PA103 - Object-oriented Methods for Design of Information Systems

Introduction to object-oriented design

© Radek Ošlejšek
Fakulta informatiky MU
oslejsek@fi.muni.cz

Lecture 1 / Part 1:
Course Organization

Course Organization

Prerequisites:

- Knowledge of object-oriented programming principles
  - e.g. the basic PHP, Java, C++ or C# courses
- Core knowledge of software engineering and UML
  - **PB007 – Software Engineering I**

Follow-Up and Related Courses:

- **PV167 – Project in Object-oriented Design of Information Systems**, spring
- PA017 – Software Engineering II, autumn
- PV260 – Software Quality, spring

About the course

Lectures:

1. Course organization, OO design vs. structured design, OO fundamentals, OO modeling vs. ER modeling.
2. Interface as contract, introduction to components, from classes to components.
3. Object Constraint Language.
4. Code refactoring („refactoring to patterns“).
5. Software re-use, software patterns at various stages of software life cycle (analysis, design, architecture, coding).
6. Design patterns in detail.
7. Analysis patterns, Java patterns, anti-patterns.
8. Software architectures, architectural patterns.
9. Component systems. Qualitative attributes and their evaluation.
10. Object-oriented methods for software development, application of UML models in RUP.
11. Special methods and architectures: MDD, FDD, SOA, ...
12. Model-Driven Architecture (MDA), employing OCL in MDA.

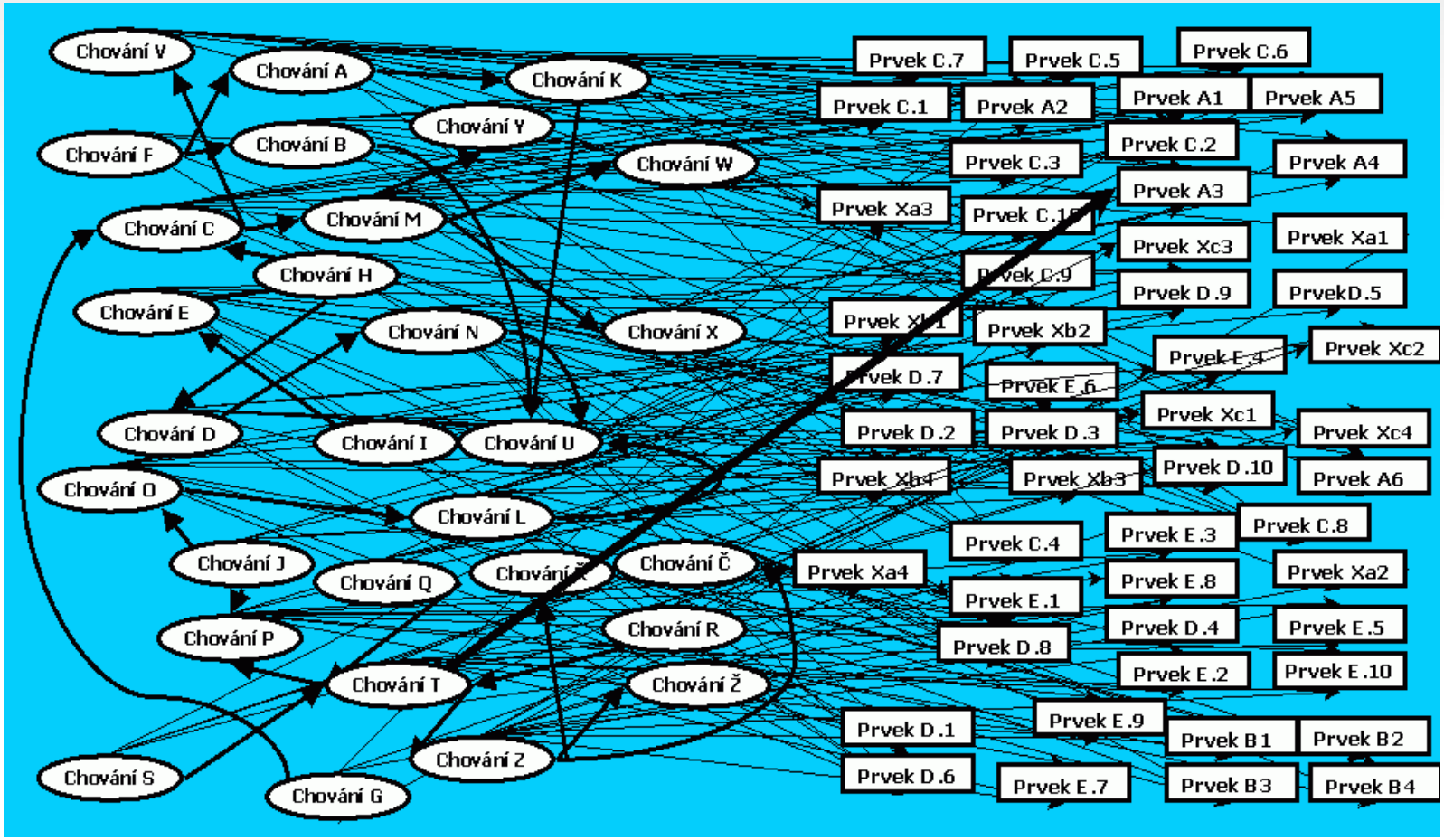
Evaluation:

- Exam = multichoice test + practical question(s), 90 min.
- Grades: A: 100-90 B: 89-80 C: 79–70 D: 69-60 E: 59-50 F: 49-0

Lecture 1 / Part 2:
Structured vs. Object-oriented Paradigms

Why software models?

- Information systems are always composed of **data** and **operations**, which are responsible for data manipulation and presentation to users
- Many relationships => it's infeasible to treat a complex system as a whole
- Modeling = controlling the complexity by the “divide et impera” principle



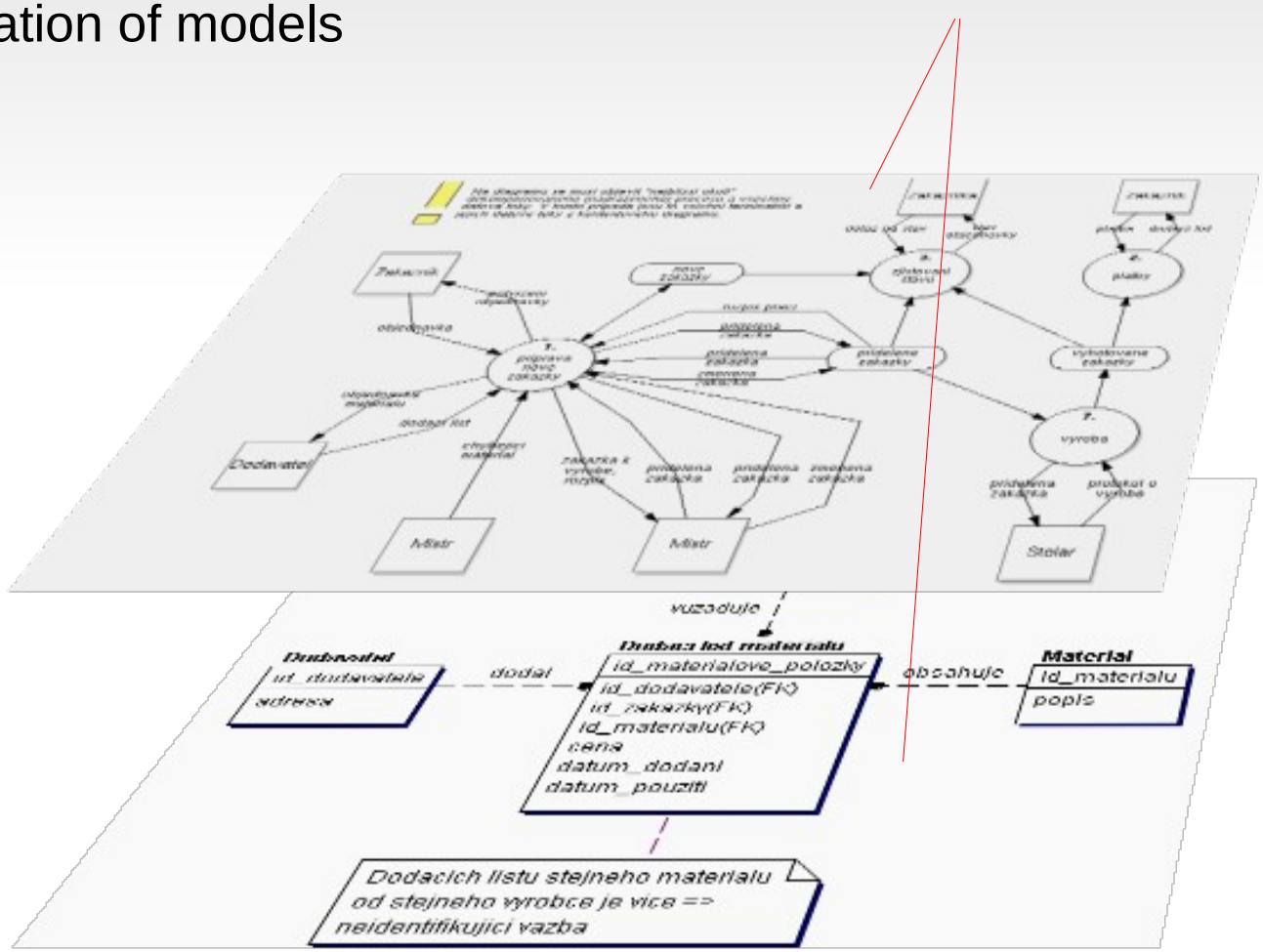
Source: objekty.vse.cz

Structured Modelling

- Separate functional and data models
  - Context diagram, data flow diagram, events, functional requirements, ...
  - Entity-relationship diagram, data vocabulary, ...
- Continuous particularization of models
- Consistency checking
  - Within models
  - Between models

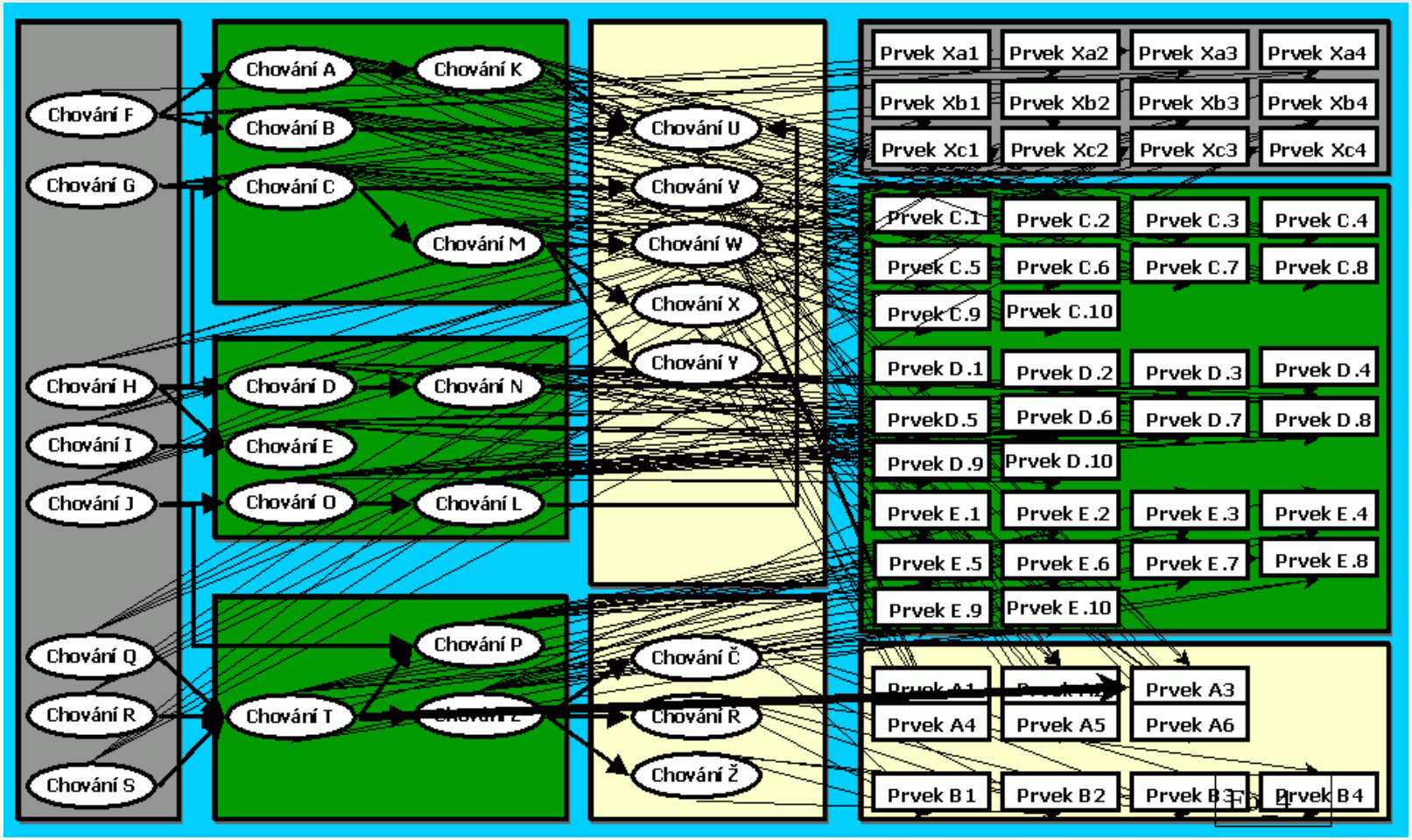
Consistency within models

Consistency between models



Structured Modelling (cont.)

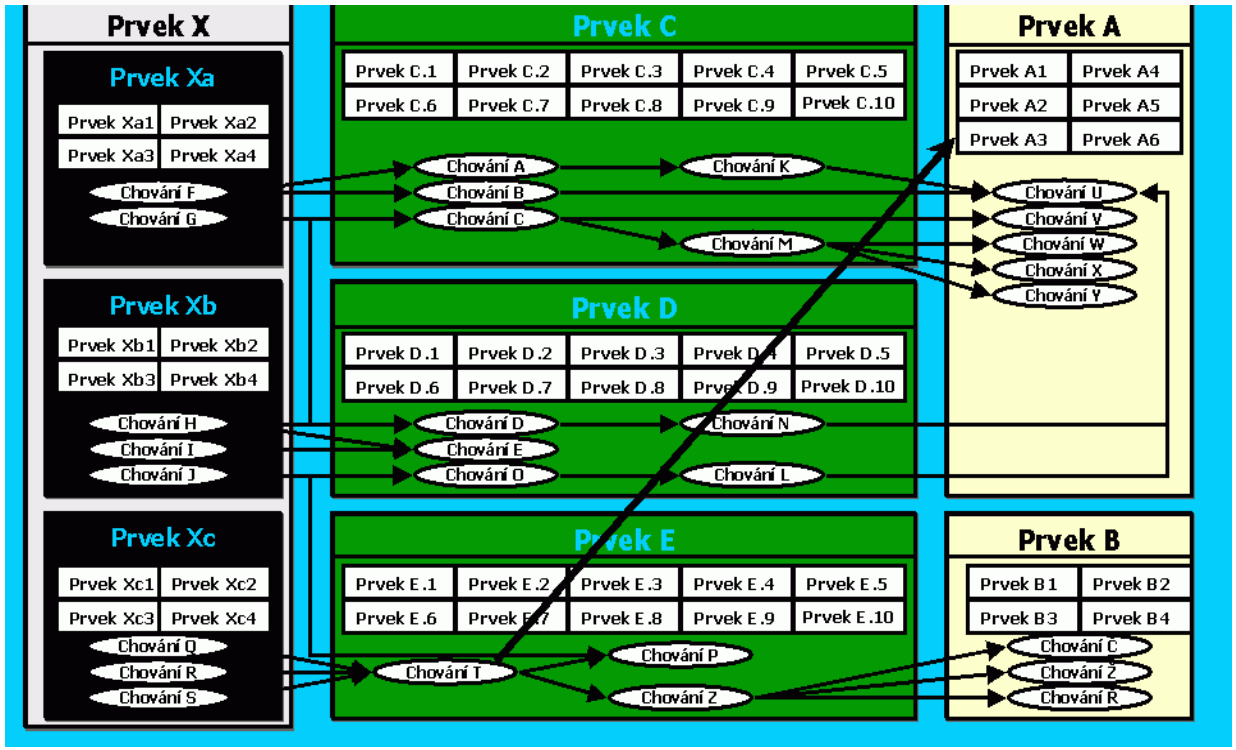
- Functional **hierarchies** and data **clustering** help to organize functional and data models.
- Still too complex relationships mainly between functional and data models.



Source: objekty.vse.cz

OO Modelling

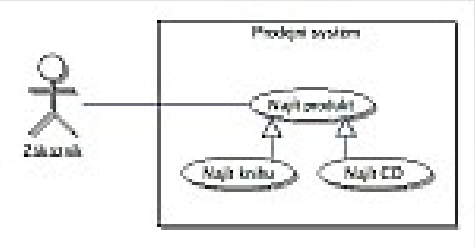
- Division of system into objects handling data as well as operations => function-data dependencies are internal, hidden inside objects.
- Object-to-object relationships are simplified.
- Hierarchical clustering of objects/classes into packages and components brings even more “clarification” of the system. On the other hand, components bring much more complicated communication dependencies then objects/classes.
- **Network of objects** and their relationships as opposed to layers in structured design



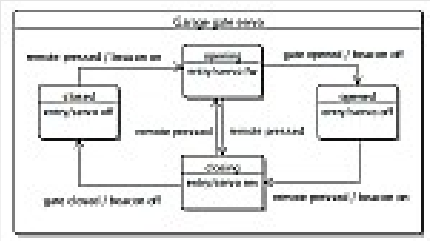
OO Modelling (cont.)

- More models than in the case of structured modeling
- Not all models, e.g. UML models, are always used. Some models are relevant to only selected phases of software life cycle and/or selected parts of the system.
- Continuous particularization of models
- Consistency checking
  - Inside models
  - Between models
- Class diagram as the main model. Other models just help to design correct final class diagram.
- Incremental and iterative development
- complex life cycle management

Use case diagram

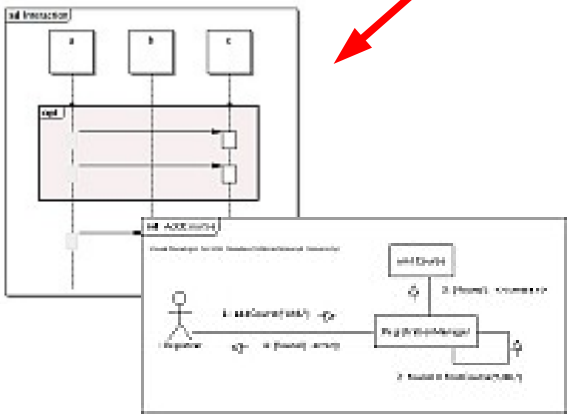


State machine diag.

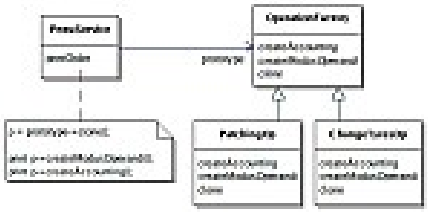


Consistency checking and particularization of models

Interaction diag.



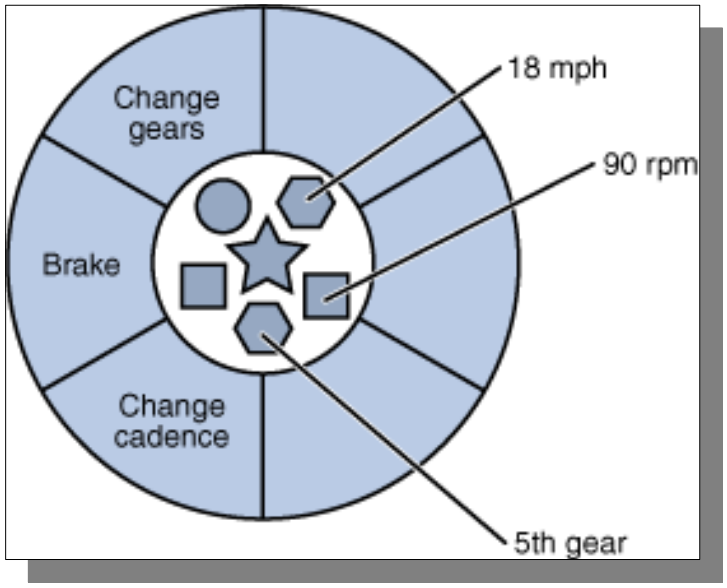
Class and object diag.

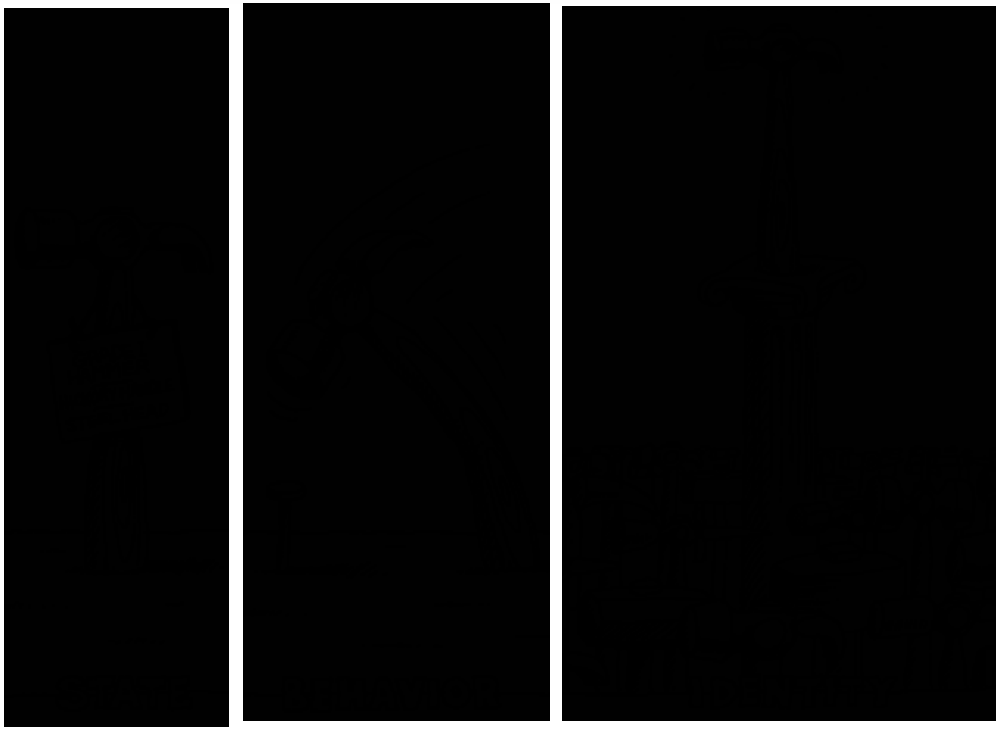


Lecture 1 / Part 3:
OO Fundamentals

Objects

- Object is the smallest unit combining (encapsulating) data and functions and instantiating classes.
- Classes represent static view (design-time entities), while objects represent dynamic view (run-time entities)
- Objects store data in field behind the “layer” of functions (operations).
- Concrete data (values of fields) define object state.
- Methods define behavior.





Objects cooperation

Structured program:

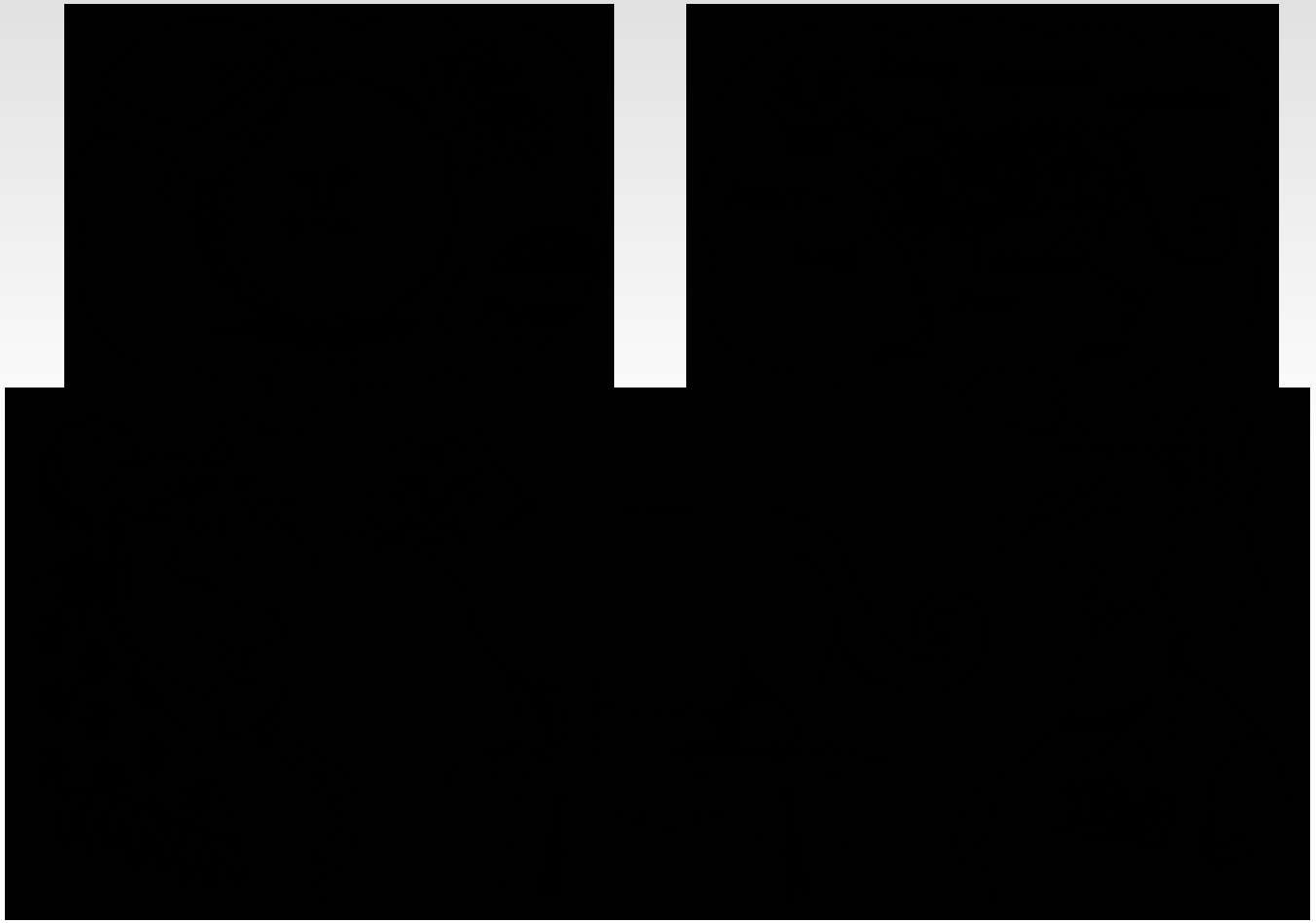
- Code of procedures is allocated in memory before the program is executed. Procedures then read/write data and calls other procedures.

OO program:

- Objects are instantiated and removed dynamically. Initial object which is instantiated by OS or interpreter is responsible for the instantiation of other objects.
  - Nodes of invocation tree are dynamically allocated and removed.
- Objects cooperate in order to successfully respond to method invocation.
  - Methods/objects can instantiate other objects.
  - Methods typically send messages to other objects by calling their methods and waiting for response.
- Data and responsibilities are distributed among objects.

Abstraction

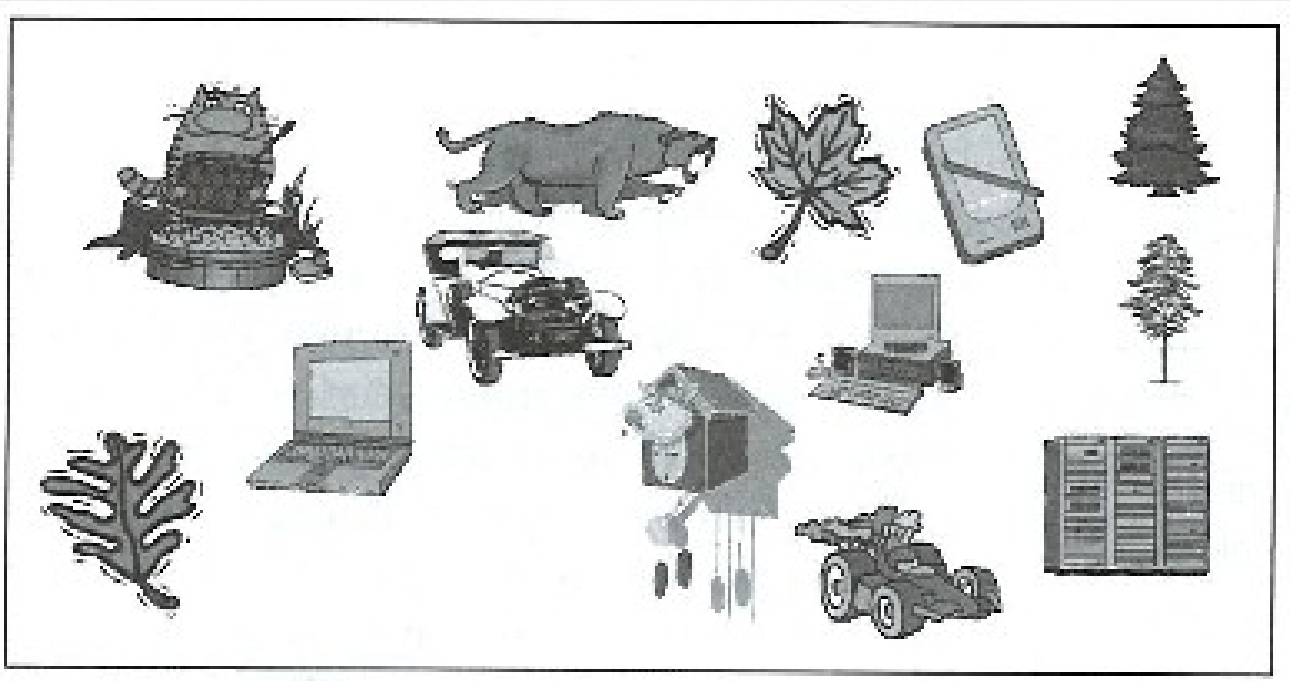
Abstraction = Classification of objects and classes



Proposal of suitable classification scheme is the key task for object-oriented analysis and design

Abstraction (cont.)

- Proposal of suitable classification scheme is the key task for object-oriented analysis and design.
- How many classes do you see in the picture?

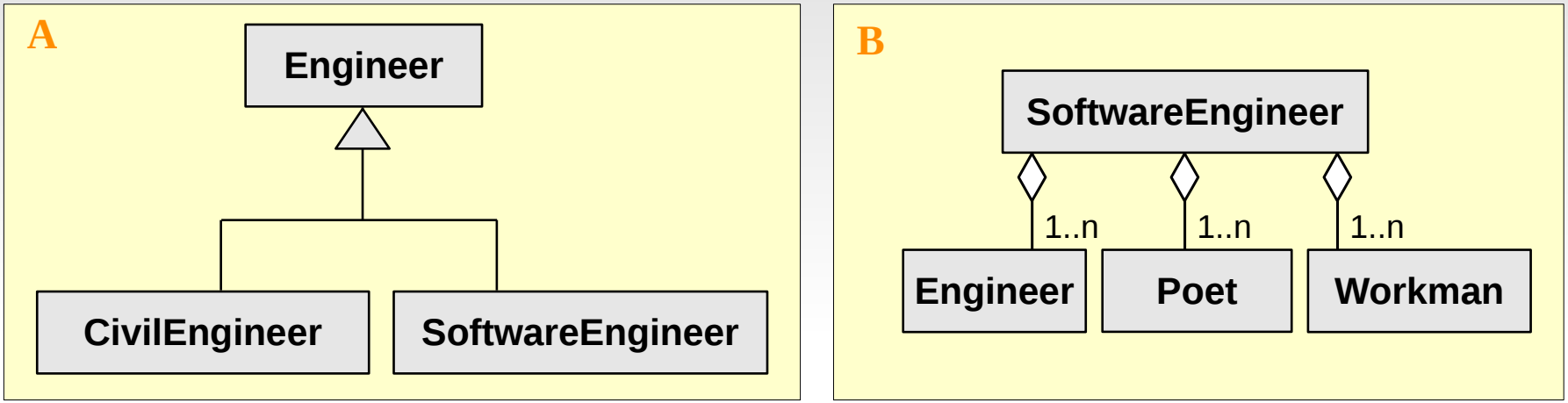


- Trees, leaves, ...
- Electronic devices vs mobile devices
- Cats vs fast moving objects – how to classify the lion?
- ...

Inheritance vs association

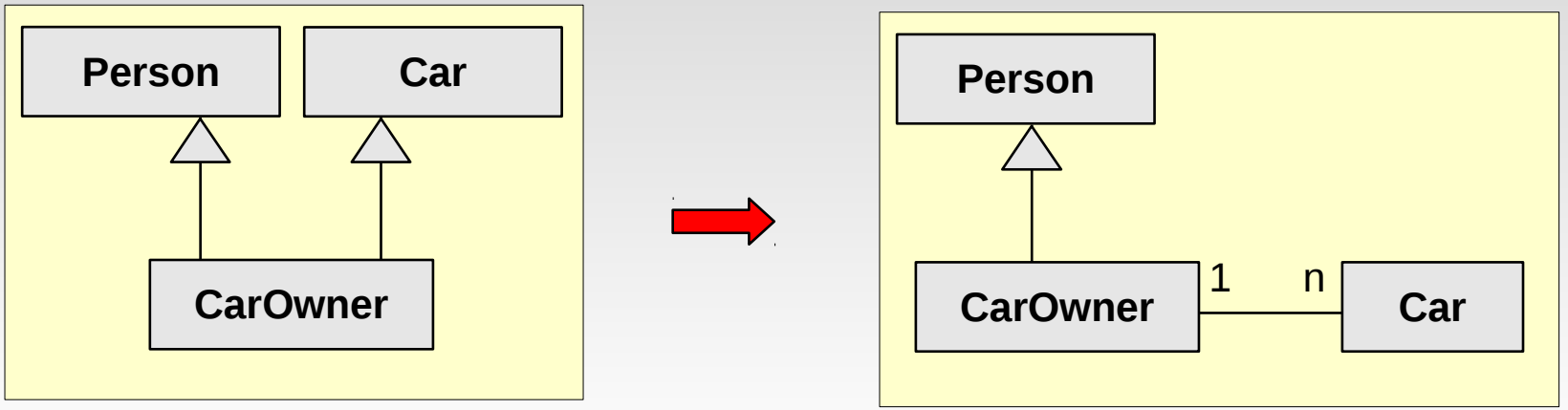
A. Every software engineer is engineer

B. Every software engineer has engineering skills



- Inheritance can be always replaced by association.
  - New trend in higher (component) level is dependency injection
  - Liskov substitution principle
- Association is more flexible because links are created at run-time.
- Never use inheritance if object's role can vary in time, e.g. one day the SW engineering is rather poet while another day he/she is rather Workman
  - **Objects can never change affiliated class** (i.e. the type) during their life time!

Object role



Every object instantiating sub-class must be **always** usable in the context of its super-class(es)

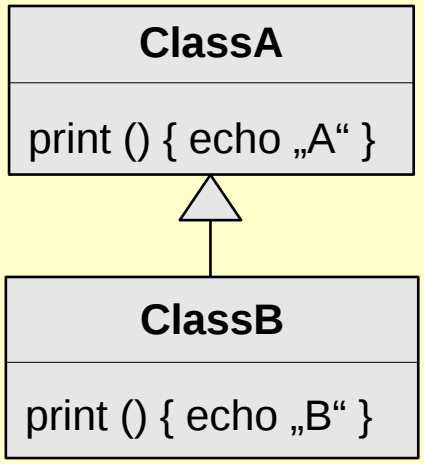
Q: Is CarOwner always Person?
Q: Is CarOwner always Car?

Polymorphism

- Concept from the theory of types
  - „+“ means the same for real as well as for integer
  - „+“ is has different implementation (behavior) for real and integer
- Polymorphism is a product of inheritance and dynamic connection
  - Sub-class inherits name of the method
  - Biding the method name with its implementation is accomplished at runtime.

Q: What is the output of the following code?

```
ClassA object;
object = new ClassB();
object.print();
```

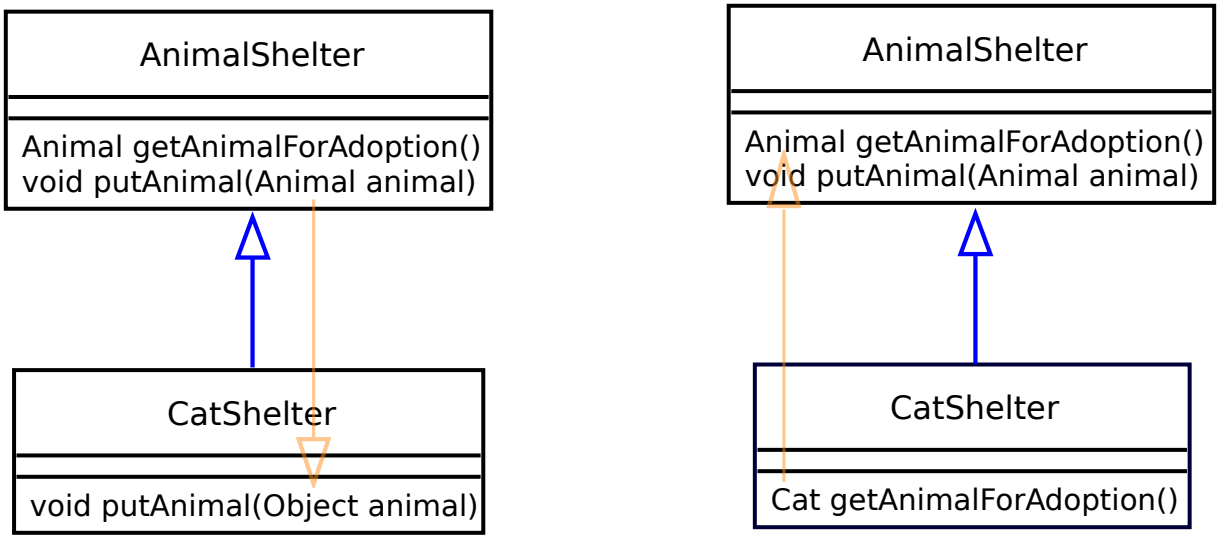


Liskov Substitution Principle

- If *S* is a subtype of *T*, then objects of type *T* in a program may be replaced with objects of type *S* **without altering any of the desirable properties of that program**.
- Requirements on method signatures
  - Usually restricted directly by OO programming language.
- Behavioral conditions of subtypes
  - Their satisfaction depends on the designer/programmer

Liskov Substitution Principle – signatures

- Requirements on method signatures:
  - Contravariance of method arguments in the subtype.
  - Covariance of return types in the subtype.
  - No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.



Liskov Substitution Principle

- Behavioral conditions of subtypes:
- … will be discussed in detail during the „interface as contract“ lesson.
  - Preconditions cannot be strengthened in a subtype.
  - Postconditions cannot be weakened in a subtype.
  - Invariants of the supertype must be preserved in a subtype.
  - History constraint (the "history rule"). Objects are regarded as being modifiable only through their methods (encapsulation). Since subtypes may introduce methods that are not present in the supertype, the introduction of these methods may allow state changes in the subtype that are not permissible in the supertype. The history constraint prohibits this.
    - **Violation example:** *Square* inheriting from *Rectangle* with *height* and *width* setters. If a *Square* object is used in a context where a *Rectangle* is expected, unexpected behavior may occur because the dimensions of a *Square* cannot (or rather should not) be modified independently.
    - Note that if *Square* and *Rectangle* had only getter methods (i.e., they were immutable objects), then no violation of LSP could occur.

The Open Closed Principle

- Software entities like classes, modules and functions should be open for extension but closed for modifications.
- Adding new functionality would involve minimal changes to existing code.
- Most changes will be handled as new methods and new classes.
- Designs following this principle would result in resilient code which does not break on addition of new functionality.

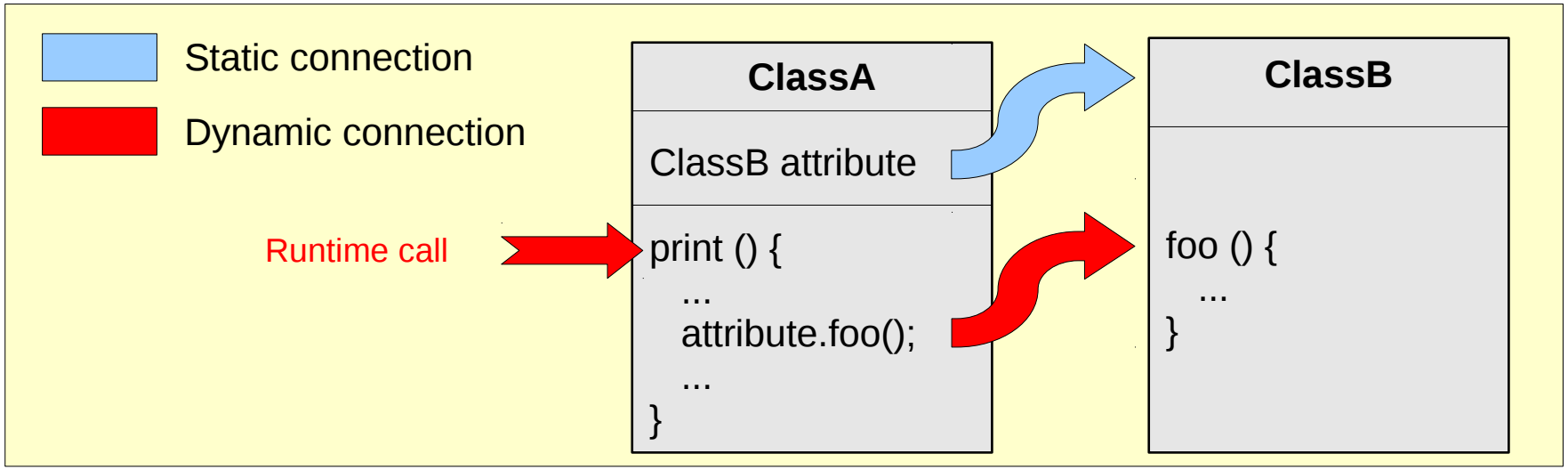
```
class ResourceAllocator {

    public int allocate(int resourceType) {
       int resourceId;

       switch (resourceType)  {
       case TIME_SLOT:
           // do something
           break;
       case SPACE_SLOT:
        // do something else
           break;
      default:
           // do something
           break;
       }
    }

}
```

Types of object connections

Object Connection - physical or conceptual link between objects. Denotes the possibility of (client) object to use services of another (server, supplying) object or to navigate the object.

When the connection is established:

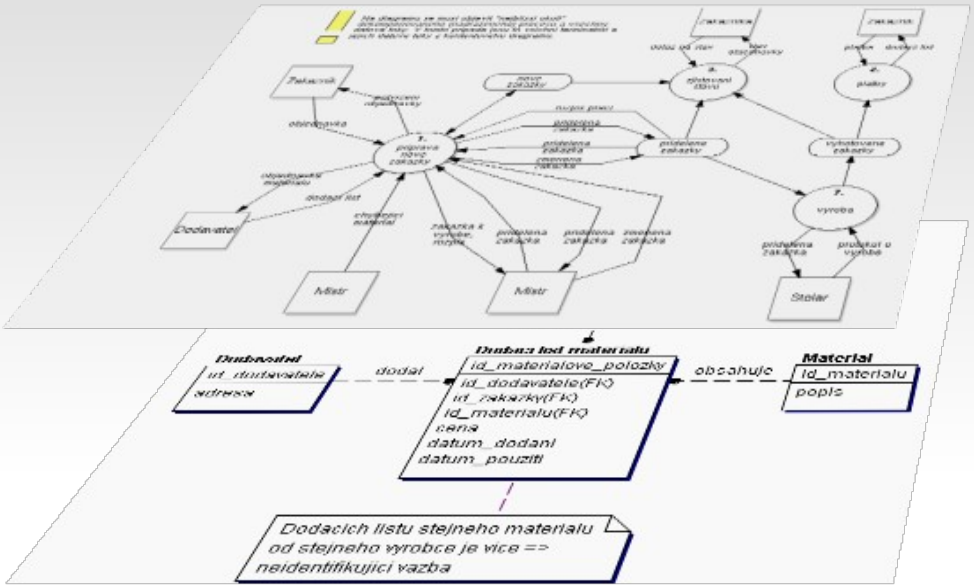
- At design time – „really“ static connection
  - Embedded classes, inheritance,
- At compile time – static connection
  - Association, aggregation, composition
- At runtime – dynamic connection
  - Dependency
  - Methods call

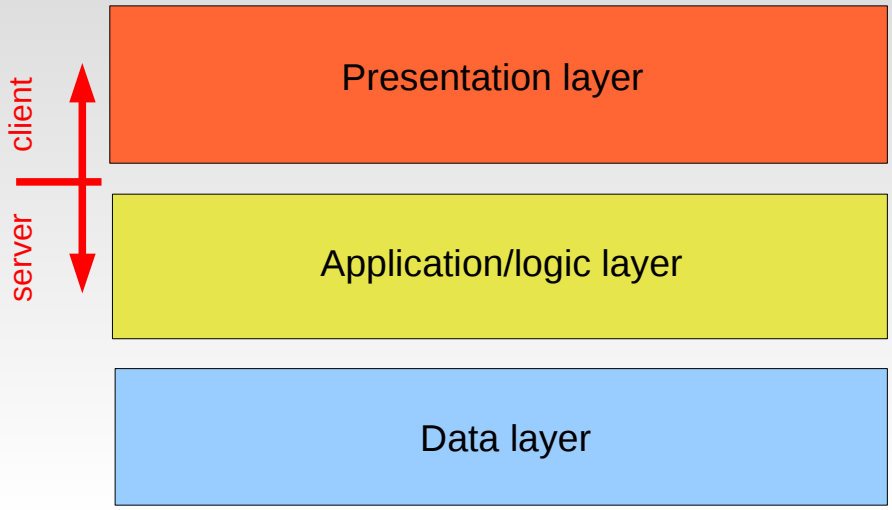


Lecture 1 / Part 4:

Software Architectures – Key Concepts

Multi-layered Architecture (I)

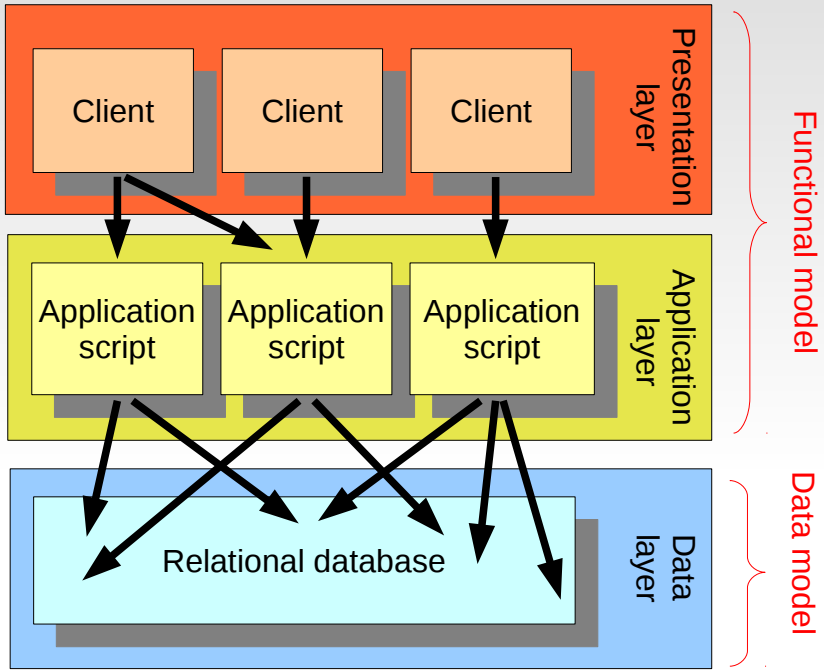




- Based on structured models
- Suitable for client-server applications
- **Multi-layered vs multi-tier Architecture:** The concepts of **layers** and **tiers** are often used interchangeably. However, one fairly common point of view is that there is indeed a difference, and that a layer is a logical structuring mechanism for the elements that make up your software solution, while a tier is a physical structuring mechanism for the system infrastructure

Multi-layered Architecture (cont.)

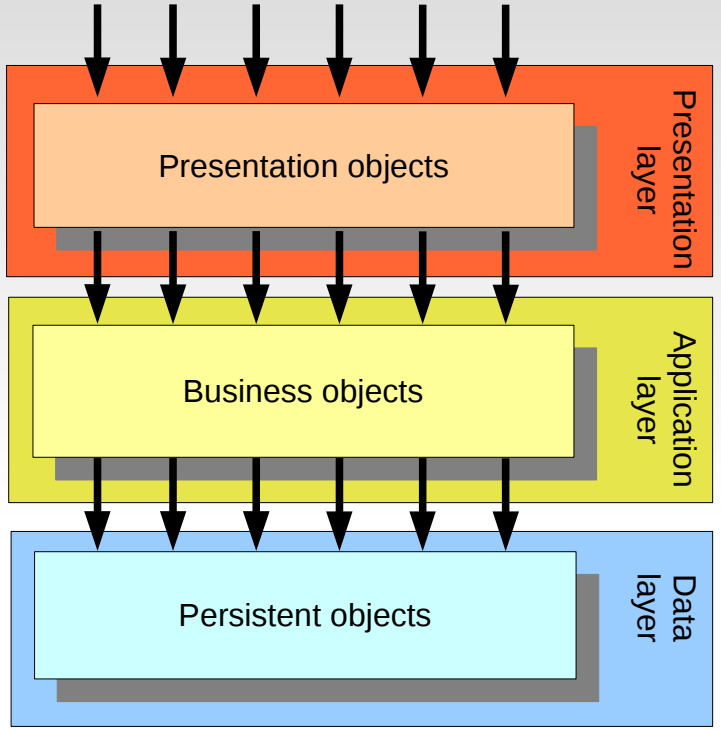
- **Typical features:**
  - Strong dependences in DB
  - Communication through DB
  - Autonomous clients
  - Complex SQL queries
- **Realization:**
  - Forms (HTML, XML, CSS, ...)
  - Scripting (PHP, ASP, ...)
  - Relational databases
- **Common use:**
  - PHP-based web pages
  - Client-server applications



Interconnection through data layer:
+ rapid implementation
+ utilization of known development processes
+ proven technologies
- single table is handled by multiple scripts
- complex database scheme
- poor scalability

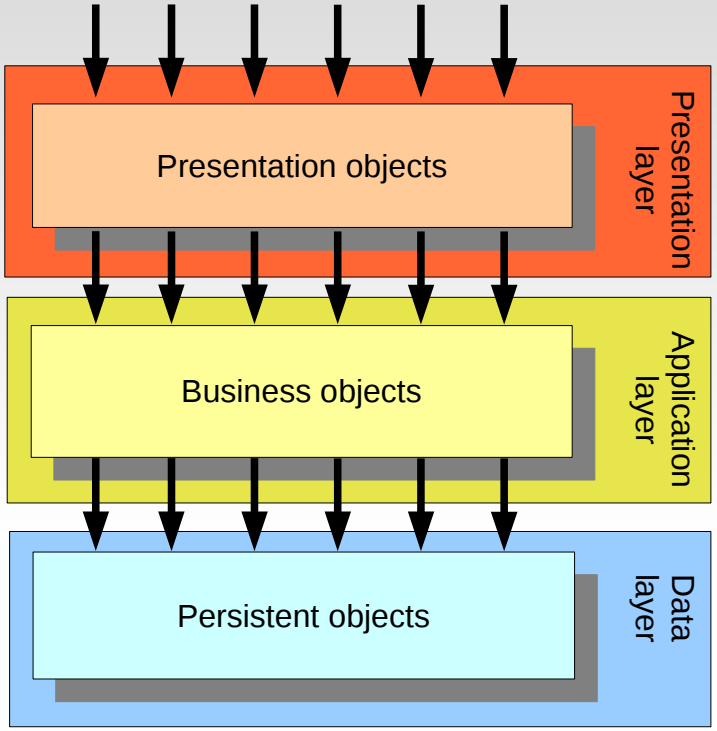
Multi-layered Architecture and OO Design

- Naive adoption of multi-layered architecture to OO design
- Interaction
  - Upper layers play the role of clients to their lower layers
  - Lower layers play the role of severs to their upper layers
  - **Object should not depend on objects from upper layers**



Multi-layered Architecture and OO Design (cont.)

- Presentation objects
  - User input/output
- Business objects
  - Forms conceptual structure of the system
    - Independent from presentation
    - Independent from data store
- Persistent objects
  - Forms persistent layer of the system
    - Data storage and their accessibility
    - Locking
    - Integrity checking

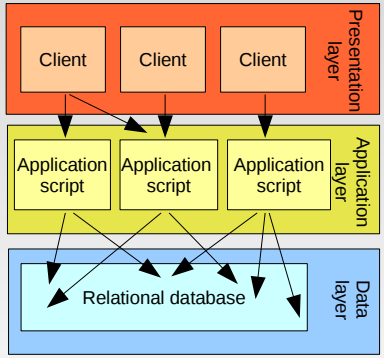


- **Q: Where to verify the input data?**

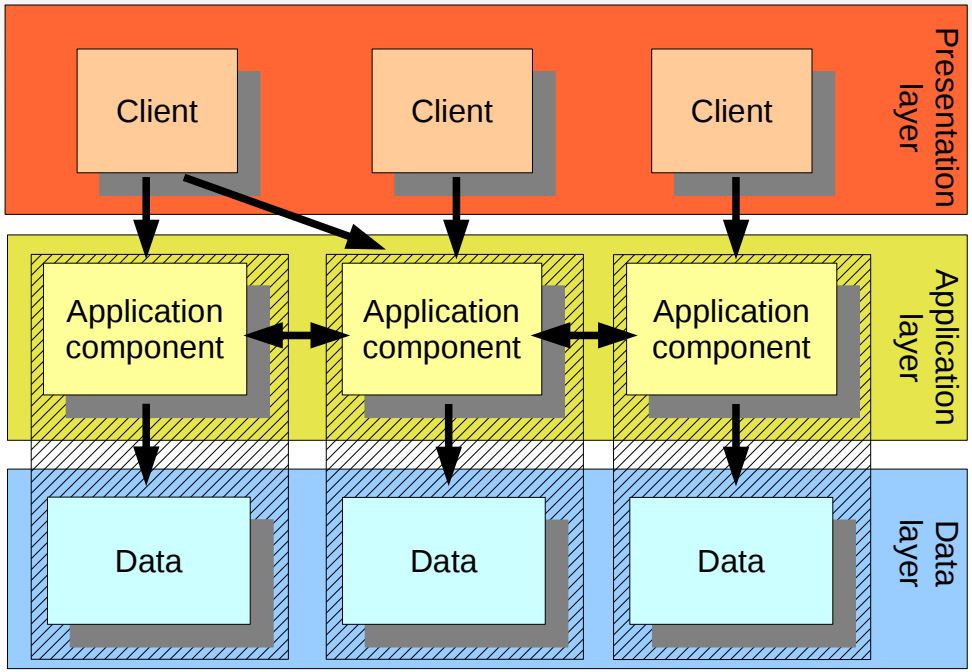
1) In the presentation layer, application layer just handles the data by passing them to data layer.
   - Data verification is not typical responsibility of presentation objects.
   - Duplication of the verification code across many presentation objects.
   - Application layer relay on valid data => is dependent on presentation layer
2) In the application layer, presentation layer just reads the input and show results.
   - Intensive client-server communication, slow response

Multi-layered Architecture and Components

- Utilization of component technologies
  - Interconnection through the application layer
  - Handling complexity of connection via components
- Typical feature:
  - Data separation
  - Context management in the application layer
- Realization:
  - CORBA, DCOM, SOAP/XML



Software Architectures



Interconnection through application layer:

+ robustness and scalability
+ maintenance and extensions
+ parallel development
+ easy integration with other systems
- complex application layer
- require modern approaches for development and management
- it's not feasible to utilize advanced features of modern relational databases

What is not multi-layered Architecture

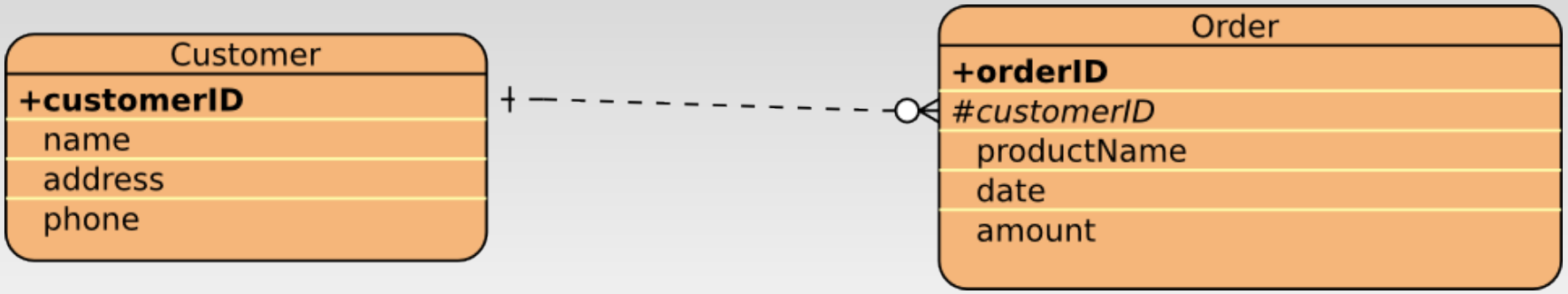
Presentation, application and data logic in single class

ORM: Object-Relational Mapping

- In the real world, the OO software is often combined with relational databases
  - **Relational databases** present proven, tuned and highly optimized technology (efficiency, scalability, data integrity, etc.)
  - => It's necessary to map object model to entity-relational model
  - => Object-Relational Mapping, ORM
    - Java Persistence API, Hibernate, …

- Note1: Although **object databases** exists a long time, they still play a minority role.
- Note2: **NoSQL databases** represent a new trend in dynamic data storage, e.g. in facebook and other social sites. They have no fixed relational scheme. Instead, they the information scheme is based on ontologies (SQRL, OWL, ...). Query languages, e.g. SparQL, SQWRL enables to query data and also support automatic inference.

ORM: Tables vs. objects (I)

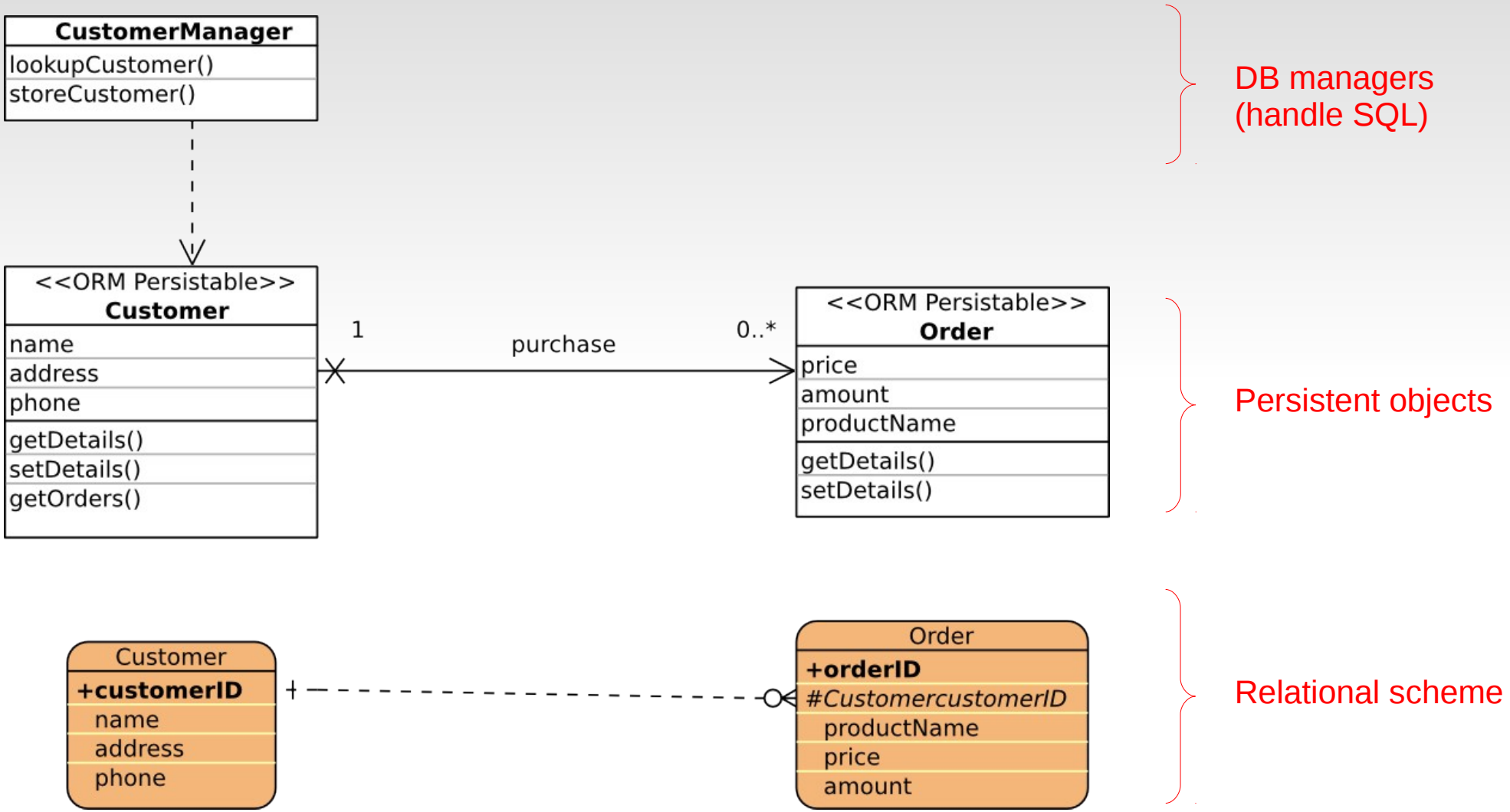


- Relational Technology
  - Data are stored in tables
  - Rows represent records, columns represent values of concrete types
  - Tables are connected by relations
  - Primary/foreign keys
  - Cardinality of relations
  - Relational algebra and SQL for data retrieval
- OO technology
  - Classes contain data as well as operations
  - Associations with cardinality
  - Inheritance
  - Associations and objects are in memory => data manipulation is based on object interaction.
  - Ex.: get all students enrolled in given course – difference between SQL and object interaction

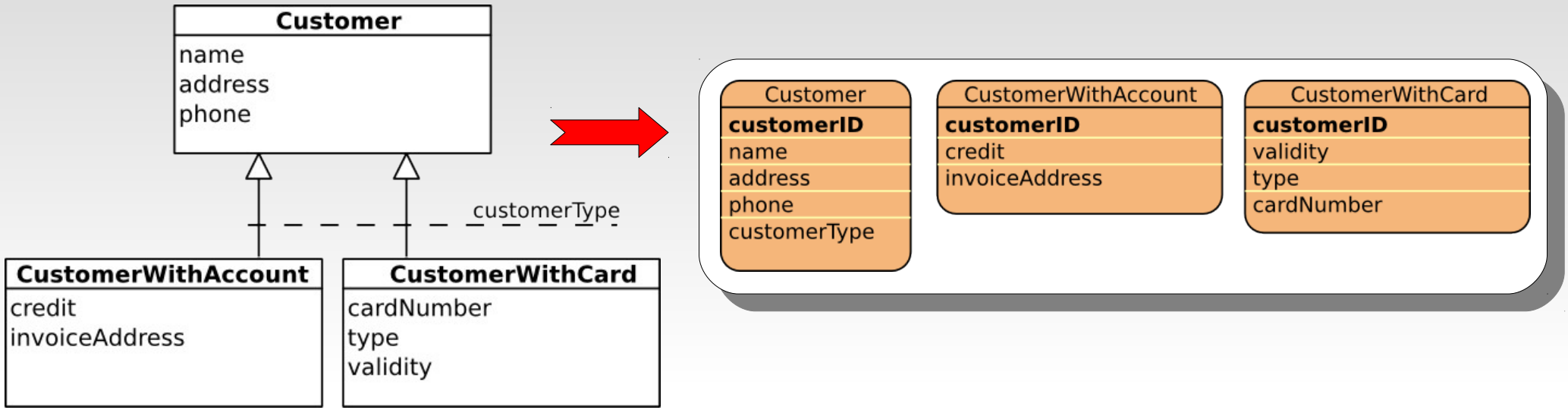
ORM: The Basic Mapping Principles

- Persistence class = entity set (table)
- Object = entity (record, line in the table)
- Primitive class attribute = entity attribute (column in the table)
- Key is selected from primitive attributes or is created a new one
- Association/aggregation/composition defines relation (interconnection of tables by means of foreign keys)
    - M:N associations must be decomposed
- Mapping of class inheritance:
    - 1:1 mapping
    - Combining to super-class
    - Splitting to sub-classes

ORM: class diagram vs. ER diagram

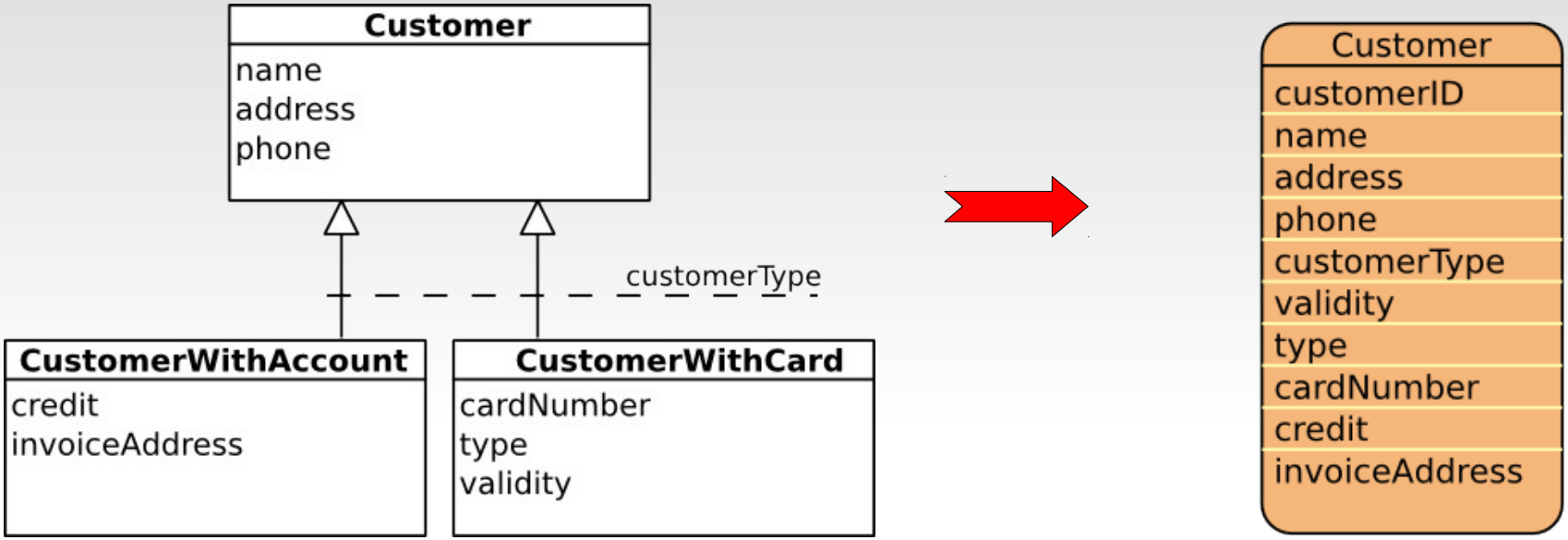


Inheritance mapping: 1:1



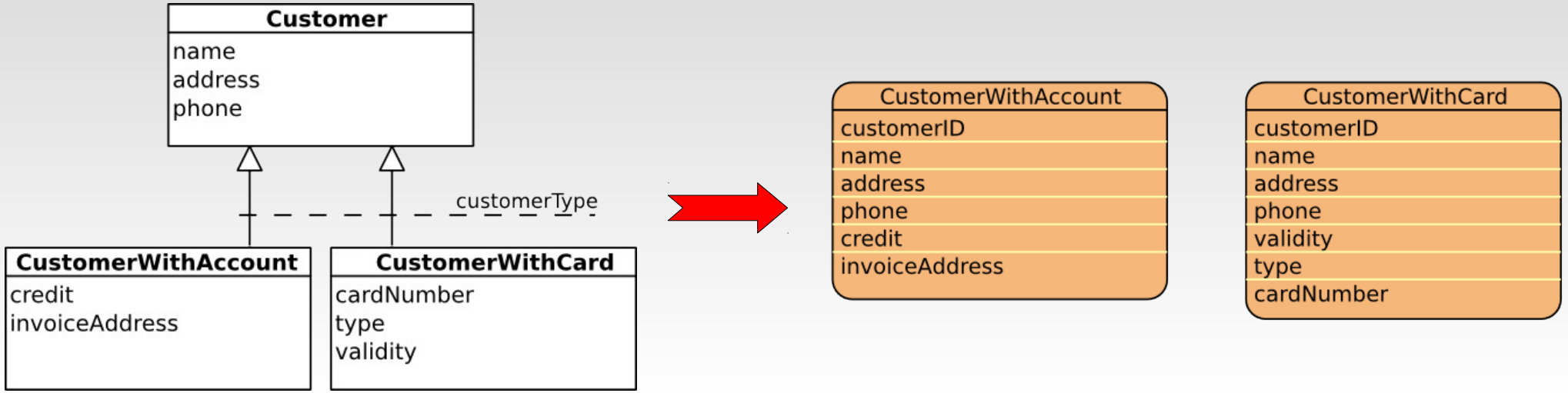
- Every class becomes a table
- All tables share the primary key
- Discriminator becomes an attribute
  - Queries search in the table of the concrete sub-class and its super-class
- Data of single instance is stored in multiple tables
  - Complex data retrieval

Inher. mapping: combining to super-class



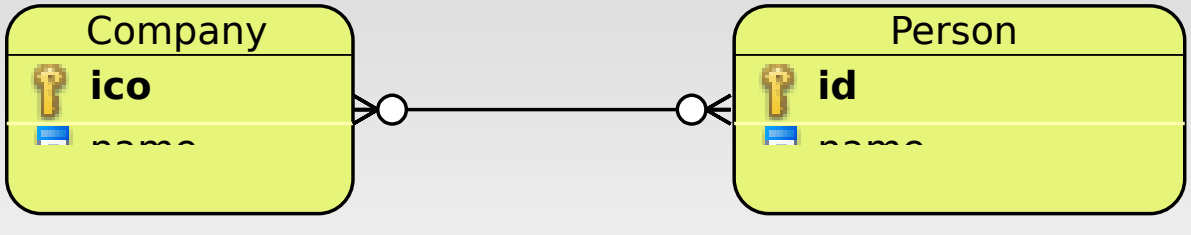
- Attributes of all sub-classes are stored in single table
- Some attributes can by NULL
  - 4NF violation
- Suitable for class hierarchies with few sub-classes and few attributes

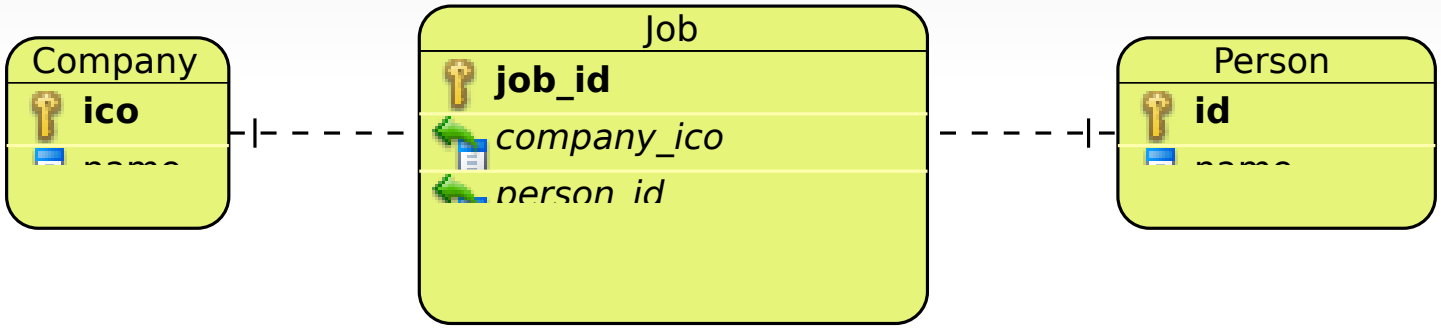
Inheritance mapping: splitting to sub-classes



- Attributes of super-class are duplicated in tables of all (non-abstract) sub-classes.
- Suitable if:
  - Super-class has few attributes
  - There exist a lot of sub-classes (spreading class hierarchy)
  - Sub-classes have a lot of specific attributes

Association vs. entity relation (I)

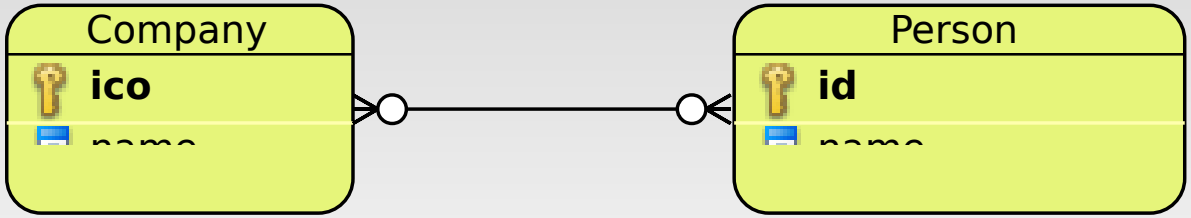





```
SELECT Person.name
FROM Company, Job, Person
WHERE ico = company_ico AND
        id = person_id AND
        Company.name = "MU"
```

```
SELECT Company.name
FROM Company, Job, Person
WHERE ico = company_ico AND
        id = person_id AND
        Person.name = "Radek Oslejsek"
```

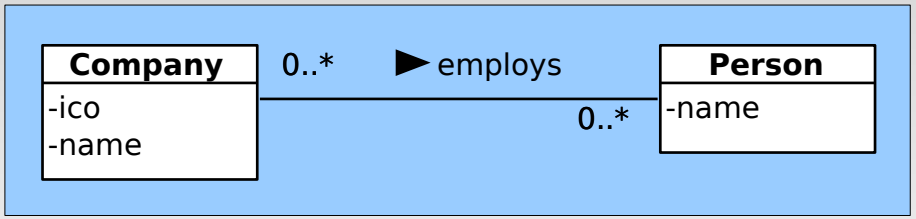
Association vs. entity relation (II)

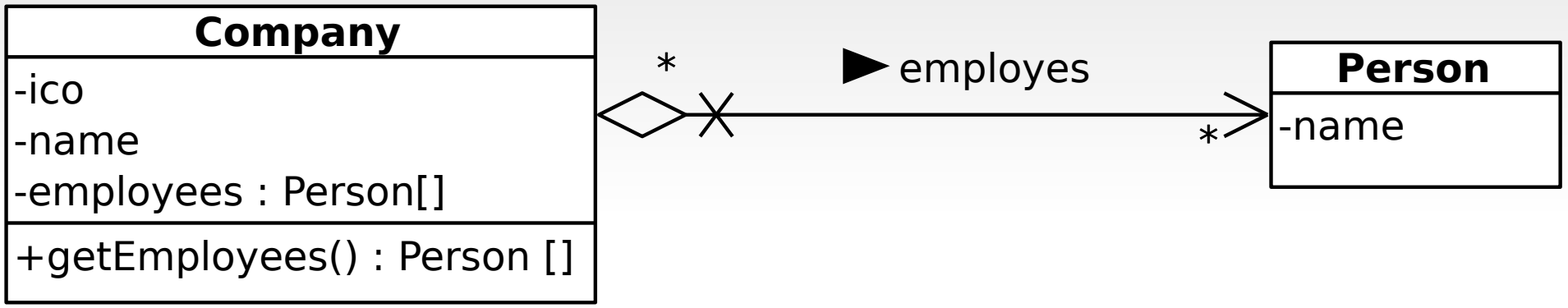


Note (often mistake): The *Person* class has no attribute *id* in the class model !!!

- **Q:** Is this model directly implementable?
- **A:** Yes. As opposed to ER model, M:N relationships pose no problem.
- For example, the *Company* class can include an array of *Persons* and vice versa. On the other hand, there are many ways to elaborate this initial decomposition.

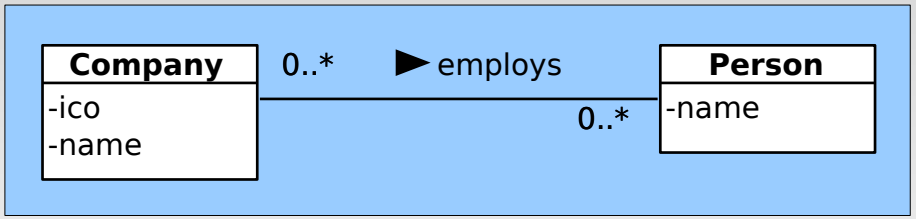
Association vs. entity relation (III)

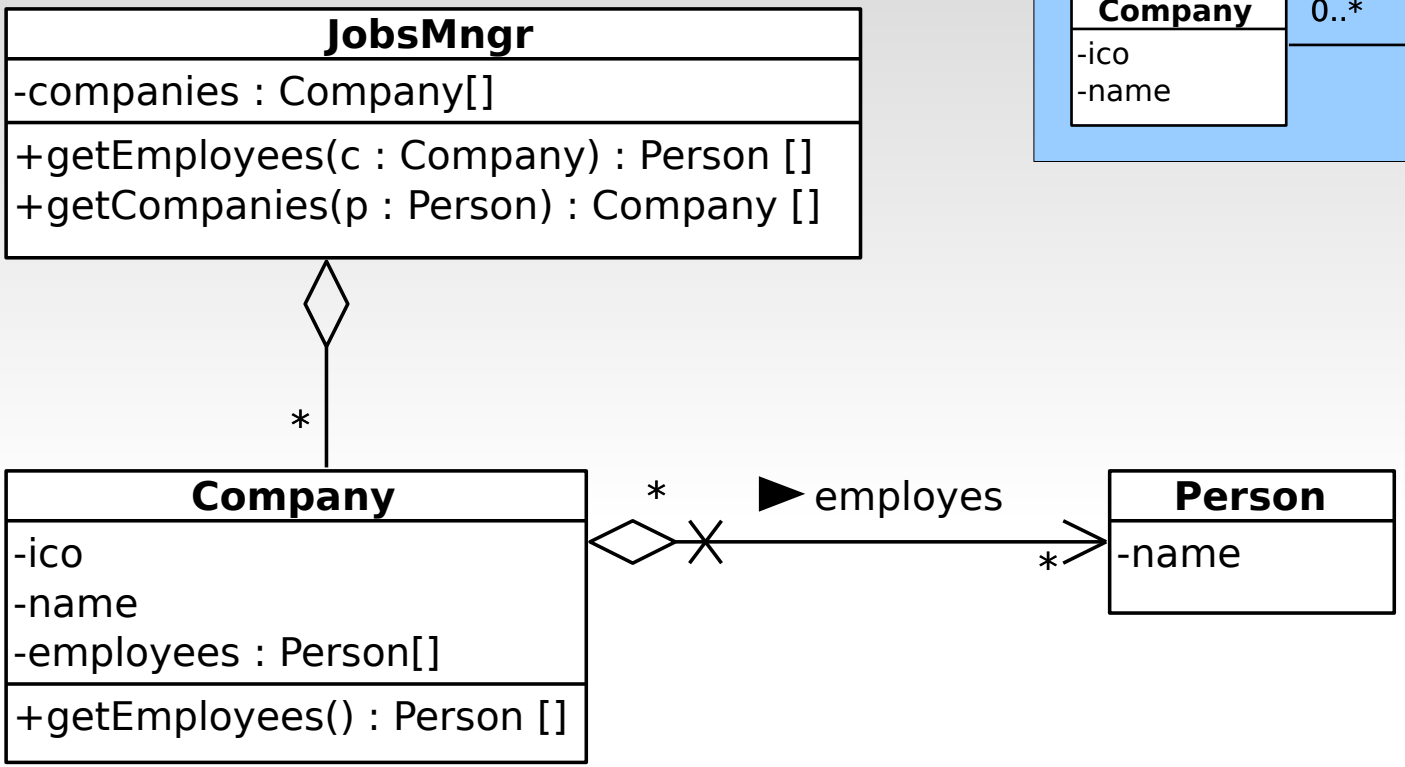




- **Approach 1, model 1:** We prefer one direction
  - Company stores persons (employees) in array
  - Person has no link to its companies
- **Problem:** There are many companies registered in the system. Where they are stored? How we get link to concrete address if we have no query mechanism?

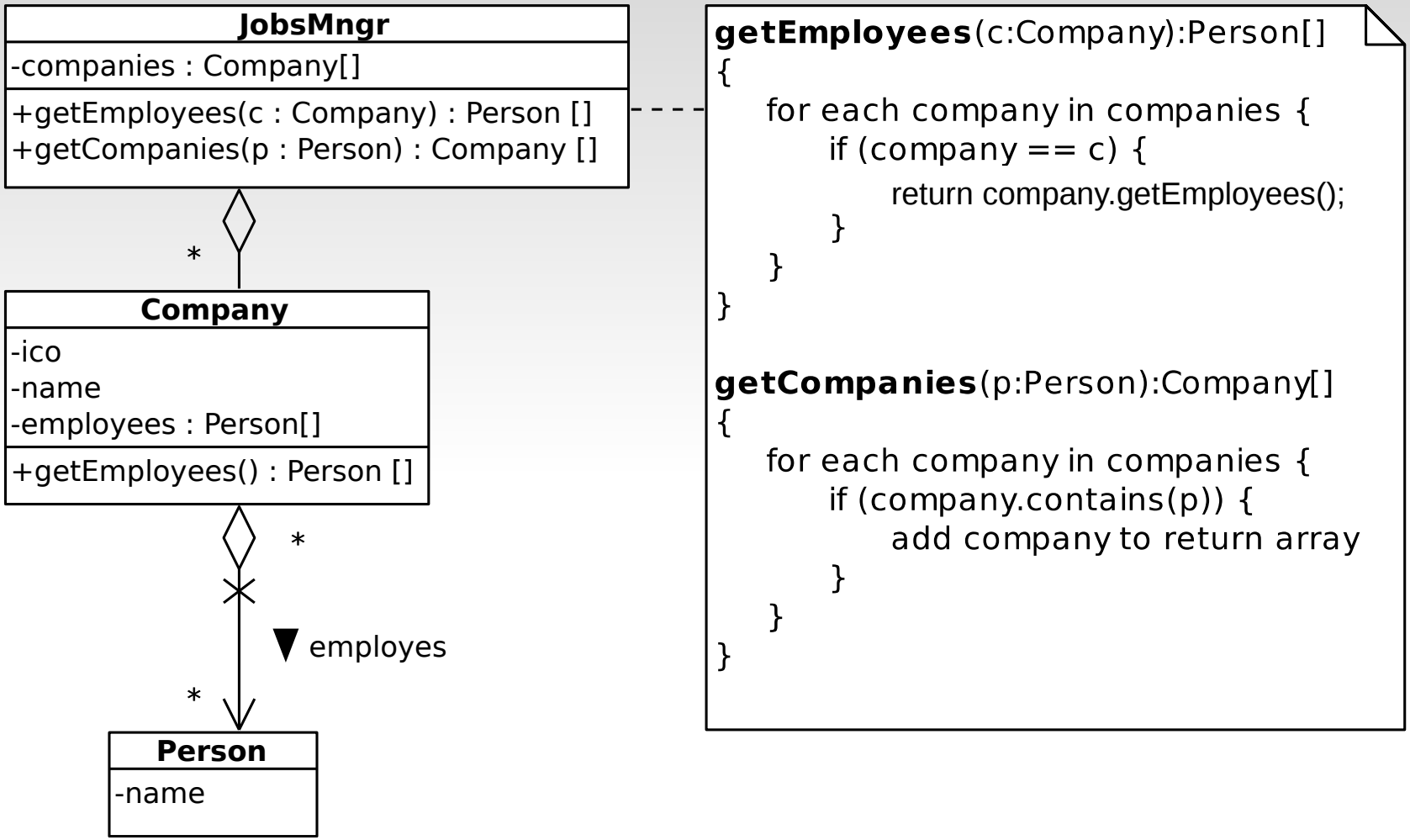
Association vs. entity relation (IV)





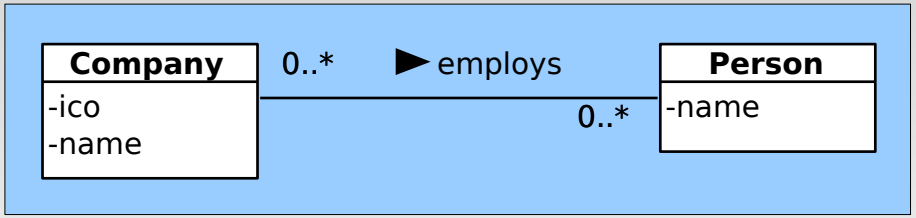
- **Approach 1, model 2:** Single *JobsMngr* stores all the companies and mediates access to companies and their employees.
- **Q:** Is the *getCompanies()* method implementable? How effectively?

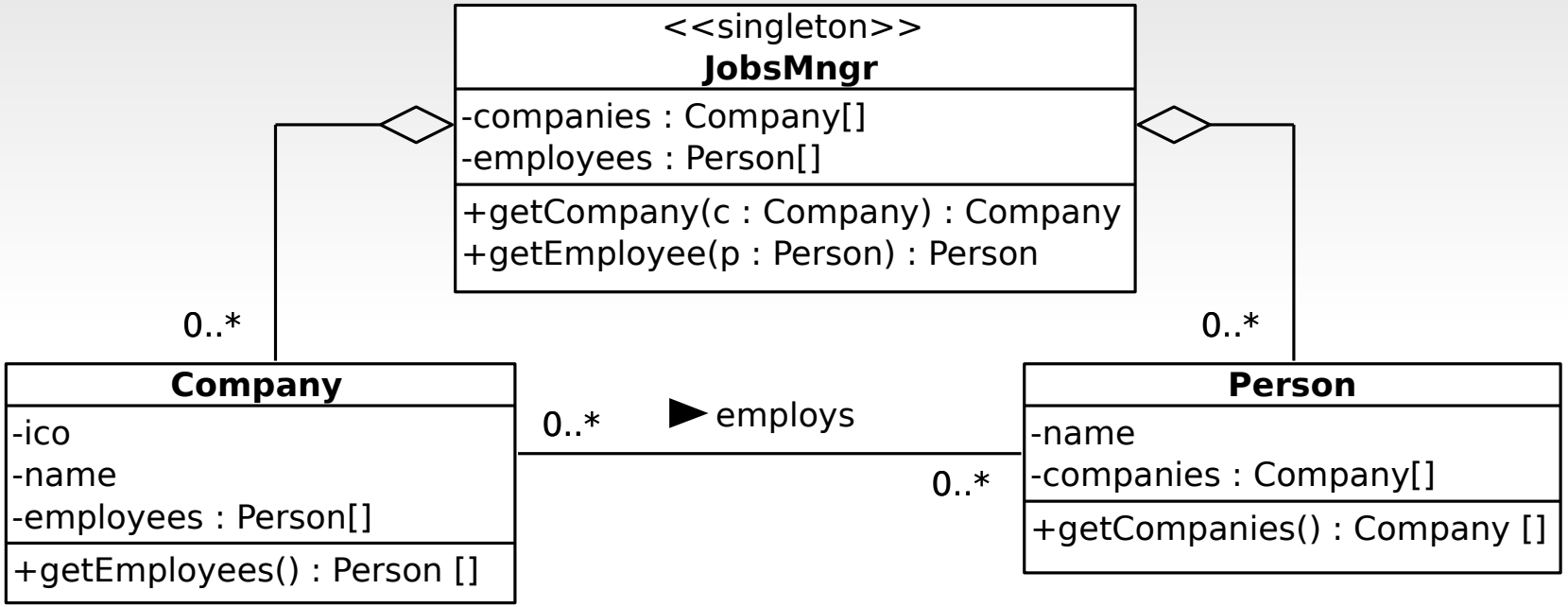
Association vs. entity relation (V)



- **Approach 1, model 2:** *getCompanies()* is less effective /O(n*n)/ than *getEmployess()* /O(n)/. The reason is that each invocation of the *company.contains()* searches in the list of employees.

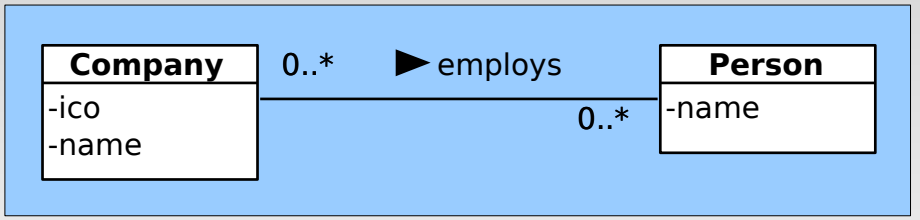
Association vs. entity relation (VI)

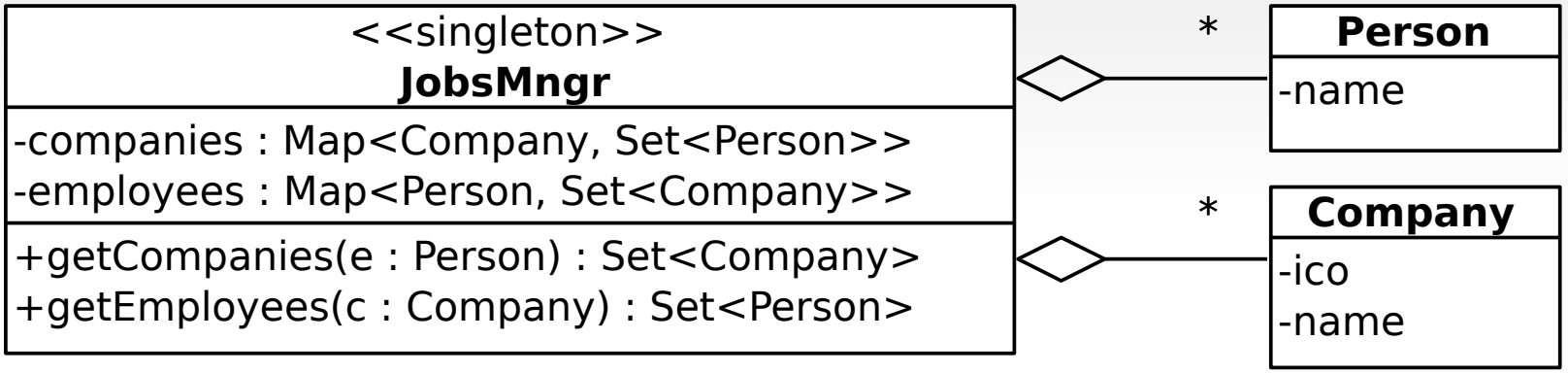




- **Approach 2, model 1:** Bidirectional association
  - **Pros:** Clear responsibilities. Responsibilities are uniformly distributed to all classes
  - **Cons:** Very complicated memory management, especially without automatic “garbage collection“

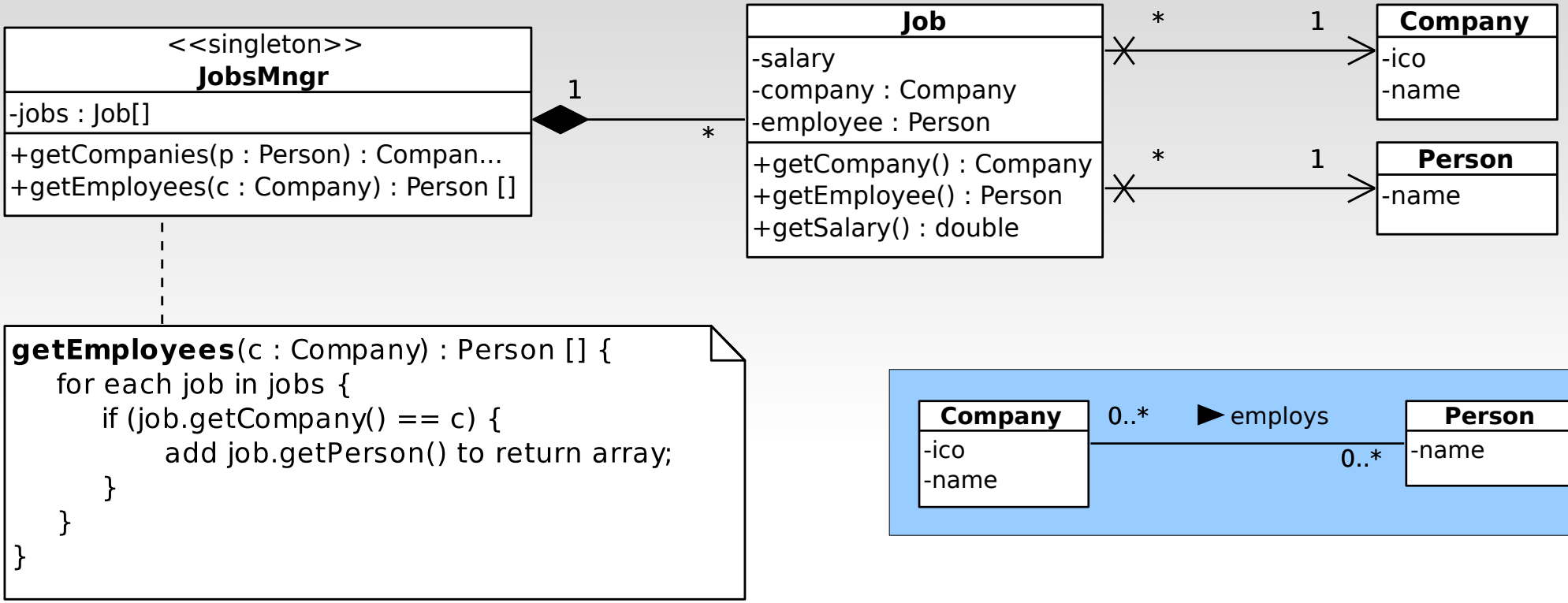
Association vs. entity relation (VII)





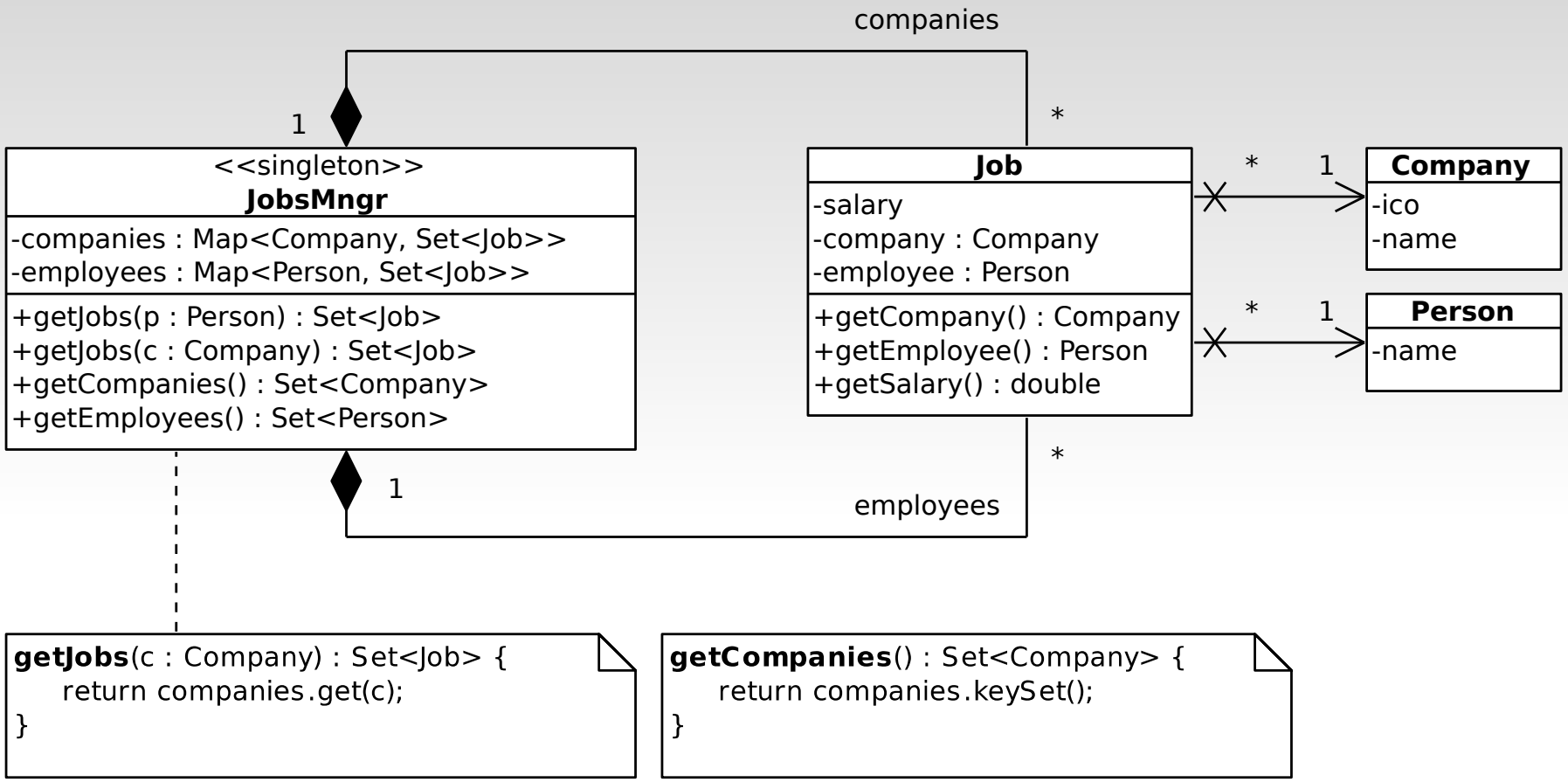
- **Approach 2, model 2:** Preserved bidirectional association, responsibility located in a big “God” object.
  - **Pros:** (a) Management code located in *JobsMngr* => maintainability.
    (b) Efficiency.
- **Q:** Where to store salary?

Association vs. entity relation (VIII)



- **Approach 3, model 1:** Helper class (similar to the association entity in ERD). This class links concrete couple and stores additional data related to the couple.
- **Q:** Putting jobs to list/array is not optimal. Do you know better solution?

Association vs. entity relation (IX)



- **Approach 3, model 2:** Maps provide efficient access to the individual sets of companies and employees as well as to concrete jobs (couples). On the other hand, this solution is unnecessary complicated in many situations.

  => Designer has to choose the best solution for concrete context

  => Design patterns drive the designer

Conclusion:

Entity-relational paradigm is definitely not the same as object-oriented paradigm. Therefore, ER diagrams are definitely not the same as UML class diagrams, although they look similar.

Questions?

Three Engineers

There are three engineers in a car going for a drive. The first is a mechanical engineer, the second an electronics engineer and the third is a software engineer.

Fortunately, the **mechanical engineer** is driving because the brakes fail as they are going downhill. The mechanical engineer eventually brings the car safely to a halt and gets out to examine the hydraulic systems.

The **electronics engineer** gets out and checks the body computer, ABS system and the power train CAN bus.

The **software engineer** stays in the car and when queried about it says that they should all just get back in the car and see if it happens again!