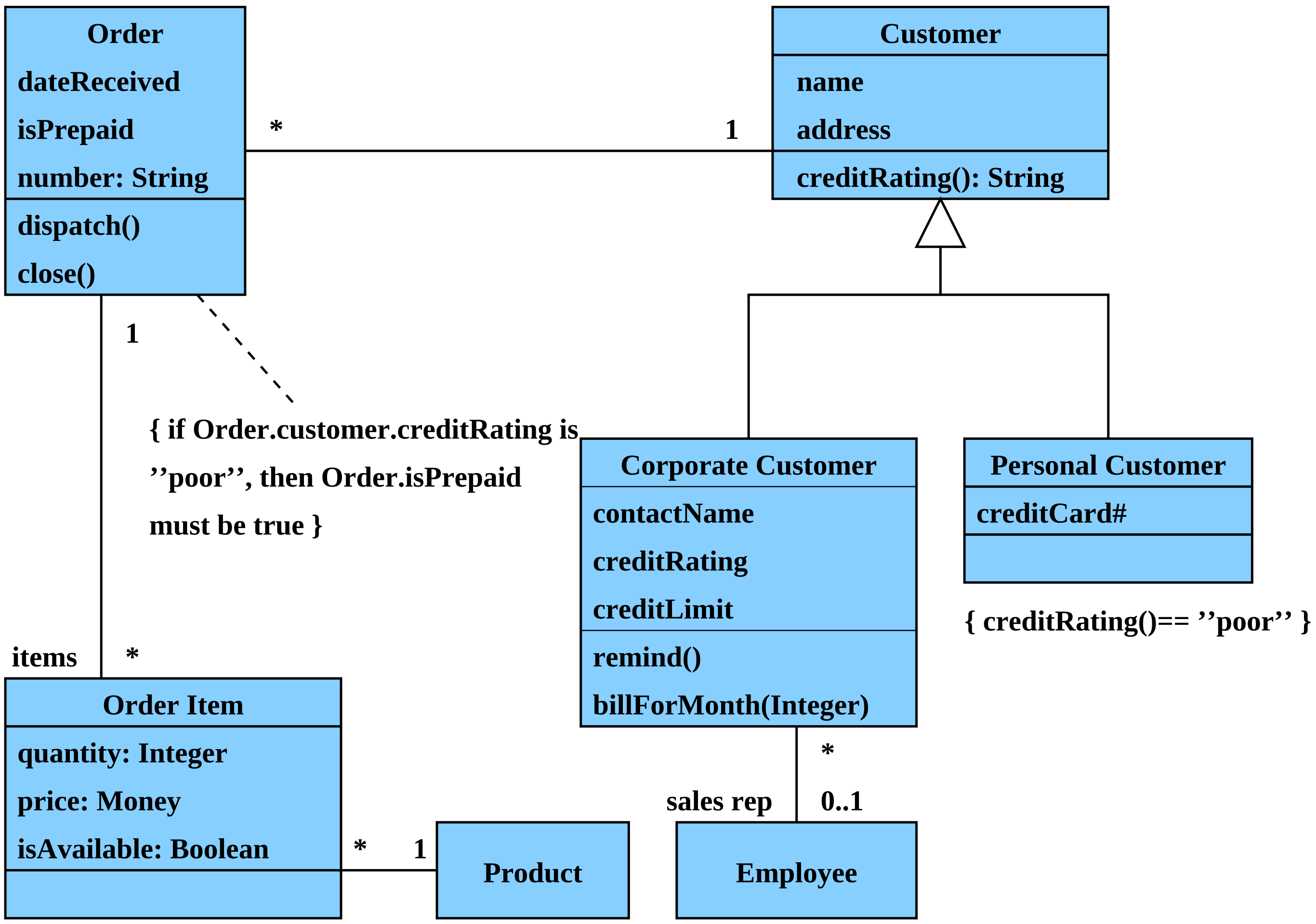
  postcondition: {**result * result == this.value**}

- invariants
  maintained by each operation
  Ex: {**balance == sum(entry.amount());**}

Responsibilities

- **Precondition** assigns responsibility to caller
- operation responsible for **postcondition** if precondition holds (analogously for invariants)
- → no duplicate or omitted checks
- explicit checking of constraints while debugging
  e.g. operation `checkInvariants`

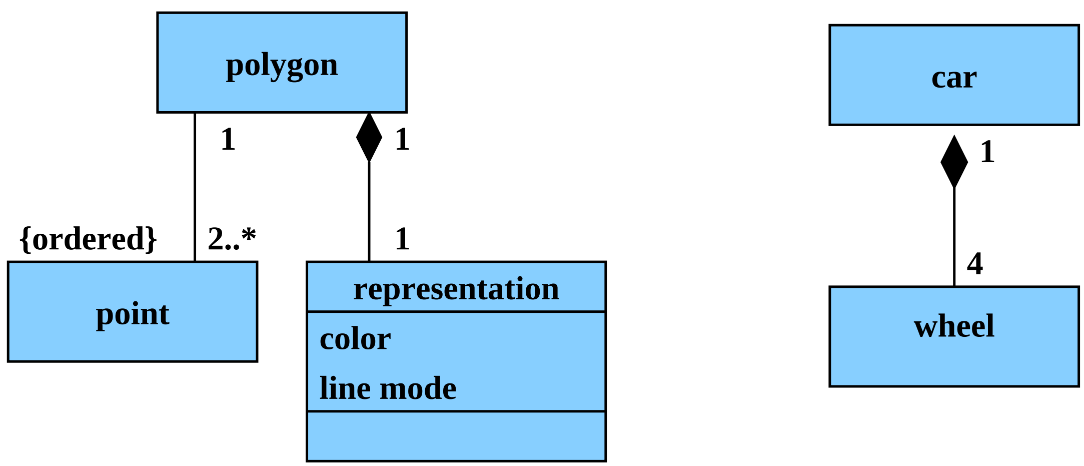
Example: class diagram with object constraints



Composition

- aggregation is a particular association **part-of**
- Meaning: object “belongs existentially” to other object
- Object and its components live and die together
- Notation: edge with black rhombus as arrow head

Example



Guidelines for Analysis Phase

- **only essential attributes and operations**
- no multiplicities, navigation, etc
- do not model trivial operations like
  - **new**: object creation
  - **delete**: object deletion
  - **set**⟨*Attribute*⟩: update an attribute
  - **get**⟨*Attribute*⟩: read an attribute
- for simplicity: each class "knows" all of its instances in OOA
- implementation may be attached to operation with a note

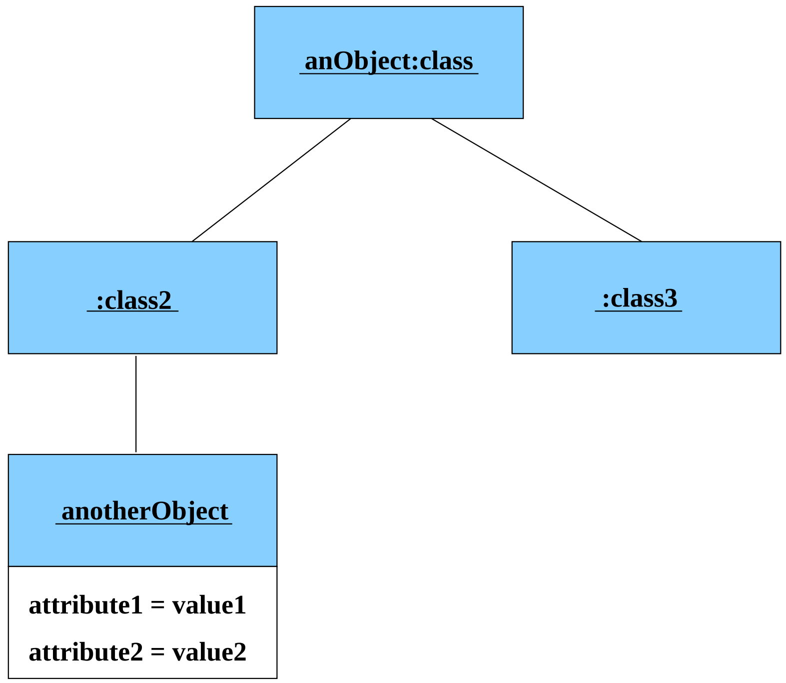
Object and Collaboration Diagrams (UML)

- notation for **objects** and their **links**
- UML notation:
  - **nodes**: objects (rectangles), labeled with **<u>object name:type</u>**
  - **edges**: links between objects
    "objects that know each other"

Properties of object diagrams

- snapshot of a system state
- configuration of a specific group of objects

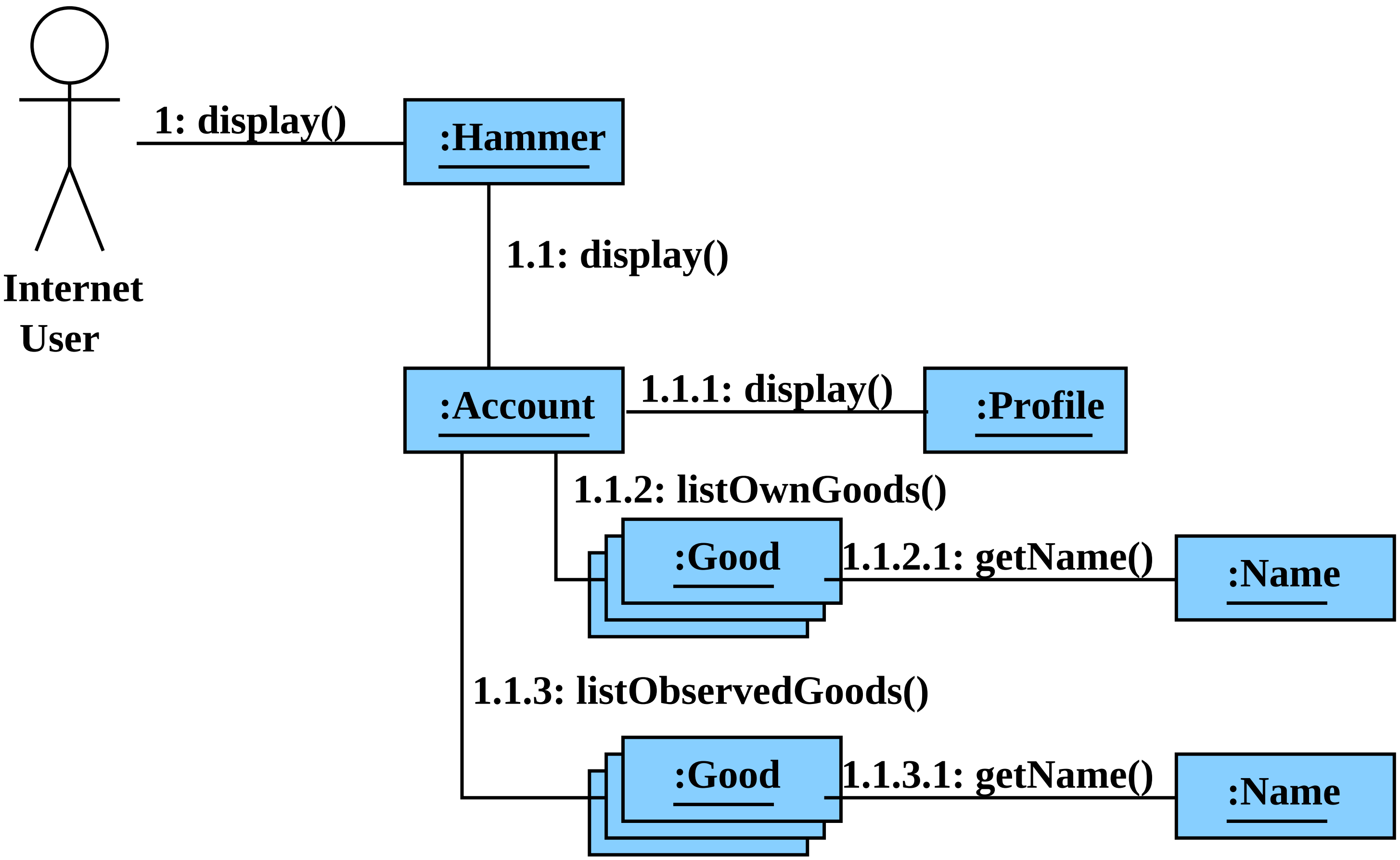
Example: Object Diagram



Dynamic properties → collaboration diagrams

- objects → **object roles**
- object notation stands for "any object of that class"
- object roles and links may be labeled with constraints
  - {new}
  - {transient}
  - {destroyed}
- labeling links with numbered operations
- numbering implies sequence of execution

Example: Collaboration Diagram



Finite State Machines (FSM, UML)

- modeling the evolving state of an object
  e.g., Statechart diagrams in UML
- starting point:
  deterministic finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ where

  Q: finite set of states

  Σ: finite input alphabet

  δ: $Q \times \Sigma \longrightarrow Q$ transition function

  $q_0 \in Q$ initial state

  $F \subseteq Q$ set of final states

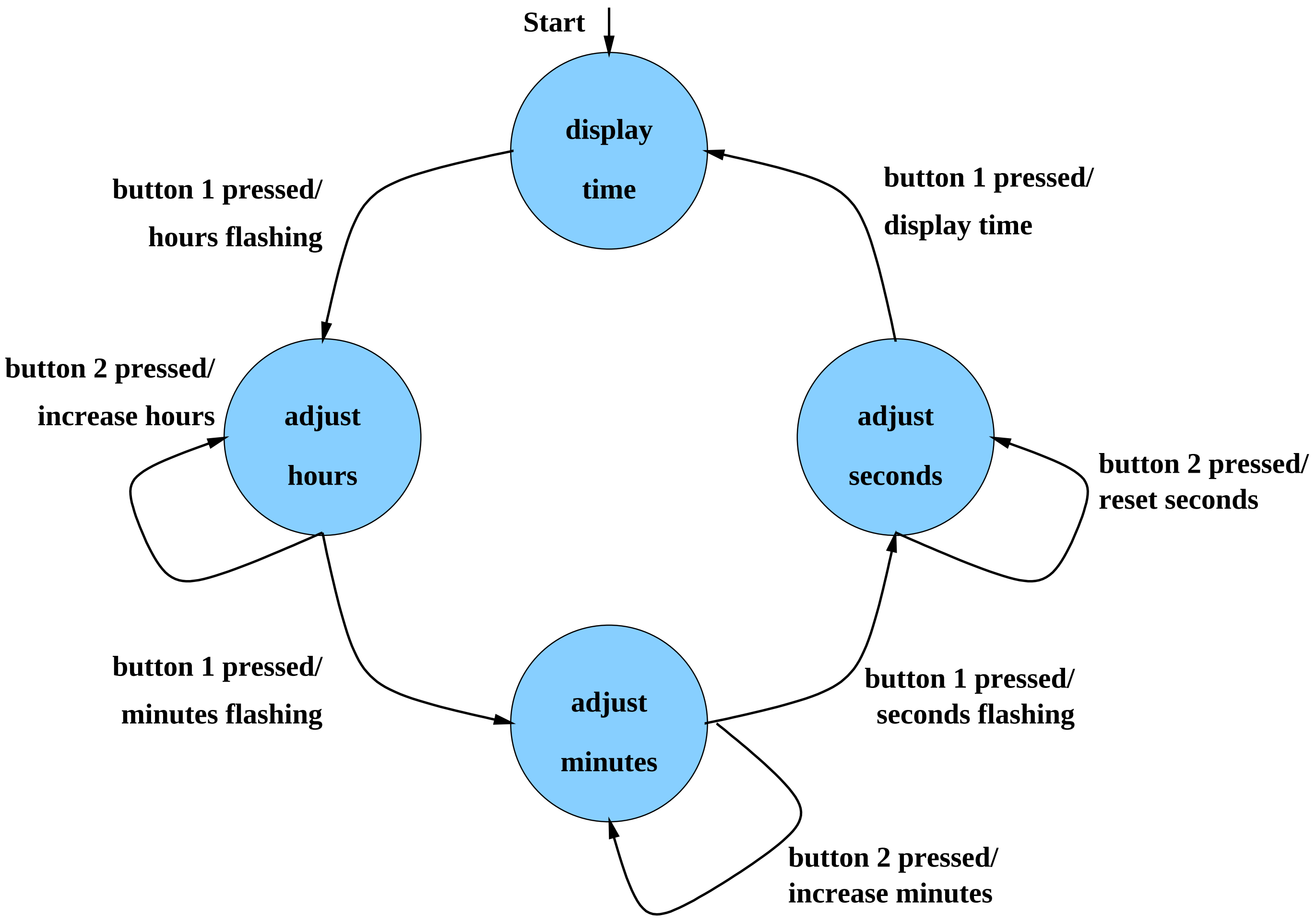
Graphical Representation of FSM

- **nodes:** states of the automaton (circles or rectangles)
- arrow pointing to q_0
- final states indicated by double circle
- **edges:** if $\delta(q, a) = q'$ then **transition** labeled a from q to q'

FSM with output specifies a translation $\Sigma^* \to \Delta^*$

- $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$
- replace final states F by output alphabet Δ and output function λ
- **Mealy-automaton:** $\lambda : Q \times \Sigma \longrightarrow \Delta$
  edge from q to $\delta(q, a)$ additionally carries $\lambda(q, a)$
- **Moore-automaton:** $\lambda : Q \longrightarrow \Delta$
  state q labeled with $\lambda(q)$

Example: digital clock as a Mealy-automaton

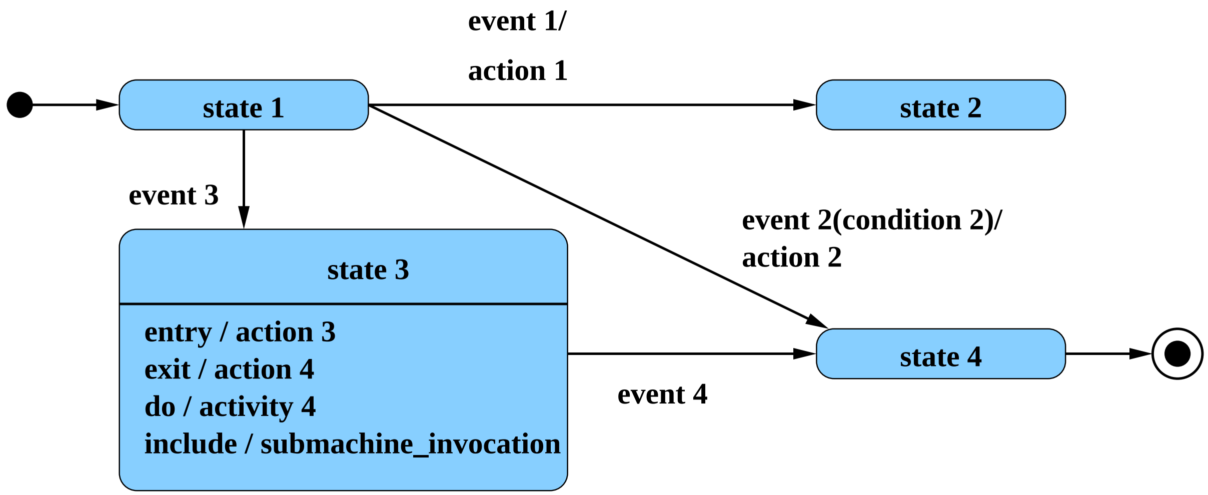


Drawback: FSMs get big too quickly → structuring required

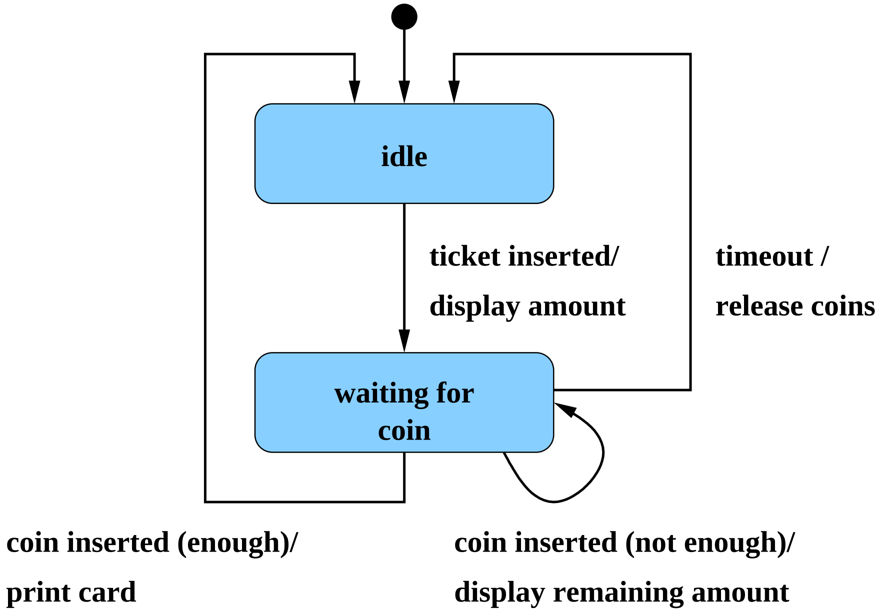
Statechart Diagram (Harel, UML)

- hybrid automata ("Moore + Mealy")
- each state may have
  - **entry action:** executed on entry to state
    $\cong$ labeling all incoming edges
  - **exit action:** executed on exit of state
    $\cong$ labeling all outgoing edges
  - **do activity:**
    executed while in state
- composite states
- states with history
- concurrent states
- optional: conditional state transitions

Example: Statechart Diagram

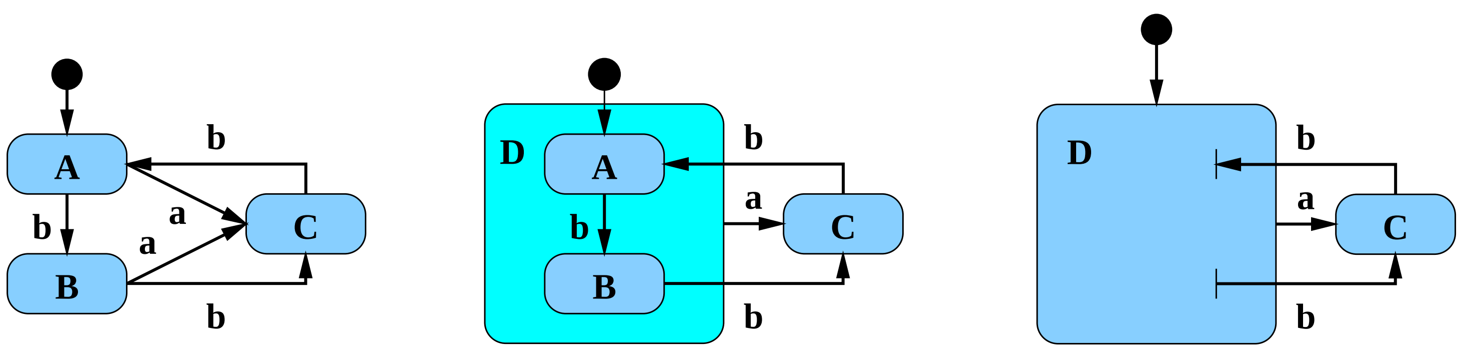


Example: parking lot

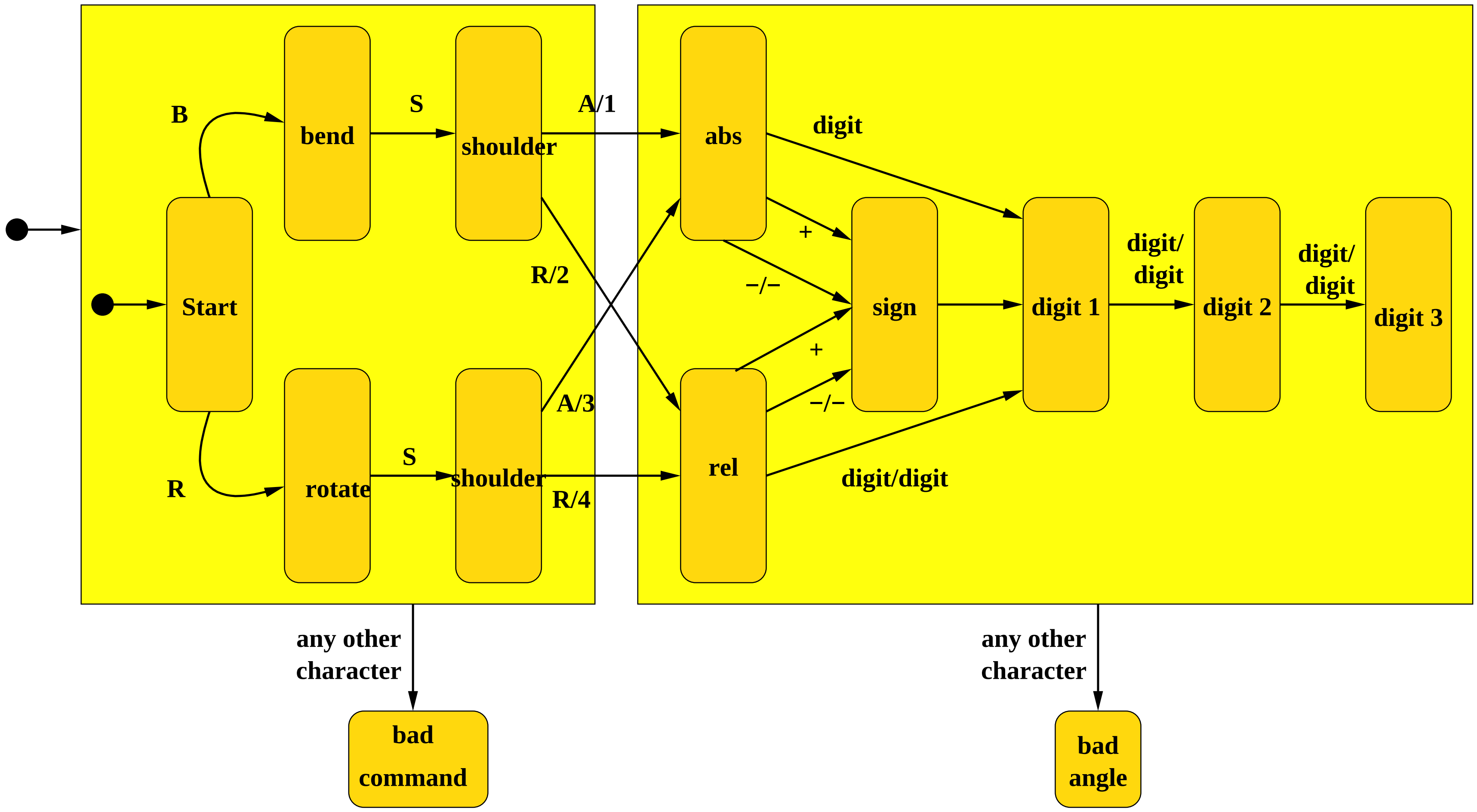


Composite States

- states can be grouped into a composite state with designated start node ($\rightarrow$ hierarchy)
- edges may start and end at any level
- transition from a composite state $\cong$ set of transitions with identical labels from all members of the composite state
- transition to a composite state leads to its initial state
- transitions may be “stubbed”

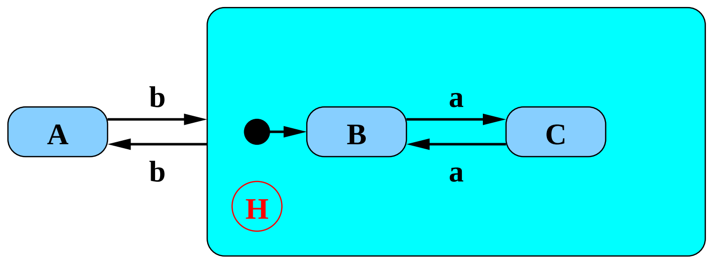


Example: Robot Control



States with History

- composite state with history — marked **(H)** — remembers the internal state on exit and resumes in that internal state on the next entry

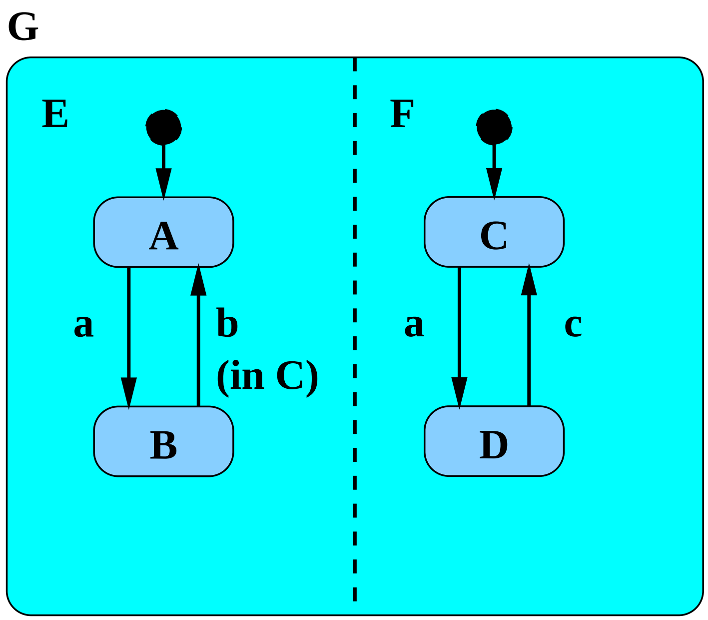


- the history state indicator may be target of transitions from the outside and it may indicate a default “previous state”
- “deep history” **(H*)** remembers nested state

Concurrent States

- composite state may contain **concurrent state regions** (separated by dashed lines)
- all components execute concurrently
- transitions may depend on state of another component (synchronisation)
- explicit synchronization points
- concurrent transitions

Example:



sequence of states on input abcb:

$(A, C), (B, D), (B, D), (B, C), (A, C)$

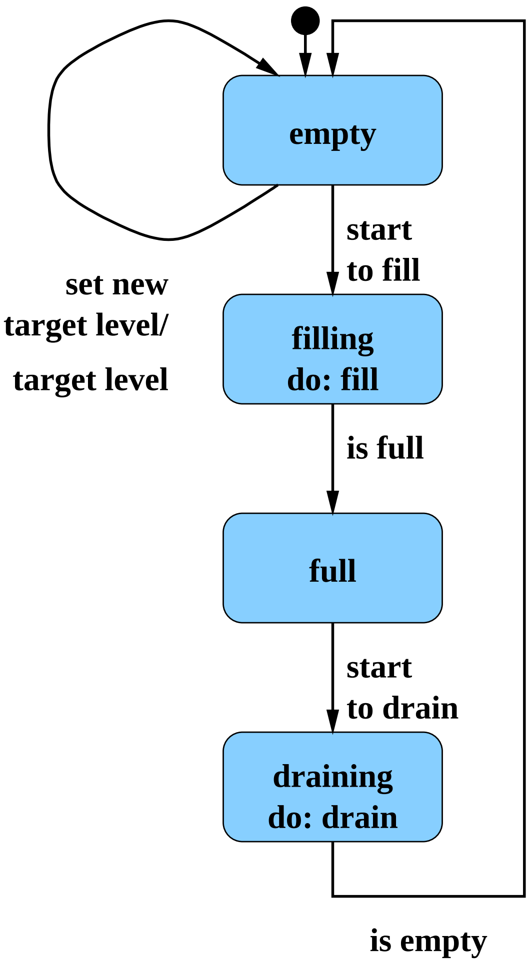
Statecharts and class diagrams

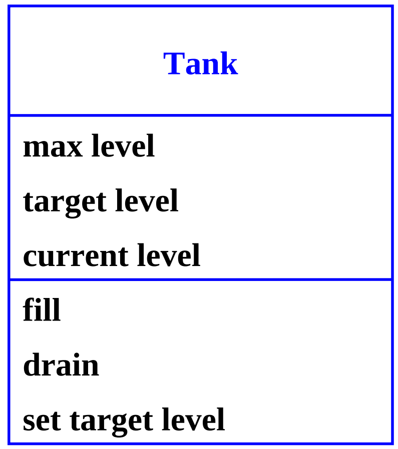
- operations can only be executed in particular state
- idea: incoming message (in class diagram) $\hat{=}$ event (in HA) that triggers the operation
- trivial event names may be dropped

Alternative use

- class has operation that determines reception of an event

Example: Tank





- can fill only if empty
- can drain only if full

Activity Diagrams (UML)

- flow diagrams + concurrency
- influenced by Petri nets, event diagrams (Odell), statechart diagrams (Harel)
- → modeling of workflow, parallel activities
- → refinement of use cases

Ex: Activity Diagram