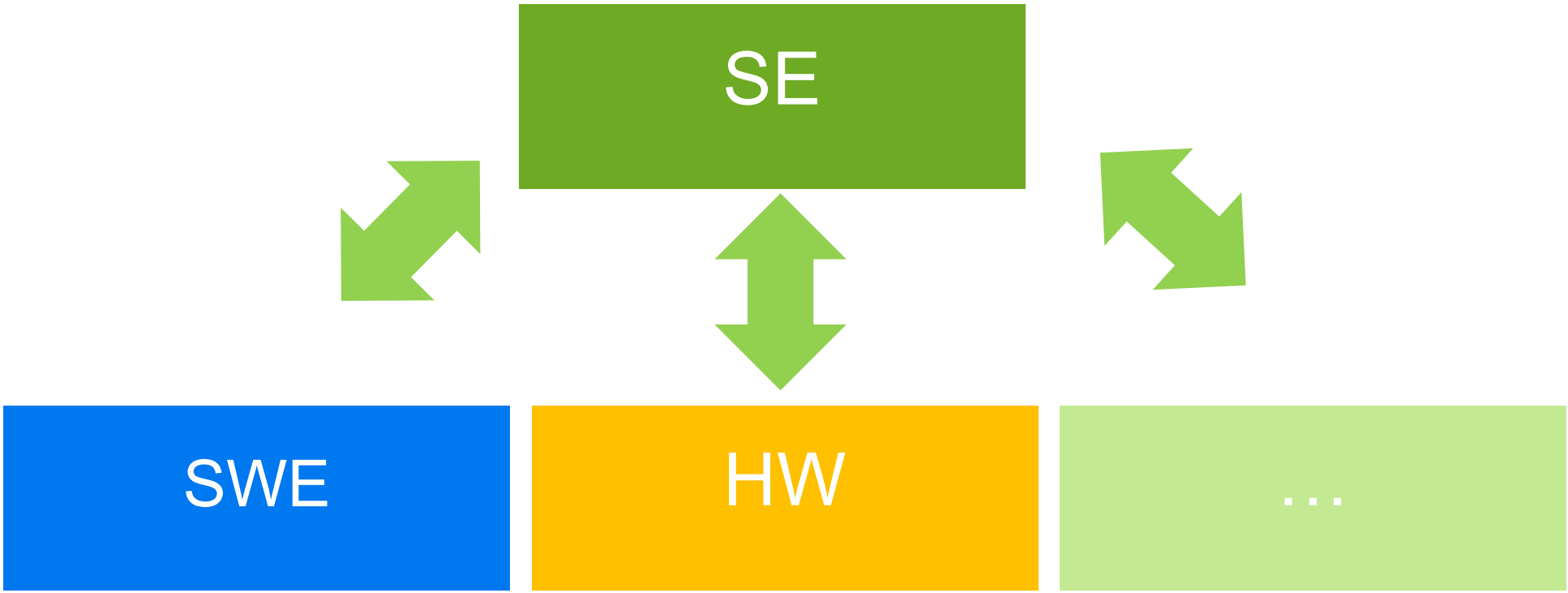
Stefan Henkler E-Mail: stefan.henkler@hshl.de

Outline

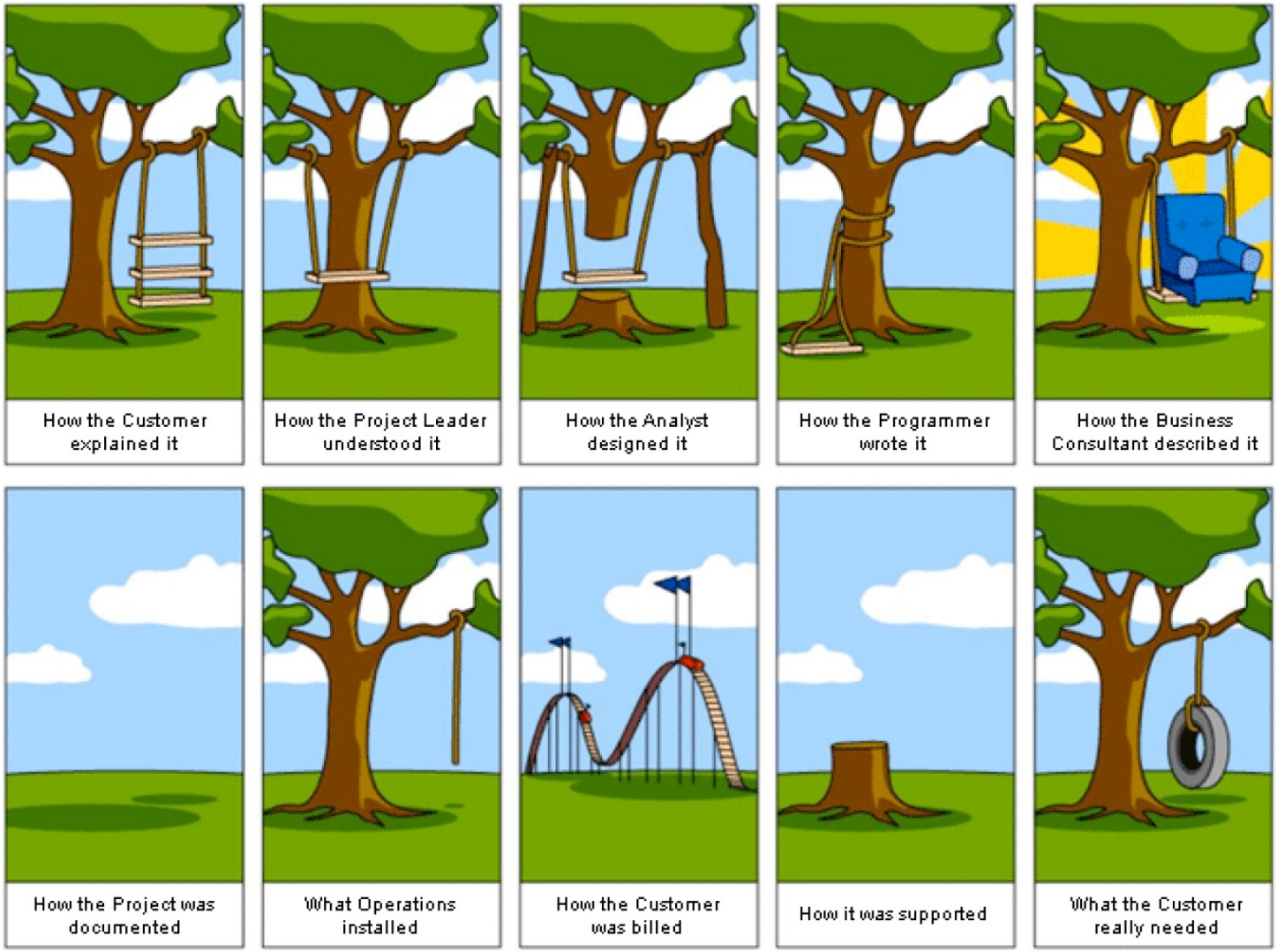
- Handling complexity
  - What is a good design? -> Design principles
- Object oriented design with UML
  - Incl. relevant mapping to implementation
- Manual implementation
  - Language comparison
  - Handling reactive principles on code level incl. memory considerations

Usage: A Systems Eng. Tool

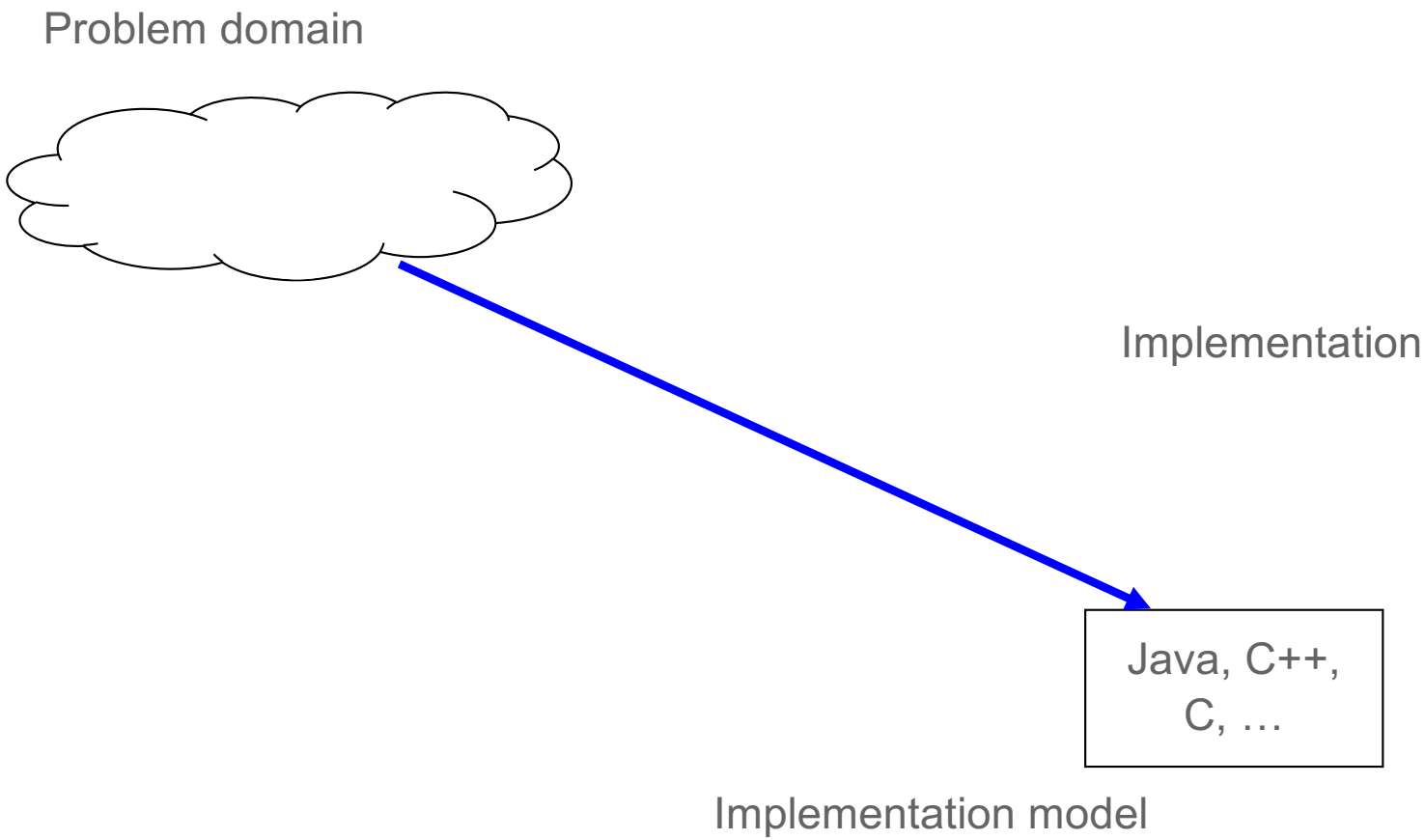
- On Systems level mapping to SWE and implementation
  - Decision on HW/SW done
  - Mapping to different engineering disciplines
    - More or less formal (but could be formal)



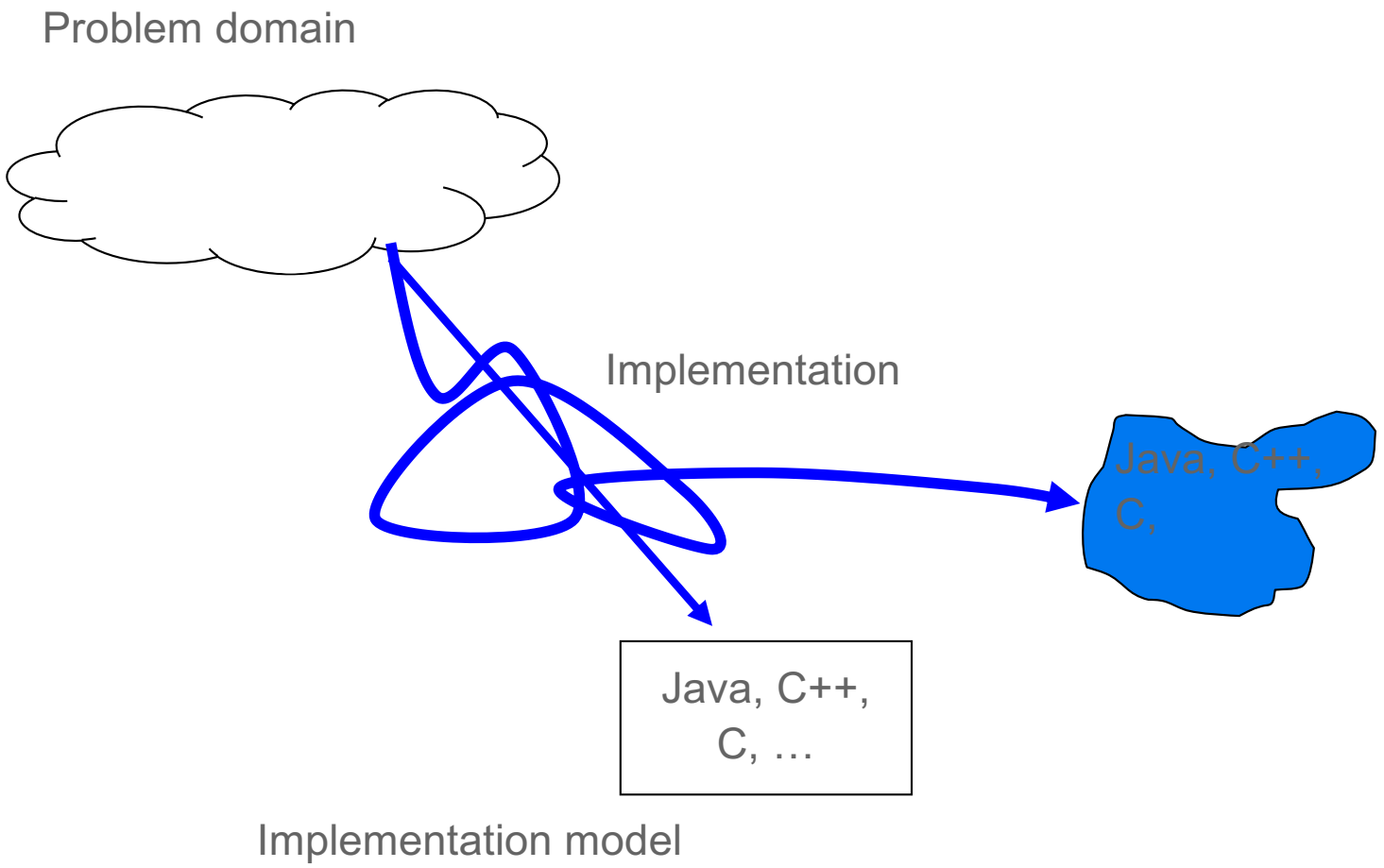
Motivation


How the Customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the Project was documented
What Operations installed
How the Customer was billed
How it was supported
What the Customer really needed

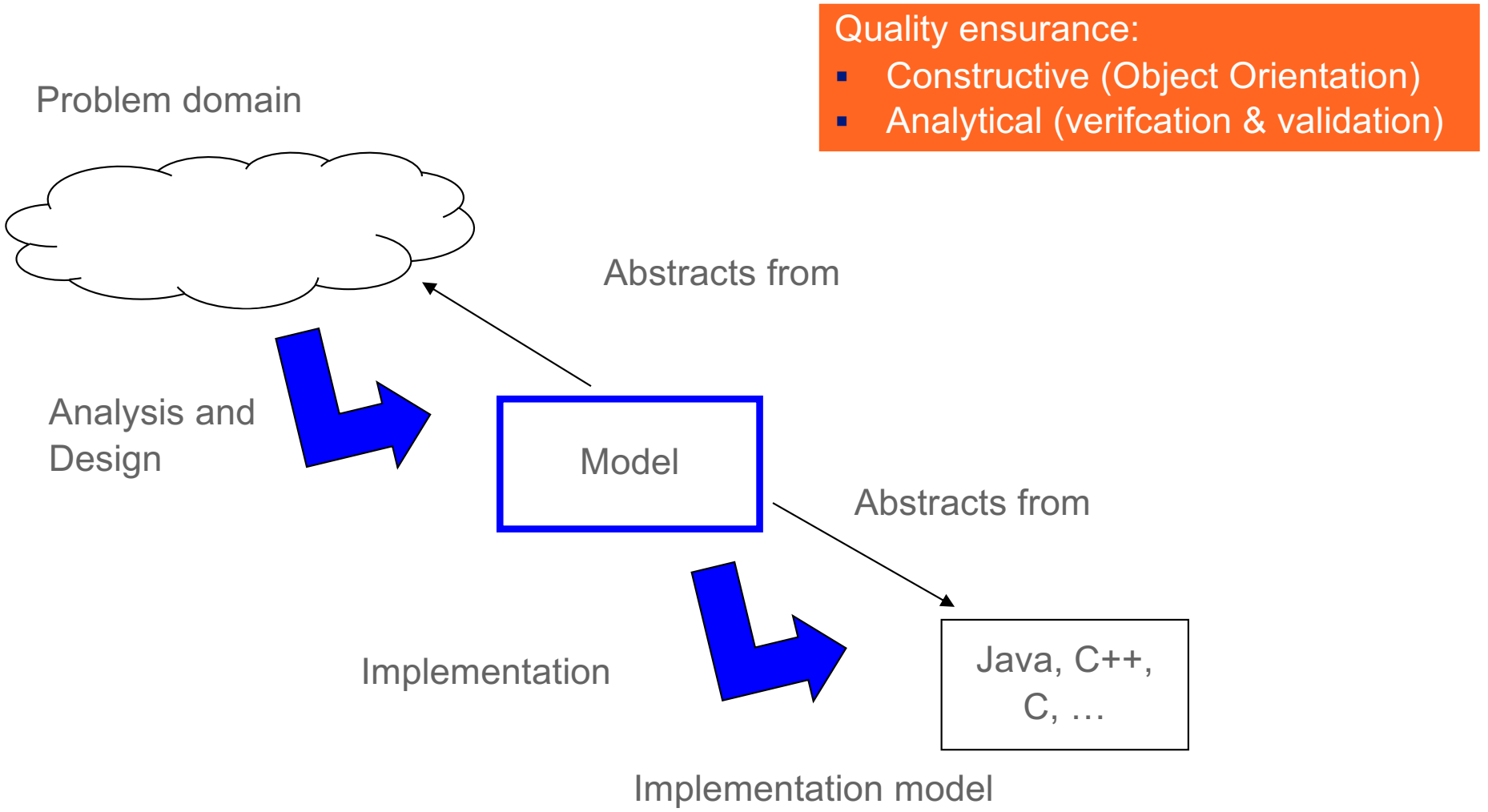
Software Development: ideal



Software Development: real



Model-based Design



What is a System

- „any organized assembly of resources and procedures united and regulated by interaction or interdependence to accomplish a set of specific functions“ US Department of Defense Architecture Framework (DoDAF)
- A system is an interconnected set of elements that is coherently organized in a way that achieves something. [Meadows2008]
- A construct or collection of different entities that together produce results not obtainable by the entities alone. International Council on Systems Engineering (INCOSE)

[LongScott2011]

Definition of the International Council on Systems Engineering (INCOSE)

- Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem:
  - Operations
  - Cost and Schedule
  - Performance
  - Training & Support
  - Test
  - Disposal
  - Manufacturing
- Systems Engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation. Systems Engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs.

Software Engineering

HOCHSCHULE HAMM-LIPPSTADT

- *The Establishment and use of sound **engineering principles** in order to obtain **economically** software that is **reliable** and works **efficiently** on real machines.*

  F.L. Bauer *in* [Buxton& Randell1969]

- software engineering. (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

  [IEEE-Std-610.12-1990]

- Looking Beyond Software?
  - For computer scientists, being told that creating software-intensive systems is difficult is not new. Even attendees at the 1968 NATO Software Engineering Conference in Garmisch, Germany, recognized the need to build, for example, a foundation for the systematic creation of software-intensive systems [3].

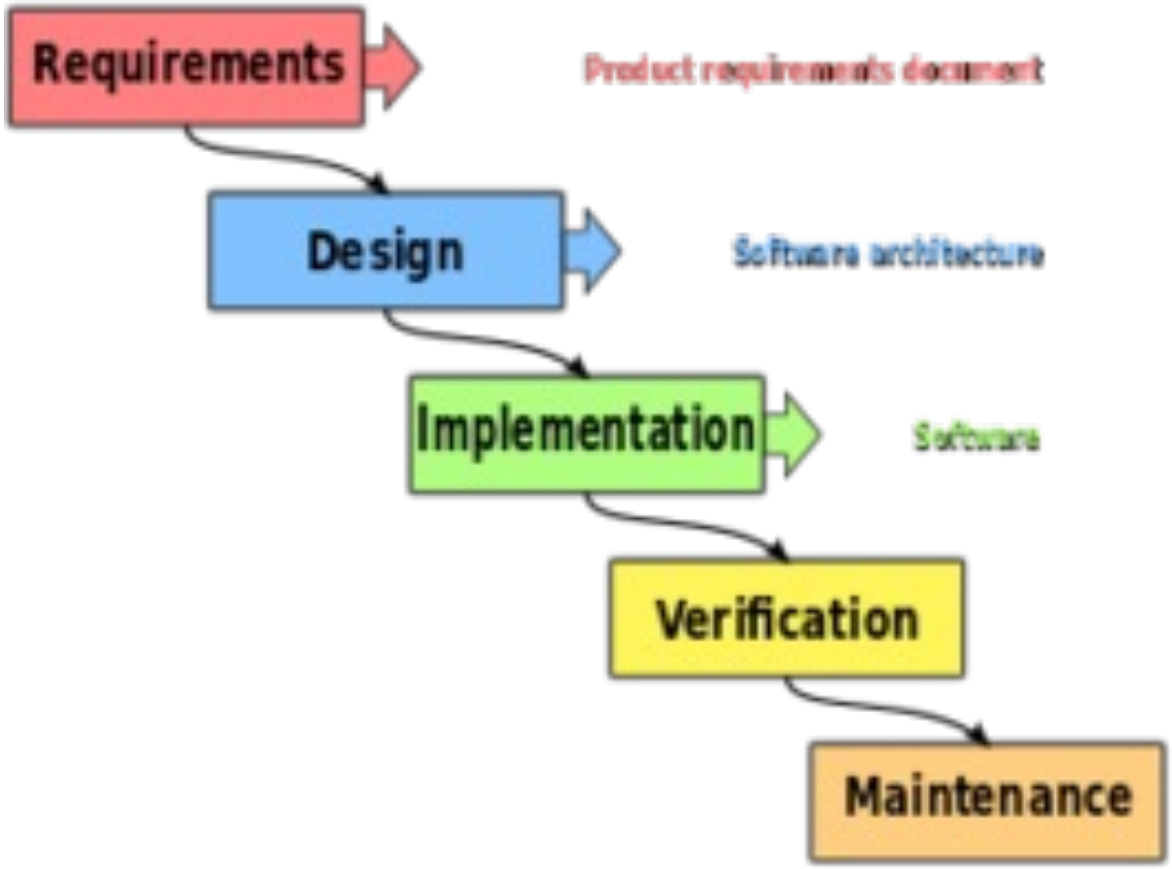
    (See [Freeman&Hart2004])

What is a Model?

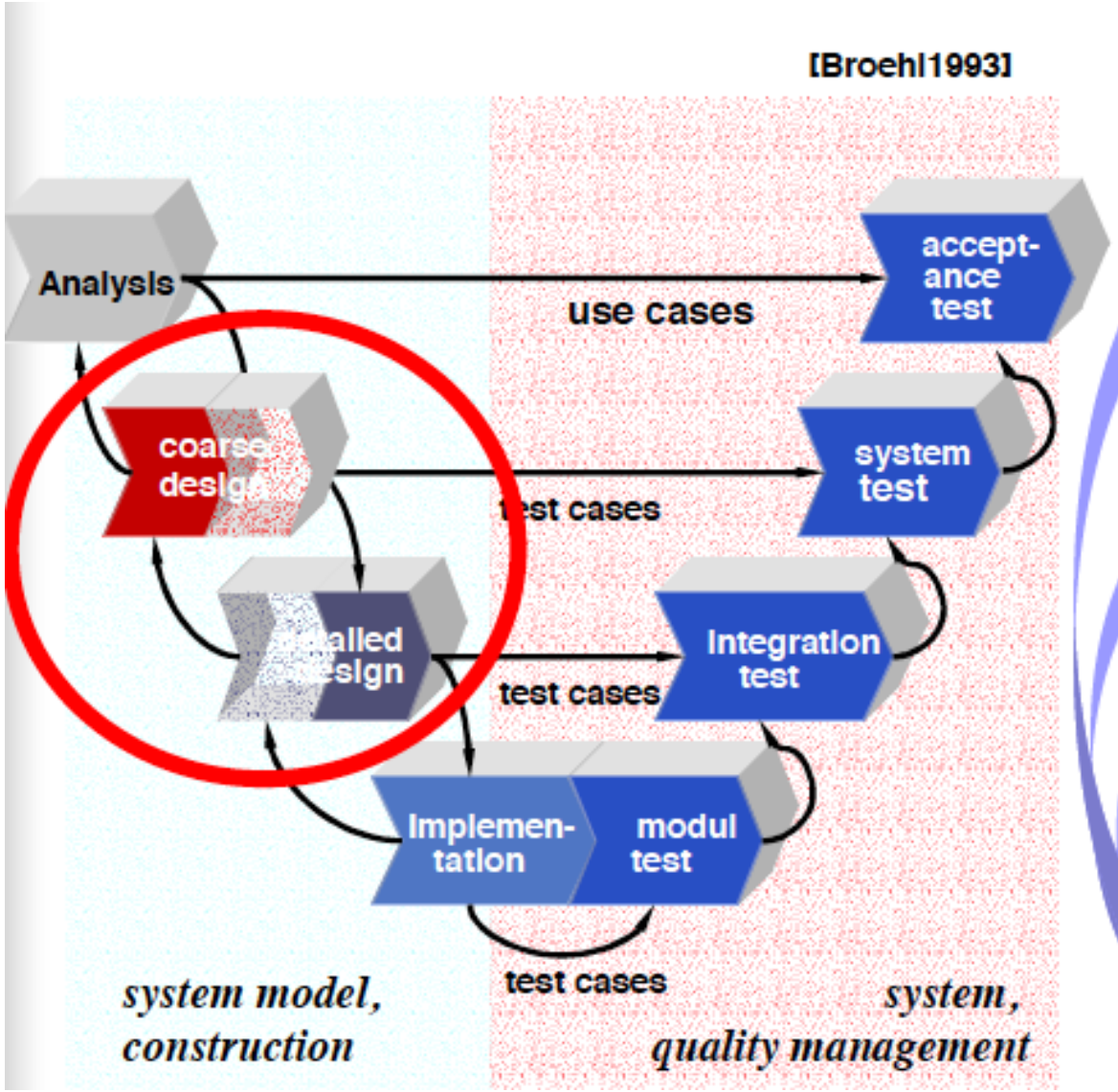
A Dictionary of Computing. Oxford University Press, 1996

Model: A simplified representation of something (the referent). The representation may be physical or abstract, and may be restricted to certain properties of the referent. In computing, models are usually abstract and are typically represented in a diagramming notation, such as dataflow diagrams (in functional design), ERA diagrams (for a data model), or state-transition diagrams (for a model of behavior)

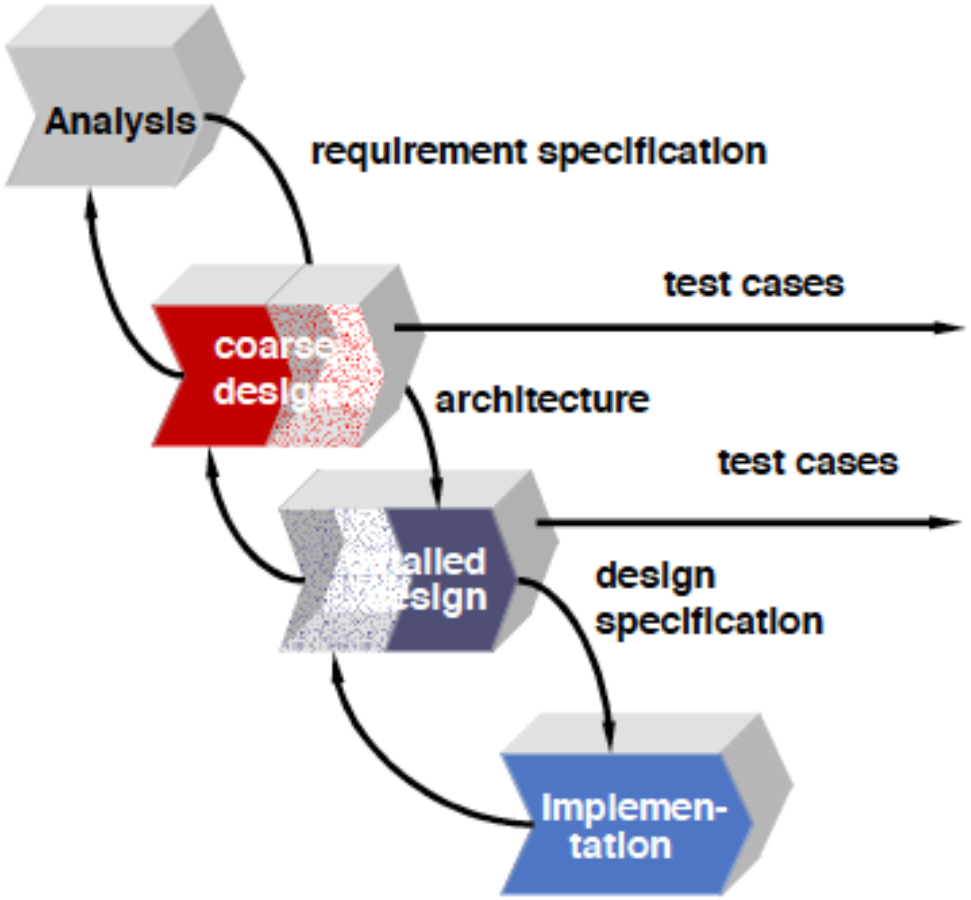
Design [IEEE-Std-610.12-1990][Taylor1959]:
(1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.
(2) The result of the process in (1).
The process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realization.
Design description [IEEE-Std-610.12-1990]:
A document that describes the design of a system or component. Typical contents include system or component architecture, control logic, data structures, input/output-formats, interface descriptions, and algorithms. .


Requirements
Product requirements document
Design
Software architecture
Implementation
Software
Verification
Maintenance

Role of Design (V-Model)



- System construction
  - Requirements
  - Design
- Quality management
  - Reviews, test, …
- Config. management
  - Manage artifacts
- Project-Management
  - Plan, control and monitor project



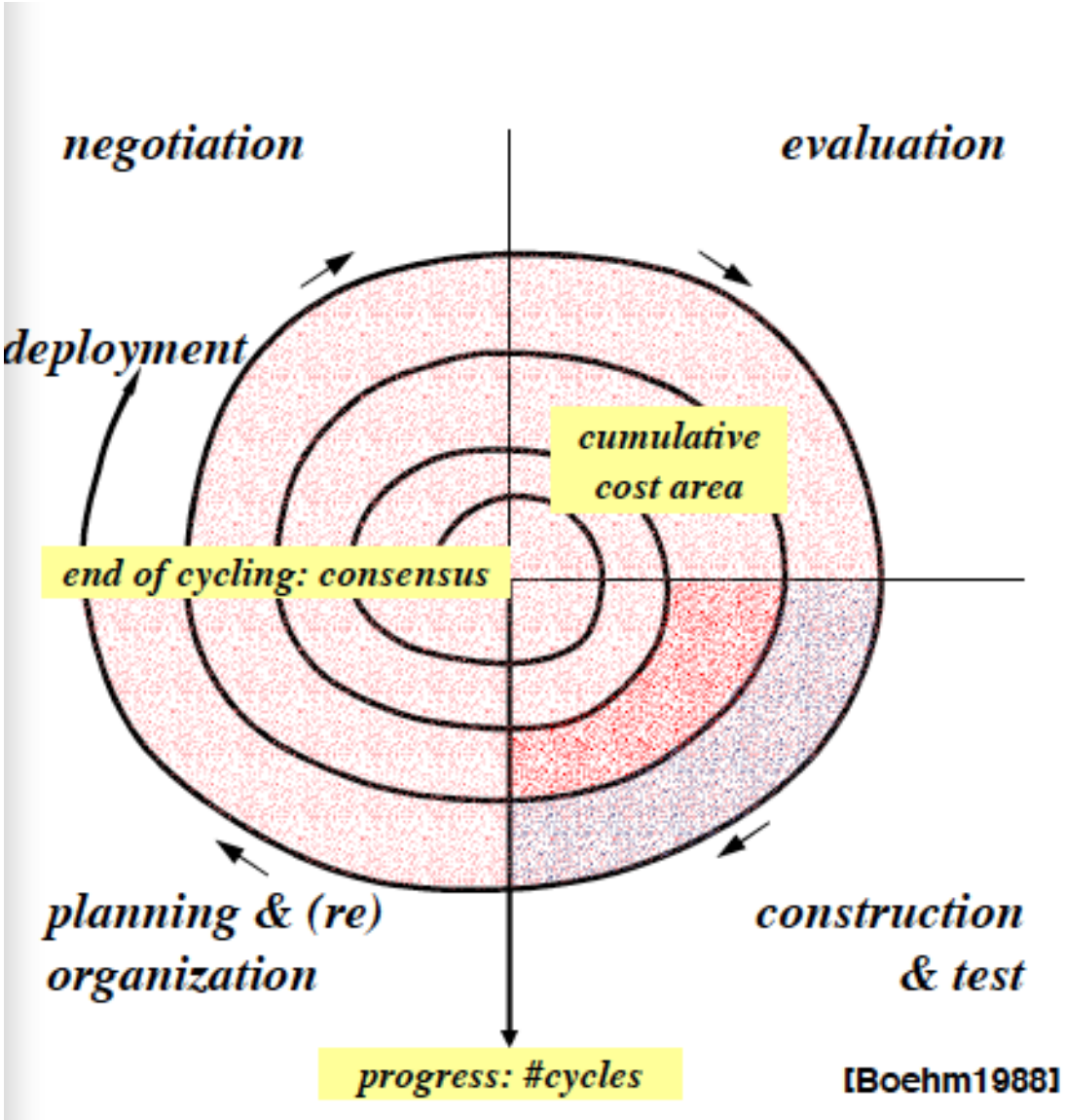
Coarse-grain Design
(Software Architecture)

- Decompose system into subsystems
- Identify components
- Relations and cooperation between components (connectors)
- Result: architectural description

Detailed Design
(Component Design)

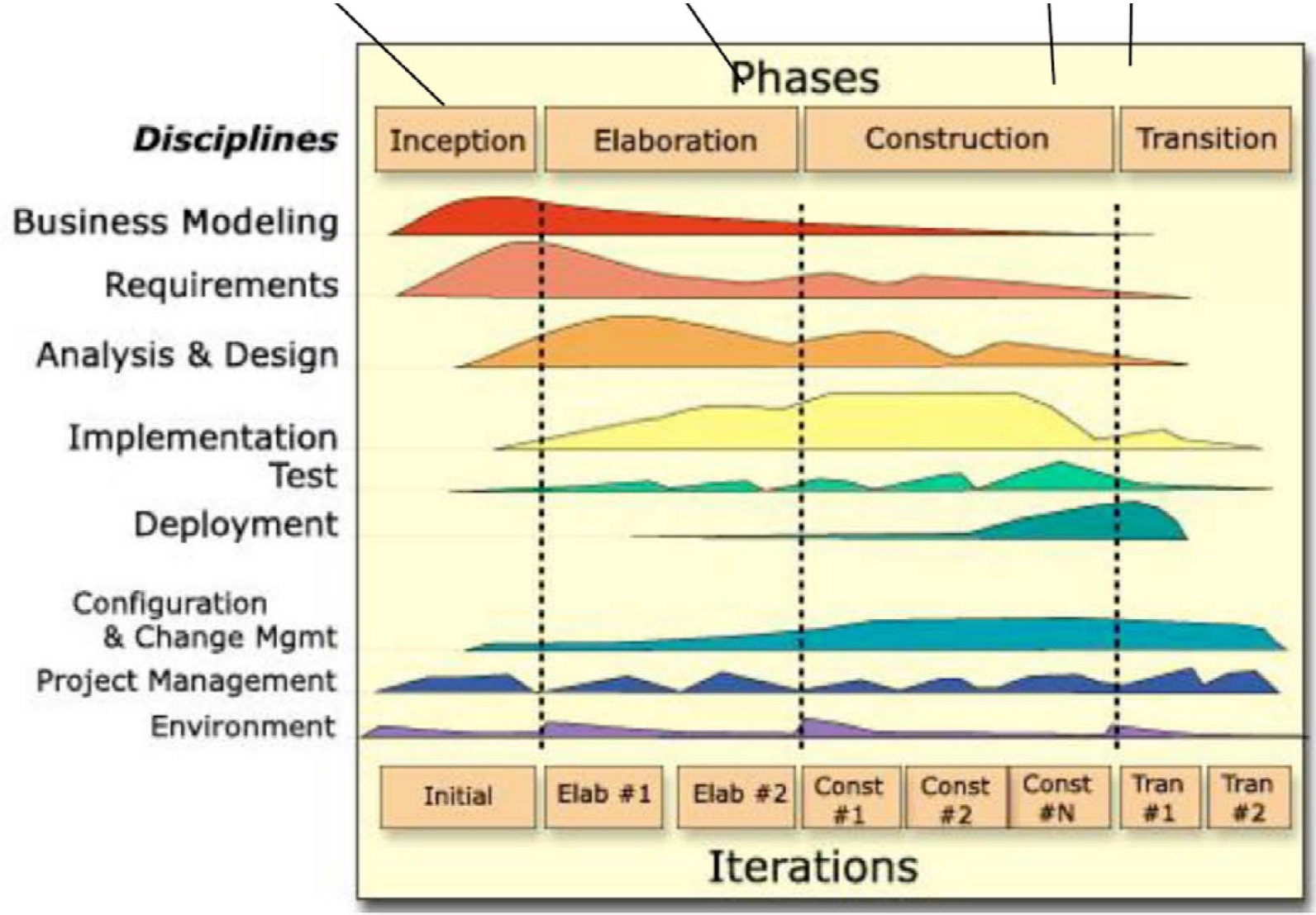
- Determine interfaces/contracts
- Result: component specification

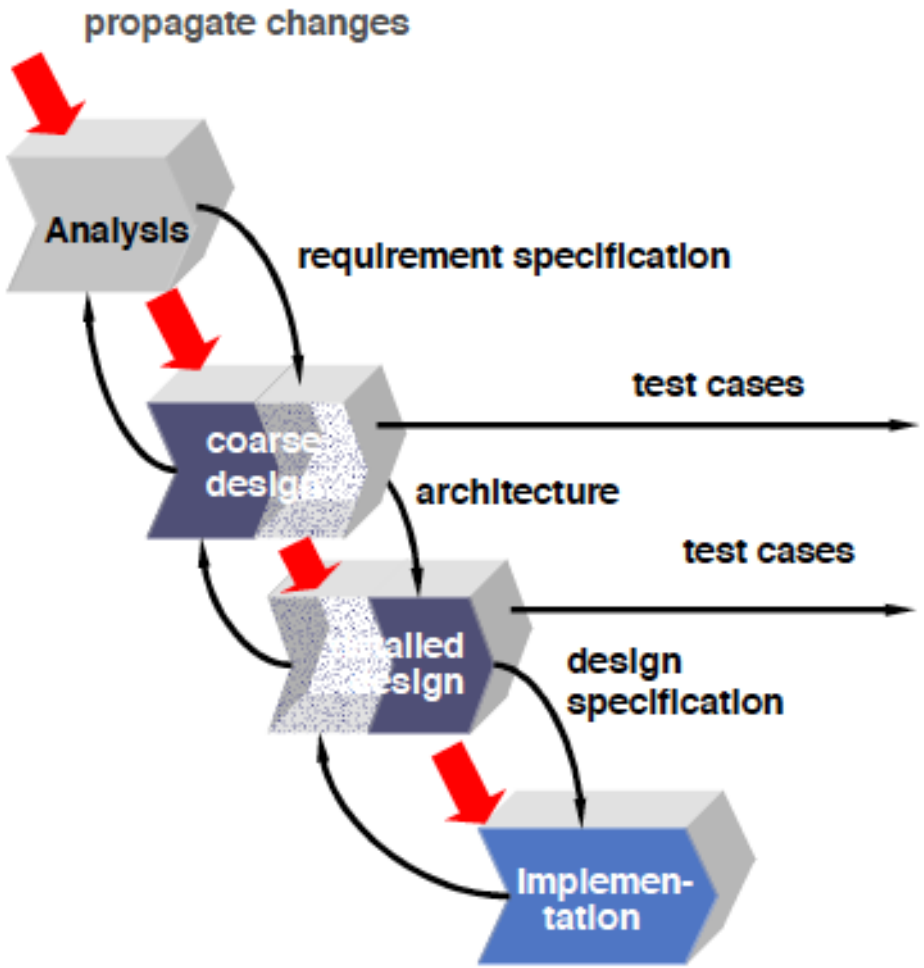
Role of Design
Spiral Model



- **Negotiation**
  - objectives, alternatives, strategies, constraints
- **Evaluation**
  - alternatives: „Make-or-Buy“, risk analysis
- **Construction & Test**
  - any SE-Process for partial or full system!
- **Planning**
  - Review,
  - Plan next phases

RUP


Phases
Disciplines
Inception
Elaboration
Construction
Transition
Business Modeling
Requirements
Analysis & Design
Implementation
Test
Deployment
Configuration
& Change Mgmt
Project Management
Environment
Initial
Elab #1
Elab #2
Const #1
Const #2
Const #N
Tran #1
Tran #2
Iterations



Iterative Process

- Realize core functionality first
- Ensure that required extensions are possible

➡ Software Architecture & detailed Design has to be flexible and extensible

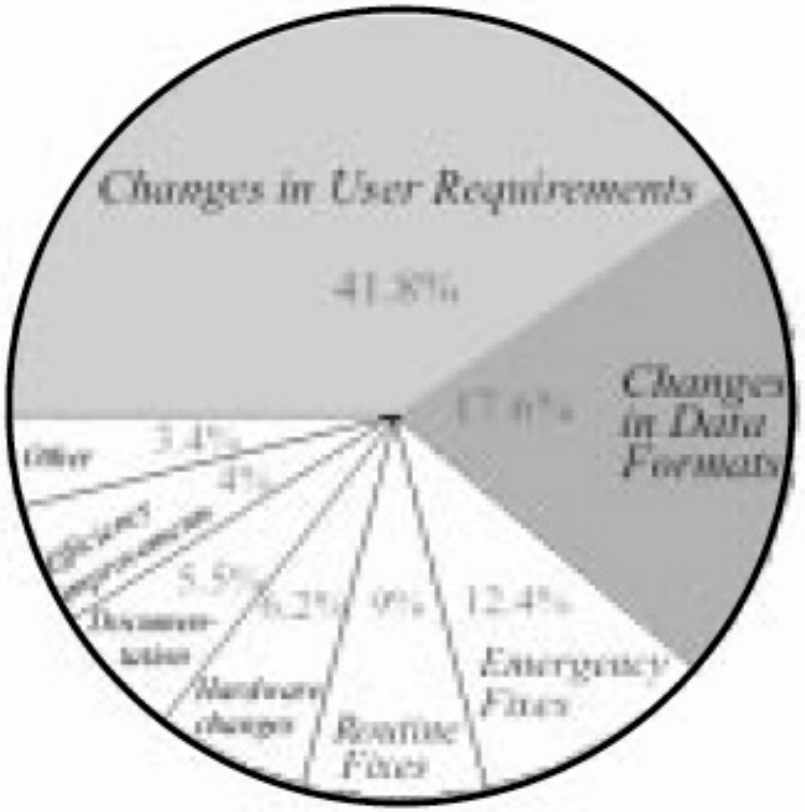
First Law of Software Evolution

A program that is used and that as an implementation of its specification reflects some reality, undergoes continual change or becomes progressively less useful.

First Law of System Engineering

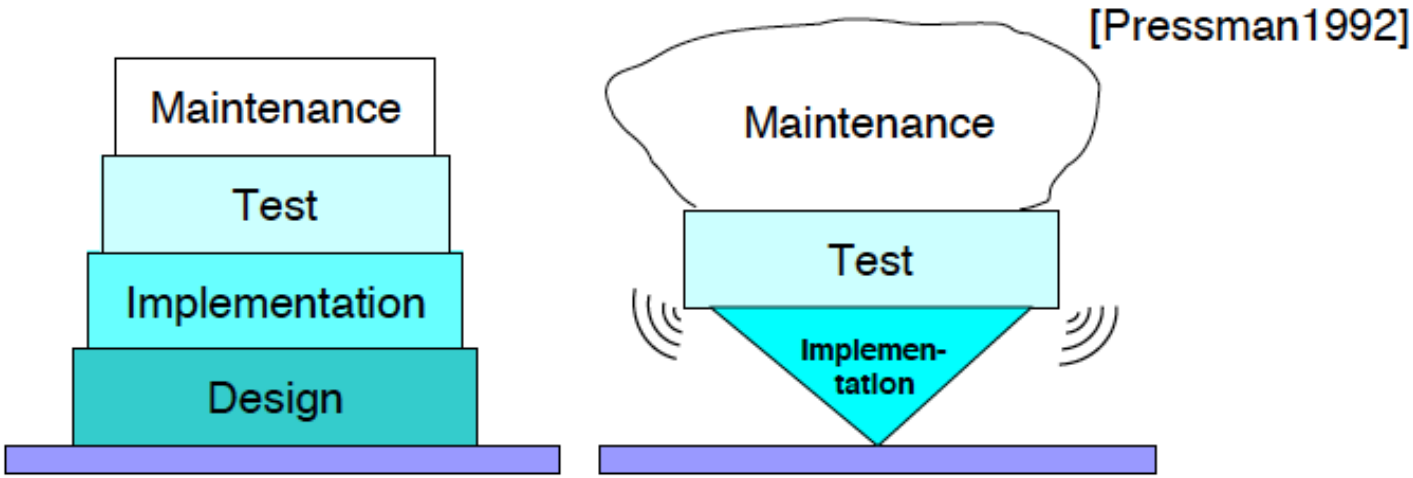
No matter where you are in the system life cycle, the system will change and the desire to change it will persist throughout the life cycle.

→ Software will be changed during maintenance



Breakdown of maintenance costs.
([Meyer1997] source [Lientz1980])

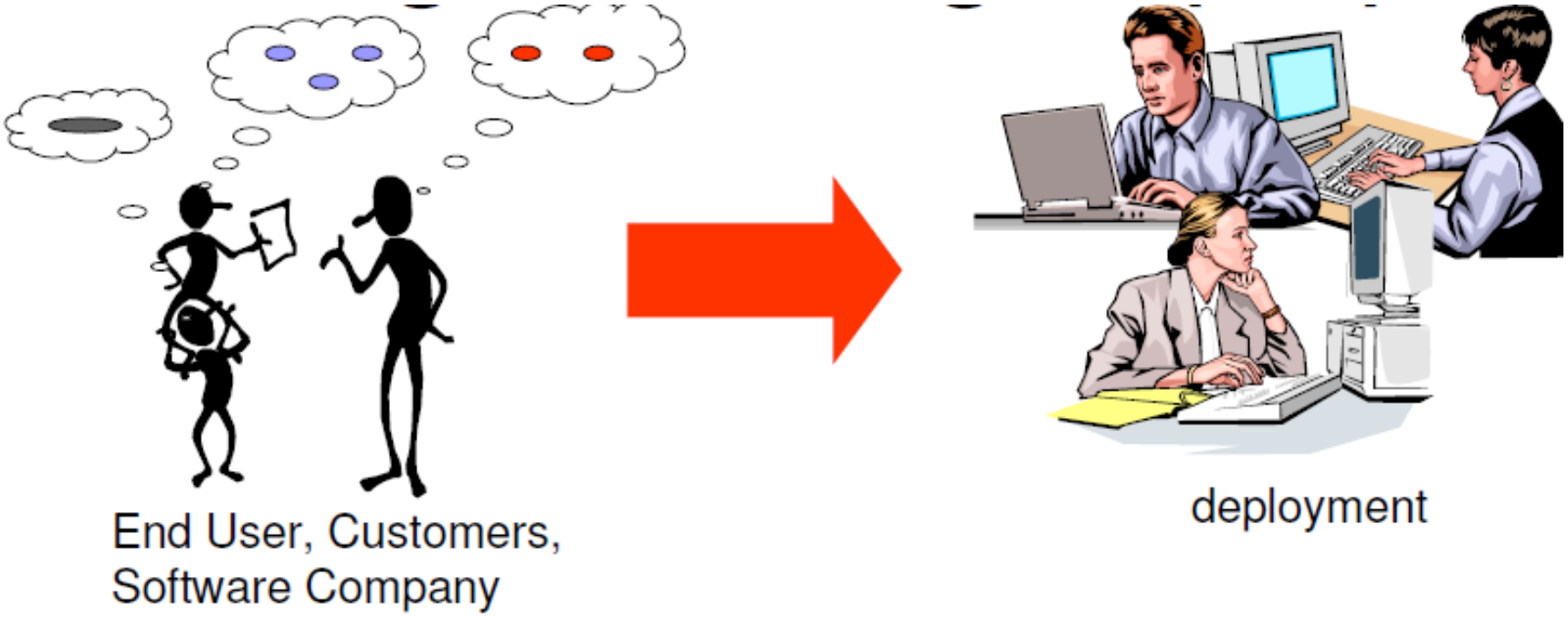
Software Quality Requires Design



Importance of software design: **quality**

- Translate accurately requirements into product
- Still stable systems for small changes
- Otherwise difficult to test

What is a good Design? (1/3)



- What qualities should the product provide
  - Properties required by the customer/user
  - Qualities required to ensure the further maintainability of the product

What is a good Design? (2/3)

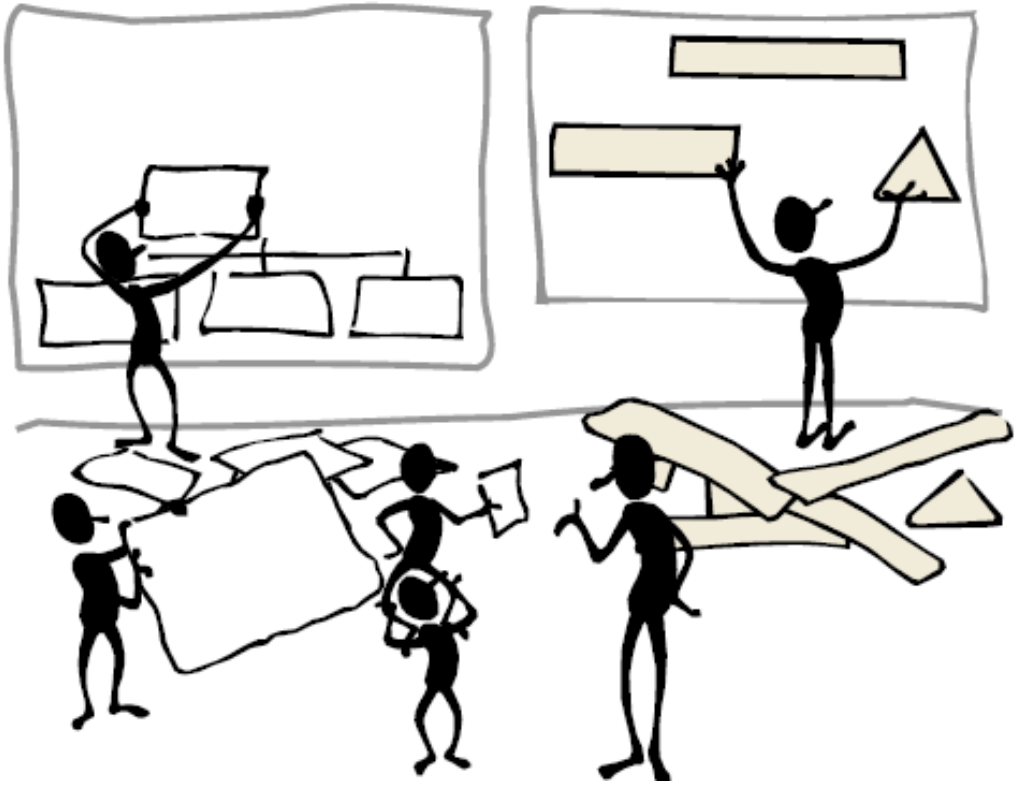
- **Software Quality: External User View**
  - Functionality
    - Correctness
    - Ease of use
  - Non-Functional
    - Reliability
    - Robustness
    - Performance
    - Efficiency
- **Deployment/Administration View**
  - Interoperability (compatibility)

[Gezzi+1991][Meyer1997]

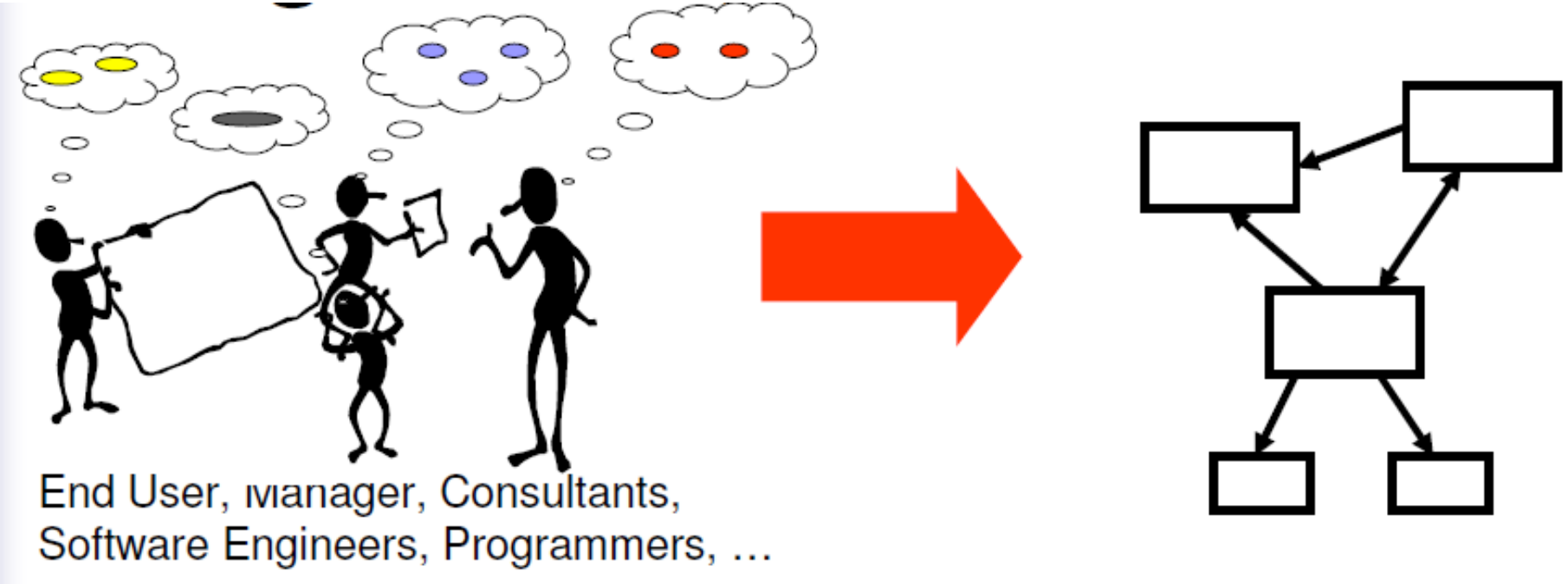
What is a good Design? (3/3)

- **Software Quality: Internal Development**
  - **Understandability**
  - Verifiability
- Maintenance
  - Repairability
  - **Evolvability**
  - **Extendibility**
  - Portability
- Impact on other projects
  - **Reusability**

[Gezzi+1991][Meyer1997]



Design Principles



- How to achieve required qualities?
  - Required functionality and non functional qualities
  - Internal qualities that guarantee an extendable and maintainable product

Design Principle: Separation of Concerns

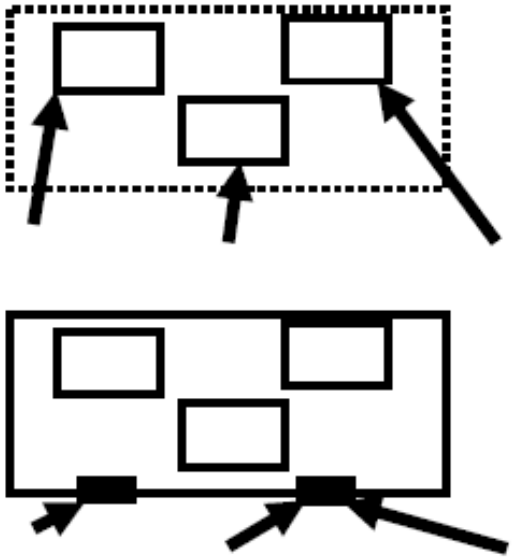
HOCHSCHULE HAMM-LIPPSTADT

- Separate different concern [Parnas1972b][Dijkstra1976]
- Group tasks that belong to the same concern
- Use elements responsible for each concern
  - reduced complexity for each concern
  - changes effect only the element realizing the concern

- Limitation: only one dimension for separation
- (see advanced separation of concerns [Tarr+1999])

Design Principle: Information Hiding

- Hide design decisions within modules especially when changes are expected.
- Access to internal modules is only provided via well defined interfaces which reveal only that information which is required to use the module.



- Examples:
  - Reveal only access points to data rather than the internally used data structures (container).
  - Specify only **that** elements are sorted not **which** sorting algorithm is used.

[Parnas1972b][Parnas1972b]

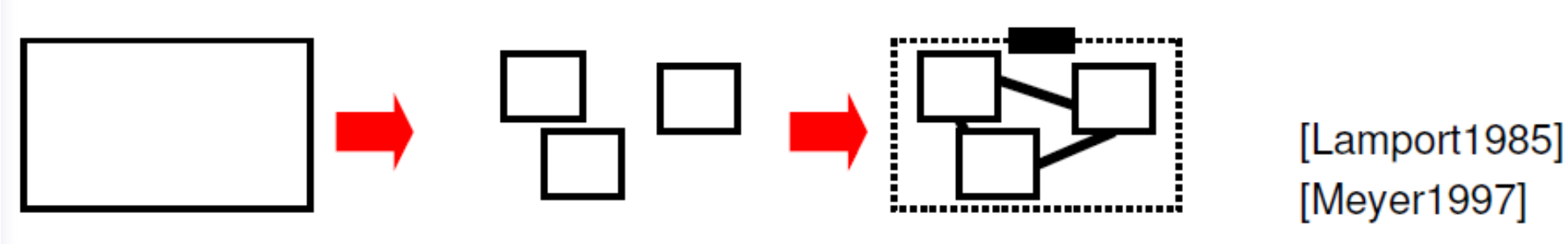
Design Principle: Abstraction

- Abstraction [IEEE-Std-610.12-1990]:
  - A view of an object that *focuses* on the information relevant to a particular purpose and ignores the remainder of the information.

- Examples:
  - Encapsulation
    - restricting direct access to elements (information hiding) – this is definition 1) of encapsulation
    - Definition 2) is given on next slide!
  - Generalisation (e.g., abstract super classes)
  - virtual machines (e.g., Java)

Design Principle: Modularity (1/2)

- Modularization [IEEE-Std-610.12-1990]:
  - The process of **breaking** a system **into components** to facilitate design and development.
  - Decomposability, Composability, Continuity, Protection, …
- Encapsulation [IEEE-Std-610.12-1990]:
  - A software development technique that consists of **isolating** a system **function** or a set of data and operations on those data within a module **and** providing **precise specifications** for the module.

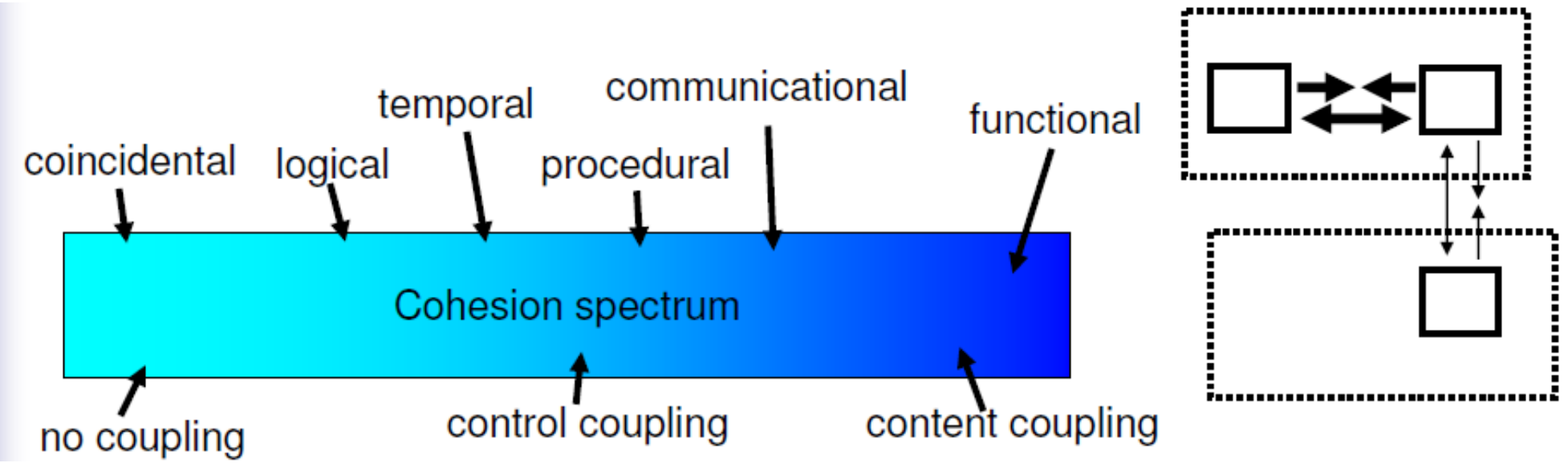


[Lamport1985]
[Meyer1997]

Design Principle: Modularity (2/2)

Cohesion and Coupling

- **Coupling:** measure for strength of connection (results in dependencies) [Pressman 1992]



Cohesion: measure for strength of relation

Design Principle: Others

- **Anticipation of Change:**
  - **Anticipate** future system **evolution** and provide a solution that is **flexible** w.r.t. these **modifications/extensions**
- **Generality:**
  - Look for the underlying more **general problem** that may be **simpler** and has more potential for **reuse**
- **Incrementallity:**
  - The **stepwise approximation** of a solution permits to integrate early customer feedback and reduce (technology) risks

HOCHSCHULE HAMM-LIPPSTADT

Historic Trends:

- Structured programming
  - Dijkstra 1968: Goto statement considered harmful
  - Keywords: top-down, functional decomposition, stepwise refinement, divide-and-conquer, …
  - SA/SD: late 1970s
- Object-oriented programming
  - Simular 67, smalltalk, C++, Object C, Eiffel, Ada, Lisp
  - Liskov, Guttag, Shaw 1970s: Abstract data type
  - Parnas 1972: Information hiding
  - Keywords: objects, classes, reusable components
  - OOA/OOD: late 1980s

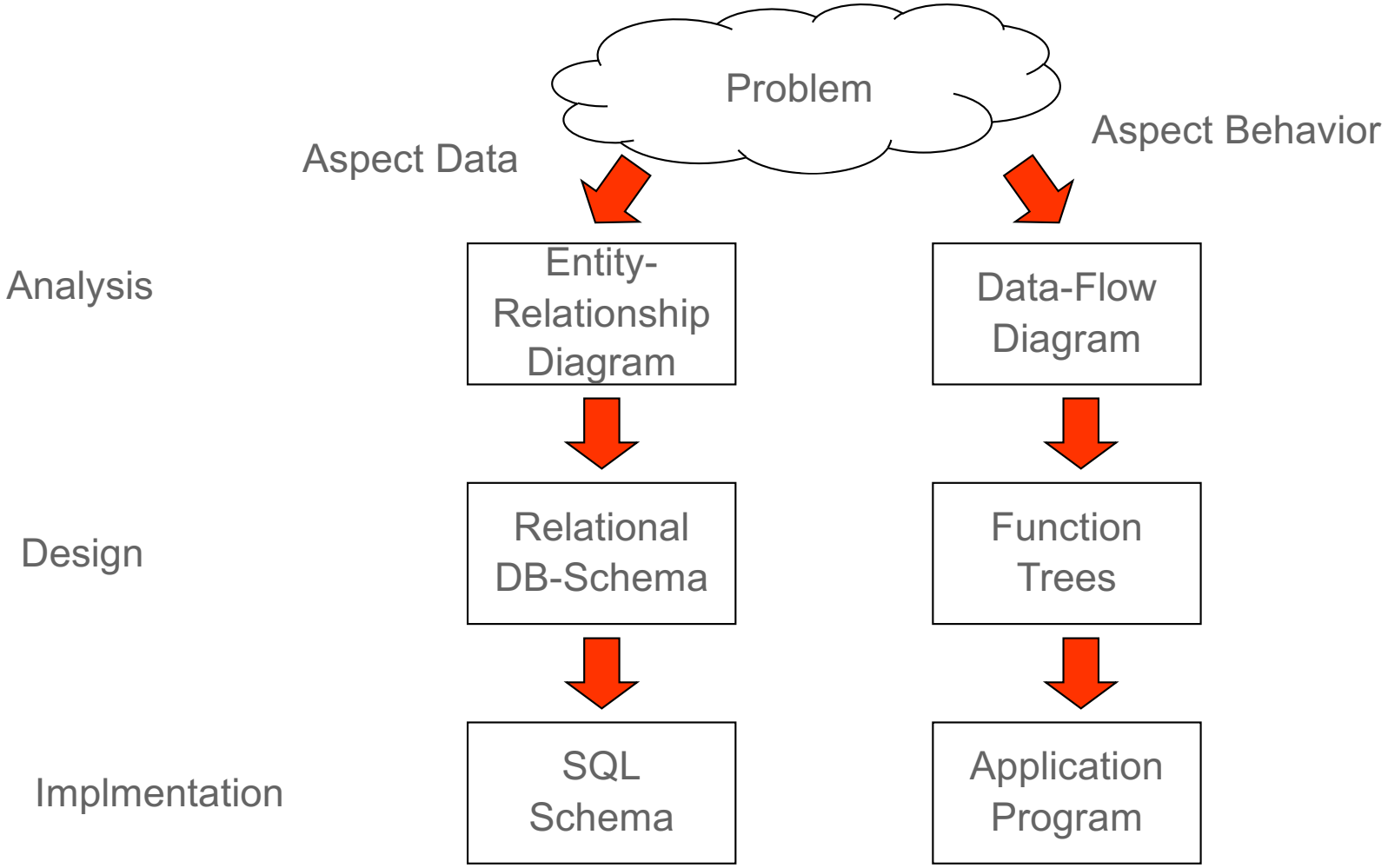
Object-orientation vs. Function-orientation

OO

- Relatively young paradigm of software development.
- Previously approach: Function-orientation

OO	Function-oriented
software: *dynamic* collection of communicating objects	software: *static* collection of procedures
objects have their own state	shared state
data-oriented architecture	action-oriented architecture
communication via message passing between objects	communication via procedure calls
actions are only possible if provided by some object	

Structured Analaysis and Design

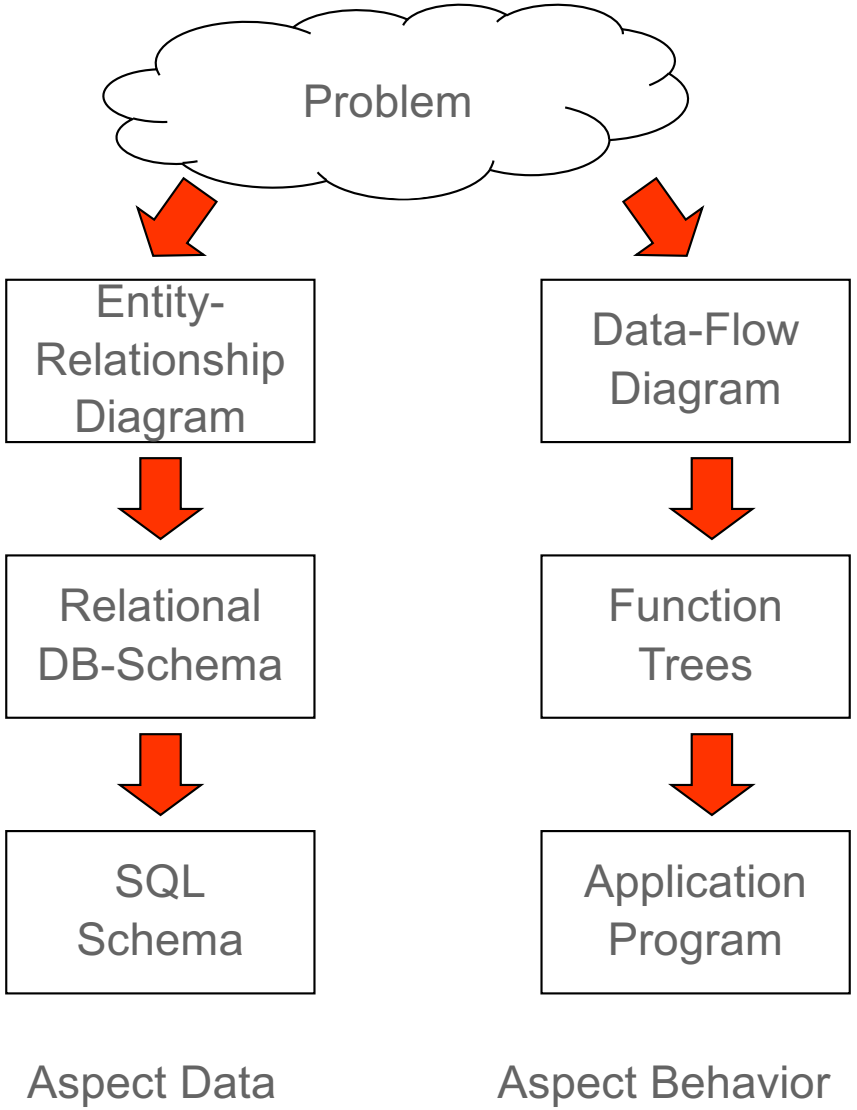


Structured Analaysis and Design: Problems

- For each phase a language
  - Well defined syntax and semantics
- Well-defined transition between phases

But:

- Integration of **heterogeneous** concepts and views necessary
- Problem: Changes on one level results in sophisticated changes on the next level
  - Especially as data (structure) aspect is loosely coupled with behavior



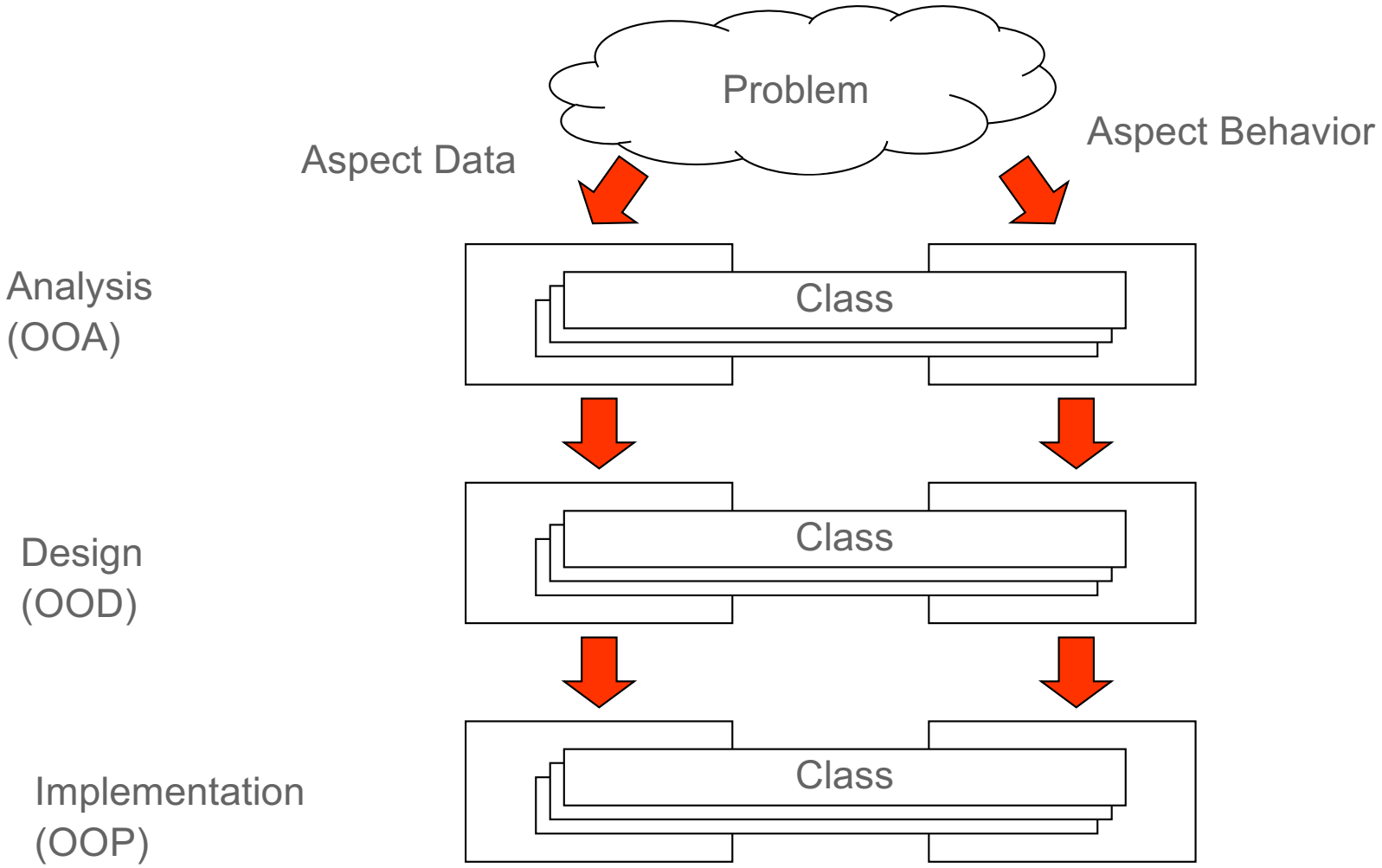
Objects combine behavior and data and therefore are more stable entities w.r.t. Changes.

Classes are used as templates for objects (instances). They abstract from uniform objects.

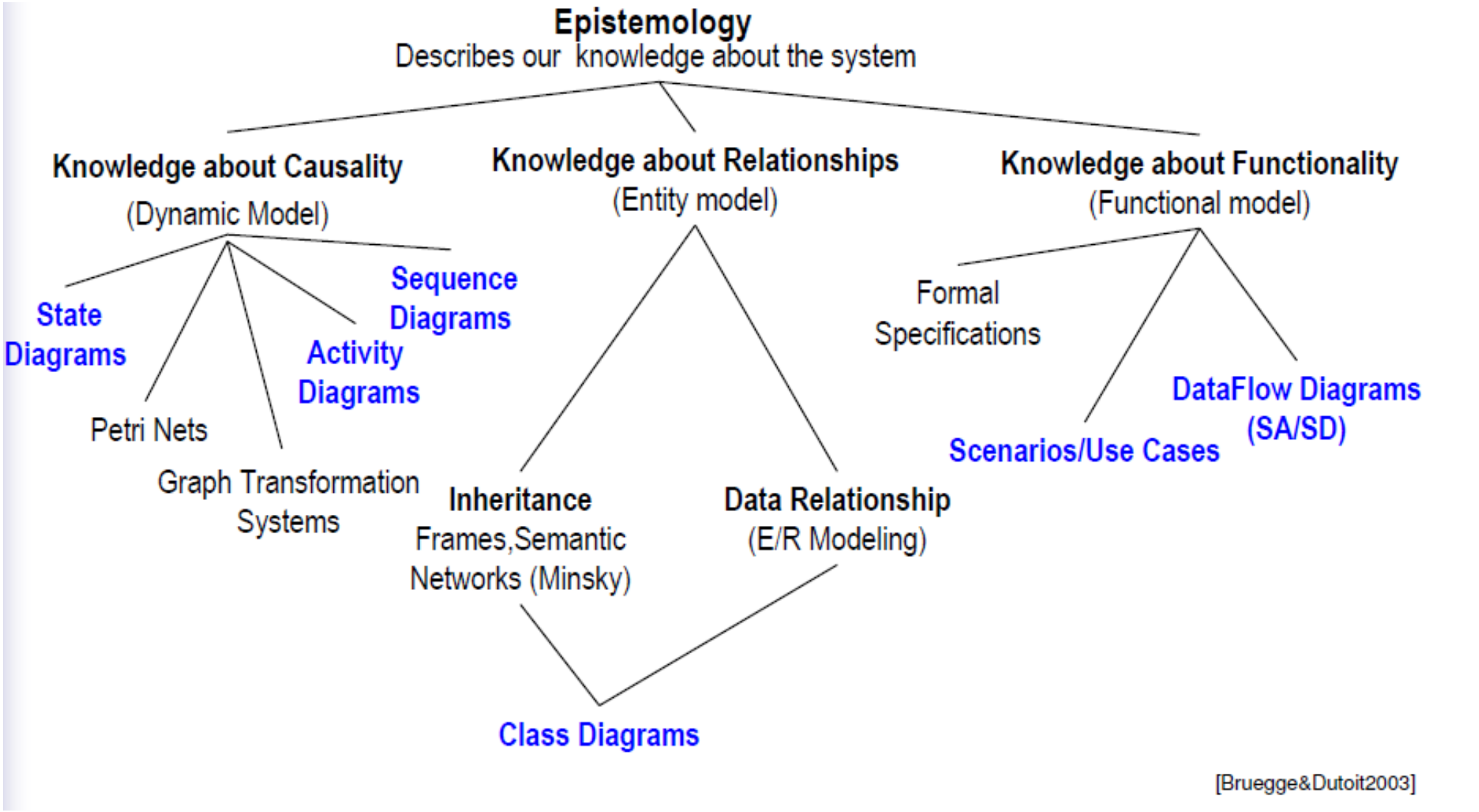
What can be an object?

- External Entities
  - that interact with the system being modeled
  - E.g. people, devices, other systems
- Things
  - that are part of the domain being modeled
  - E.g. reports, displays, signals, etc.
- Occurrences or Events
  - that occur in the context of the system
  - E.g. transfer of resources, a control action, etc.
- Roles
  - played by people who interact with the system
- Organizational Units
  - that are relevant to the application
  - E.g. division, group, team, etc.
- Places
  - that establish the context of the problem being modeled
  - E.g. manufacturing floor, loading dock, etc.
- Structures
  - that define a class or assembly of objects
  - E.g. sensors, four-wheeled vehicles, computers, etc.
- Some things cannot be objects:
  - procedures (e.g. print, invert, etc)
  - atomic attributes (e.g. blue, 50Mb, etc)

Object-oriented Analysis/Design



How to Describe Complex Systems?



[Bruegge&Dutoit2003]

Objects combine behavior and data and therefore are more stable entities w.r.t. changes.

Programming (OOP):

- Object = local state + access operations
- Class = template for objects
- Inheritance = derive one class form another
- Polymorphism = take multiple forms
- Dynamic/late binding = instance specific methods

History: Simula, Smalltalk, C++, Java, C#, …?

HOCHSCHULE HAMM-LIPPSTADT

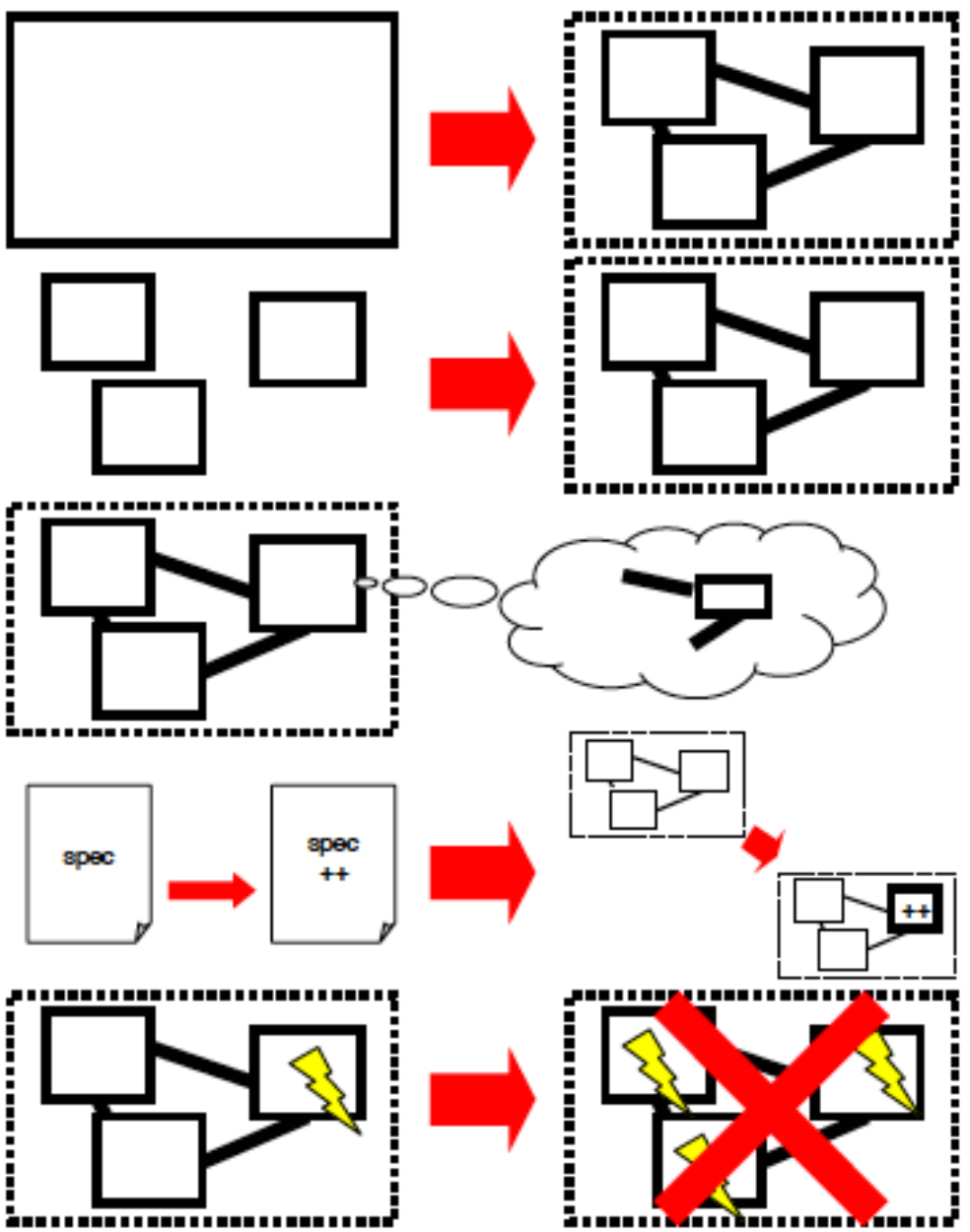
Analysis and Design (OOA/OOD):

- [Booch1993] [Jacobson+1992] [Coleman+1994]
  - Integrate entity relationship (ER) concepts data-driven, e.g., OMT [Rumbaugh+1991]
- responsibility-driven [Wirfs-Brock+1990]
- **History: Booch, OOSE, OMT … “methodology war” => UML 1.3 [UML1.3]**

- OO & Modularity: 5 Criteria, 5 Rules, 5 Principles
  - (cf. [Meyer1997])

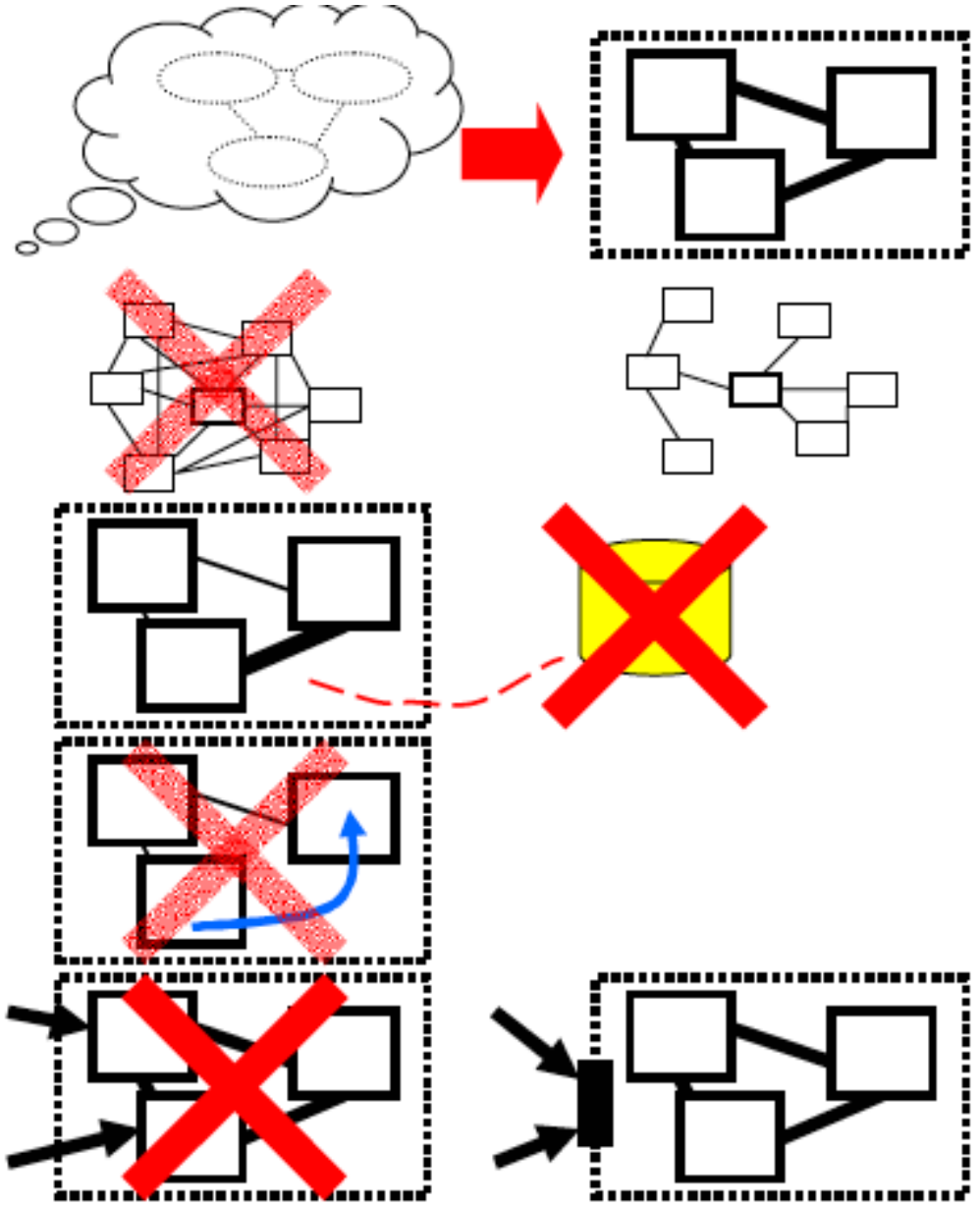
OO & Modularity: 5 Criteria

- Decomposability
  - split into small number of less complex modules
- Composability
  - Software elements that can be freely combined
- Understandability
  - Understand each module without having to know too many others
- Continuity
  - Small change in requirements effect only a small number of modules
- Protection
  - Runtime errors will only effect a few other components

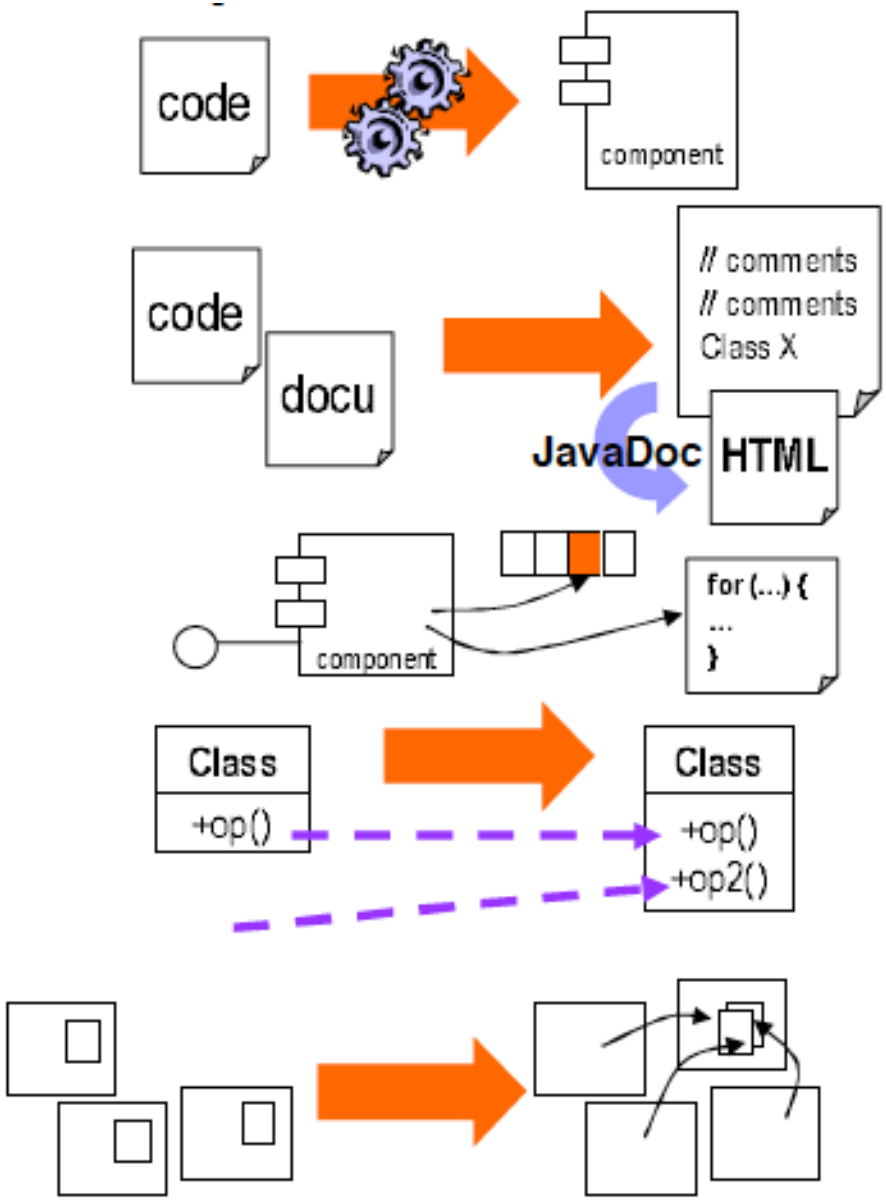


OO & Modularity: 5 Rules

- Direct Mapping
  - Problem domain and software structure should be compatible
- Few Interfaces
  - Minimize number of required interfaces
- Small Interfaces
  - Minimize exchanged information to ensure “weak coupling”
- Explicit Interfaces
  - No implicit communication between modules!
- Information Hiding
  - Reveal only the required subset of module properties that enable its usage by client modules



- Linguistic Modular Units
  - Modules should correspond to software units that can be compiled independently
- Self Documentation
  - Documentation is part of the module itself (not only code)
- Uniform Access
  - Interfaces should not betray whether how a specific functionality is realized (data or code)
- Open-Closed Principle
  - "Modules should be both **open** and **closed**." ([Meyer1997]: closed = interface does not change, open = extension possible)
- Single Choice
  - Variation points (e.g., number of alternatives) should be managed by only **one** module

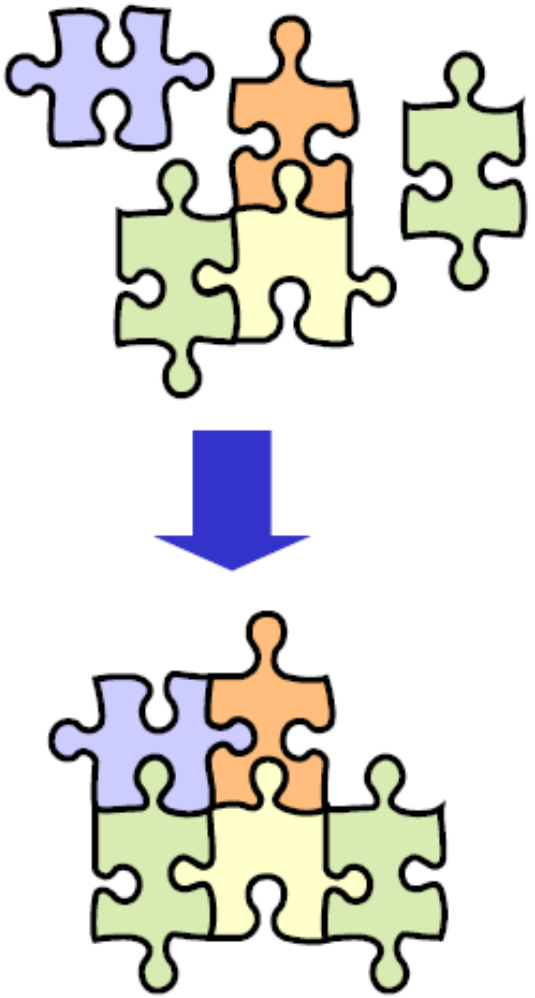


Object Structuring: Limitation

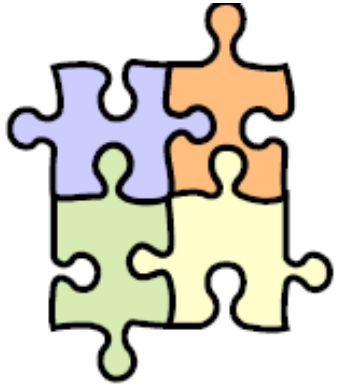
- Object abstraction results usually in **stable** entities w.r.t. requirement **changes**
- Class as unit for functional and data abstraction is often only suitable as fine-grain concept, because instead of modules only package concepts exists
  - Often focus on programming rather than design
  - Missing concepts for “programming in the large”
- Encapsulation is sometimes weak:
  - Inheritance breaks encapsulation
  - Implicit Interface specifications
- OO supports only white-box reuse

And Beyond …

- **Components:**
  - reuse of pre-defined and tested components
  - construct systems on a very high abstraction level.
  - decreases time-to-market and improves the productivity.

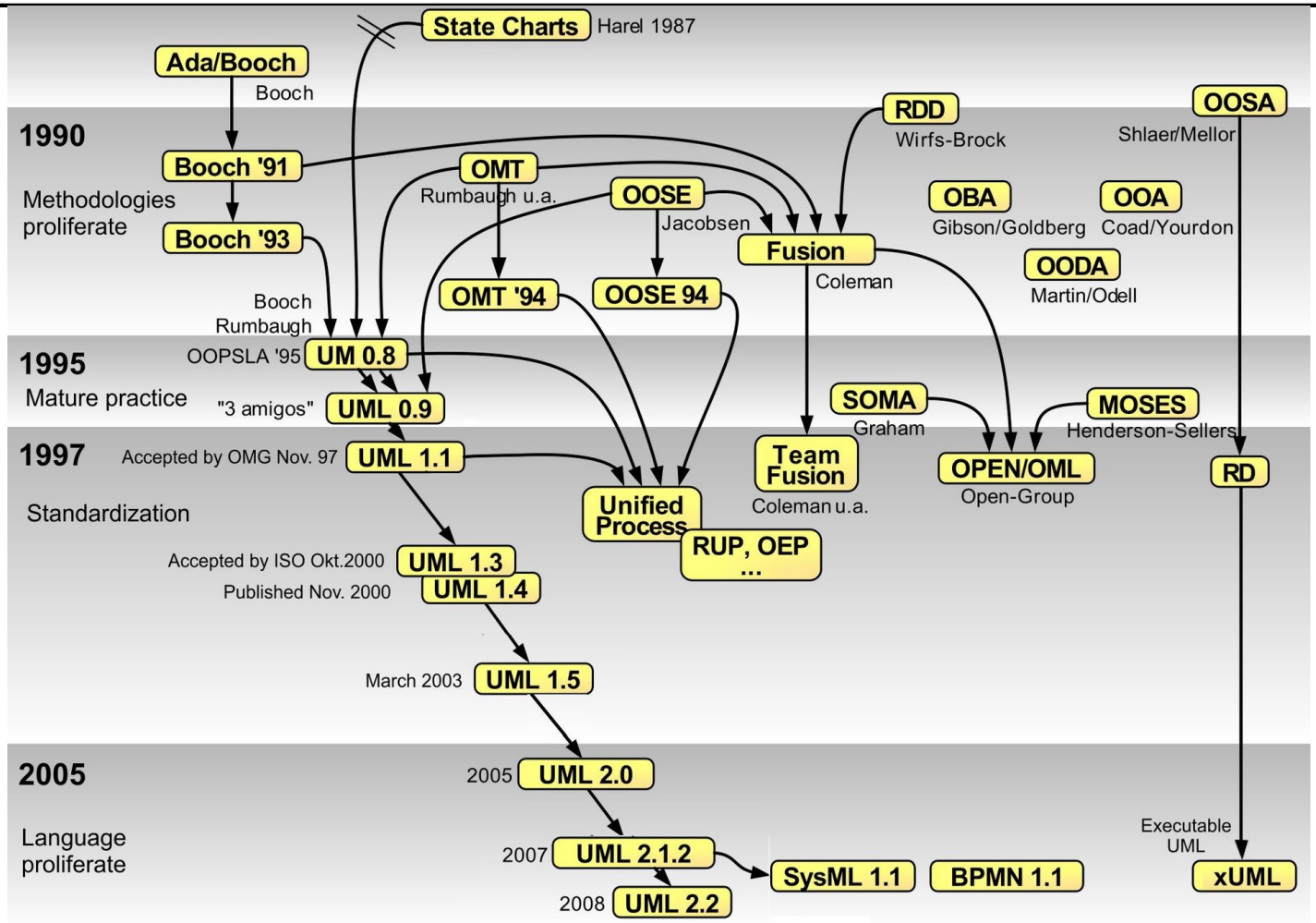


- **Szyperski defines a component as follows:** *“A Software component is an unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“ [Szyperski1998]*
- Only explicit contractual relations
  - Well-defined provided functionality
  - Only explicit specified context dependencies
- Independent deployment (beyond language)
  - Third party composition
  - Black-box reuse



Outline

The unified modeling language



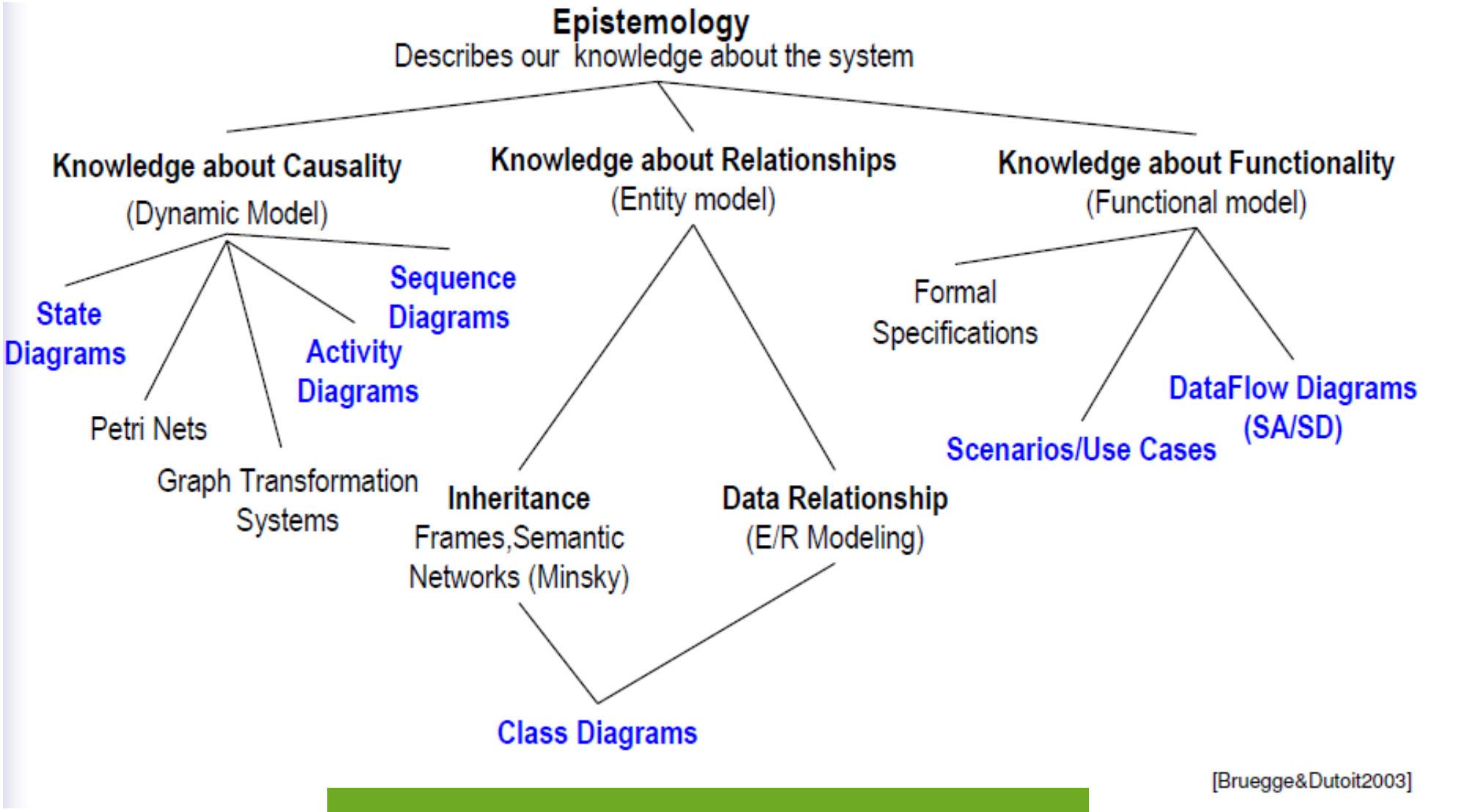
History: OMT + OOD/Booch + OOSE

[http://en.wikipedia.org/wiki/Unified_Modeling_Language#mediaviewer/File:OO_Modeling_languages_history.jpg]

Goal: industrial standard for object oriented software design

- Visual language
- Neutral w.r.t
  - Programming language
  - Problem domain
  - Methods and processes

Models and Diagrams (I/II)



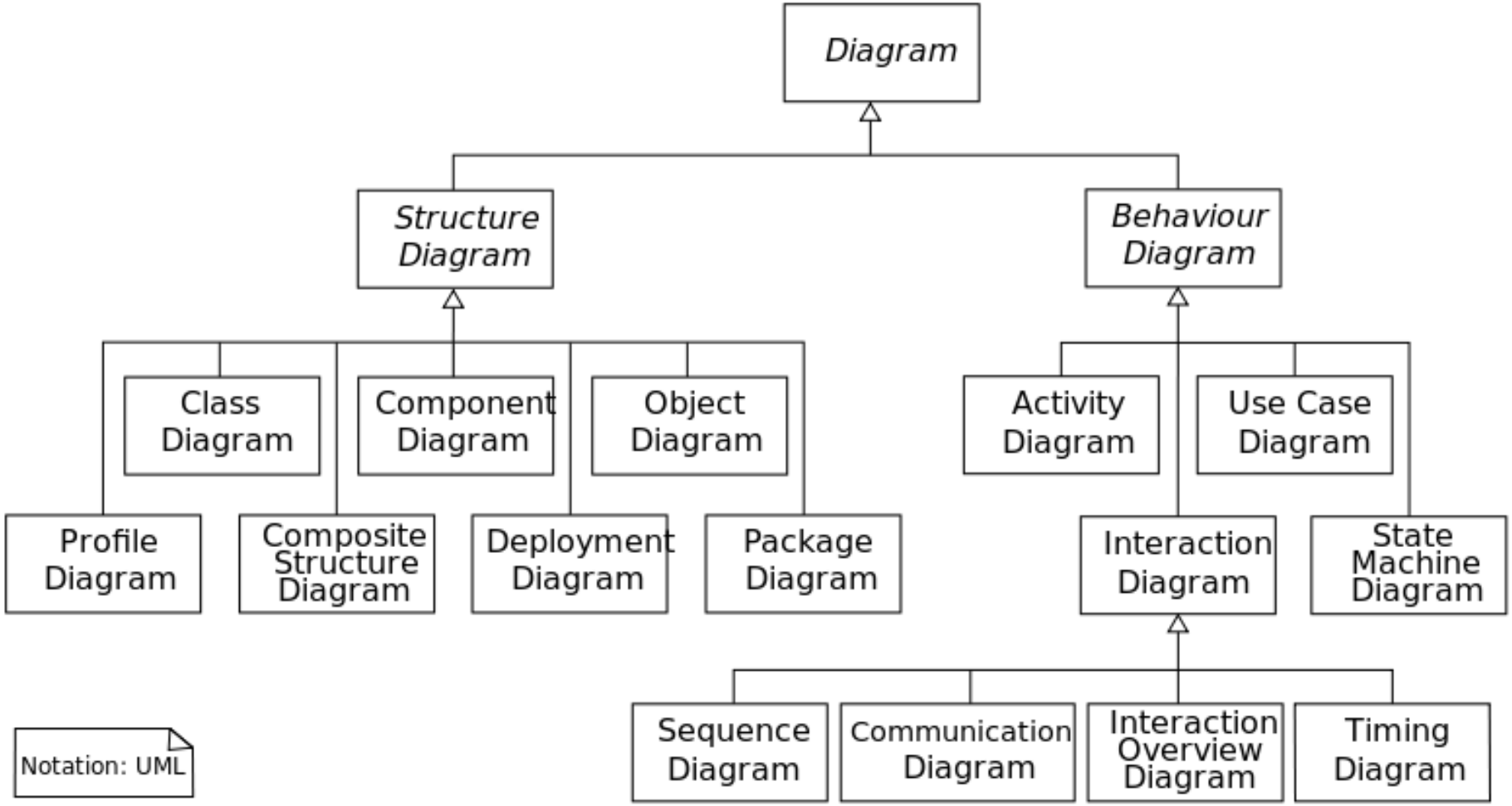
[Bruegge&Dutoit2003]

What is a model and what is a diagram?

Models and Diagrams (II/II)

- Models are abstract representations of a system
- Diagrams
  - Correspondents to one view of a model
- All the different views combined result in a model of a system
- UML diagrams illustrate the quantifiable aspects of a system that can be described visually, such as relationships, behavior, structure, and functionality
- A typical UML model can consist of many different types of diagrams, with each diagram presenting a different view of the system that you are modeling

UML Diagram Overview

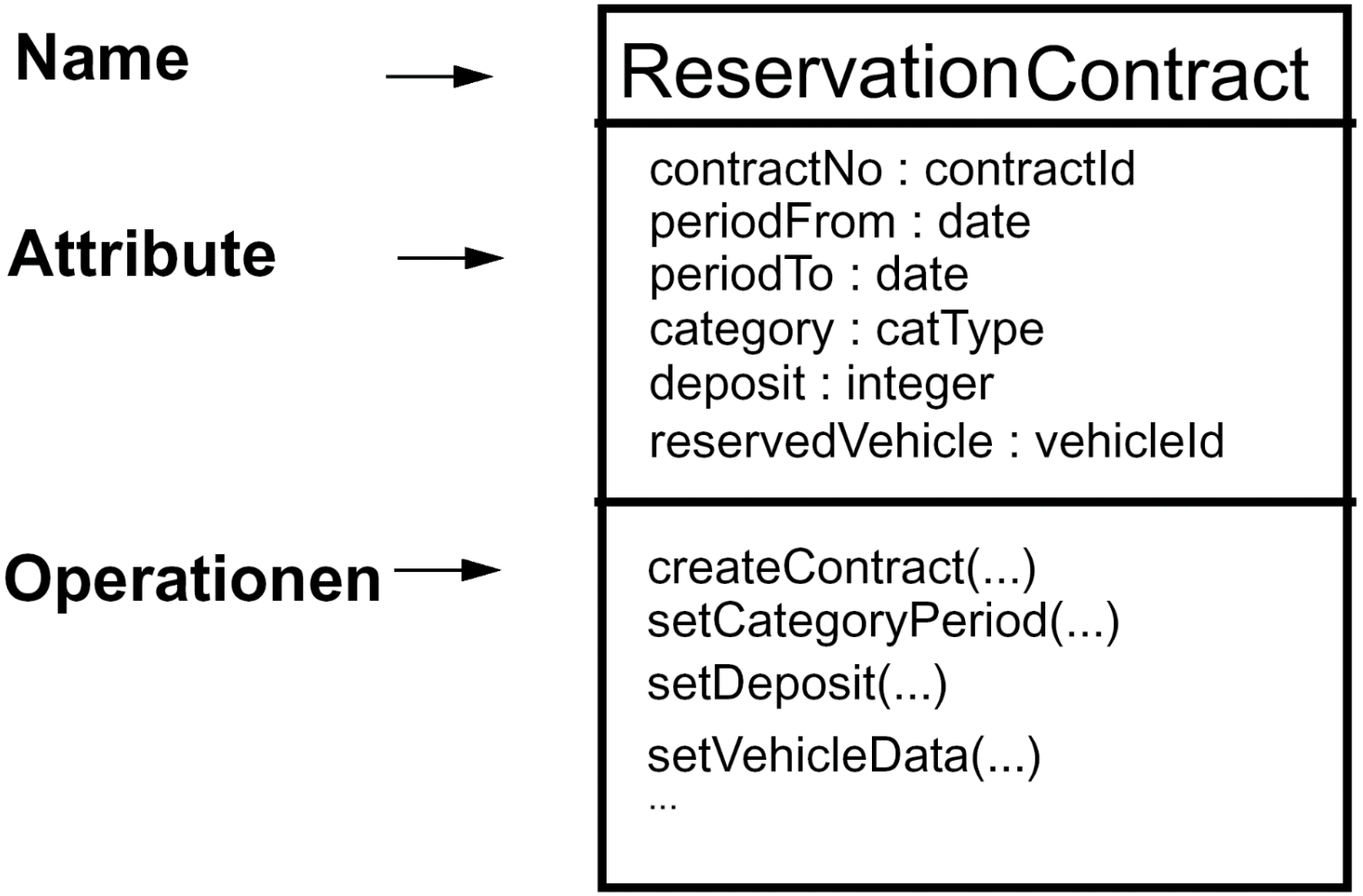


[http://en.wikipedia.org/wiki/Unified_Modeling_Language#mediaviewer/File:UML_diagrams_overview.svg]

Class diagram

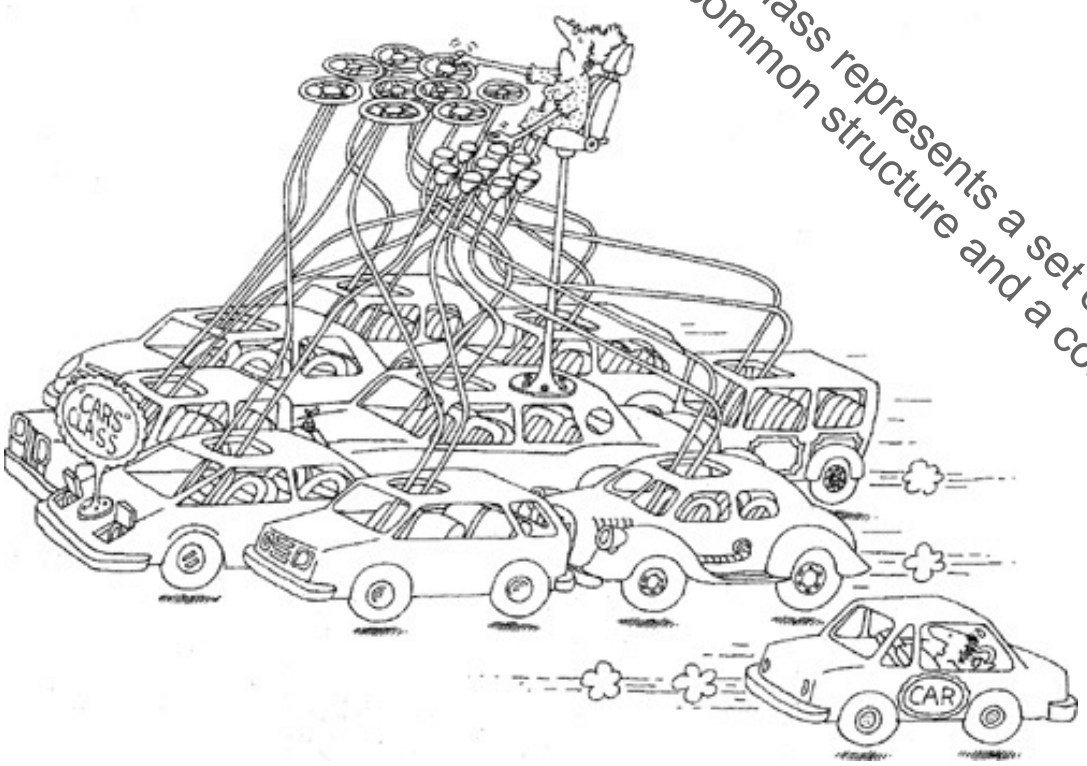
- Based on Entity-Relationship-Modell [Chen1976]
  - Conceptional model to describe the structure of data (Entities) and the relationship between the data
- Tasks
  - **Analysis**: identify object model of system and context (environment)
  - **Design**: design object model of the systems
  - **Implementation**: realize object model of the system
- UML Class diagrams:
  - Consists of definitions of classes and relation types
  - Classes: define structure (attributes) and behavior (signature of operations) of uniform objects
  - Relation types: define relation between objects

Elements of a class



Identification of possible classes

- Step 1: determination of relevant classes
- Analysis: determine classes of problem domain
  - Candidates are all substantives



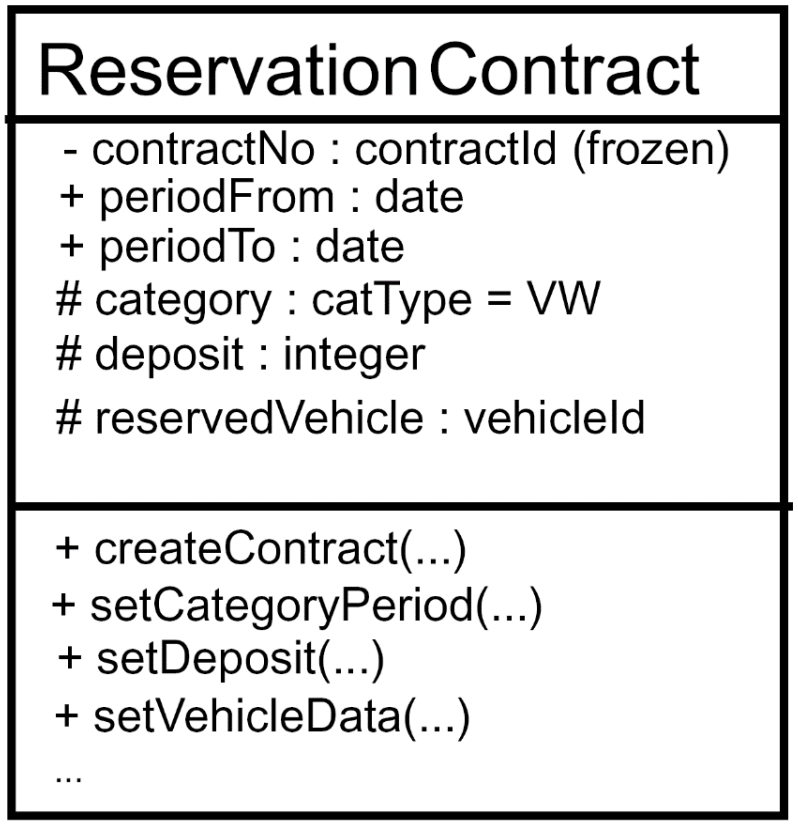
A class represents a set of objects that share a common structure and a common behavior.

Identification of relevant classes

- Step 2: eliminate irrelevant classes

HOCHSCHULE HAMM-LIPPSTADT

- Step 3: define attributes and operations



Purpose: Attributes define the local data of an object

Syntax:

<visibility> <name> : <type-expression> = <initial-value> {property-string}

Optional: <visibility>, <initial-value> und <property-string>

<visibility> defines the visibility of an attribute:

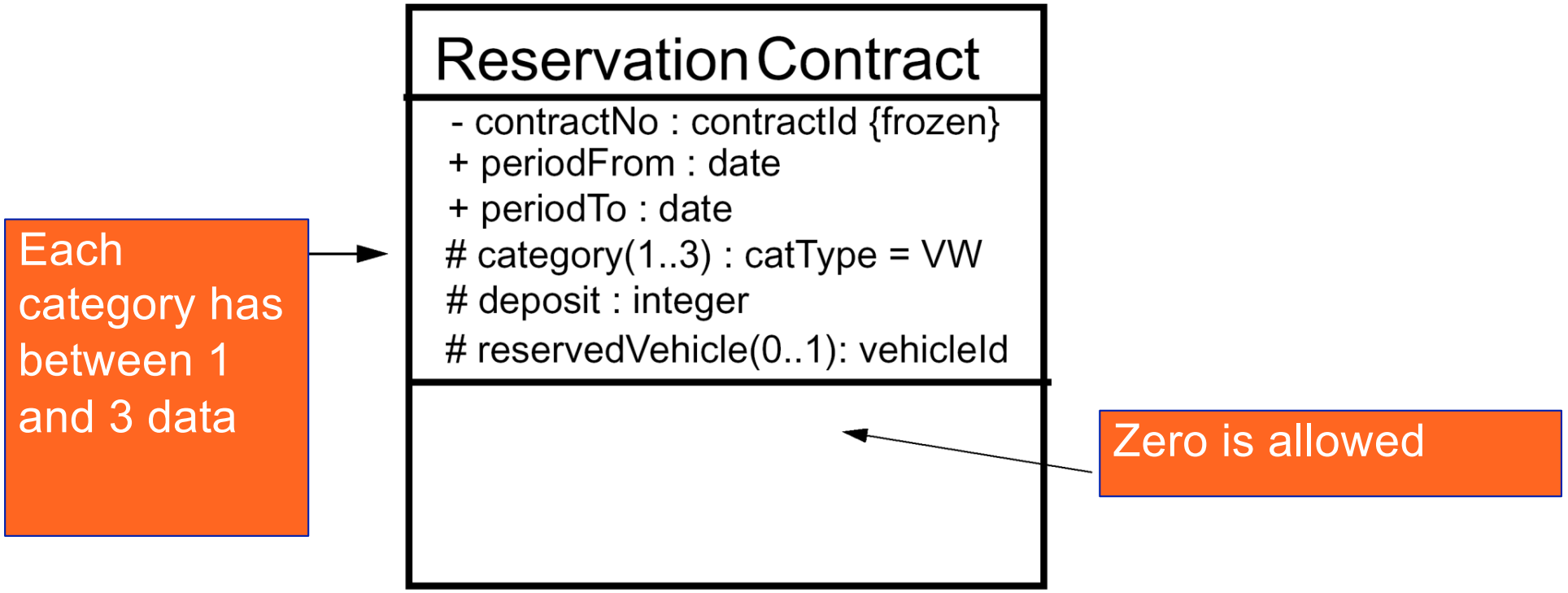
+ public

protected (as in C++)

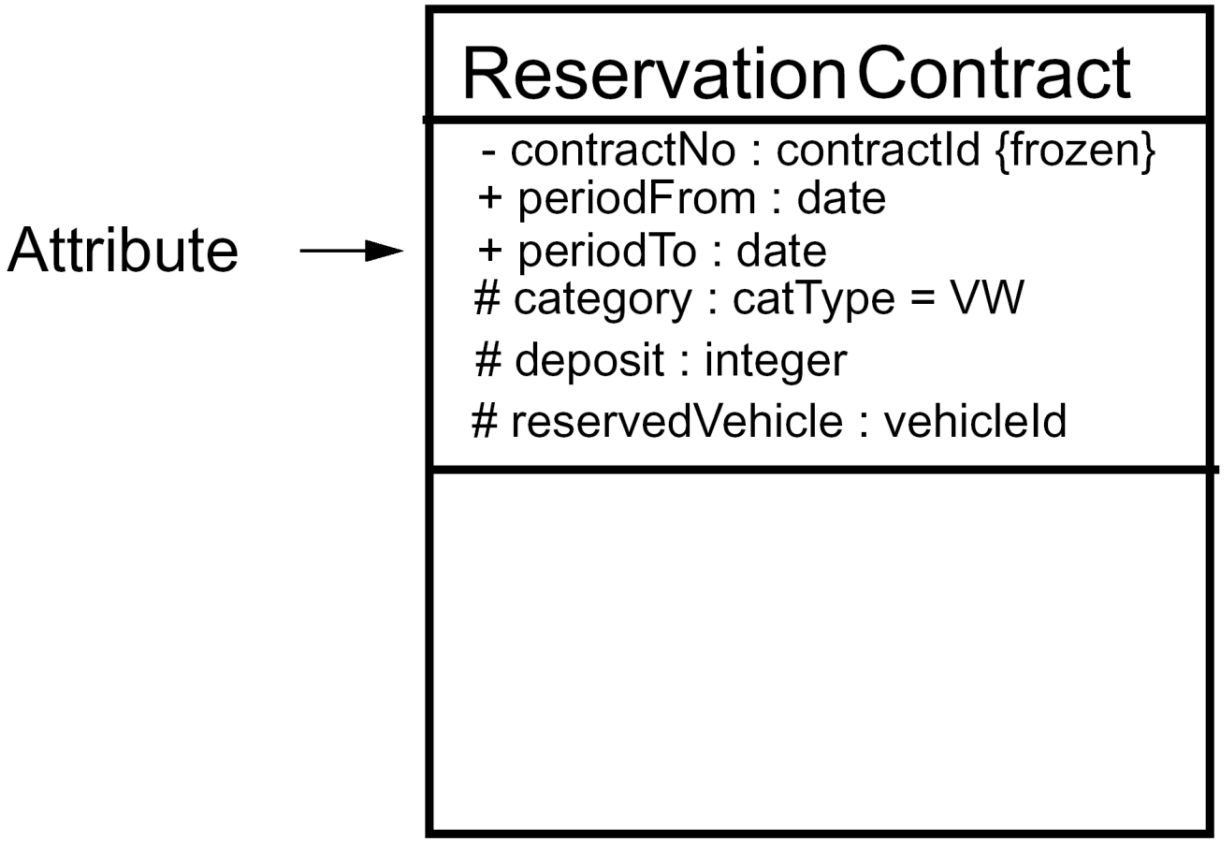
- private

HOCHSCHULE HAMM-LIPPSTADT

- The class (type) defines the multiplicity



<property-string> defines properties like „frozen“ (attribute is not modifiable)



Goal: Operations define the functionality of objects and enable read and write access of objects (and its relations)

Syntax:

<visibility> <name> (<parameter-list>) : <return-type-expression>

Syntax of paramter lists:

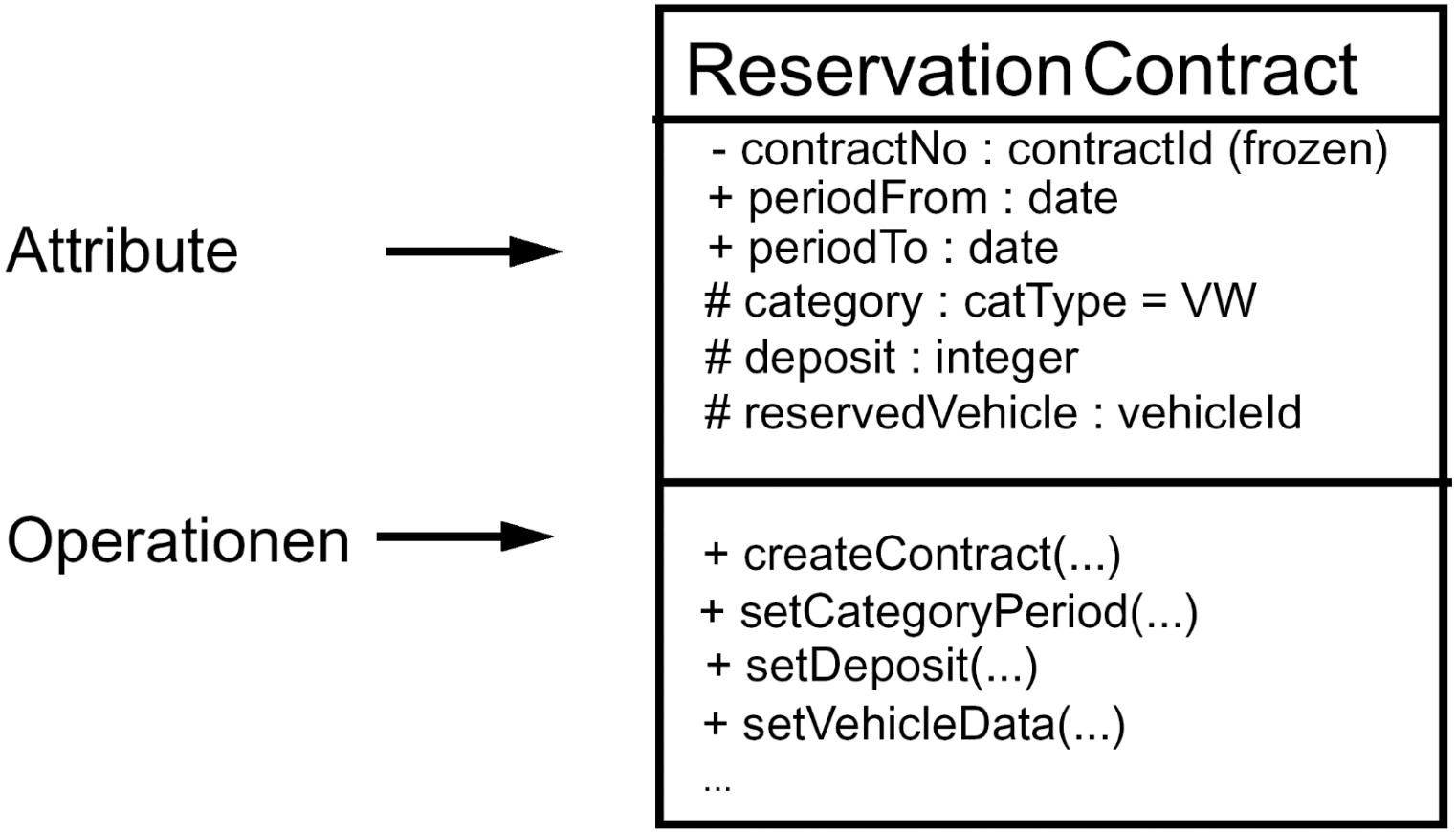
<kind> <name> : <type-expression> = <default-value>

<kind> distinguishes between in and out-put parameters

- Possible values: **in**, **out** or **inout**

Example of an operation

- addCategoryPeriod (**in** from : date, **in** to : date, **in** cat : catType)



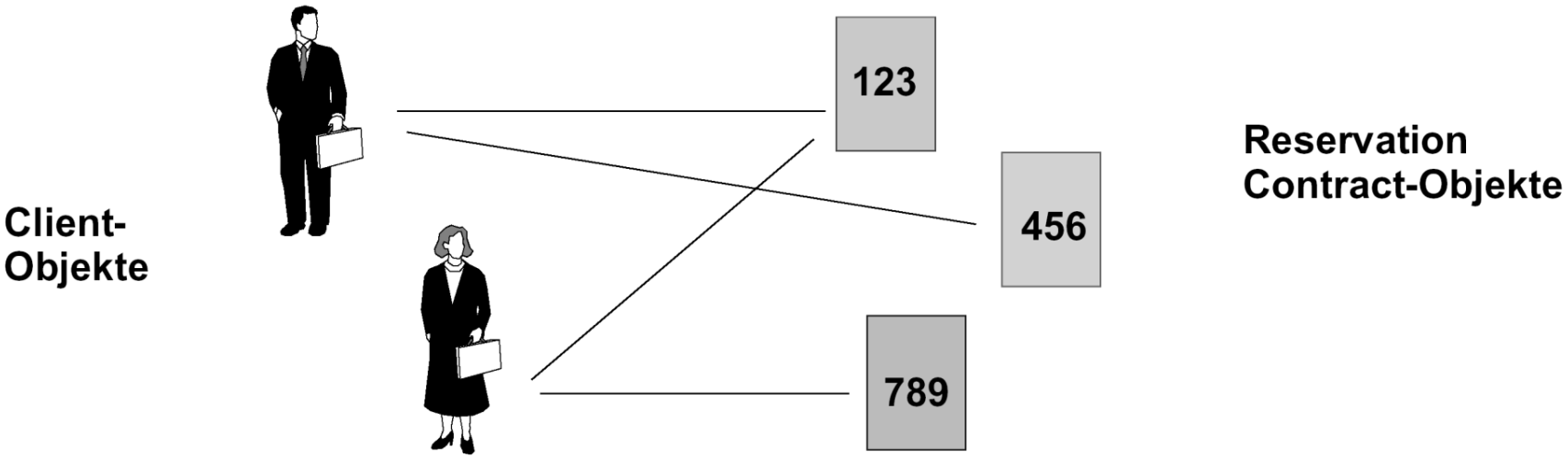
HOCHSCHULE HAMM-LIPPSTADT

Step 4: Define relations between objects

(1) Binary association

Semantics: in state σ holds

$$\sigma(\text{hasResContract}) \subseteq \sigma(\text{Client}) \times \sigma(\text{ReservationContract})$$



Additional declations of associations:

(i) Reading direction

hasResContract ▶

(ii) Multiplicities (Cardinalities)

0 .. 1	1 .. *
1	1 .. 3
0 .. * or * or *..*	1 .. 3, 7, 10 .. *

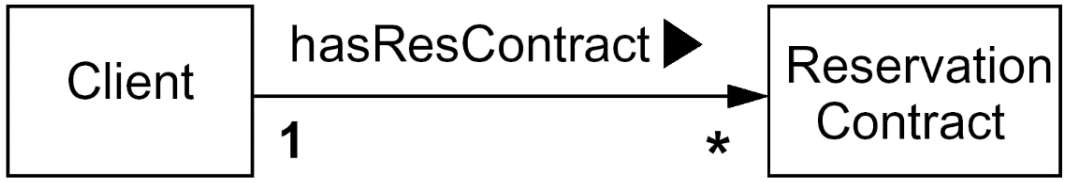
Association (2/6)

(iii) Sorting order

{unordered} un-sorted

{ordered} sorted set

(iv) Navigation direction



(v) Modifiability/Changeability

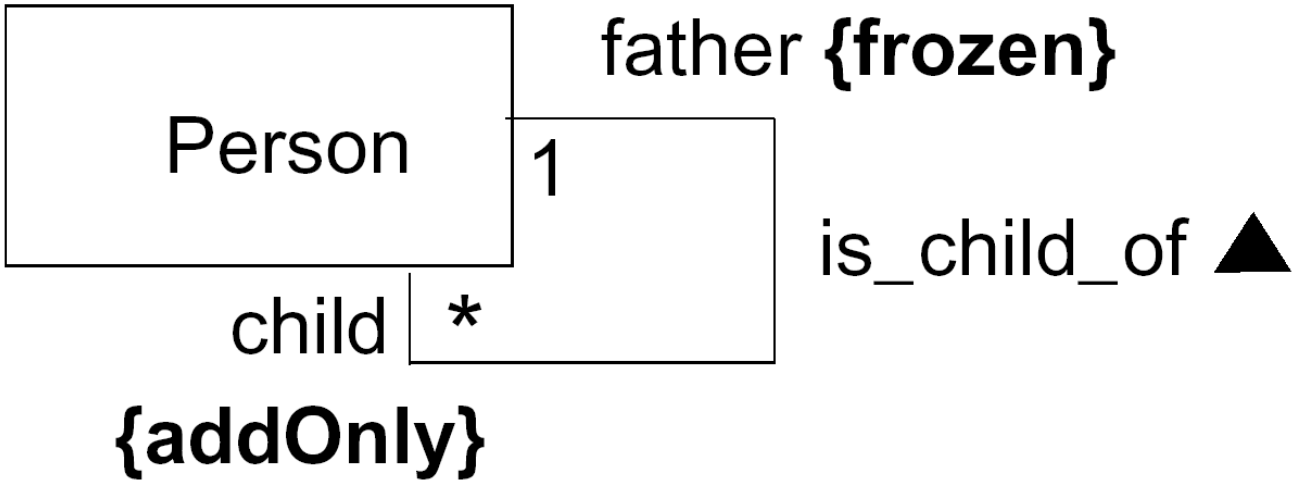
{frozen} frozen indicates that the value of an attribute or association end may not change during the lifetime of the source object. The value must be set at object creation and may never change after that.

{addOnly} add only new values (delete is not allowed)

Association (3/6)

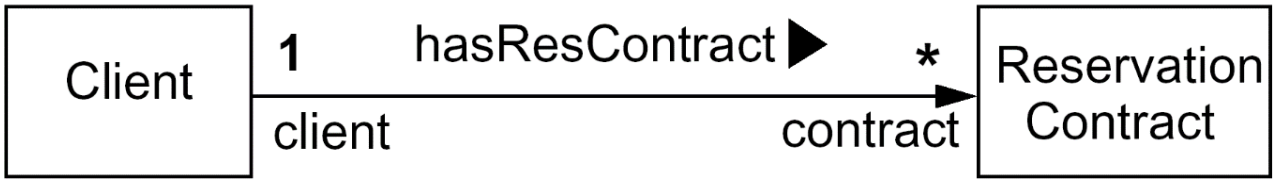
Example

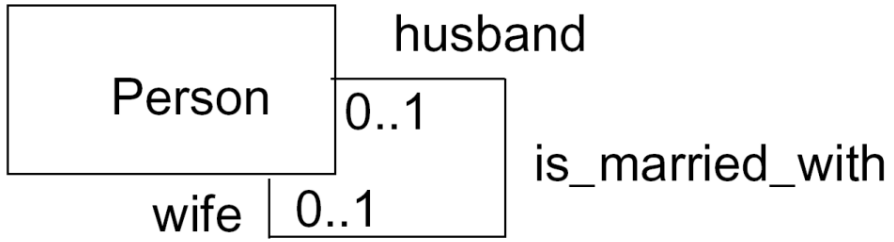
- When „creating“ a new person the father is defined and is not allowed to change
- Once chidren are recognized:
  - They will alwas be children
  - Children could only be added



(vi) Role name

describes the role of an object in a relationship



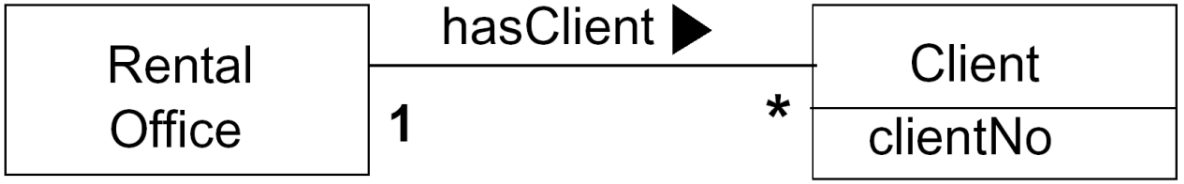


(vii) Visibility (of roles)

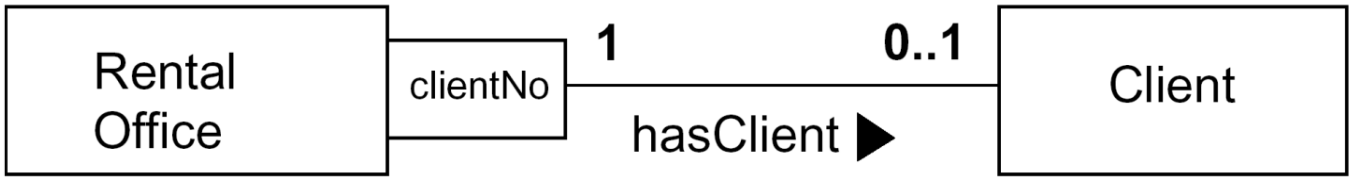
visibility of the relationship in the direction of the role name

(viii) Qualified attributes

- Attribut of a relation
- Defines a decomposition of the object set which are in relation with (other) objects
- 0..* multplicity is adapted to 0..1

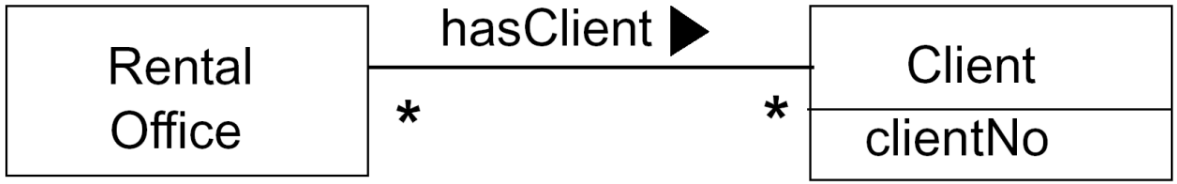


RentalOffice-Object + clientNo identifies the client object

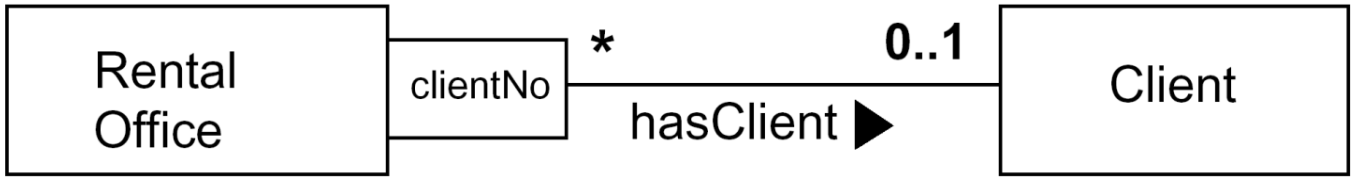


(other) example: the same person is client of different car rental services

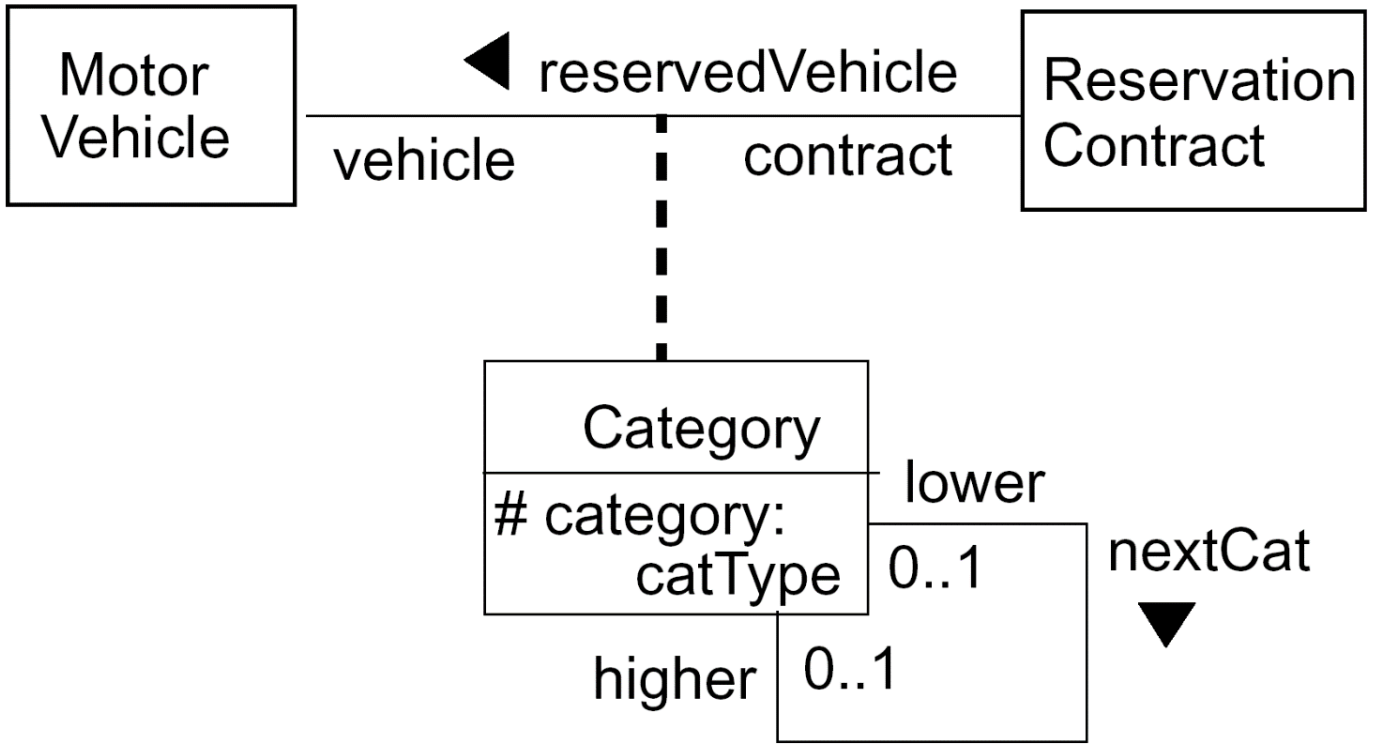
(a) With the same client number:



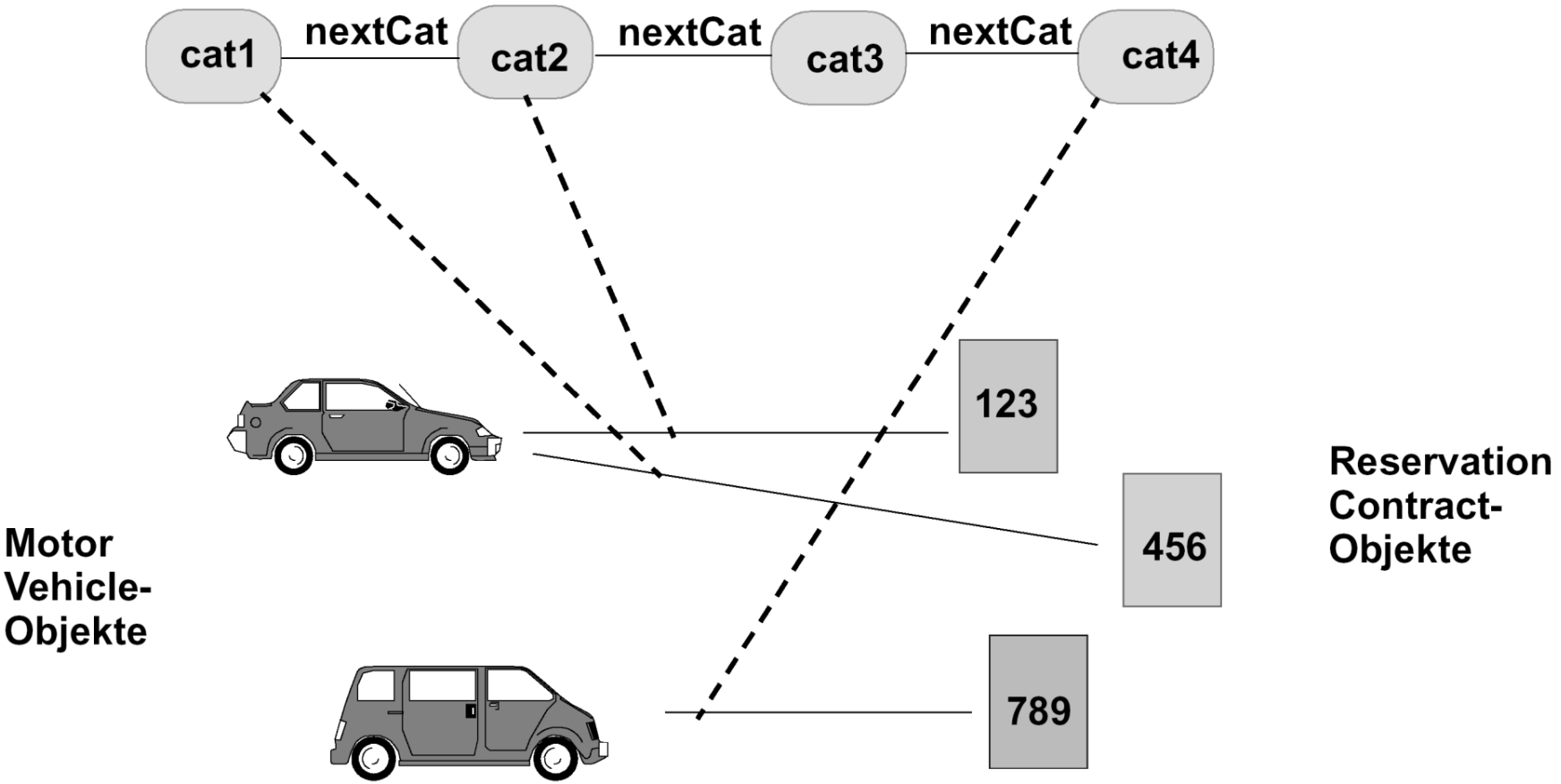
(b) With different client numbers:



- Association class: Association with properties of a class



Category example: Instance level

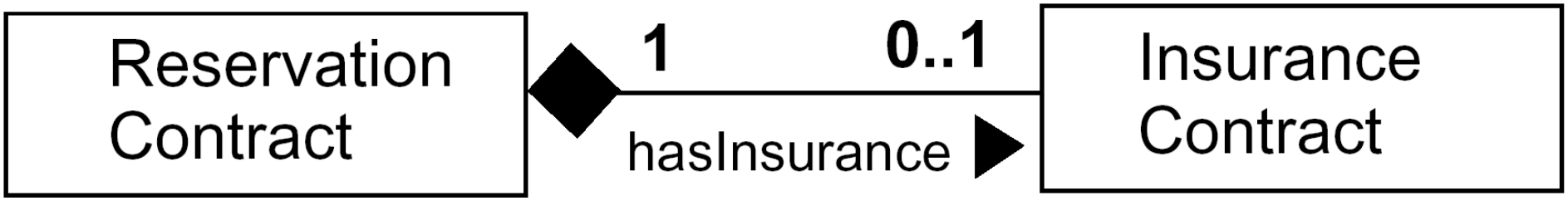


Aggregation

- Binary association with extra semantics
  - Part of relation

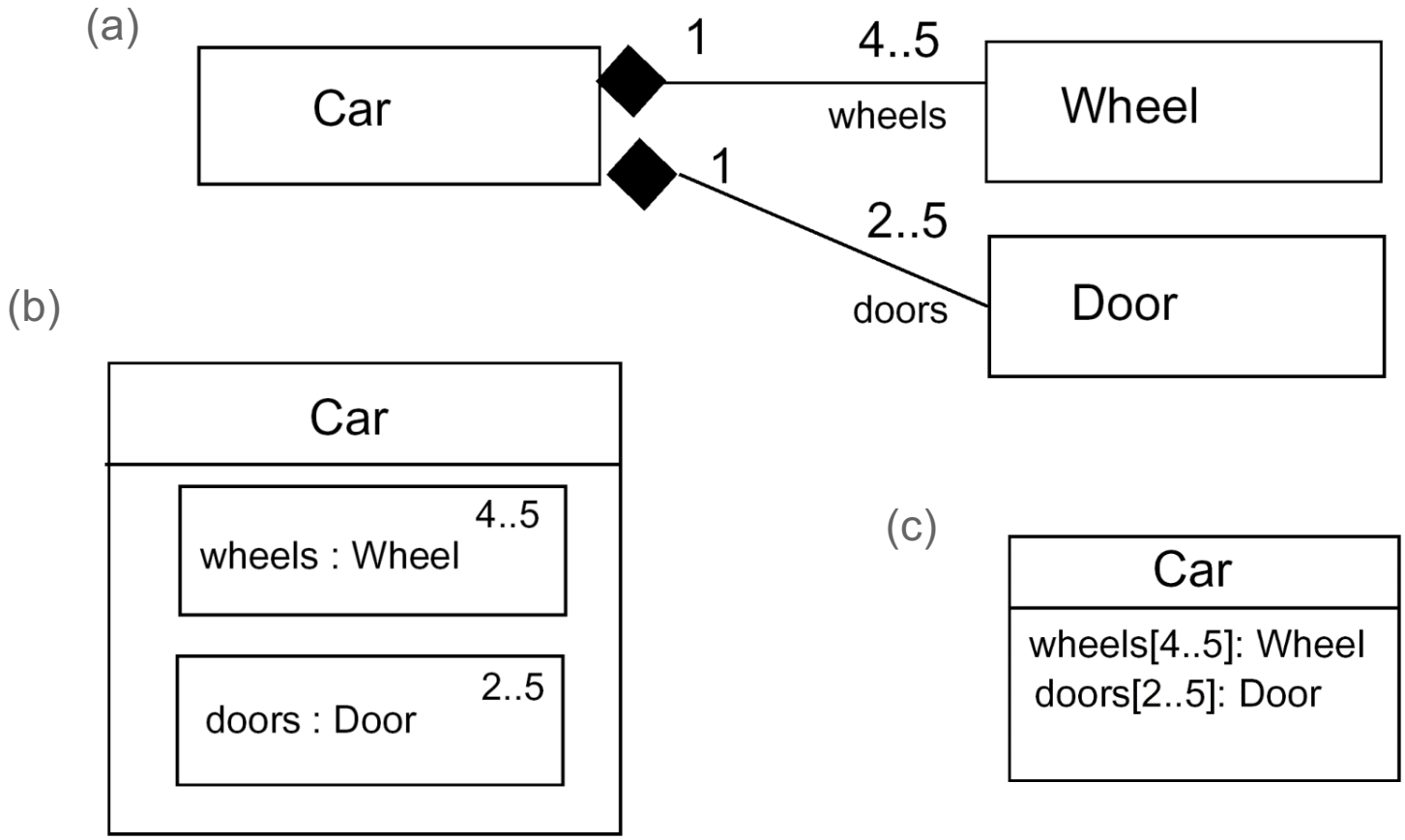
Composition

- Aggregated objects depends on the aggregating object
  - Life time is limited to one of the aggregating object



HOCHSCHULE HAMM-LIPPSTADT

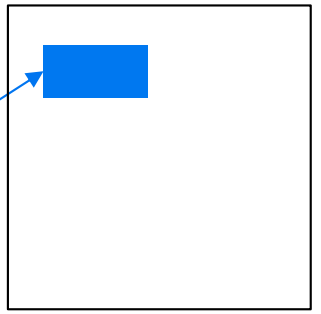
- Different representations

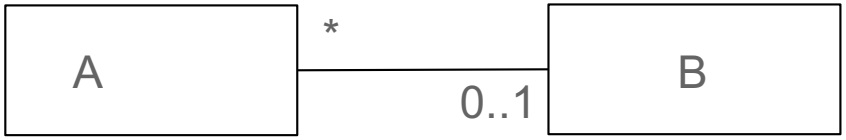


- Ideas?


```
class A{
private: B* myB;

}
```

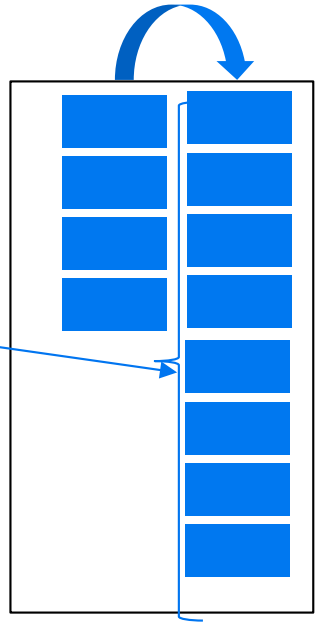
```
class B{
private: A* myA;

public: void setClassA(A* a);

}
```






```
class A{
private: B* myB;

}
```

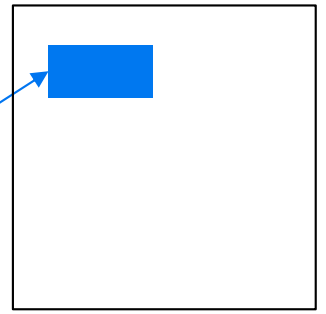
```
class B{
private: A* myA[];

public: void setClassA(A* a);

}
```



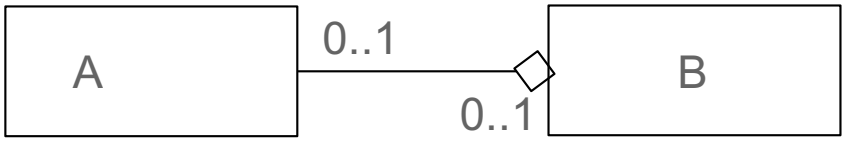
- Ideas?

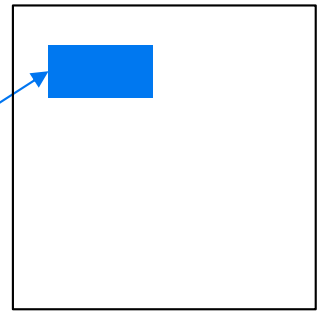

```
class A{
private: B* myB;

}
```

```
class B{
private: A* myA;

public: void setClassA(A* a);
        B(A *a){
            myA = a;
        }


}
```



- Ideas?

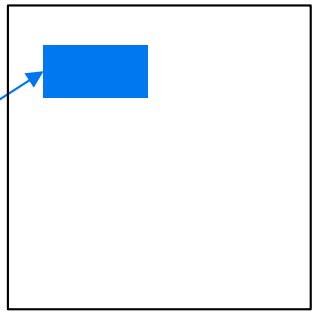




```
class A{
private: B* myB;

}
```

```
class B{
private: A* myA;

public: void setClassA();
}

class A{
private: B* myB;

}


}
```

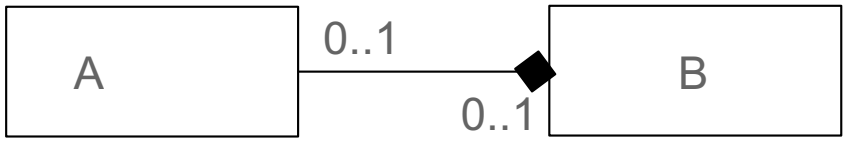
- Ideas?



```
class A{
private: B* myB;

}
```

```
class B{
private: A* myA;

public: void setClassA();
        B(){
           myA = new A();
        }

}
class A{
private: B* myB;

}


}
```

- Ideas?
- And now to C



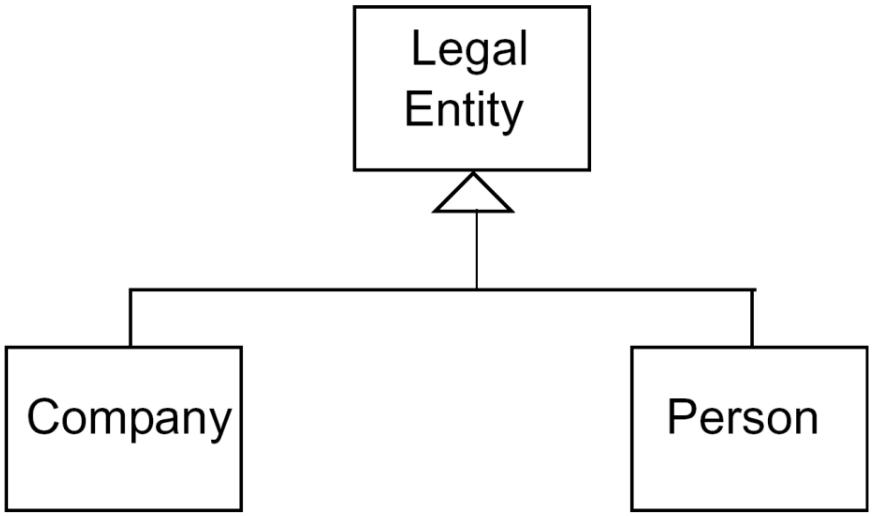
.h
struct A{
 char, …
};
Operations…
.c

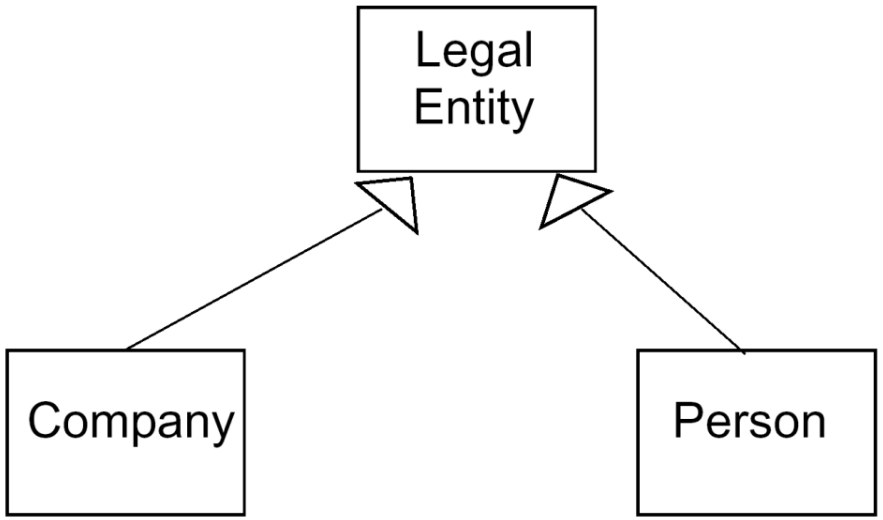
.h

.c

Step 5: Identify common properties of different classes. Define super class and inheritance relation.

(1) Single inheritance





{overlapping} an object can be part of more than one subclass

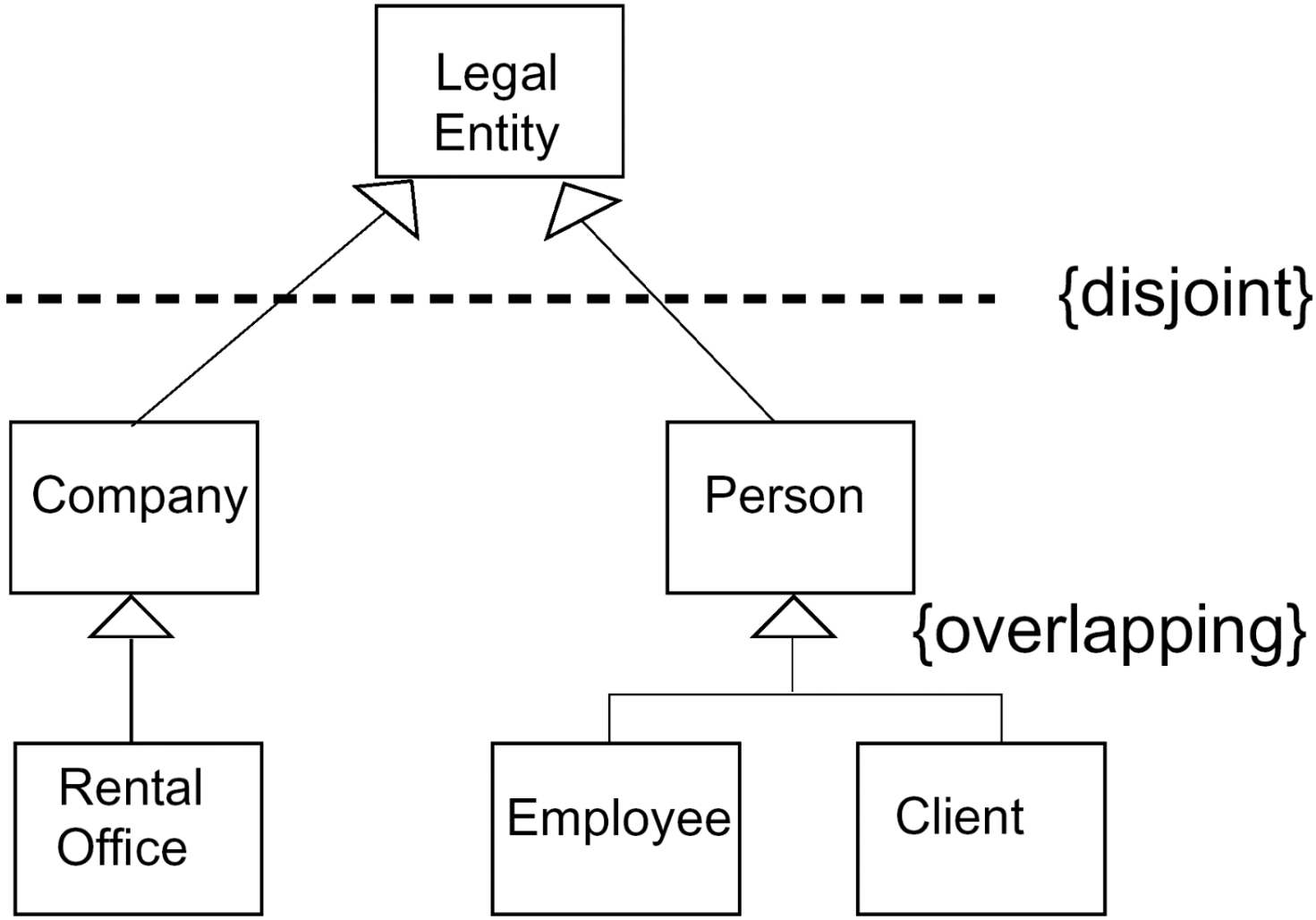
{disjoint} The objects of the subclasses are disjoint

{complete} all subclasses of the upper class are specified

{incomplete} maybe not complete

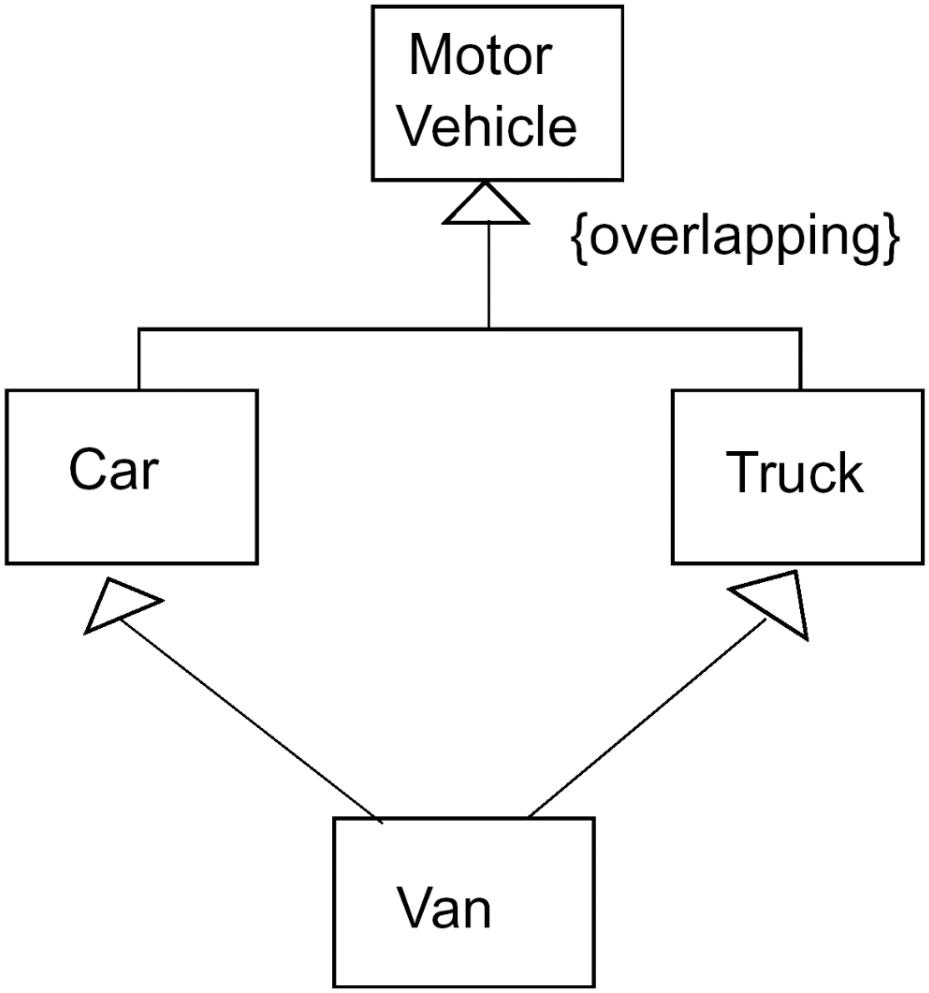
Inheritance (Generalization)

Example



(2) multiple inheritance:

a class inherits from more than on upper class



Analysis Phase: Class Diagram

- **Step 1**
  - Determine relevant classes of problem domain: all substantives are candidates for classes
- **Step 2**
  - not relevant classes are eliminated
- **Step 3**
  - determine attributes and operations
- **Step 4**
  - determine relations between objects (Association, Aggregation)
- **Step 5**
  - Determine common properties of different classes
  - Define common properties in a common super class
- **Step 6**
  - Repeat step 2 to 5 till class diagram is stable

Analysis ⇨ Analysis model of the problem domain

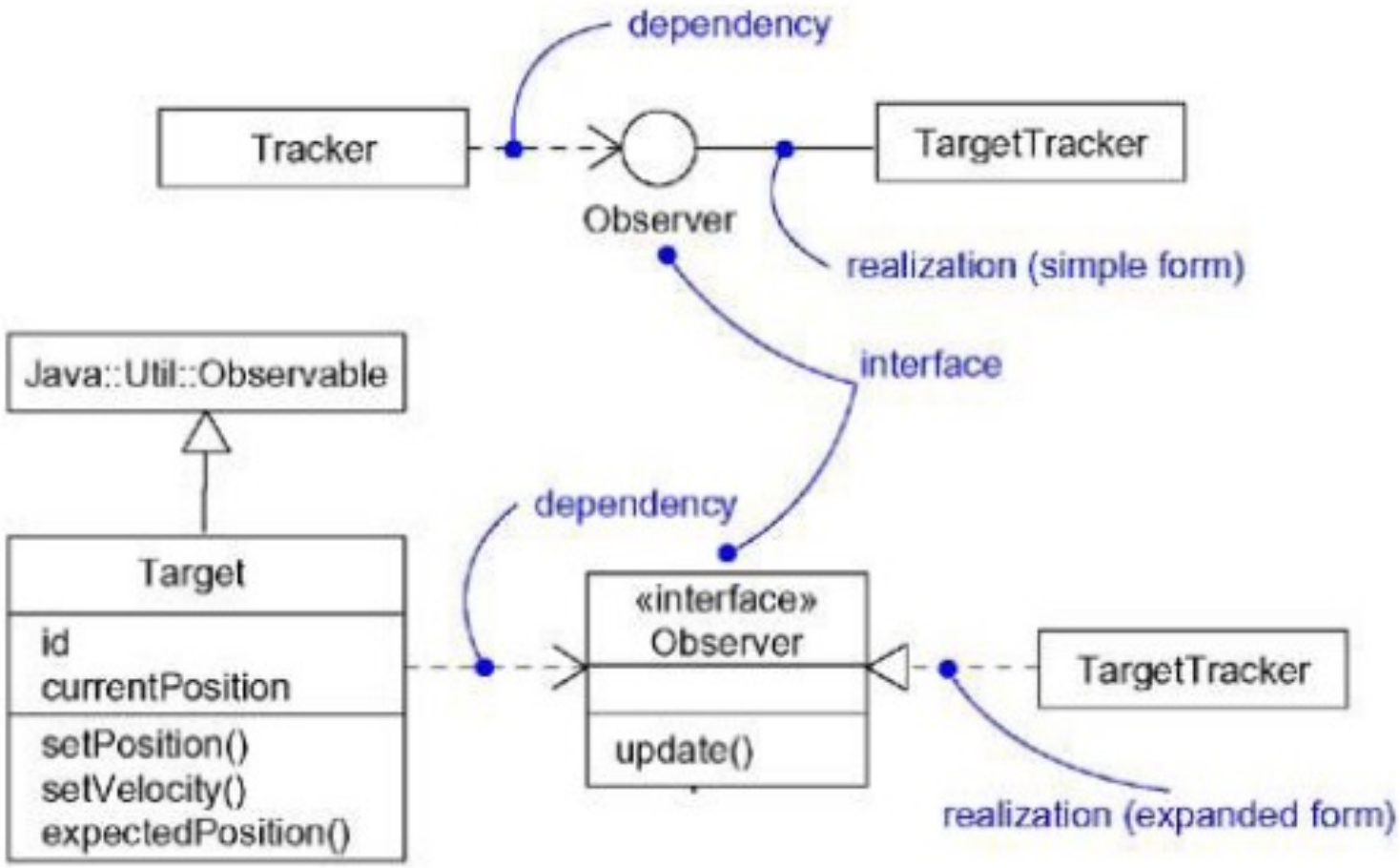
Design:

- Define the software architecture (**coarse design**)
  - Decompose system into subsystem
  - Identify components
  - Relations and cooperation between components (connectors)
  - Result: architectural description
- **Detailed Design**
  - Determine interfaces (contracts)
  - Result: component specification
- Identify properties in your system (and model) where e.g. flexibility is required. Apply design pattern

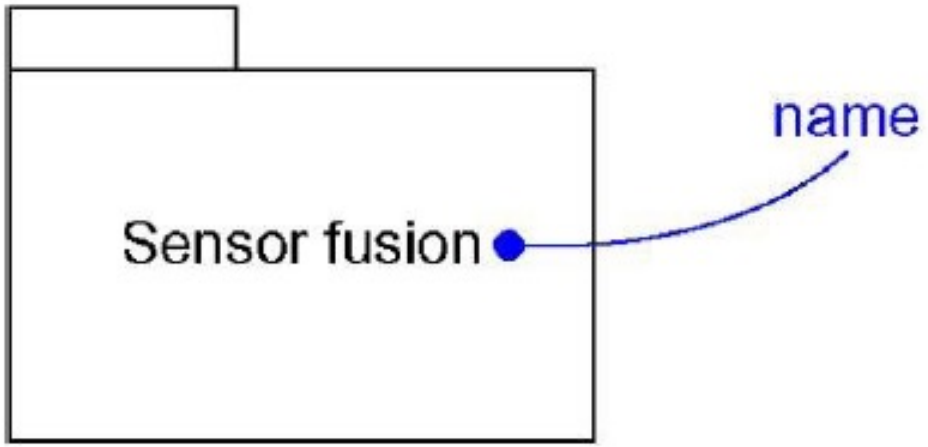
Important: Analysis model ≠ Design model!

Realization

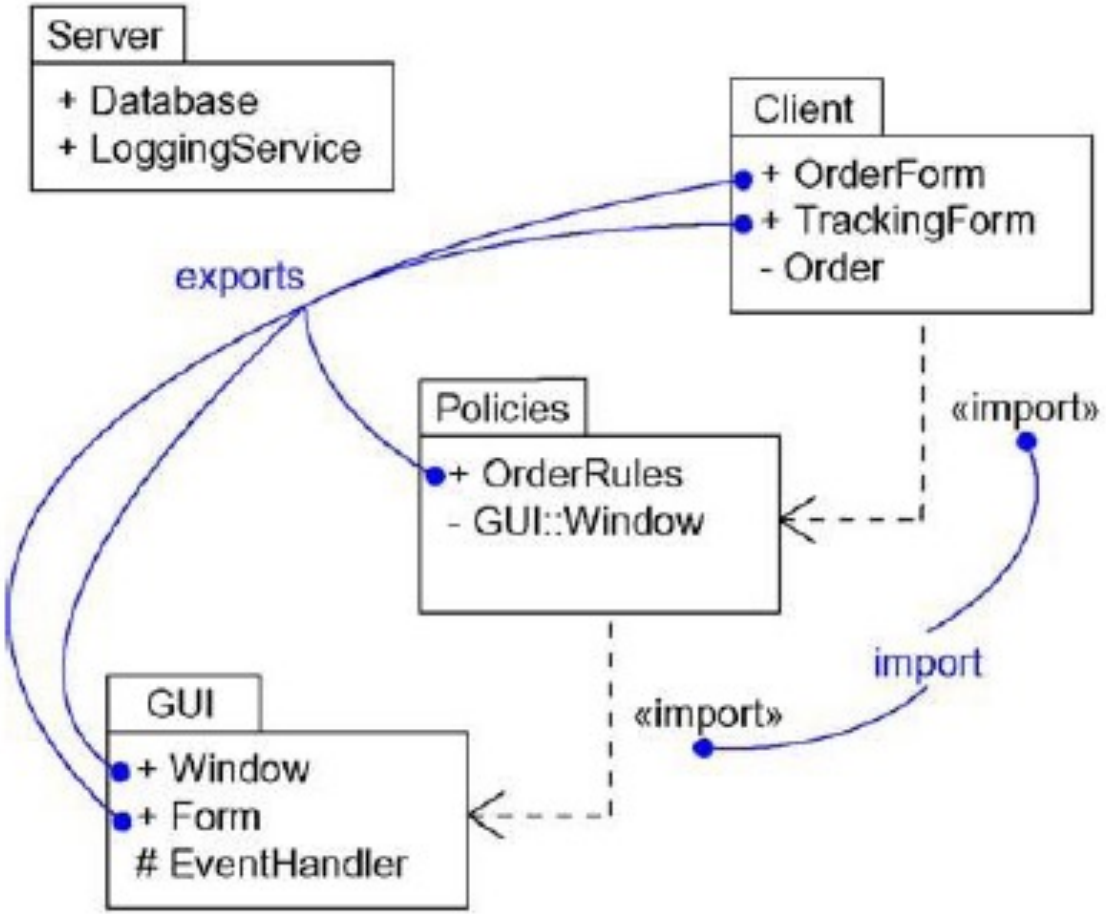
- semantic relationship between classifiers (classes) in which one classifier specifies a contract that another classifier guarantees to carry out.



- visualize groups of elements
  - can be manipulated as a whole
  - control the visibility of and access to individual elements (e.g. classes)

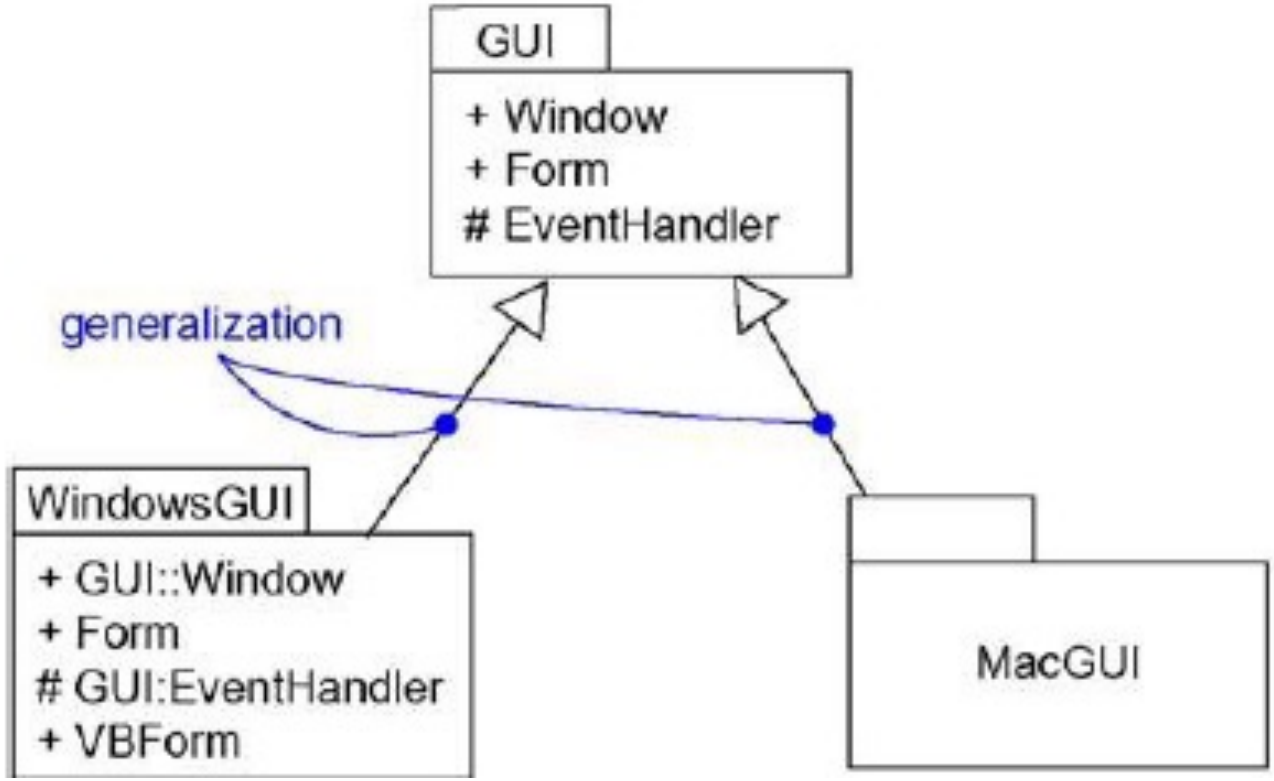


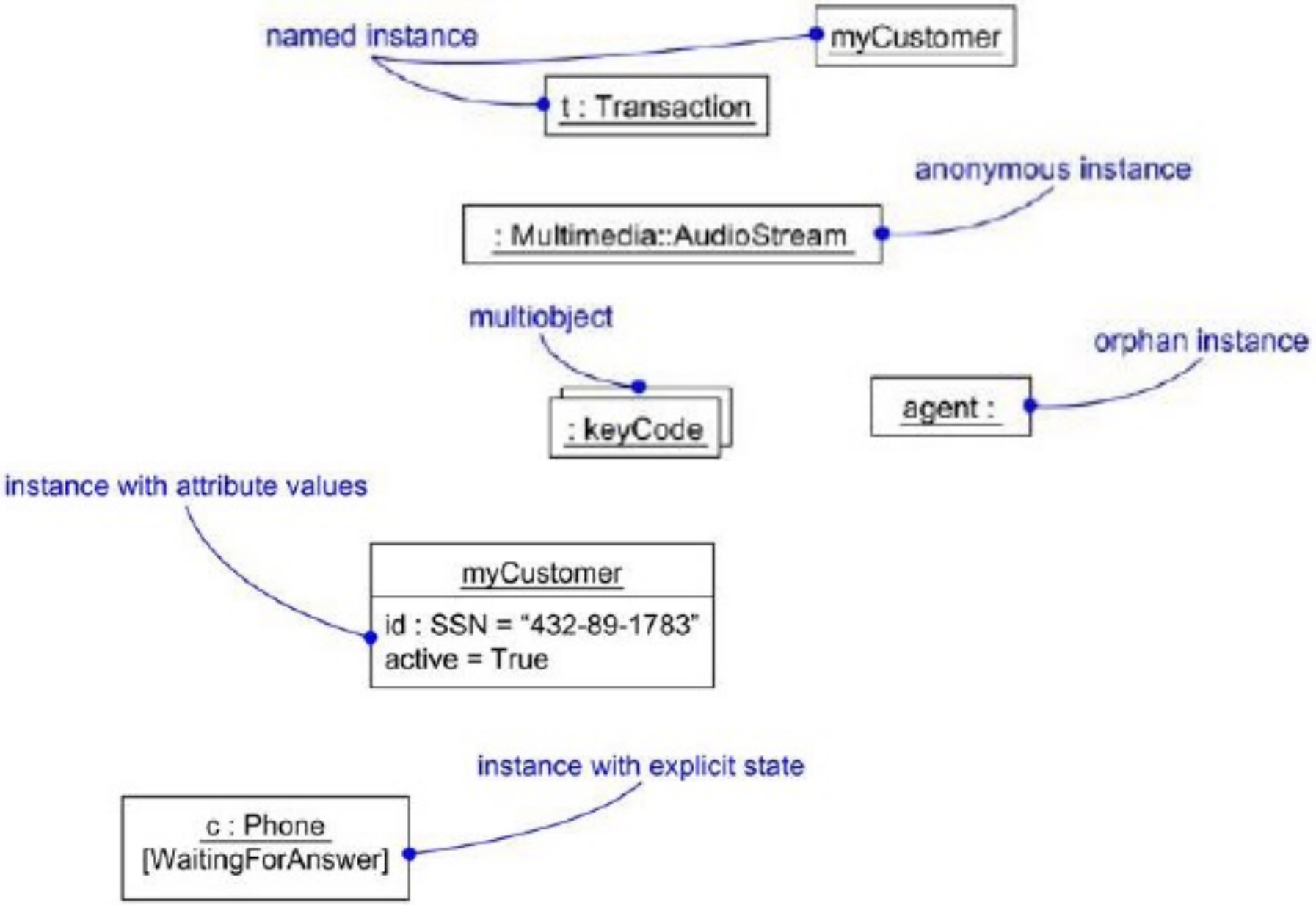
Import and Export



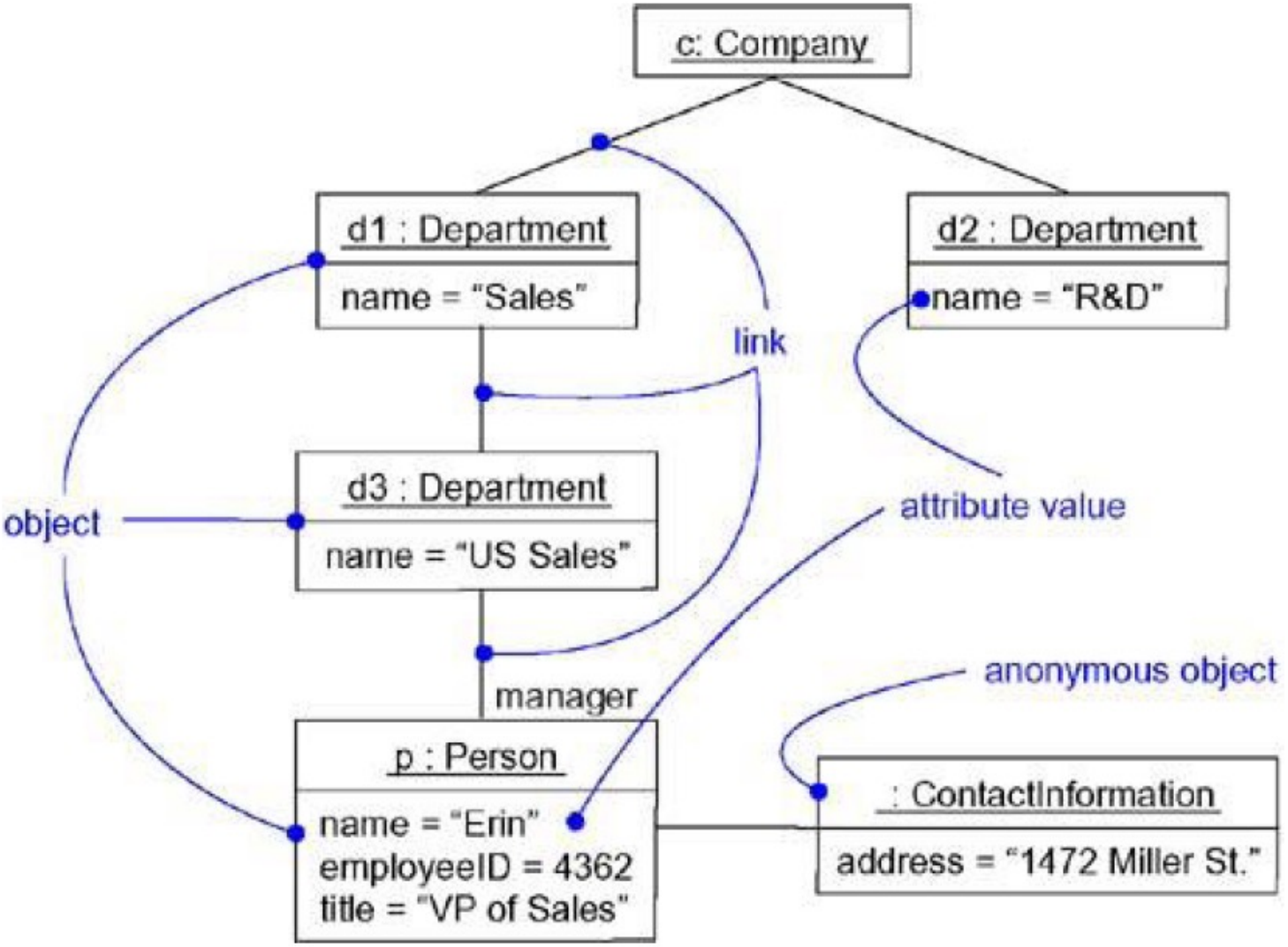
Generalization

- very much like generalization among classes




named instance
myCustomer
t : Transaction
anonymous instance
: Multimedia::AudioStream
multiobject
: keyCode
orphan instance
agent :
instance with attribute values
myCustomer
id : SSN = "432-89-1783"
active = True
instance with explicit state
c : Phone
[WaitingForAnswer]

Object diagram



HOCHSCHULE HAMM-LIPPSTADT

1. Introduction
2. Manual Coding
3. Discussion & Summary
4. Bibliography

HOCHSCHULE HAMM-LIPPSTADT

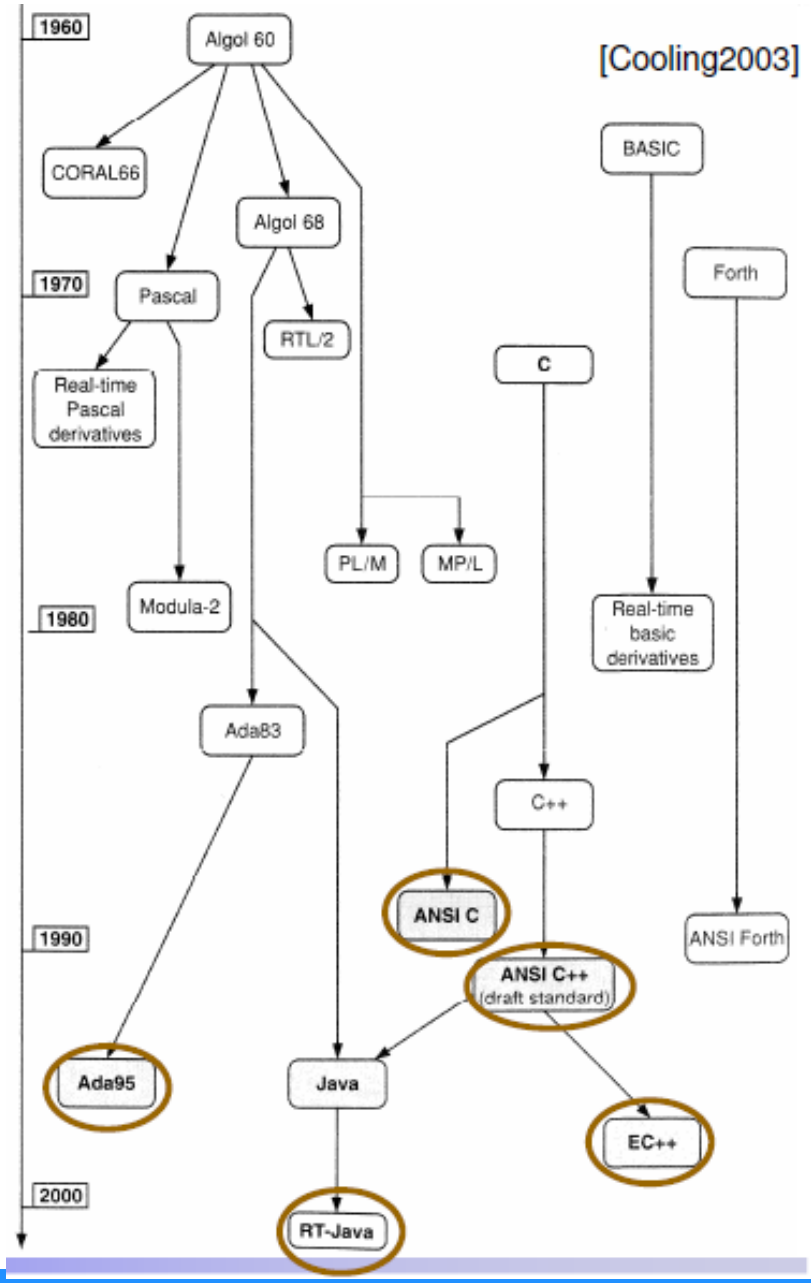
1. Assembly Language
2. (High-Level) Programming Languages
3. Coding Patterns

1) Assembly Language

- Using assembly language programming should be avoided wherever possible, why?:
  - Inherent insecure
  - Extreme weak data typing
  - Limited (or non-existent) flow control mechanisms
  - Difficult to read (weak understandability)
  - Not portable
  - Difficult to maintain
- Occasional reasons to still use it are:
  - Lack of high-level language
  - Hardware testing
  - Interrupt handling
  - Peripheral and subsystem interfacing
  - Speed of program execution
  - Program code and data size

(2) Programming Language

- **Essential features** which are required to realize the system (really needed)
- **Primary features** for correct, reliable and safe programs (needed to produce high-quality software)
- **Secondary features** which make a significant contribution to productivity, portability, maintainability, flexibility and efficiency (features useful from an economic perspective)



Language Comparison (1/3)

Essential features which are required to realize the system:

	Ada 95	C	C++	EC++	Java
Assembly language/machine code interfacing	Y	Y	Y	Y	N
Absolute addressing	Y	Y	Y	Y	N
Access to and control of hardware	Y	Y	Y	Y	N
Interrupt handling	Y	Y – but no standard method	As for C	As for C	N
Bit manipulation	Y	Y	Y	Y	N
Processor specific extensions	Y	Y – but compiler-dependent	Y	Y	N
Pointers	Y	Y – very strong feature	As for C	As for C	N – but has the 'reference'
Timing/time delays	Y	Not directly	As for C	As for C	N

Language Comparision (2/3)

Primary features for correct, reliable and safe programs:

	Ada 95	C	C++	EC++	Java
Comprehensive well-defined language standard	Yes	Yes	Yes	Yes	Yes
Modular structure with good software encapsulation	Excellent	Poor	Good	Good	Excellent
Data typing	Good	Poor	Better than C, weaker than Ada, Java	As for C++	Good
Separate compilation with type checking	Good	Poor	Much better than C, weaker than Ada, Java	As for C++	Good
Behaviour and control structures	Good	Poor	Poor	As for C++	Good
Maths model	Good	Poor, insecure	Weak, but superior to C	As for C++	Similar but superior to C++
Exception handling	Yes	No, but can be devised	Yes	No	Yes
RTOS interfacing	No, but can be devised	As for Ada 95	As for Ada 95	As for Ada 95	As for Ada 95
Memory exhaustion checks	Yes	No, but can be devised	As for C	As for C	Yes, including garbage collection
Subset for safety-critical applications	Yes – SPARK Ada	Yes – MISRA C	No	No	No

HOCHSCHULE HAMM-LIPPSTADT

Secondary features which make a significant contribution to productivity, portability, maintainability, flexibility and efficiency.

	Ada 95	C	C++	EC++	Java
Readability	Good	Moderate to very poor (highly programmer-dependent)	As for C	As for C	Better than C++ (no pointers) but inferior to Ada
Comprehensive standard library	No – only a basic library	Yes – very good	Yes – very good	Uses a restricted set of C++ libraries	Yes
OOP constructs	Yes	No	Yes	Yes	Yes
Multitasking – language and run-time support	Yes	No	No – compiler-dependent	As for C++	Yes – the Java threading model (but must run on top of a host OS)
Interfaces to 'foreign' HLLs	Yes – but compiler defined	As for Ada 95	As for Ada 95	As for Ada 95	Yes

HOCHSCHULE HAMM-LIPPSTADT

- **Relevant aspects:**
- a) Reactivity
- b) Memory management

a) Reactivity

- Observation: The control state of a real-time task and its reactive behaviour w.r.t. events and timing is essential for embedded real-time systems.
- **State Pattern**

State Pattern (1)

- **Intent:** Allow an object to alter its behaviour when its internal state changes.
- **Also Known as:** Objects for States
- **Applicability:**
  - The object behaviour depends states
  - Operations have large multipart conditional statements that depend on the object state.

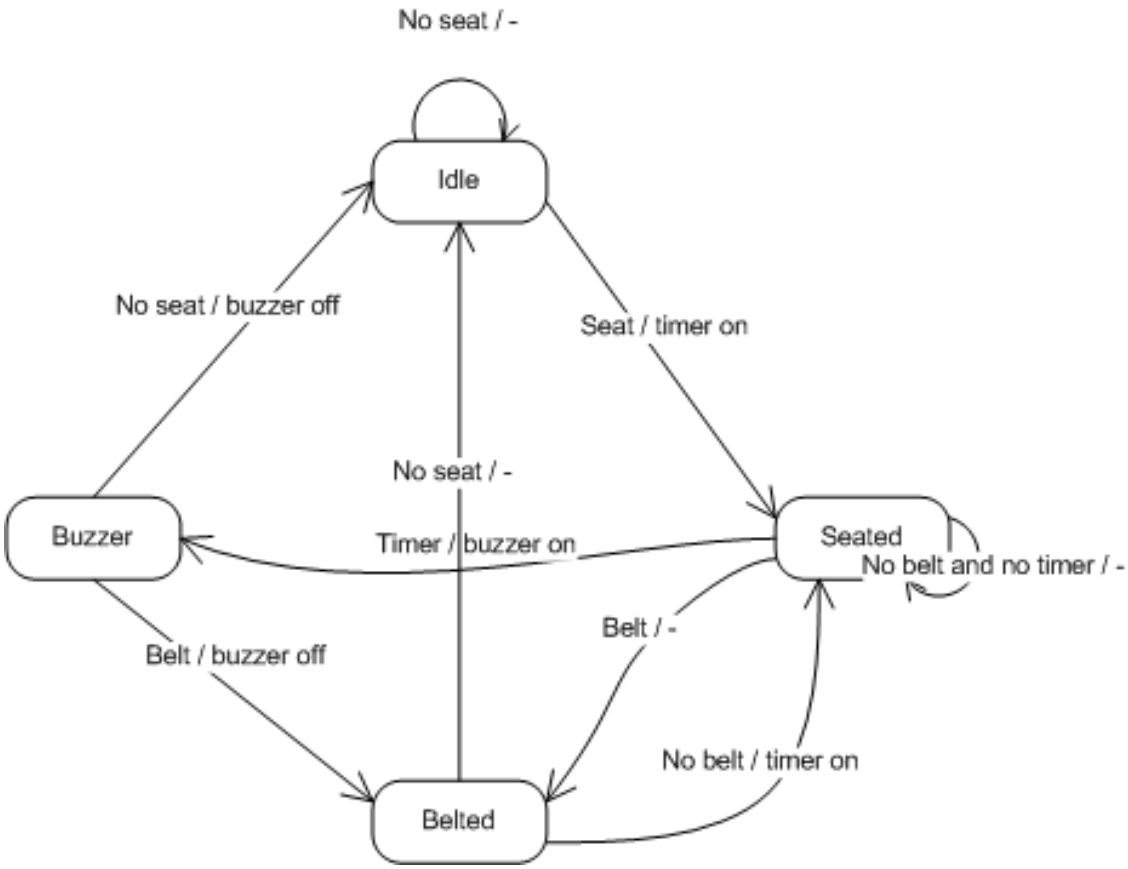
- **Participants:**
  - **Context**: defines interface of interest and handles current state
  - **State**: defines an interface encapsulating the behaviour associated with each particular state
  - **ConcreteState**: specific implementation of state dependent behaviour

State Pattern (2)

- Implementation:
  - In simple cases: switch/case statements
- Examples:
  - Reactive objects with a main event loop (as common in embedded system)
  - State Pattern is often used to implement statechart designs:
    - Explicit "Transition" objects with code
    - Nested States
    - Parallel States
    - History-mechanism

- **Known Uses:**
  - TCP connections
- **Related Pattern:**
  - State objects are often Singletons

State Pattern example



Pattern „switch-case“

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3

statemachine()
{
   /* Declaration of states */

   switch (state) {
          case IDLE:
                    if (seat) { state = SEATED; timer_on = true; }
                    break;
          case SEATED:
                    if (belt) state = BELTED;
                    else if (timer) state = BUZZER;
                    break;
          /* … ? */
   }
}
```

Problems:

- Necessity of managing data resources in a safe manner as **heap fragmentation** can arise when different sized blocks are allocated and released asynchronously from a heap.
- Explicit dynamic memory management is notoriously error prone.

- **Static Allocation**
- Fixed-Size Allocation
- Smart Pointer
- …

Static Allocation (1)

Solution:

- Pre-allocate most or all objects in the system. If message objects are needed, some maximum number of them are created as the system starts up and then placed in a ring buffer.

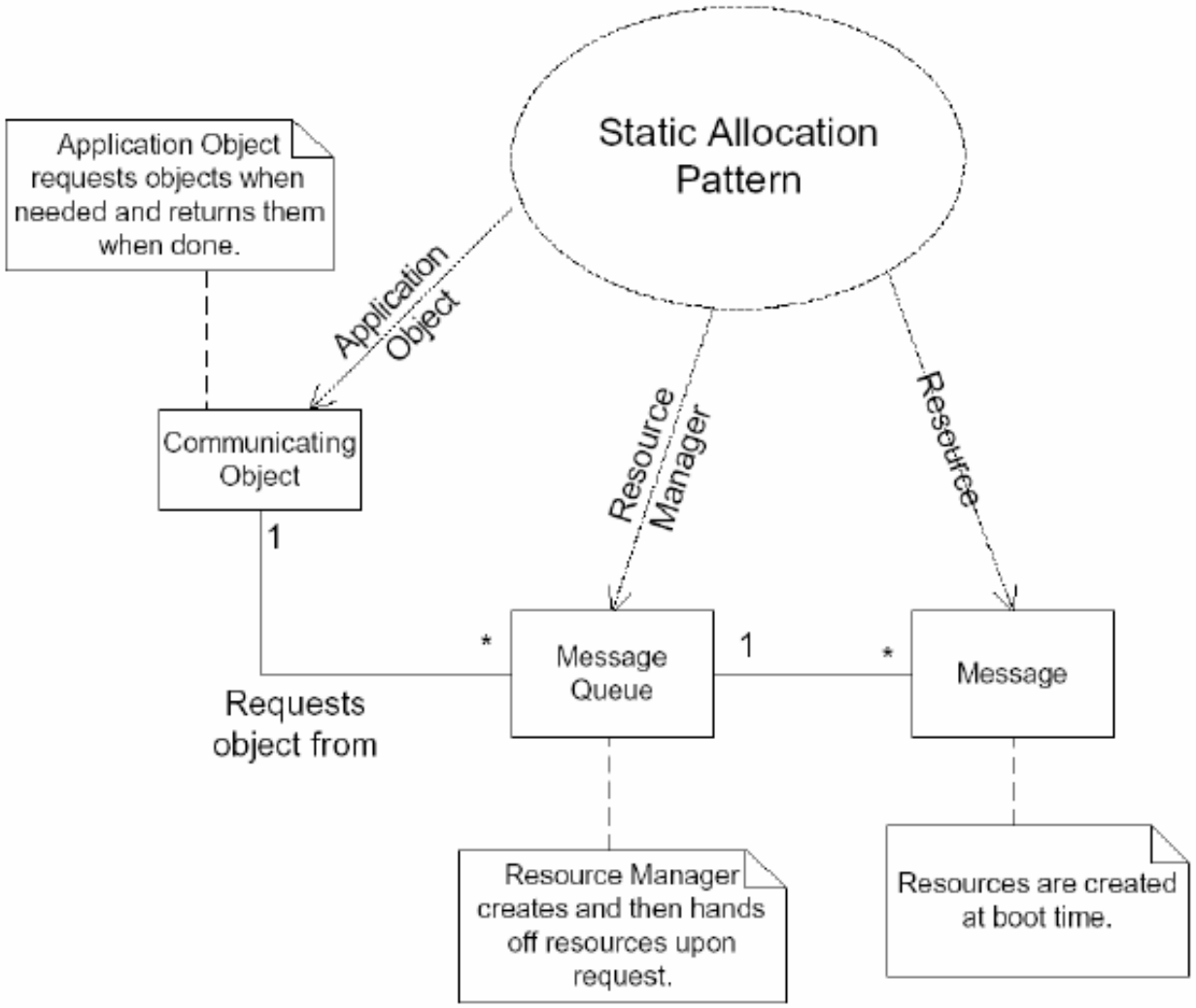
Remarks:

- This pattern can be realized with the Factory Pattern

- [Gamma+1994]

Static Allocation (2)

[Douglass1999]



[Douglass1999]

Pros & Cons

Pros:

- Because memory is never released, heap fragmentation cannot occur
- Overhead is minimized during run-time because time need not be taken to call object constructors.

Cons:

- One problem with this pattern is that it may not scale up well to large problems.
- It may be impossible to pre-allocate all possible needed objects because of a lack of total memory.
- Many systems can operate in a much smaller amount of total memory if they can dynamically create and destroy objects as needed.

- High-level programming languages should be employed when manual coding where possible. Assembly language is only justified where this is not possible.
- Code generation is state-of-the-art for continuous as well as discrete behavior. For continuous behavior the numerical approximation and floating-point/fix-point realization usually require that the models are adjusted! Usually the timing constraints are not considered during code synthesis.

[Cooling2002] Jim Cooling, Software Engineering for Realtime Systems. Addison Wesley, November 2002

[Douglass1998b] Bruce Powel Douglass. Safety Critical Systems. In Embedded Systems Conference, Spring 1998.

[Douglass1999] Bruce Powel Douglass. Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns. Addison- Wesley, May 1999.

[Rammig01] F. Rammig: http://www.inf.ufrgs.br/~flavio/ensino/cmp502/TutorialRammig.pdf

[Wietzke&Tran2005] J. Wietzke and M.~T. Tran, Automotives Software Engineering : Embedded Systems, Modell-Framework, Vernetzte Systeme. Springer, 2005. 1 edition.