

Approaches to Analysis

- standard approaches combine fundamental techniques
- state-of-the-art:
 - **OOA (Object Oriented Analysis)**:
class diagrams, ER, FSM, sequence diagrams,
(pseudo code, decision tables)
 - **SA (Structured Analysis)**:
hierarchy of DFDs (→ function tree), DD, decision tables,
decision trees, pseudo code
 - **SA/RT (Real Time Analysis)**:
SA, FSM, (ER)

3.3 Object Oriented Analysis

- after introduction of OOP: need for OOA and OOD
- initially many different approaches:
Booch, Rumbaugh (OMT), Coad/Yourdon, Jacobson (OOSE), Wirfs-Brock, ...
- now: Standardization **UML** (Unified Modeling Language, Booch/Jacobson/Rumbaugh)
- combines
 - **class diagrams** extended with concepts from ERD
 - **interaction diagrams**
 - **Harel automata**
 - **use cases** (Jacobson)
 - **activity diagrams**
 - **deployment diagrams** (*Einsatzdiagramme*)

3.3.1 Perspectives of a Class Diagrams

Meaning of a concept in a class diagram depends on chosen perspective

- conceptual perspective:
application perspective — without regard for eventual implementation
- specification perspective:
interface perspective — identifies functionality (ADT) but not implementation
- implementation perspective:
code perspective

Conceptual Perspective

- boxes represent concepts of the application related to implementation classes, but not necessarily a direct mapping
- attributes and associations indistinguishable
- no distinction between values and references
- association denote relationship, not necessarily navigation
- language independent
- identifies only high-level features
- only important attributes (not types, values, visibilities)
- only important operations (names only)

Specification Perspective

- software interface view / responsibility view
types rather than classes, several implementations possible
- attributes and associations distinct:
attribute — value, association — reference
- refers to interfaces rather than classes
- most attributes and operations (except trivial ones)

Implementation Perspective

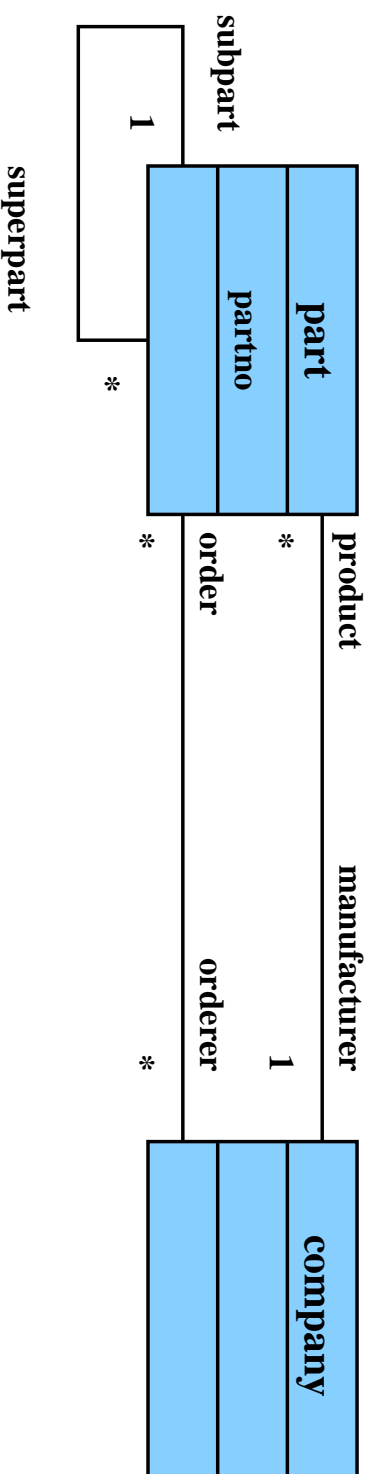
- concepts = classes
- associations no longer imply navigability; navigation through explicit operations and attributes
- operations and attributes as defined in classes

3.3.2 Associations

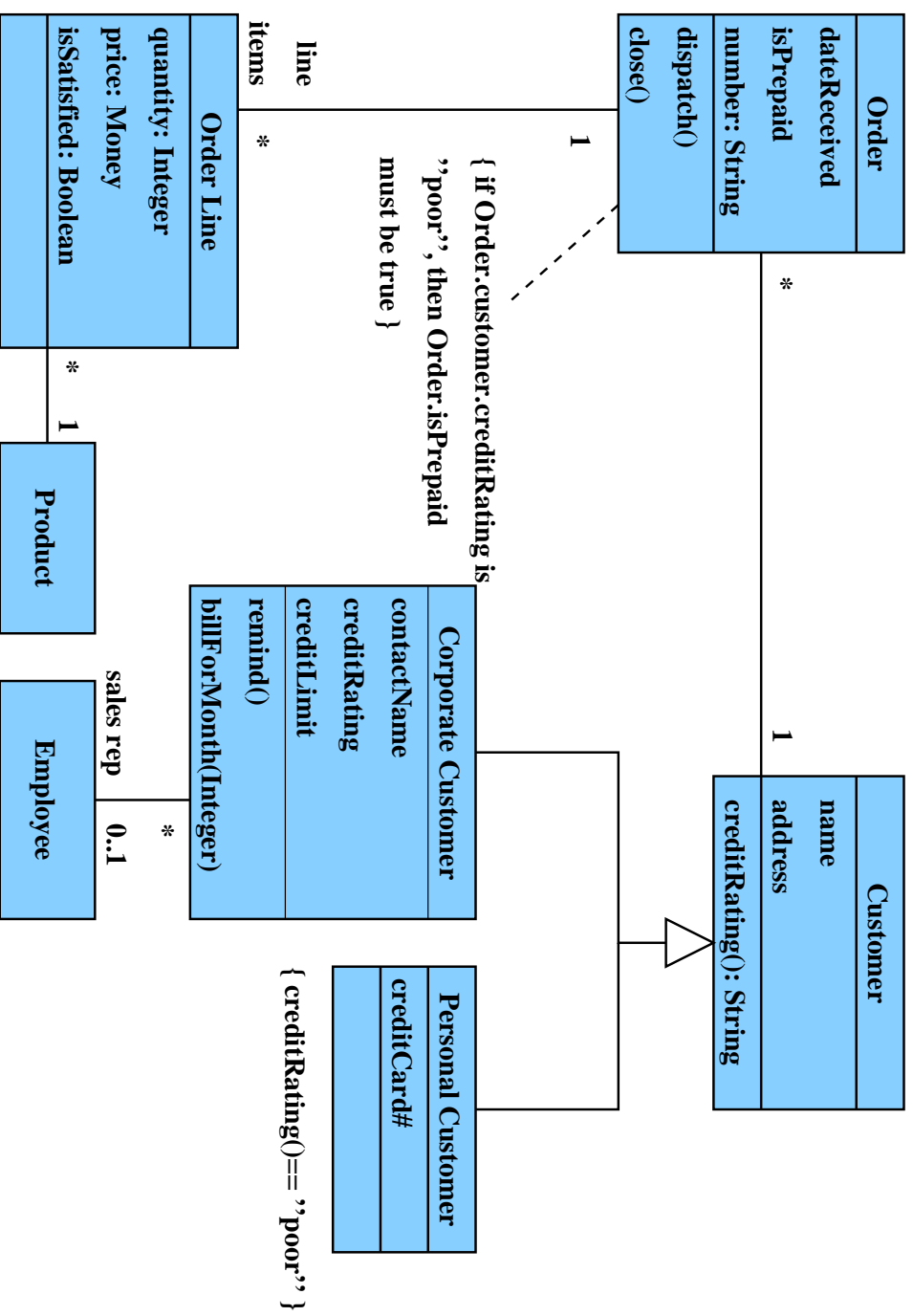
Conceptual Perspective:

- relation between objects of two classes
- from point of view of each class one **role name**
default: name of the neighboring class
required, if more than one association between two classes
otherwise, improves clarity
- alternatively: named association
- role corresponds to attribute (object or set of objects)
- **cardinality:** 1, n, 0..1, n..m, * ($\hat{=}$ 0.. ∞)
or sequence thereof: 1, 3..5, 7

Example: class diagram with associations



Example 2: class diagram with associations



Constraints

- Constraints (*Restriktionen*) wrt object state or association using comments
 - Notation: `{constraint}`
Ex: `{sorted}`, `{immutable}`, `{read-only}`
 - natural language, pseudo code, predicate logic, ..., **OCL**
- Design by Contract (Bertrand Meyer, Eiffel)

Possible Constraints

- 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

Associations: Specification Perspective

- correspond to responsibility to supply operations to access neighboring classes and to maintain the relation
- Ex: company has operation to identify parts manufactured
- convention:
 - unary relation
operation yields neighboring object
 - multi-ary relation
operation yields collection of neighboring objects
 - Ex: (Java)

```
class Teil {  
    public Part partOf();  
    public PartList subParts();  
    public Company manufacturedBy();  
    public CompanyList orderedBy();  
}
```

- Specification perspective does not fix an implementation
- in Ex. several possible implementations of `partOf()`:
 1. Part contains a pointer to `partOf()`, or
 2. scanning all parts and determine if current part (`this`) is a subpart

Operations to maintain relation:

- constructor of Part inserts link to manufacturer
- operation `addCustomer` adds customer

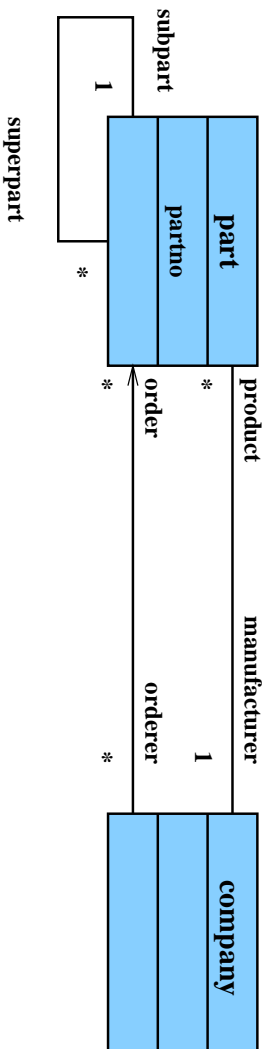
Collections

- default: Set
- using constraints:
`{bag}`, `{ordered bag}`, `{dag}`, ...

Association: implementation perspective

- corresponds to pointer (OID) to neighboring object (or collection thereof)

Constraining Navigation of Associations



Example:

- in Ex: possible to navigate from company to ordered parts, but not the other way round
- directions should only be specified in design or implementation phase

3.3.3 Attributes

Conceptual View: similar to associations

Specification View:

- object enables access and manipulation of attribute value (but not from attribute to object)
- **value semantics** (instead of **reference semantics** as with associations)

Implementation View:

- attribute is part of object state (instance variable)
- value semantics

UML syntax: `<visibility> <name> : <type> = <defaultValue>`
`<visibility>`, `<defaultValue>`, and `<type>` may be dropped

3.3.4 Operations

UML syntax:

```
<visibility> <name> (<parm list>) : <return type> {<properties>}  
only <name> is required
```

Derived Attributes and Associations

- attributes and associations, which are computed from others

Ex: /duration from start time and end time

- specification view does not fix implementation

3.3.5 Aggregation and Composition

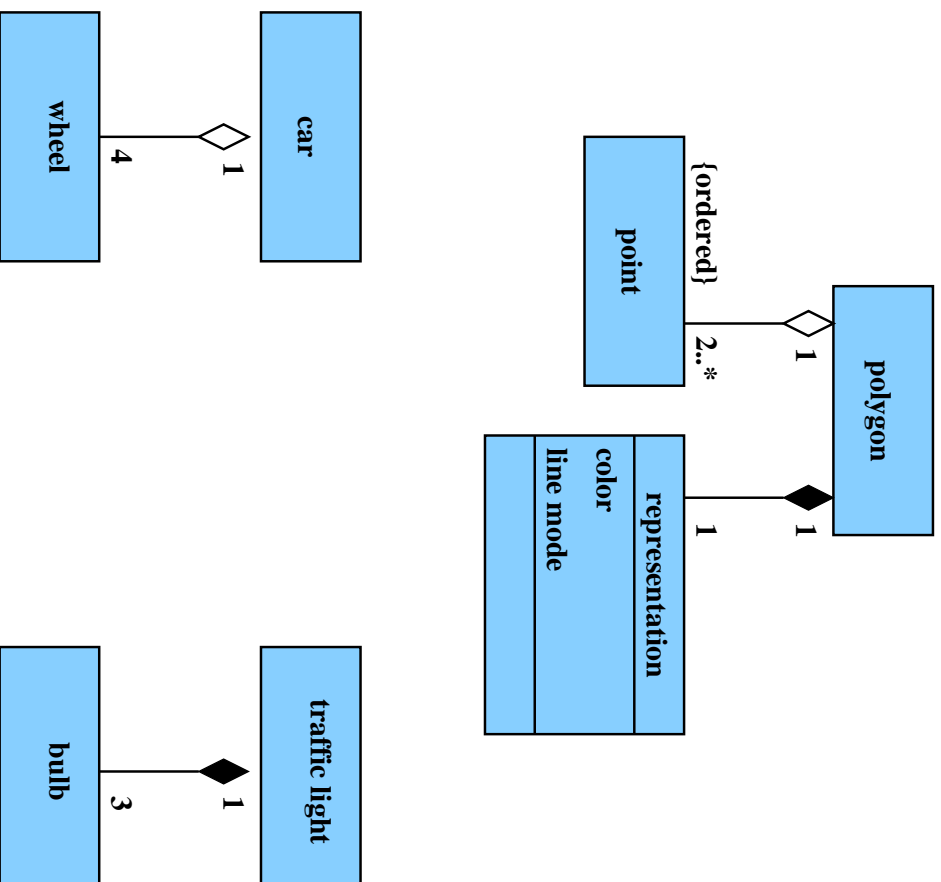
Aggregation

- aggregation is a particular association **part-of**
- Meaning: object “belongs” to other object
- Notation: edge with white rhombus as arrow head

Composition

- special case of aggregation, where the parts belong “**existentially**” to the container
- components and object live and die together
- Notation: edge with black rhombus as arrow head

Example: Aggregation and Composition



3.3.6 Inheritance and Classification

inheritance specifies generalization relation between types

transitive relation: if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.

classification relationship between object and its type

not transitive (relates different sets)

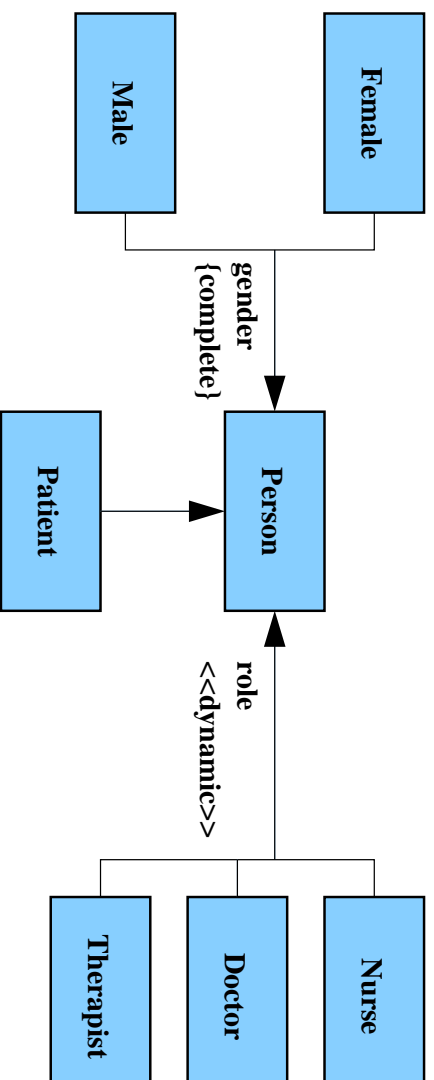
multiple inheritance a type can have more than one generalization

multiple classification an object can have more than one type (**role**)

dynamic classification the type (role) of an object changes over time

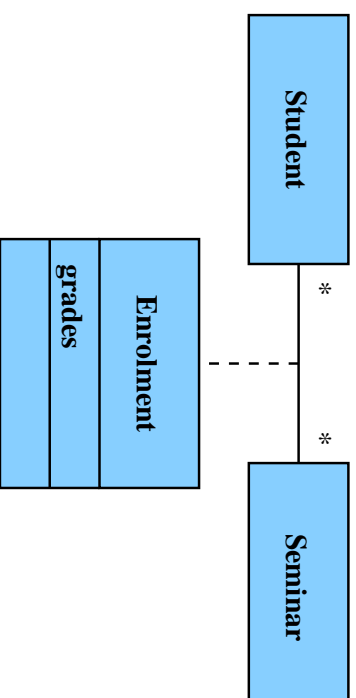
Multiple Classification

- OOP asks for single static classification (i.e., inheritance only wrt one aspect at a time, object cannot change subclass)
- in analysis sometimes better: classification wrt multiple aspects



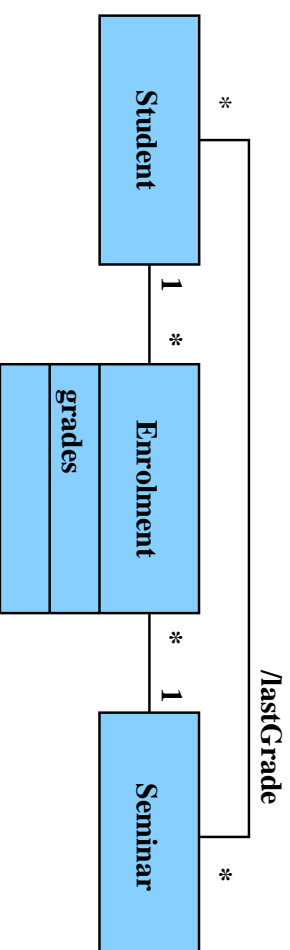
A Person object may be Female, Patient, and Nurse at the same time. Role may change **dynamically** (useful in analysis, problematic to implement).

3.3.7 Association Classes



- attributes and operations linked to association
- **only one** association object per pair of associated objects
- in Ex: each enrolment of each student has exactly one instance

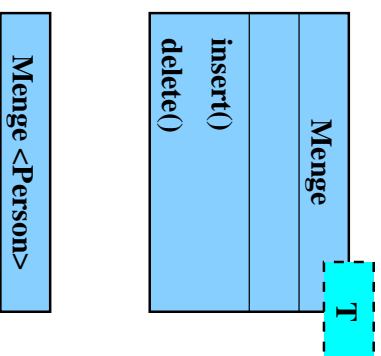
Similarly:



but: multiple instances of an enrolment for each student possible (not desired)

3.3.8 Parameterized Classes

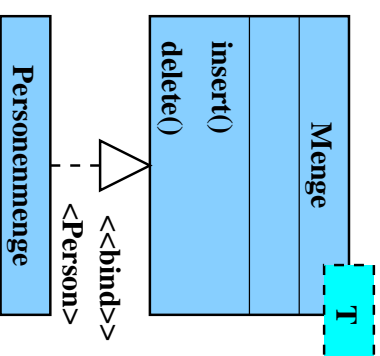
- add parametric polymorphism
- in particular for collection classes



Implementation in C++: Templates

```
class Set <T> {  
    void insert(T element);  
    void delete(T element);  
}  
:  
:  
Set <Person> persons;
```

Alternative Representation:



- difference to inheritance relation:
neither changed attributes nor operations
- replacement at compile time (\leftrightarrow late binding)
- \rightarrow mostly used in design/implementation phase

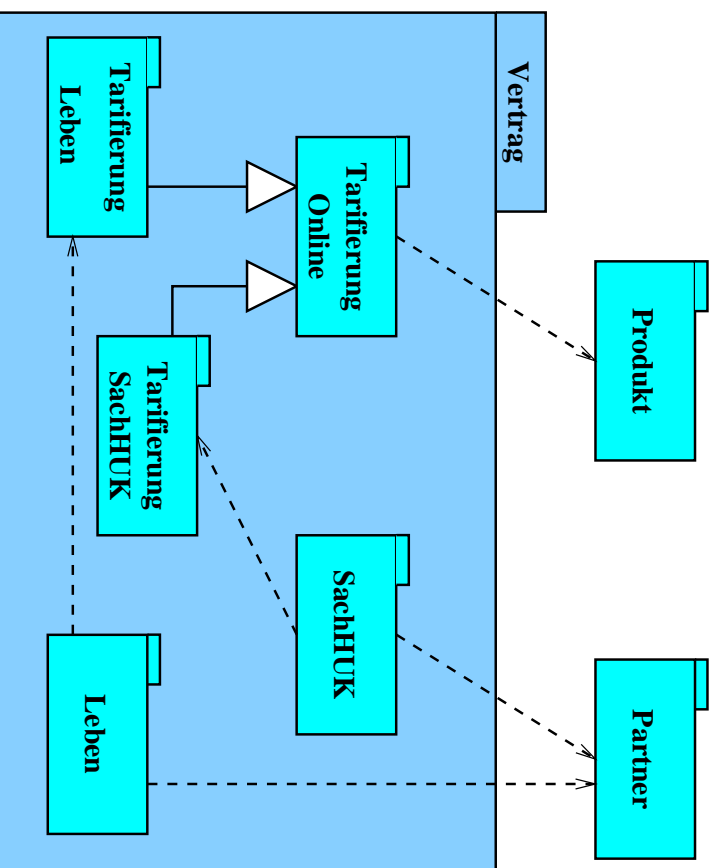
3.3.9 Visibility

- in UML: visibility similar to C++
- attributes/operations:
 - **public** (UML: +): visible everywhere
 - **private** (UML: -): only visible in own class
 - **protected** (UML: #): visible in own class and its subclasses
- → mostly used in design/implementation phase
- problem:
each OO-language has different visibility rules (→ adaption)

3.3.10 Structuring of Large Systems

- goal: decomposition of large system into **subsystems**
- minimization of dependency between subsystems
- in UML:
 - **package**
 - * part of system with its own class diagram (potentially *subpackages*)
 - * representation as filing card
 - dependency between packages
 - * change of one package may require change in neighboring package
 - * representation: dashed arrow
- decomposition into packages: Java (visibility: package)

Example: Decomposition in Packages



P_2 depends on P_1 if:

- class C_2 in P_2 sends message to C_1 in P_1
- operation in C_2 has argument of type C_1
- dependencies are not transitive, in general
- \rightarrow inheritance of packages, \rightarrow abstract packages

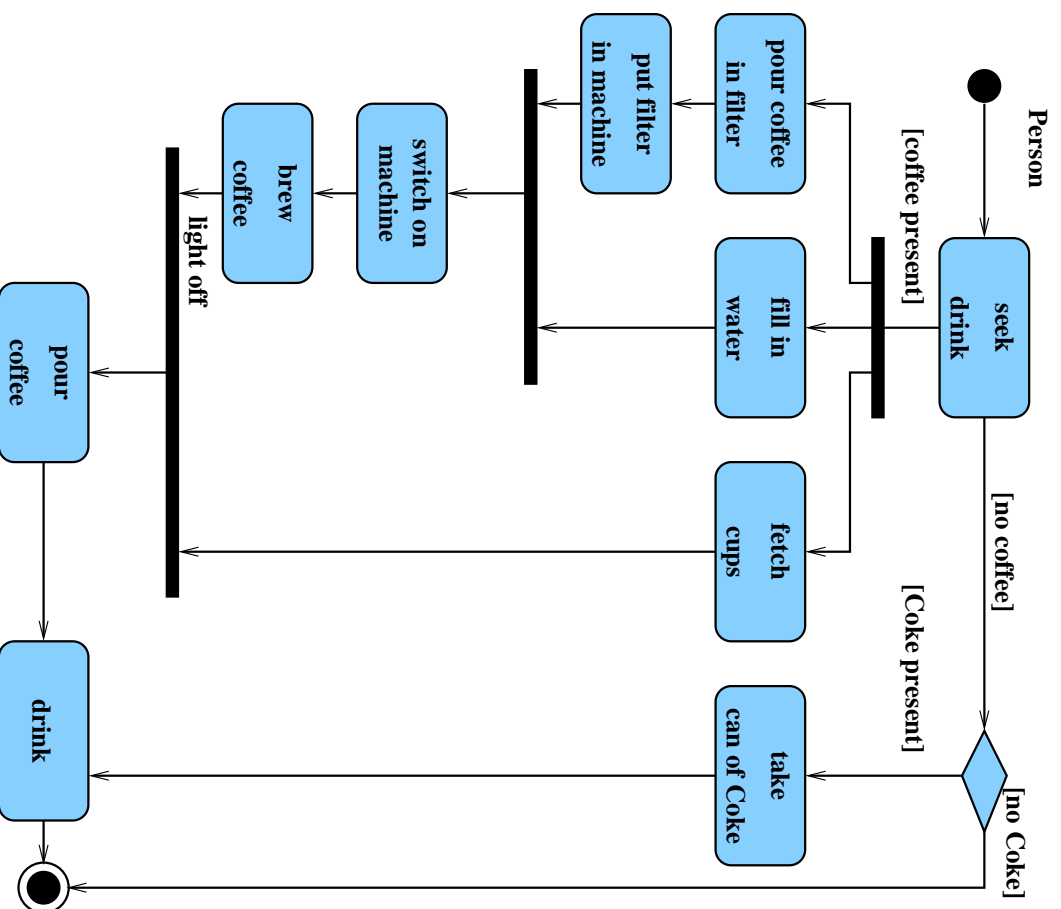
Decomposition into Packages

- good decomposition difficult
- rules of thumb
 - logically connected topics
 - implementation and test in isolation
 - inheritance hierarchy should be cut at most vertically (if at all)
 - no aggregation should be cut
 - cut as few associations as possible

3.3.11 Activity Diagrams

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

Ex: Activity Diagram



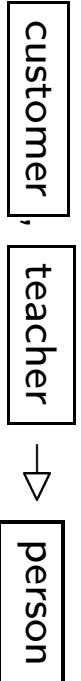
3.3.12 OOA Patterns

- certain constellations of classes with fixed responsibilities and interactions recur during analysis
- constellations abstracted into **patterns**, which encapsulate common analysis solutions

Pattern 1: Abstract Superclass

Situation: many classes have identical attributes and operations

Approach: transfer attributes and operations to new abstract superclass

Ex:  customer, teacher \rightarrow person

Pattern 2: Concrete Superclass

Situation: there is a set of classes, whose attributes and operations are supersets of the attributes and operations of class C

Approach: C becomes superclass

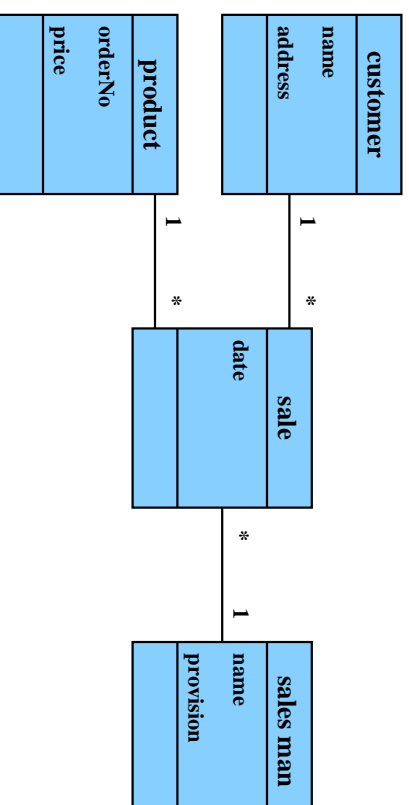
Pattern 3: Associations with Properties

Situation: information about an association required

Approach: use association class **or** introduce new class, which contains the required information

Pattern 4: coordination of objects

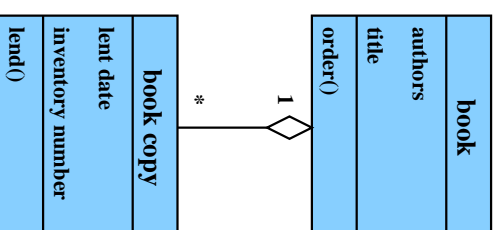
- on a very high level of abstraction, classes have few attributes and operations
- they rather coordinate interaction between other classes



Pattern 5: Exemplar Type

Situation: attribute values are shared between many objects

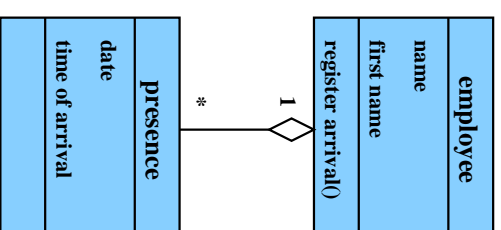
Approach: create abstract notion to factor out commonalities, implement as separate class, model using aggregation



Pattern 6: Register Events

Situation: register events for an object

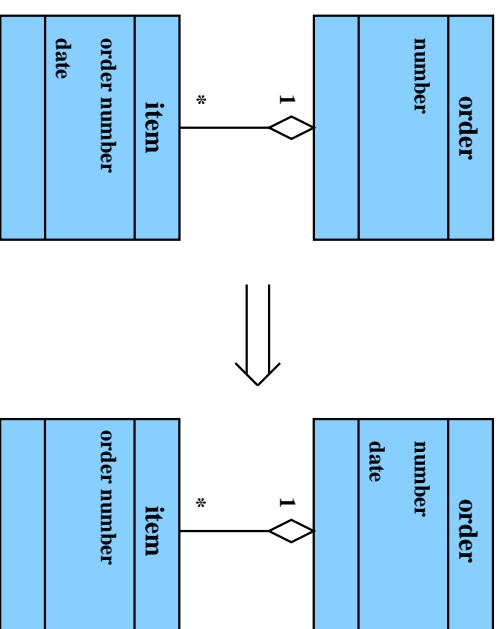
Approach: aggregate class of object with event class



Pattern 7: Lifting of attribute values

Situation: all components of an aggregation have a common attribute value

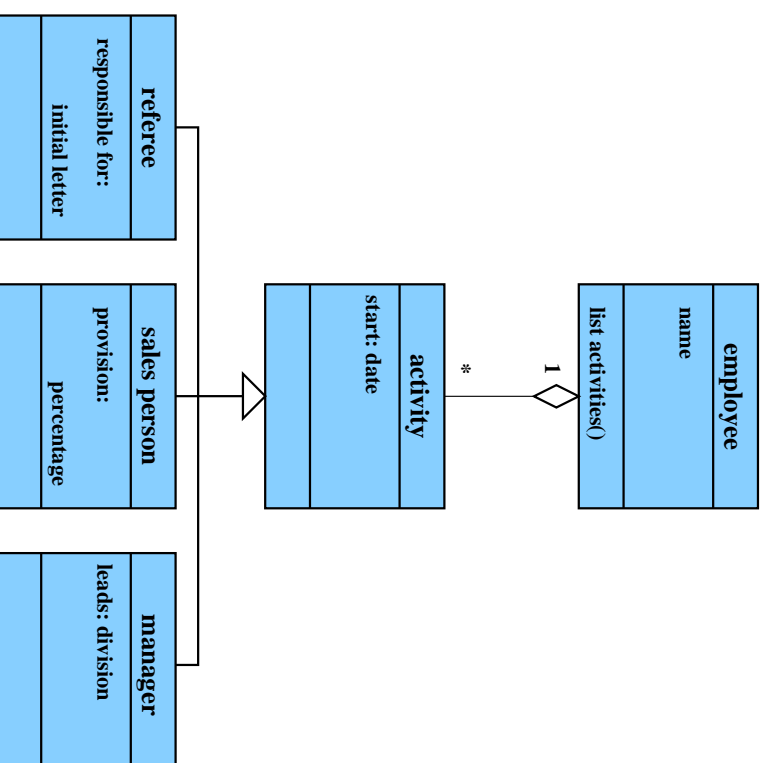
Approach: lift this attribute into the superclass



Pattern 8: History of Dynamic Classification

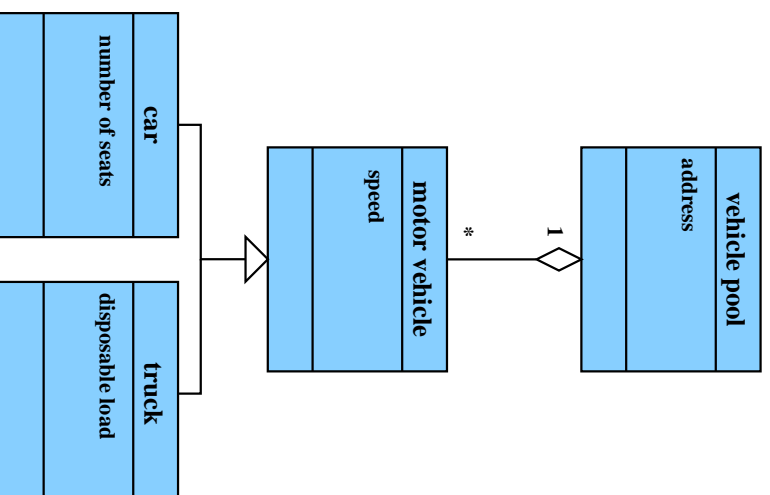
Situation: track the dyn. classification of an object

Approach: class aggregates history



Pattern 9: “Power-Types”

class, which aggregates object of subclasses of another class



3.4 OOA Methodology

- no generally accepted methodology

1) data-centric approaches

- Rumbaugh'91
- Shlaer/Mellor'92

2) function-oriented approaches

- Wirfs-Brock'90
- Jacobson'92 (use cases)
- Rubín/Goldberg'92

3) Synthesis of 1) and 2)

- Coad/Yourdon'91
- Booch'94

3.4.1 Methodology of Heide Balzert: 10 Steps Towards an OOA Model

1. identify classes
2. identify associations and aggregations
3. attributes and operations for each class
4. construct object life cycle
5. introduce inheritance
6. identify internal operations
7. specify operations (e.g. using pseudo code)
8. check inheritance
9. check associations and aggregations
10. decomposition in subsystems

Step: Identify Classes

1. identify concrete objects
 - in technical systems: physical objects
 - in commercial systems: forms
2. top-down:
 - scan verbal requirements for attributes / operations?bottom-up:
 - collect attributes (data) and operations
 - combine into classes
3. name of class: concrete noun, singular, describes all objects (no roles)
4. classes related via invariable 1:1 associations may be joined

Step: **Associations and Aggregations**

- permanent relations between objects
- scan verbal requirements for verbs
- technical subsidiarity: aggregation
- communication between objects → association
- determine roles
- snapshot / history required?
- constraints?
- are there attributes / operations for association?
- determine cardinalities

Seminar-Example: Form-Analysis

booking form

participant: title first name name

seminar number subject date

acknowledgement and bill to:

title first name name

company street

post code city phone

participant
title
first name
name

Seminar
seminar number
subject
date

billed party
title
first name
name
company
street
post code
city
phone

From Verbal Requirements:

customer
customer number
name
address
function
sales
first entry() change() delete()
create address stickers() create letter()

seminar booking
registered on
acknowledgement on
...
register() change() delete()
send acknowledgement() cancel() send message() invoice()

seminar type
subject
...
first entry() change() delete()

company
name
address
sales

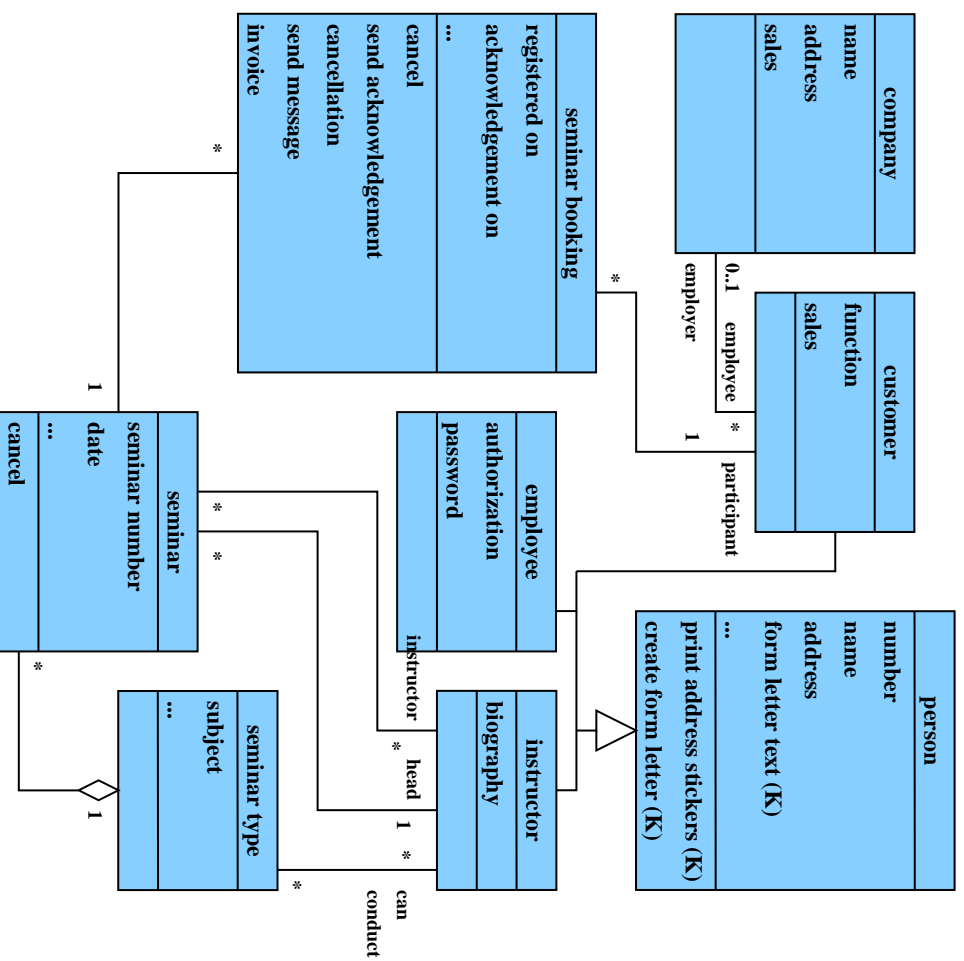
employee
name
authorization
password

delay in payment
billing date
billing amount
enter delayed payment()

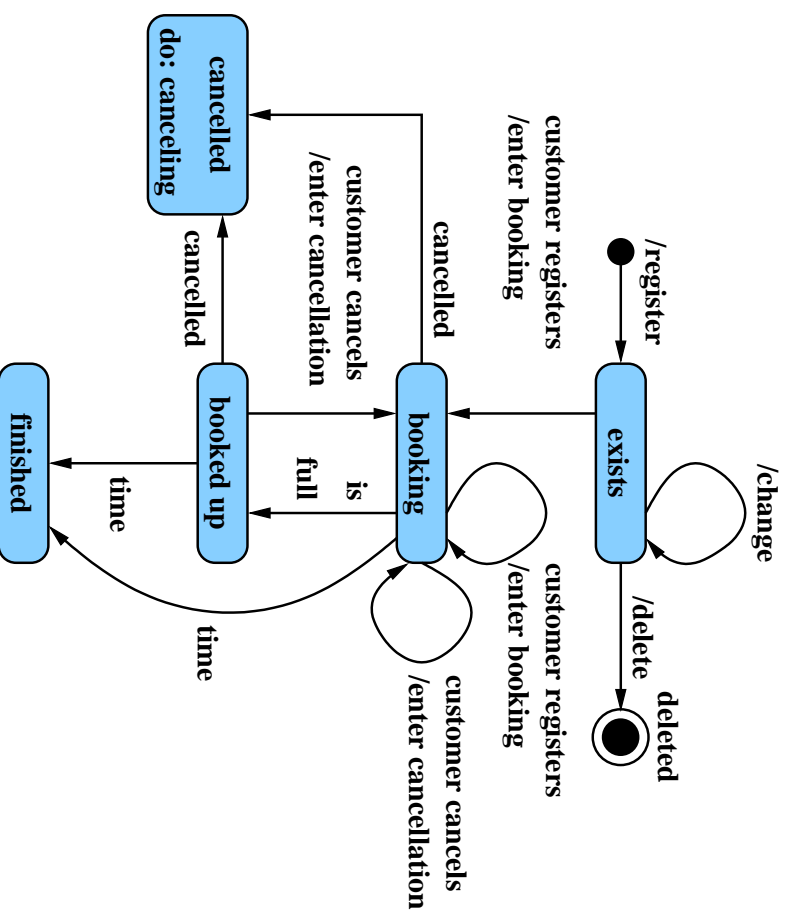
seminar
seminar number
date
...
first entry()... cancel() note execution() list participants()

instructor
name
address
biography ...
first entry() ... associate seminar() associate seminar type() create form letter()

Class Diagram from Ex. (simpl'd)



Automaton from seminar administration



→ internal operations enterBooking() and enterCancellation()

Operations

```
operation enterBooking(out result):  
  if state = exists or state = booking  
    then incrementNrParticipants()  
       result := "ok"  
  else result := "no booking possible"  
  if state = exists  
    then state := booking  
  if nrParticipants = maxParticipants  
    then state := booked up
```

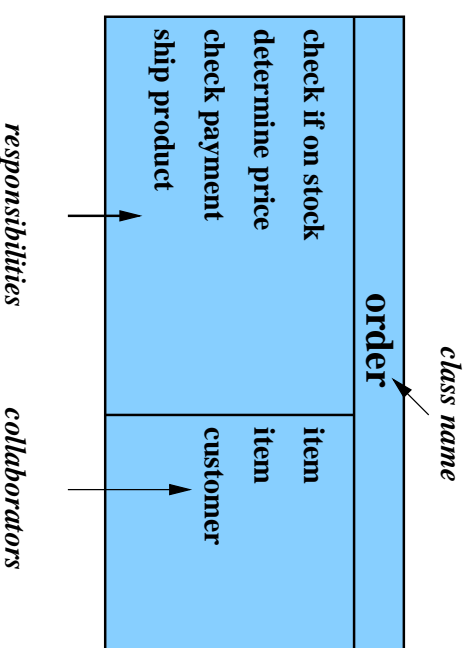
3.4.2 Methodology of Jacobson

Three kinds of objects:

1. **interface objects**
communication with outside world
2. **control objects**
coordination of other objects
guided by use cases
3. **entity objects**
data encapsulation
low probability of change

3.4.3 CRC Cards

- CRC = Class-Responsibility-Collaboration (Wirfs-Brock)
- initially, a class is assigned **responsibilities** and **collaborators**
- collaborator is a class cooperating to fulfil responsibilities
- **at most three** responsibilities per card (class); otherwise: split class



3.5 Structured Analysis (SA)

- classical approach for requirements analysis
- (still) industry standard
- well supported by CASE-tools
- simple to learn and understand
- Tom DeMarco ('78)

SA combines fundamental techniques:

- data flow diagrams (DFDs)
i.e.: consistent hierarchy of DFDs
→ function tree (→ design)
- data dictionary (DD)
- decision tables
- pseudo code

Approach:

1. physical model of existing system
2. logical model of existing system
3. new logical model
4. new physical model (→ design)

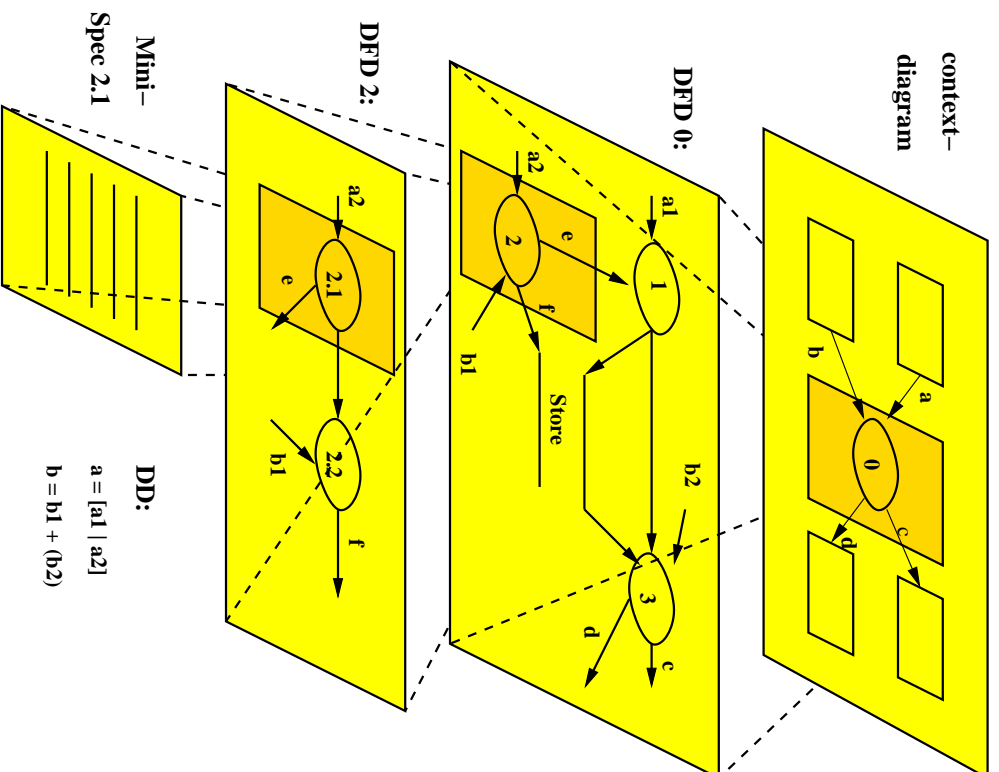
Drawbacks of SA:

- ignores flow of control; only data flow
- too expensive for analyzing heritage systems
- insufficient data modeling
- top-down only
- integration of existing components difficult
- function oriented; transition to OOD difficult due to (lack of) data abstraction

Extensions of SA

- SA/RT (SA/Real Time, Ward/Mellor'85, Hatley/Pirbhai'87)
- modern Structured Analysis (MSA, Yourdon'89)

Example: SA model



Hierarchy of Diagrams

- stepwise refinement
- context diagram
 - only process 0, ≥ 1 interfaces, no store, “appropriate, invariable level of abstraction”
- DFD refines process of superior diagram
- no refinement of store, but repeated in refined diagram
- no refinement of interfaces (perhaps reptd.)
- ≤ 7 processes / diagram
- DD consistent wrt. refinement of data flows and stores (\rightarrow checked by CASE-tool)

Hierarchy of diagrams (cont'd)

- **level consistency**: incoming and outgoing data flows of a diagram correspond to those in superior diagram (or part of them → DD) neither add, nor drop!
- “atomic” process described by MiniSpec using pseudo code or decision table
- in MiniSpec: I/O consistent to superior diagram
- no implementation details in MiniSpec
- tree-structured numbering scheme for diagrams (e.g. 2, 2.1, 2.1.3; → CASE-tool)

3.6 SA/RT

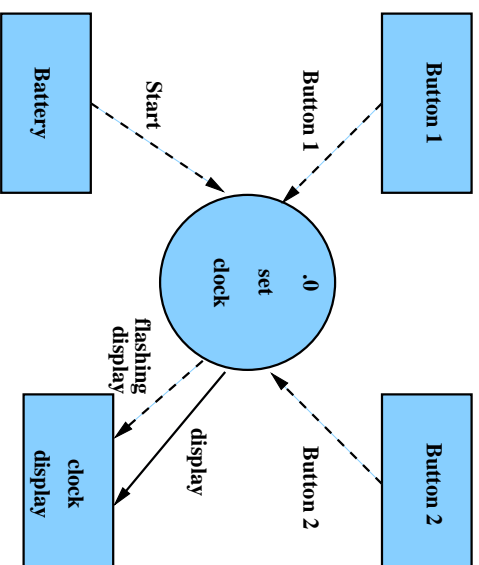
- industry standard for technical systems
- extends SA with
 - finite state machines
 - control flows in DFDs
- suitable for event-driven systems (name “RT” misleading)

Data Flows and Control Flows

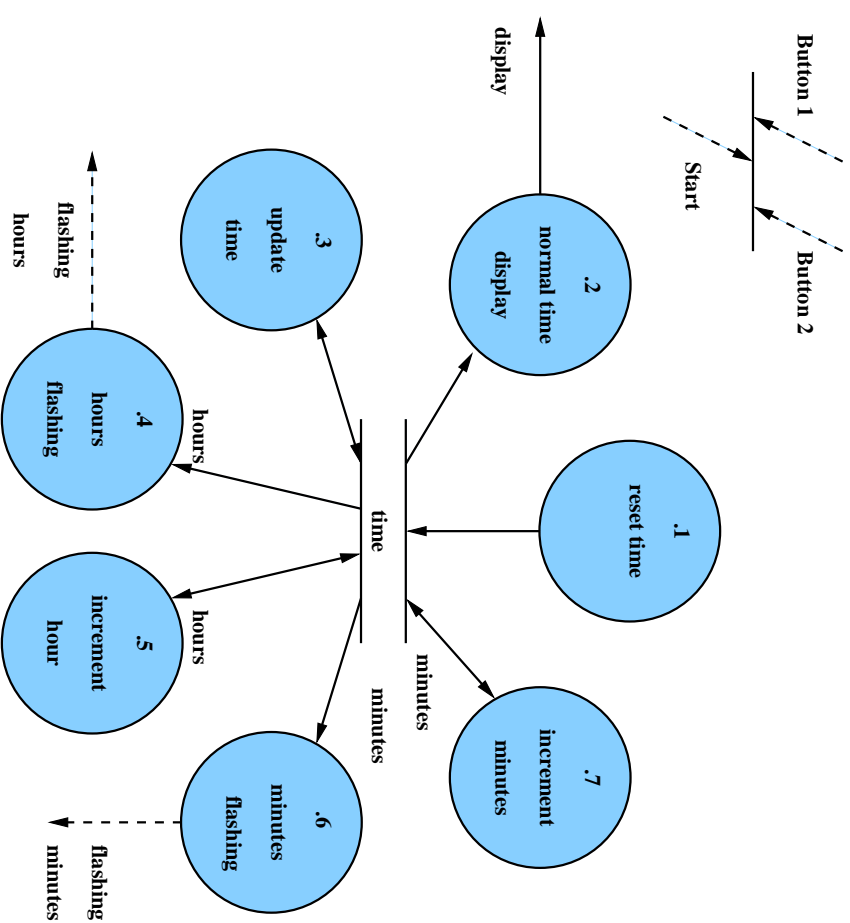
- data flow (continuous or discrete range)
- control flow
 - discrete (even finite) range
 - represented by dashed line
 - entered into DD (requirements dictionary, RD)

Example: digital clock using SA/RT, DFDs

context diagram:



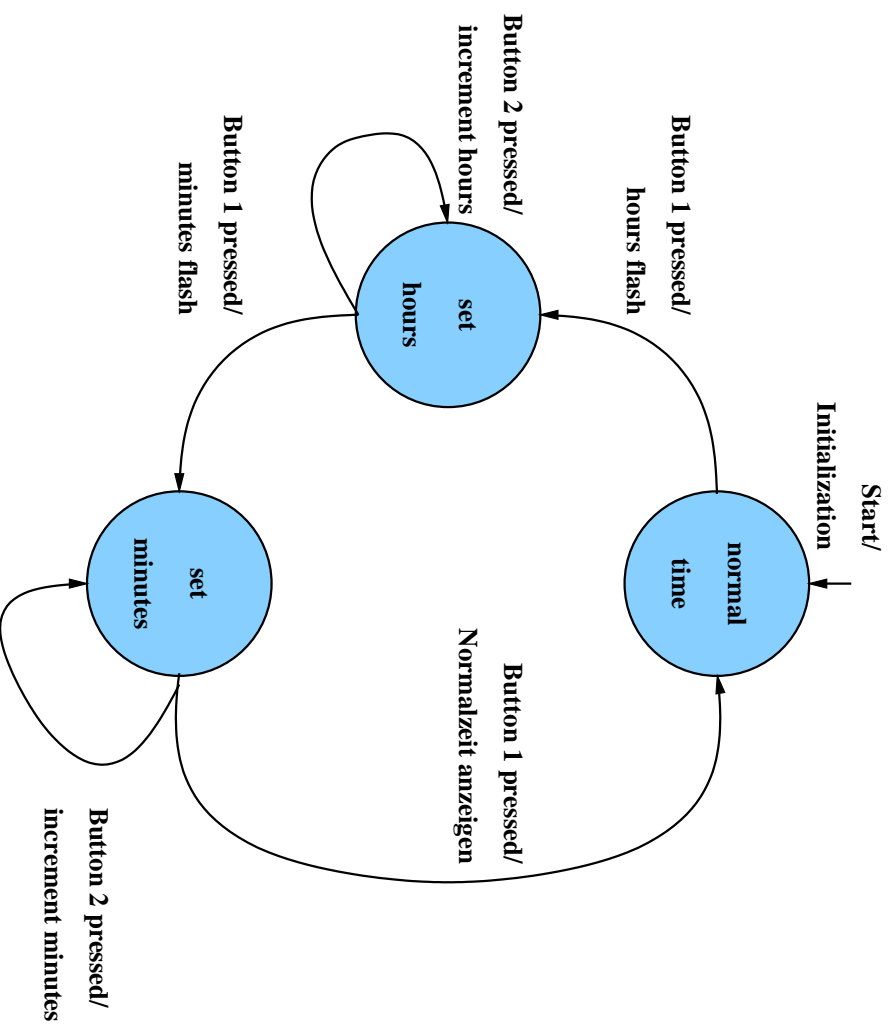
DFD 0:



Data Dictionary

$\langle \text{button1} \rangle$	=	[pressed released]
$\langle \text{button2} \rangle$	=	[pressed released]
$\langle \text{start} \rangle$	=	[electricity no-electricity]
$\langle \text{display} \rangle$	=	$\langle \text{hours} \rangle + \langle \text{minutes} \rangle$
$\langle \text{flashing display} \rangle$	=	[[$\langle \text{flashing hours} \rangle$ $\langle \text{flashing minutes} \rangle$]]
$\langle \text{time} \rangle$	=	$\langle \text{hours} \rangle + \langle \text{minutes} \rangle$
$\langle \text{hours} \rangle$	=	0..23
$\langle \text{minutes} \rangle$	=	0..59
$\langle \text{flashing hours} \rangle$	=	0..23
$\langle \text{flashing minutes} \rangle$	=	0..59

CSpec 0 = Finite State Machine + ...



CSpec 0 = ... + Decision Table

control actions	processes						
	.1	.2	.4	.5	.6	.7	
initialization	1	2	0	0	0	0	
increment hours	0	0	2	1	0	0	
increment minutes	0	0	0	0	2	1	

Process Specification (PSpecs $\hat{=}$ MiniSpecs)

PSpec 1; set time to 0

issue time := 00:00

PSpec 2; display time

display := time

PSpec 3; update time

read time, increment minutes, and write back

PSpec 4; hours flashing

flashing hours := hours

PSpec 5; increment hours

issue hours := (hours + 1) % 24

PSpec 6; minutes flashing

flashing minutes := minutes

PSpec 7; increment minutes

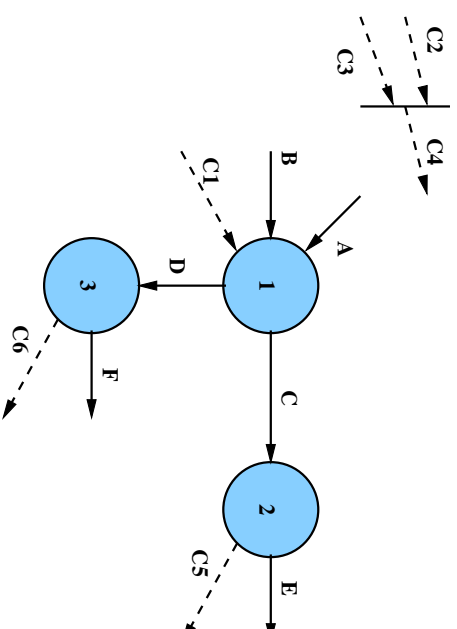
issue minutes := (minutes + 1) % 60

DFDs and CSpecs

- CSpec consists of finite state automaton + decision table
- bar notation specifies connection between CSpec and DFD:
 - incoming control flow of CSpec (event) \cong arrow **to bar**
 - outgoing control flow of CSpec \cong arrow **from bar**
- action of an FSM or DT corresponds to
 - activation of a process in DFD,
 - triggering of an event in DT or automaton, or
 - generation of an outgoing control flow
- processes remain active until state changes or until they deactivate themselves (marked with **issue** in PSpec)
- processes not mentioned in CSpec are active throughout (Ex.: **.3**)

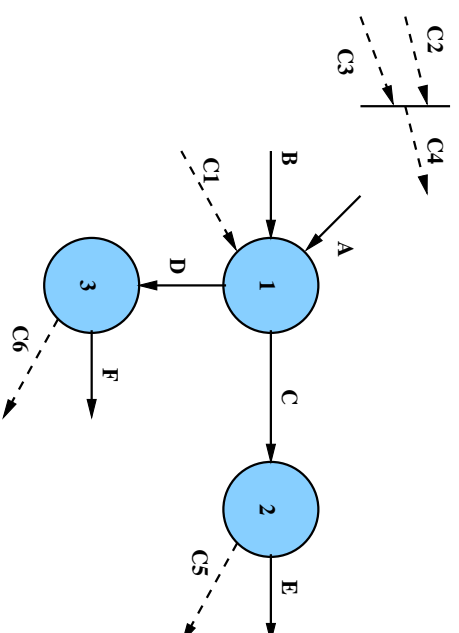
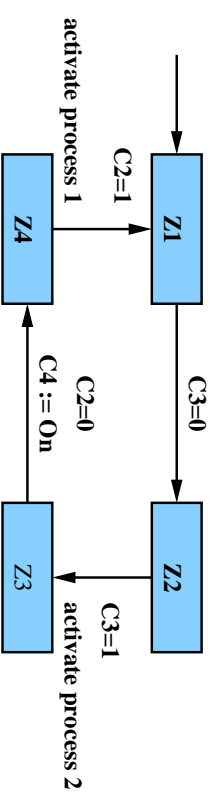
Example: DT and DFD

input	C2	C3	output	C4	process		
					1	2	3
0	0	0	On	1	1	0	
0	1	1	On	3	2	1	
1	0	0	On	0	2	1	
1	1	1	On	1	0	0	



C5 and C6 described in PSpec2 and PSpec3, respectively.

Example: automaton and DFD



C5 and C6 described in PSpec2 and PSpec3, respectively.

Timing Specification in SA/RT

- specification of cycle times and response times
- no checking

Cycle times in DD (RD) for output control flows in context diagram

$\langle hours \rangle = 0..23$ Rate: every 200 msec

$\langle minutes \rangle = 0..59$ Rate: every 200 msec

Response Times in time specification table

input	event	output	event	response time
button 1	pressed	fl. hours	display number	< 200 msec
button 1	pressed	fl. minutes	display number	< 200 msec
:	:	:	:	:
:	:	:	:	:

3.7 Modern Structured Analysis

Fundamental Techniques:

- hierarchy of DFDs extended with control flow
- DFDs describe **events** (cf. use cases in OOA)
→ middle-out instead of top-down
- ER-Diagrams, Data Dictionary
- hierarchical FSM
- pseudo code for atomic processes

Methodology immediate consideration of new system
(no analysis of existing system)

Consistency Checks for Model Integration: (→ Tools)

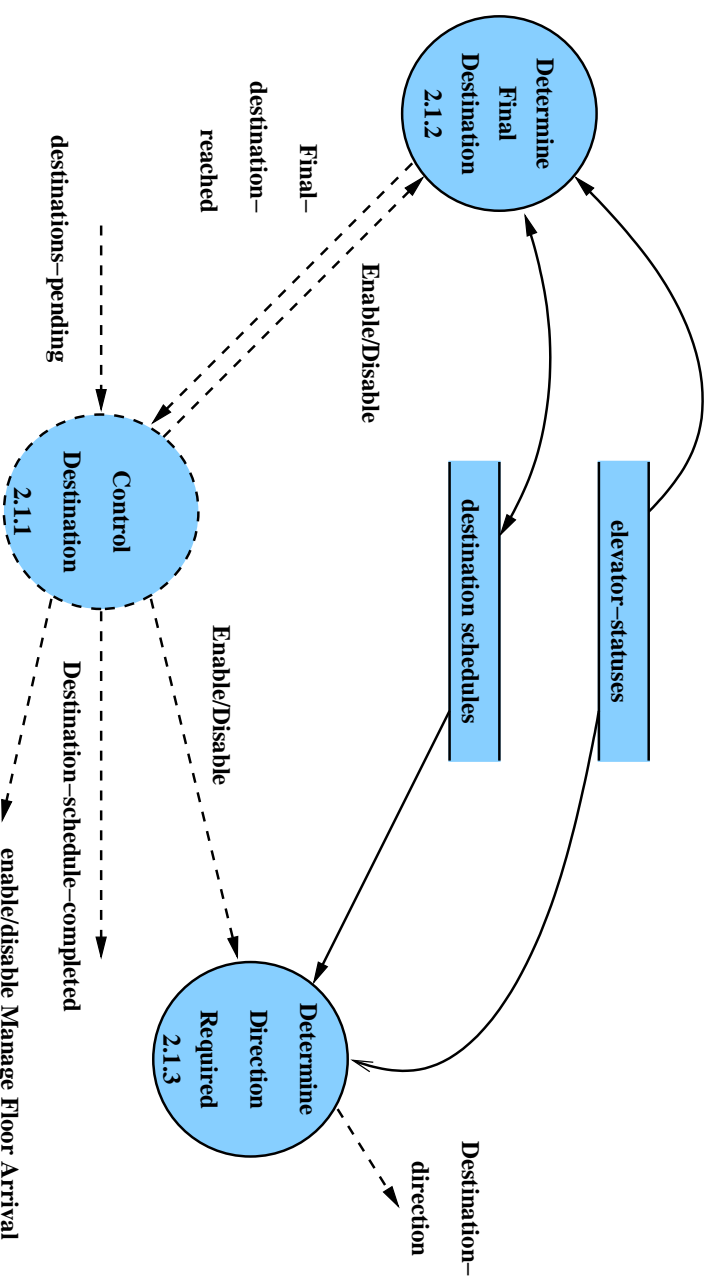
- each data flow and store from DFD described in DD
- all data from DD must be used in DFD, PSpec or DD
- each process from DFD is either refined or described by PSpec
- each PSpec belongs to atomic process from DFD
- PSpec must be consistent to refined process from DFD wrt input/output
- each data in PSpec corresponds to (a component of) an incoming data flow of DFD or a store (or local)

Consistency Checks: (cont'd)

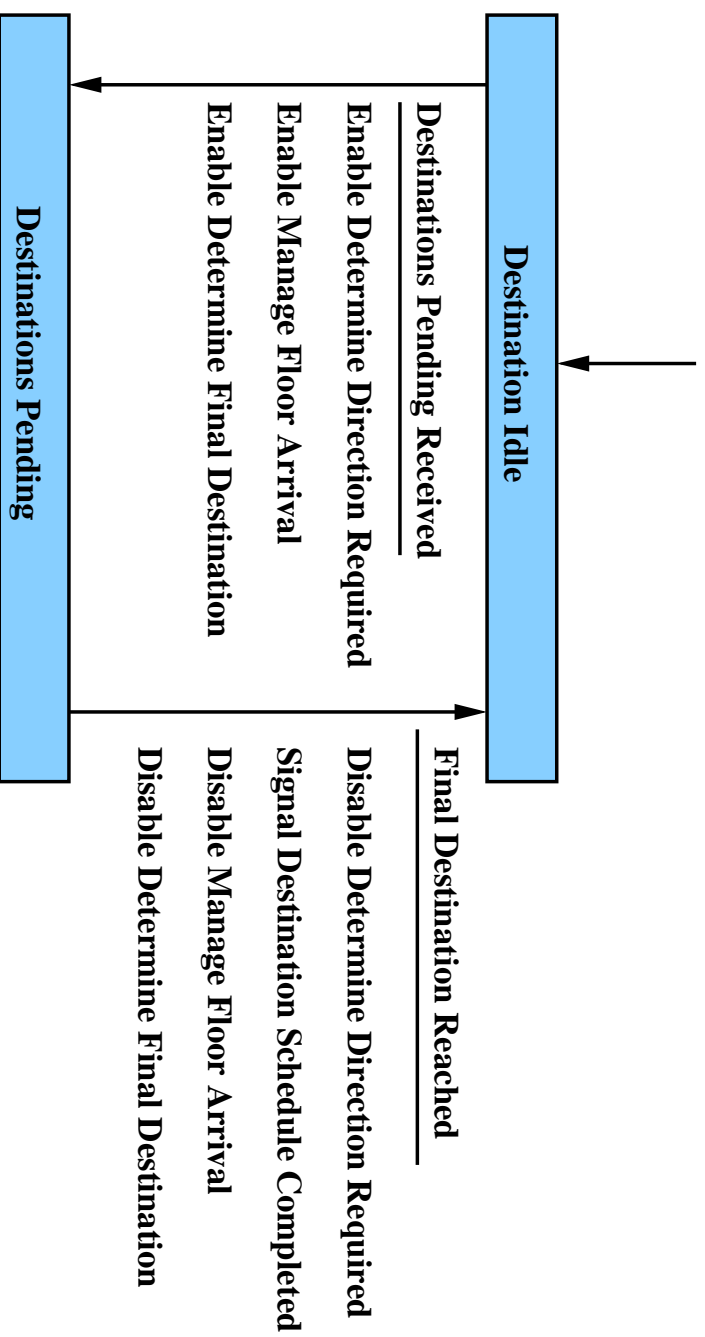
- each store of DFD corresponds to an entity set or to a relation in the ER-Diagram, and vice versa
- there is an FSM for each atomic control process in DFD, and vice versa
- each condition in the automaton corresponds to an incoming control flow in its control process, and vice versa
- each action of an automaton corresponds to an outgoing control flow of its control process, and vice versa

Example: Elevator Control

DFD 2.1:



Automaton 2.1.1:



PSpec 2.1.3: Determine Direction Required

Local term `match` is a matching `elevator-number`

in `destination-schedules` and `elevator-status`

Precondition 1

`match` exists and

there exists in `destination-schedules` a

`destination-floor > current-floor`

in `elevator-status`

Postcondition 1

`destination-up` is produced

Precondition 2

`match` exists and

there exists in `destination-schedules` a

`destination-floor < current floor`

in `elevator-status`

Postcondition 2

`destination-down` is produced

Data Dictionary:

```
destination-schedules = {destination-schedule}
destination-schedule = elevator-number +
    {destination-floor} +
    destination-pending
destination-pending = [ on | off]
destination-floor = [ 1|... | 40]
elevator-number = [ 1|... | 4]
elevator-statuses = {elevator-status}
elevator-status = elevator-number + elevator-state + current-floor
elevator-state = [ parked | moving-down | moving-up | out-of-service]
current-floor = [ 1|... | 40]
destination-direction = [ up | down]
:
:
```

drop unary signals (e.g., `Final-destination-reached`)

ER-Diagram *trivial*