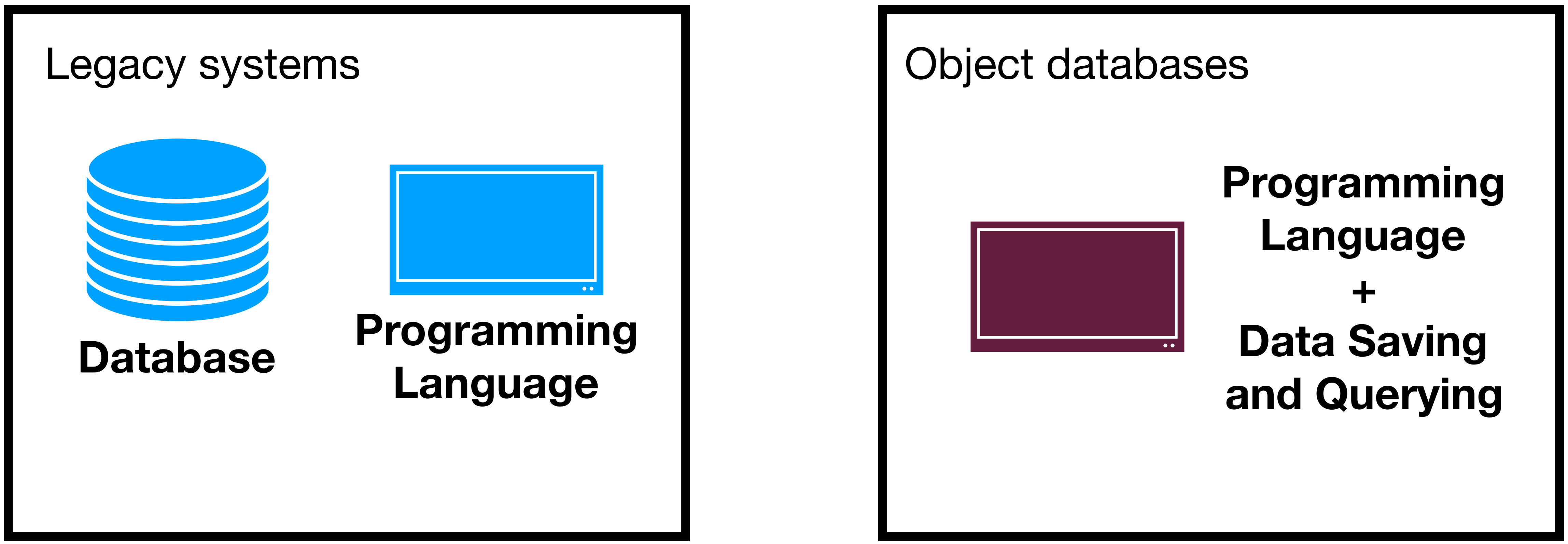
In The Beginning There Was Light

Introduction

Matematikos
ir informatikos
fakultetas

Arūnas Janeliūnas

Object Databases


Legacy systems
Database
Programming Language
Object databases
Programming Language
+
Data Saving and Querying

Programming Language
+
Data Saving and Querying

~~*INSERT*~~

~~*UPDATE*~~

~~*DELETE*~~

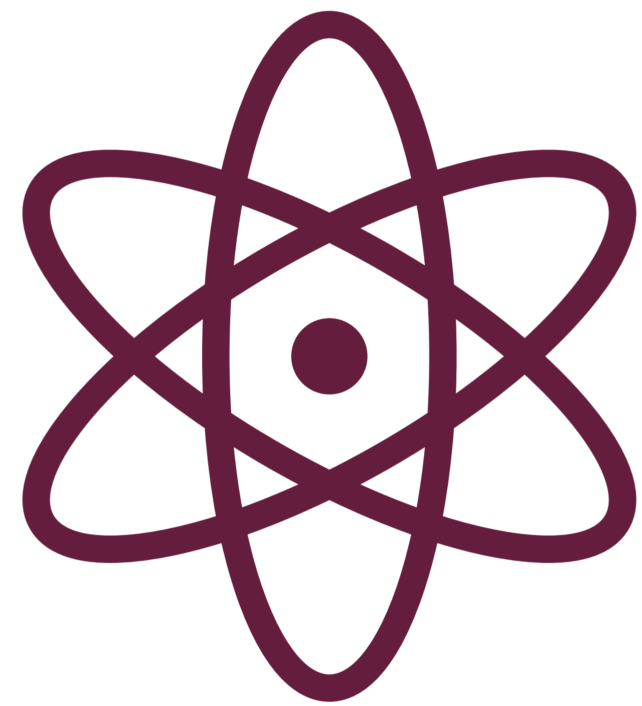
SELECT

The Object-Oriented Database System Manifesto

- *Malcolm Atkinson, University of Glasgow*
- *Francois Bancilhon, Altar*
- *David DeWitt, University of Wisconsin*
- *Klaus Dittrich, University of Zurich*
- *David Maier, Oregon Graduate Center*
- *Stanley, Zdonik Brown University*

Complex objects

Thou shalt support complex objects

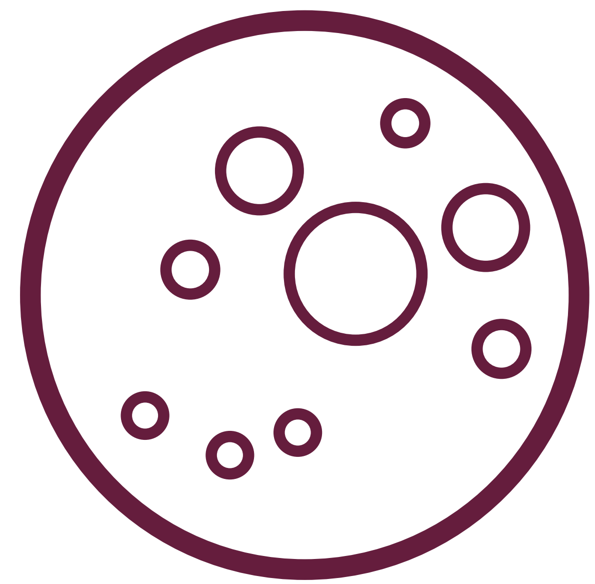


Object identity

Thou shalt support object identity

Encapsulation

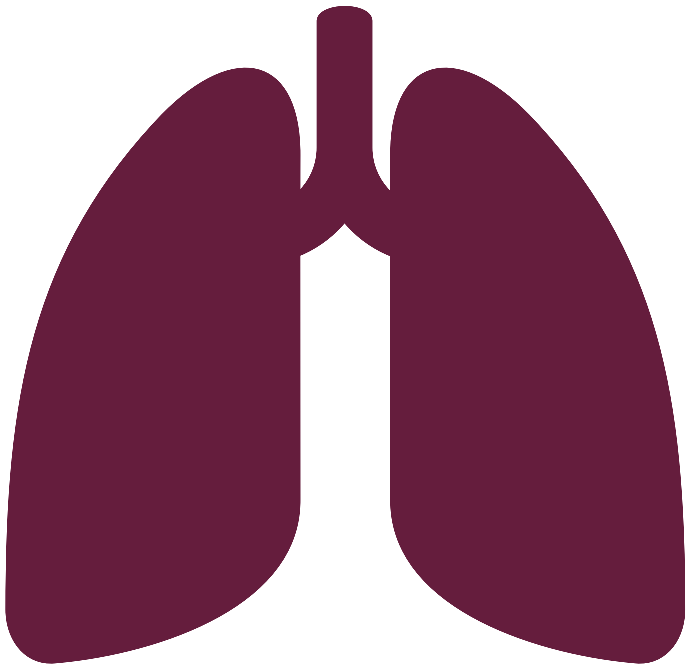
Thou shalt encapsulate thine objects



Arūnas Janeliūnas

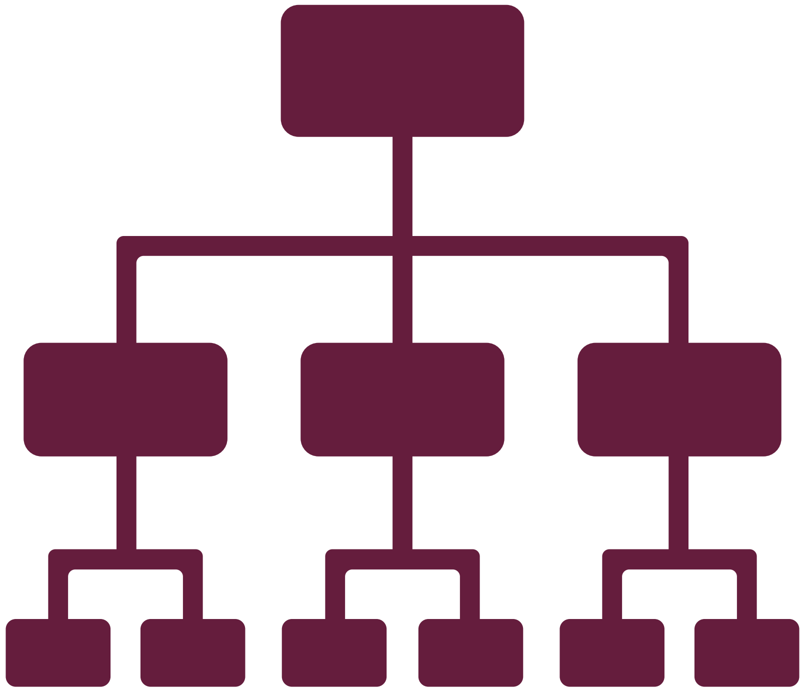
Types and Classes

Thou shalt support types or classes



Class or Type Hierarchies

Thine classes or types shalt inherit from their ancestors



Arūnas Janeliūnas

Overriding, overloading and late binding

Thou shalt not bind prematurely

Computational completeness

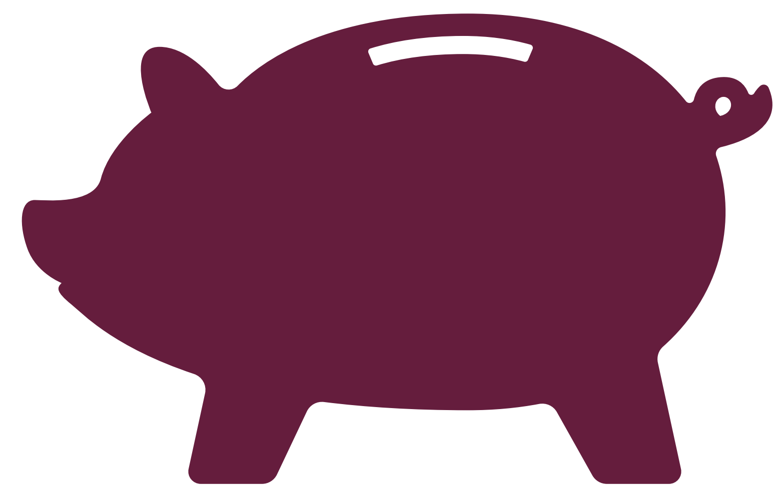
Thou shalt be computationally complete

Extensibility

Thou shalt be extensible

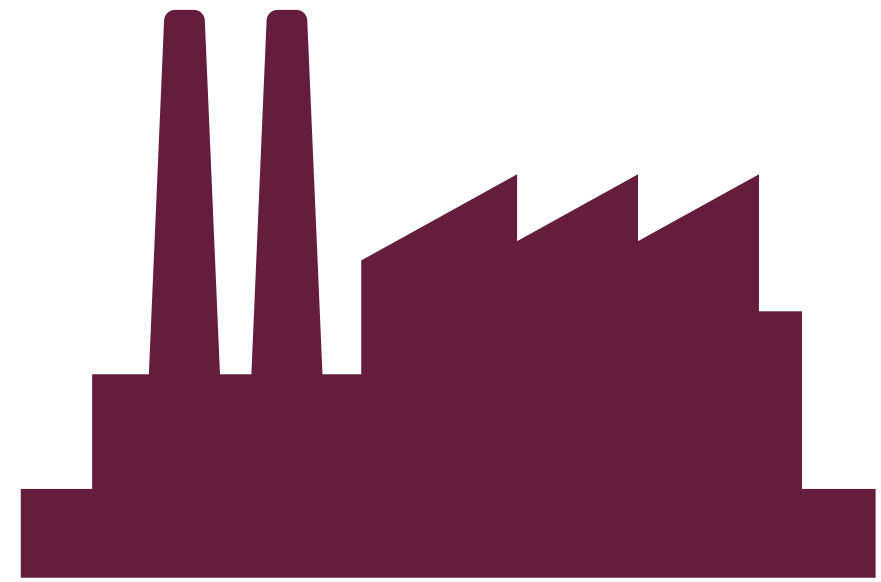
Persistence

Thou shalt remember thy data



Secondary storage management

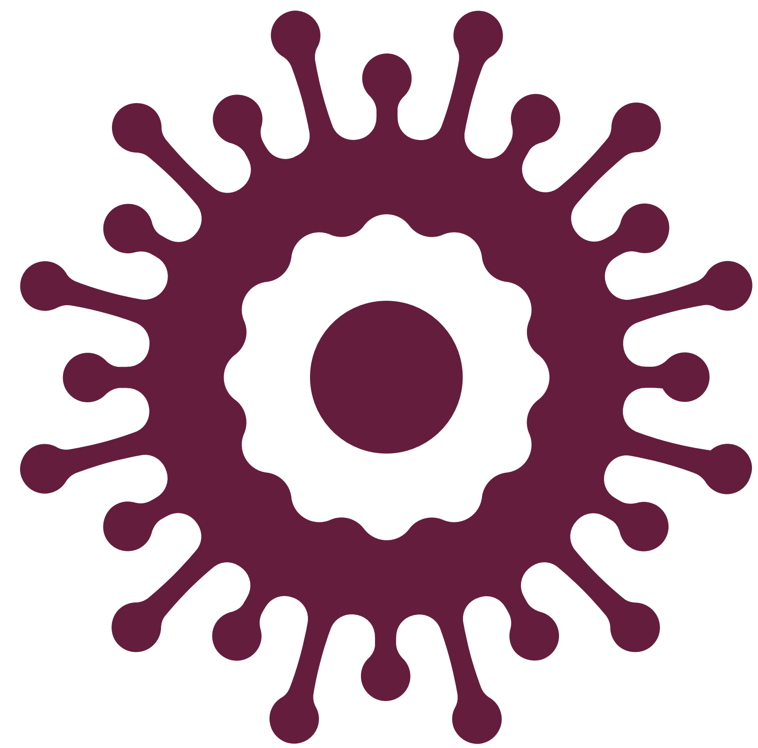
Thou shalt manage very large databases



Matematikos
ir informatikos
fakultetas

Concurrency

Thou shalt accept concurrent users



Recovery

Thou shalt recover from hardware and software failures

Ad Hoc Query Facility

Thou shalt have a simple way of querying data

Optional features: the goodies

Multiple inheritance

Type checking and type inferencing

Distribution

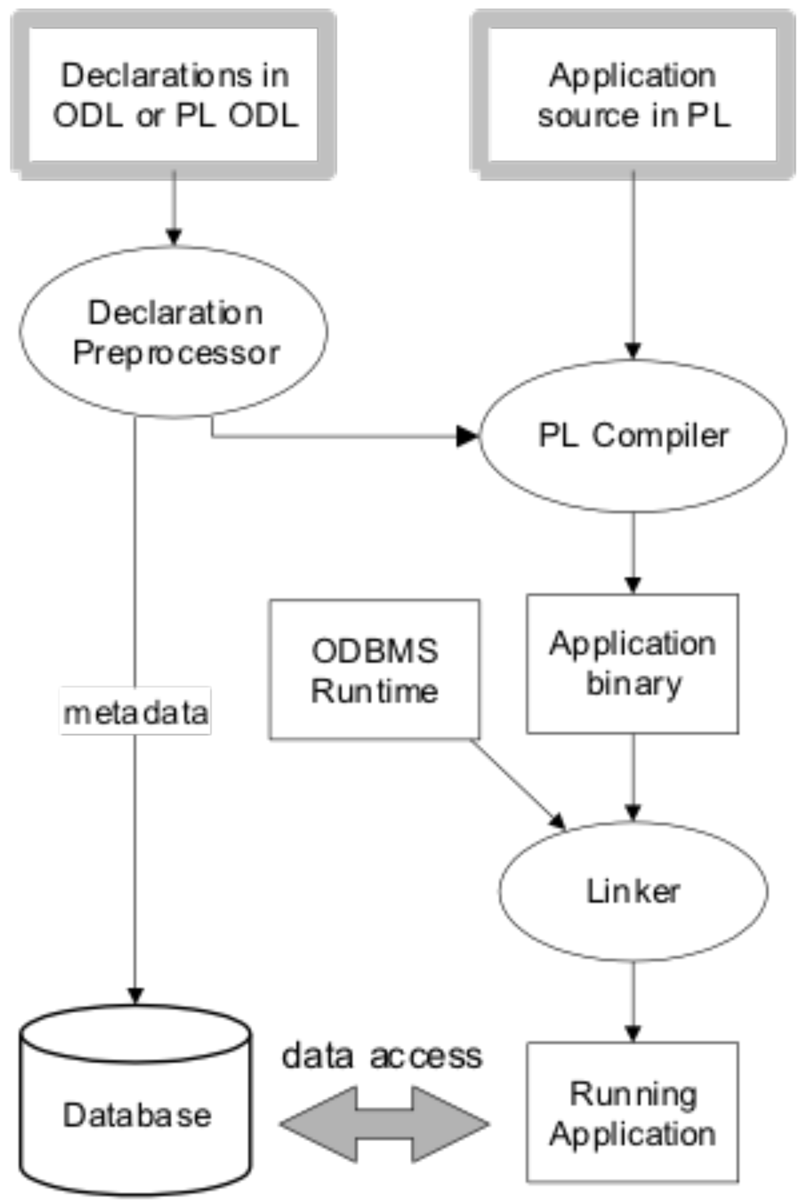
Design transactions

Versions

ODMG Standard

- 1.0 (1993), 2.0 (1997), 3.0 (2004)
- From database API to object storage API
- Main components:
  - Object model (based on OMG model)
  - Object definition language (based on IDL)
  - Object Query Language
- Interfaces to programming languages
  - C++
  - Java
  - SmallTalk
- Appendixes
  - OMG data model vs. ODMG data model
  - Interface to OMG ORB

ODMG Standard



Semantic Models

Before the object data model…

Hierarchical
data model

Graph
data model

Relational
data model

Semantic
data model

Object
data model

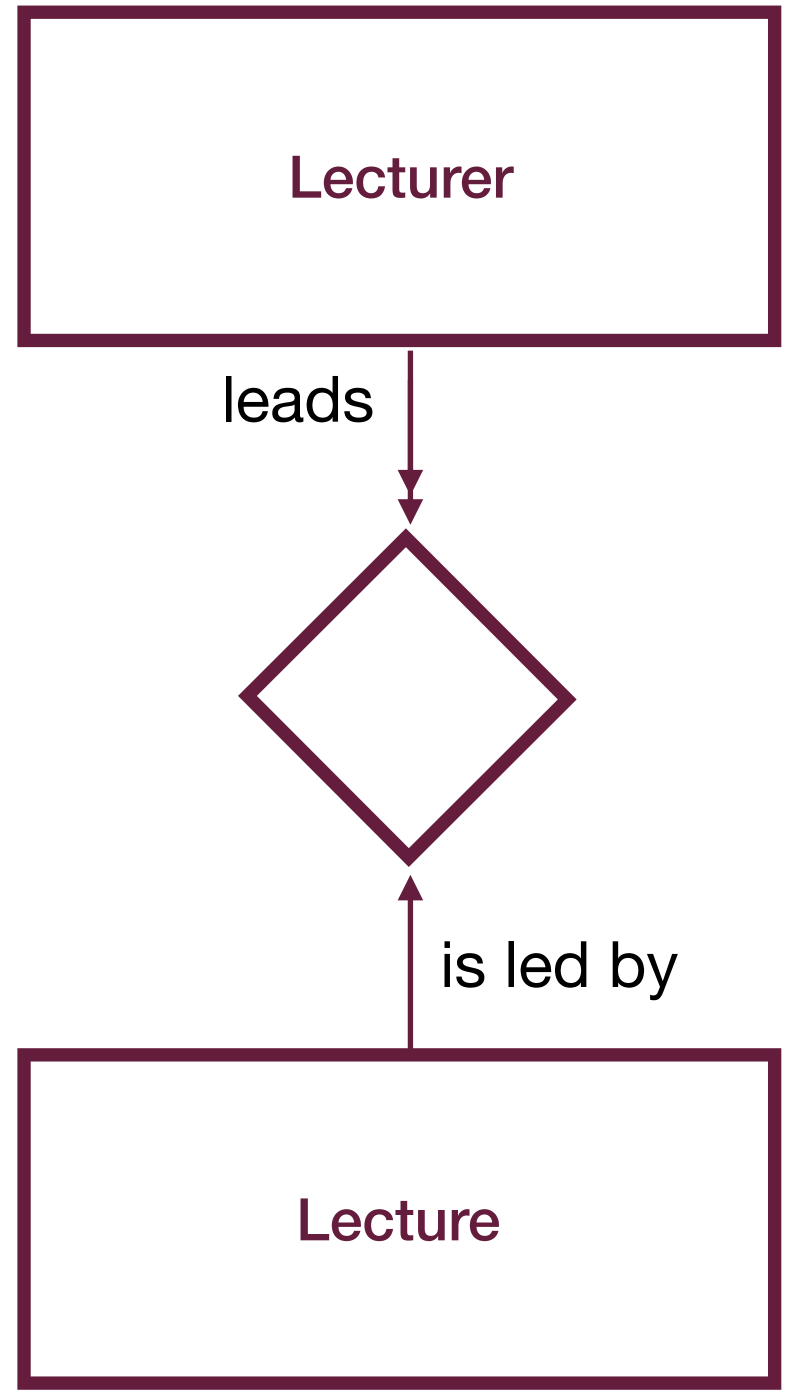
Entity

Represents . . . entity

Associations

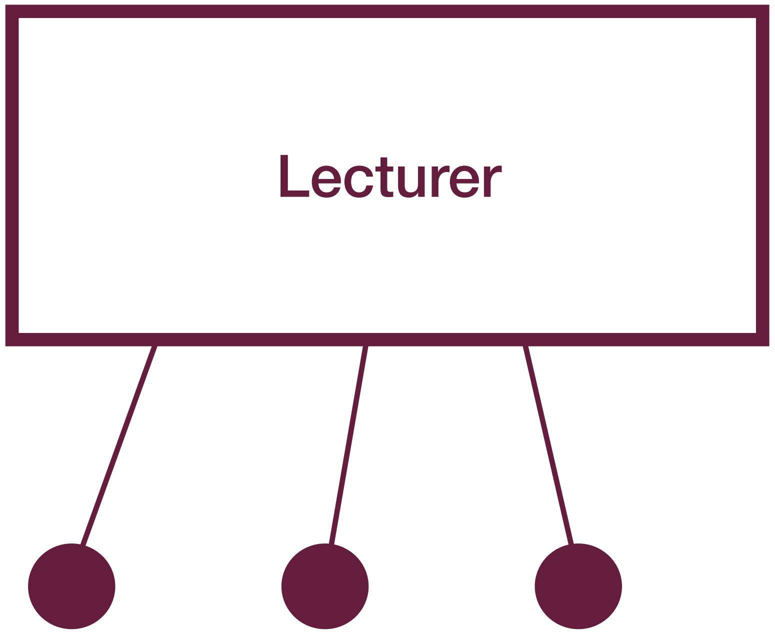
Links between entities

- Single
- Multiple



Attributes

- Key attribute *vs.* non-key
- Mandatory *vs.* facultative
- Simple *vs.* complex
- Single *vs.* multiple
- Descriptive *vs.* association
- Defined *vs.* derived
- Constant *vs.* modifiable



Abstraction techniques

- Classification
  A. *Extensional* aspect: class is just a set of some objects
  B. *Intentional* aspect: all objects in a class have similar structure
- Agrégation
- Generalisation / specialisation

  Iterative process

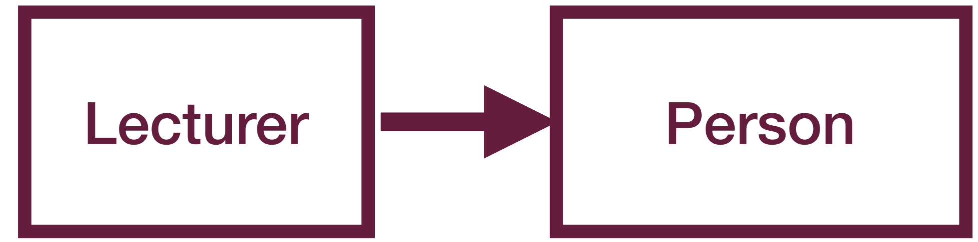
Generalisation / specialisation

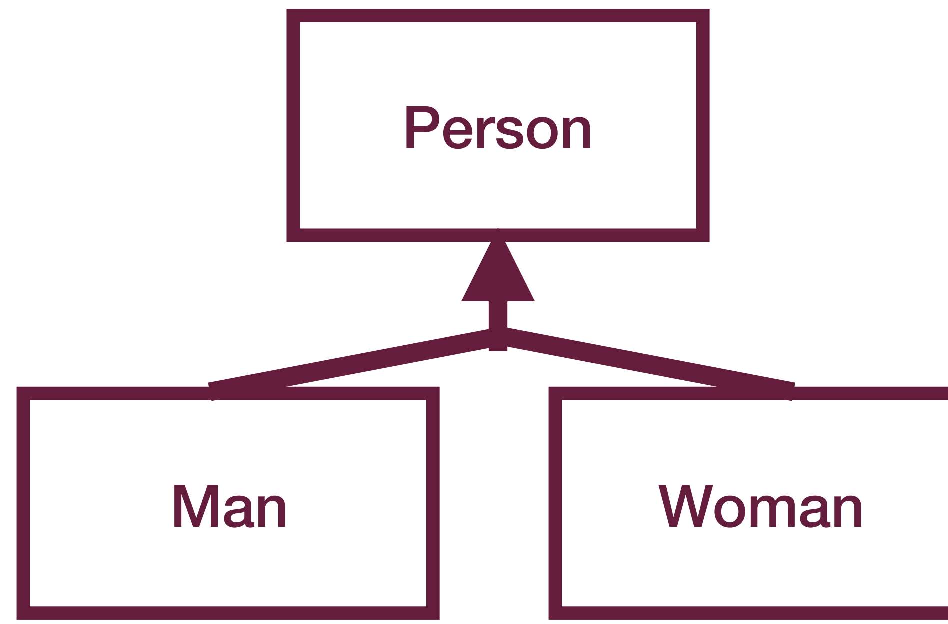
- Two golden rules
    A. If a class ***c*** is a sub-class of ***C***, then ***c*** is a sub-set of the class ***C*** (*extensional* aspect)
    B. If a class ***c*** is a sub-class of ***C***, then ***c*** inherits all properties from the class ***C*** (*intentional aspect*)
- Agrégation
- Generalisation / specialisation

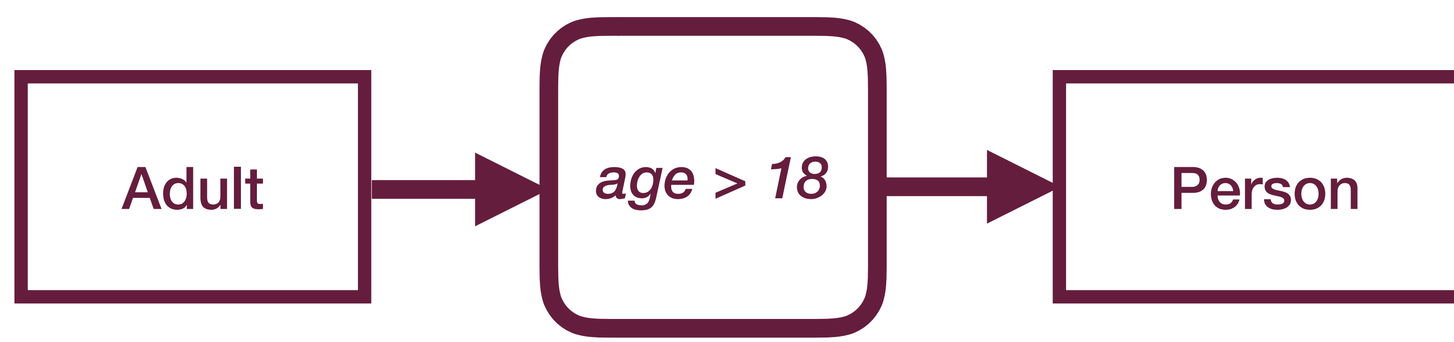
    Iterative process

Generalisation / specialisation

- Inclusion (*Is A*)
- Division
- Constraint







Object Databases Data Model

Mathematical representation

Arūnas Janeliūnas

Things out of model

Infinite sets of:

- object identifiers **obj** = { o_1 , o_2 , … };
- class names **class** = { c_1 , c_2 , … };
- attribute names **att** = { a_1 , a_2 , … };
- method names **meth** = { m_1 , m_2 , . . . }.

Types

Atomic data types

- Long,
- Short,
- Unsigned long,
- Unsigned short,
- Float,
- Double,
- Boolean,
- Octet,
- Char,
- String,
- Enum.

Values of those types constitute a set denominated by **dom**.

Values (*literals*)

Given a set $O \subset$ **oid**, the set of values over O is defined as:

1. *nil* is a value over O;
2. all values from **dom** are values over O;
3. all elements from O are values over O;
4. if $v_1, \dots, v_n$ are values over O and $a_1, \dots, a_n$ are attribute names from **att**, then the tuple $[a_1 : v_1, \dots, a_n : v_n]$ is a value over O;
5. if $v_1, \dots, v_n$ are values over O then the collection $\{v_1, \dots, v_n\}$ is a value over O.

The set of values over O is denoted by **val**(O).

Value examples

```
1,

„Some Value",

oid12,

[ cinema: oid12,
  time: “16.30”,
  price: nil,
  movie: oid4
],

{ “G.Massina”, “S.Loren”, “M.Mastroianni” },

[ title: “La Strada”,
  director: “F.Fellini”,
  actors: {oid25, oid14, oid51}
]
```

Values

Objects

Objects

Object is a pair <*id, val*>**,** where *id* is an element of **oid, **and *val* is a value of the form of a tuple or a collection

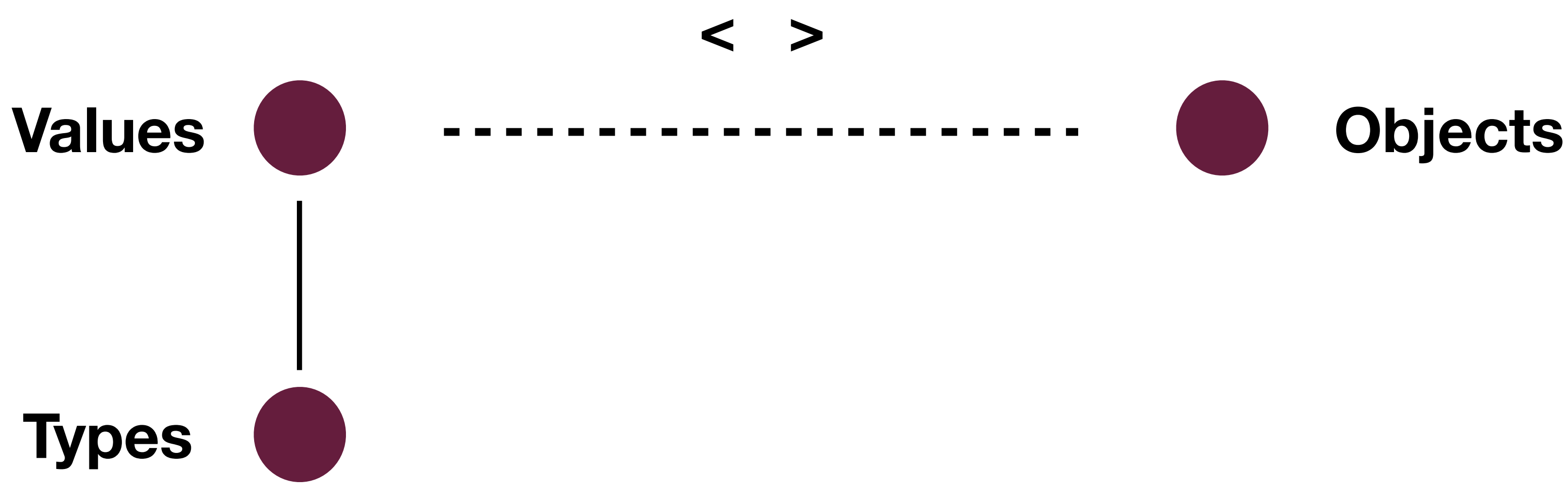
< >

Values - **Objects**

Object examples

```
< oid123, { “G.Massina”, “S.Loren”, “M.Mastroianni” } >,

< oid672354, [ title: “La Strada”,
               director: “F.Fellini”,
               actors: {oid25, oid14, oid51}
             ]
>
```


< >
Values
Objects
Types

Types

Given the set of class names $C \subset$ **class**, types over C are defined as:

- class name **any** is a type over C;
- all atomic types (**short, long, unsigned short** ir t.t.) are types over C;
- class names from C are types over C;
- if $t_1 , \dots , t_n$ are types over C and $a_1 , \dots , a_n$ and $a_1 , \dots , a_n$ are attribute names from **att**, then the tuple $[a_1 : t_1 , \dots , a_n : t_n]$ is a tuple type over O
- if t is a type over C then $\{t\}$ is a collection type over C.

All types over C are denoted by **types**(C).

Collections

ODMG data model has several types for collections:

- *Set*;
- *Bag* (multi-set);
- *List* (has an order in it);
- *Array*.

Tuple types

ODMG data model also has several predefined tuple types:

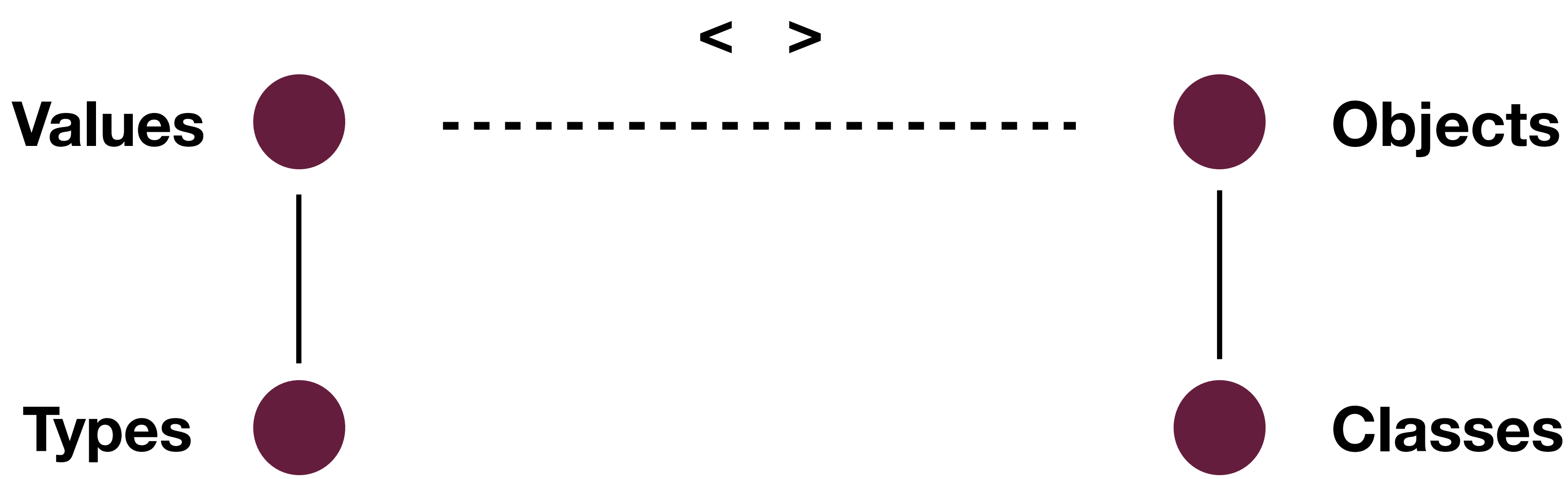
- *Date*;
- *Interval*;
- *Time*;
- *Timestamp*.

Type examples

```
Cinema,  // class name

{ Time },

[ cinema: Cinema,
    time: String,
   price: Short,
   movie: Movie   // yet another class name
]
```


< >
Values
Objects
Types
Classes

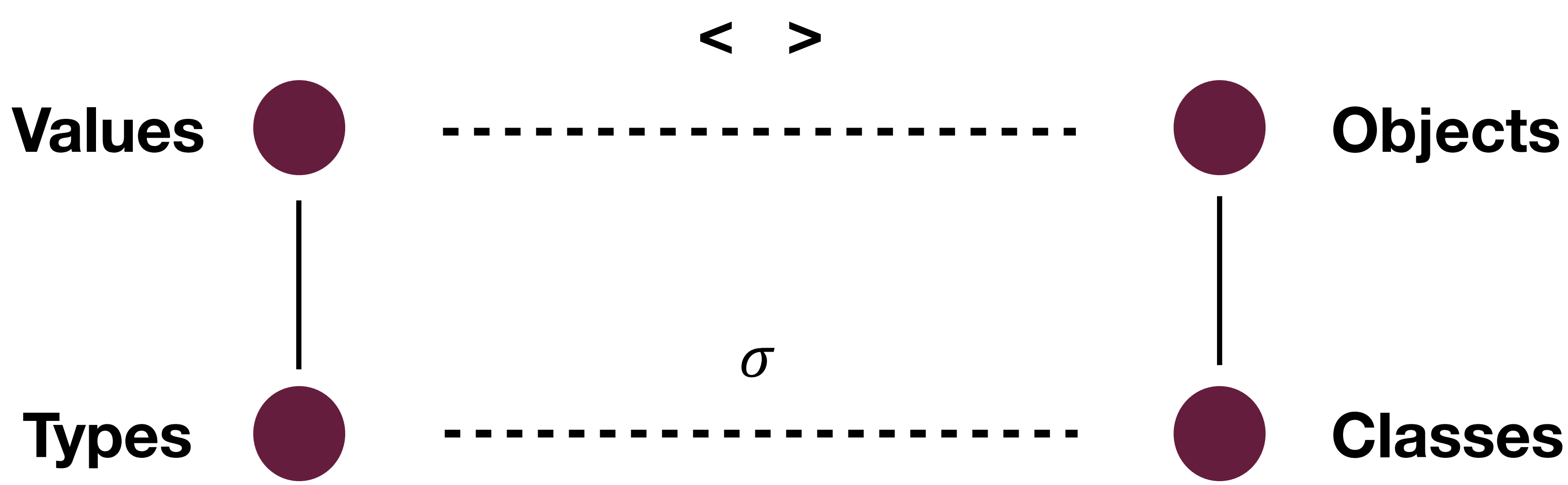
Classes

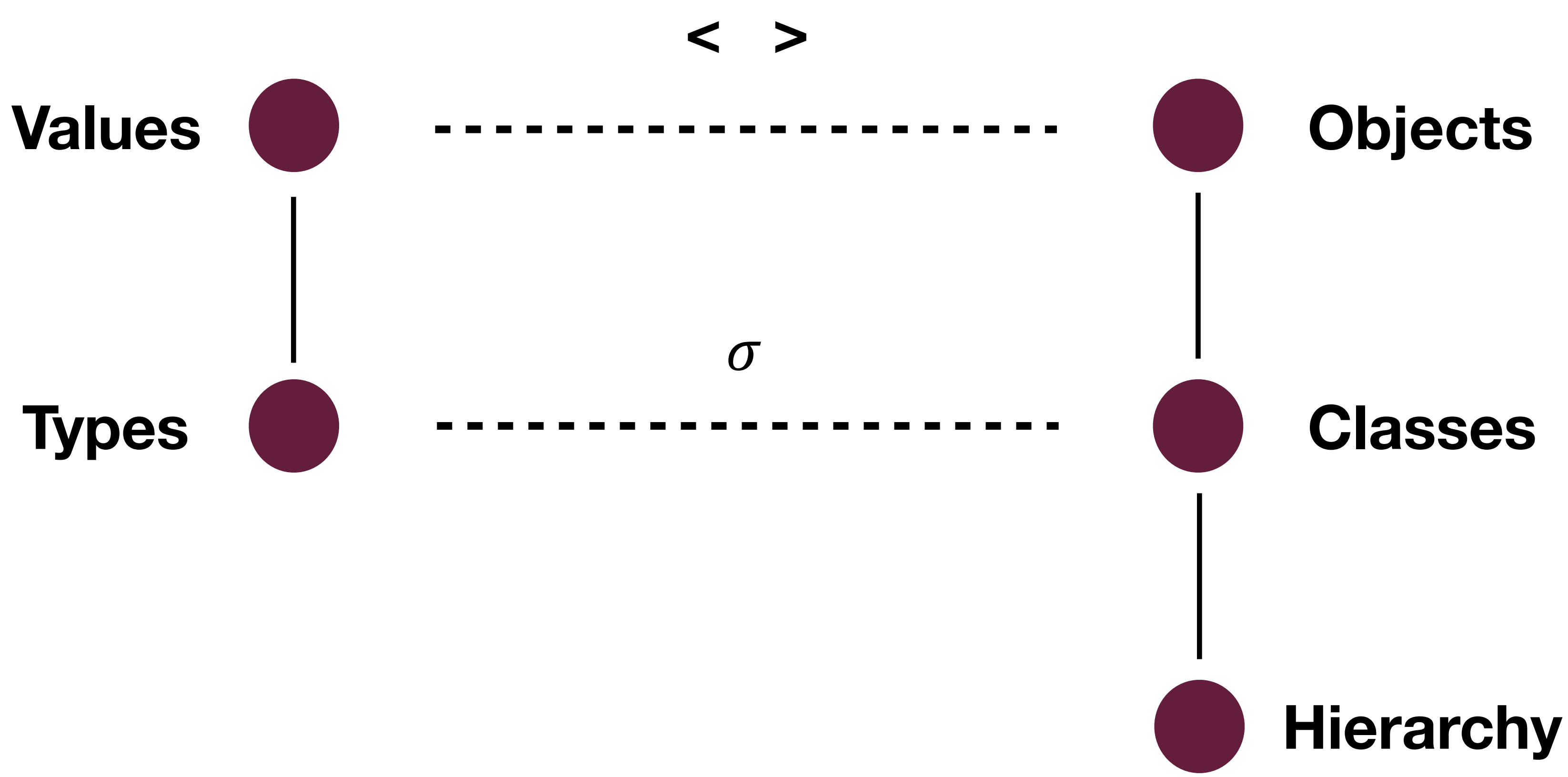
Class is a set of objects holding inside values of the same type.

Classes / types

If C is a set of class names $C \subset$ **class**, then $\sigma(C)$ is a function

$$\sigma : C \to \mathbf{types}(C)$$


< >
Values
Objects
σ
Types
Classes



< >
Values
Objects
σ
Types
Classes
Hierarchy

Class hierarchy

Class hierarchy is a triplet $< C, \sigma, < >$, where:

- C is a finite set of class names,
- $\sigma : C \to$ **types**(C),
- $<$ is a partial order relationship in the set C.

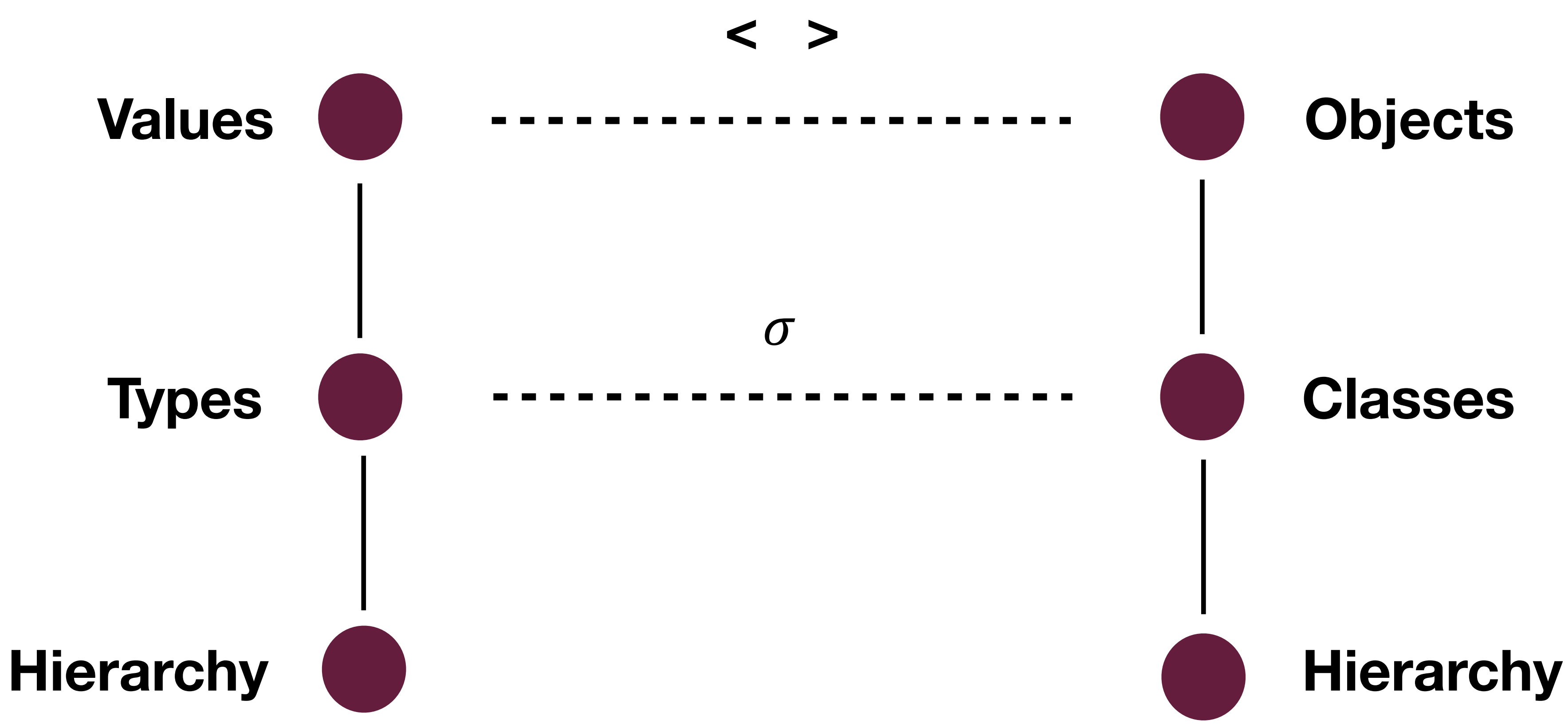
Transitional and non-comutative relationship in the set is called an *order*. The order relationship in the set which exists between any given pair of the set elements is called *total order* and *partial order* otherwise.

Class hierarchy

Can you see $< C, \sigma, < >$ here?

```
class Person {
   String name;
   Integer age;
};

class Lecturer extends Person {
   String title;
};
```


< >
Values
Objects
σ
Types
Classes
Hierarchy
Hierarchy

Type hierarchy

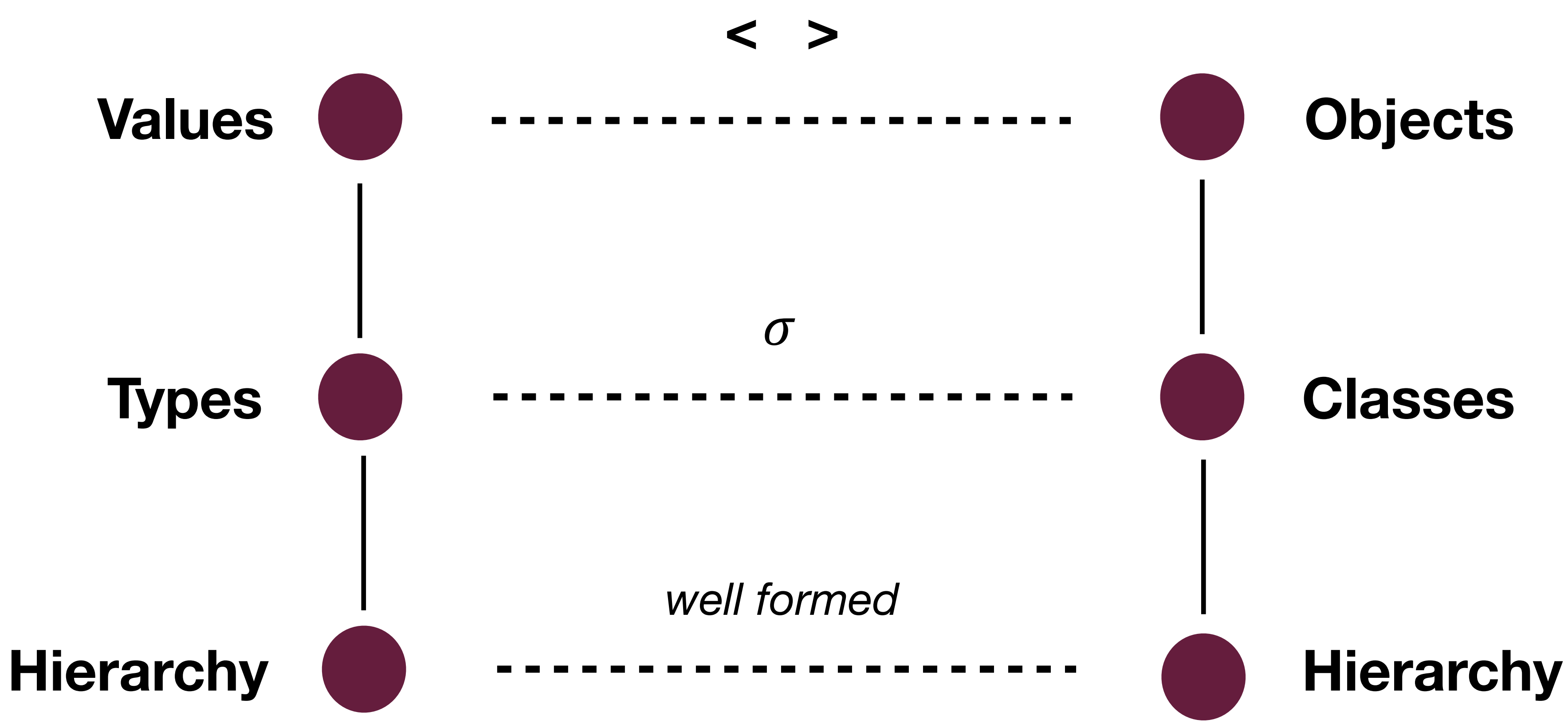
Let $< C, \sigma, < >$ be a class hierarchy. Then the sub-type/super-type relationship $\leq$ is a partial order in the set **types**(C), described by the following rules:

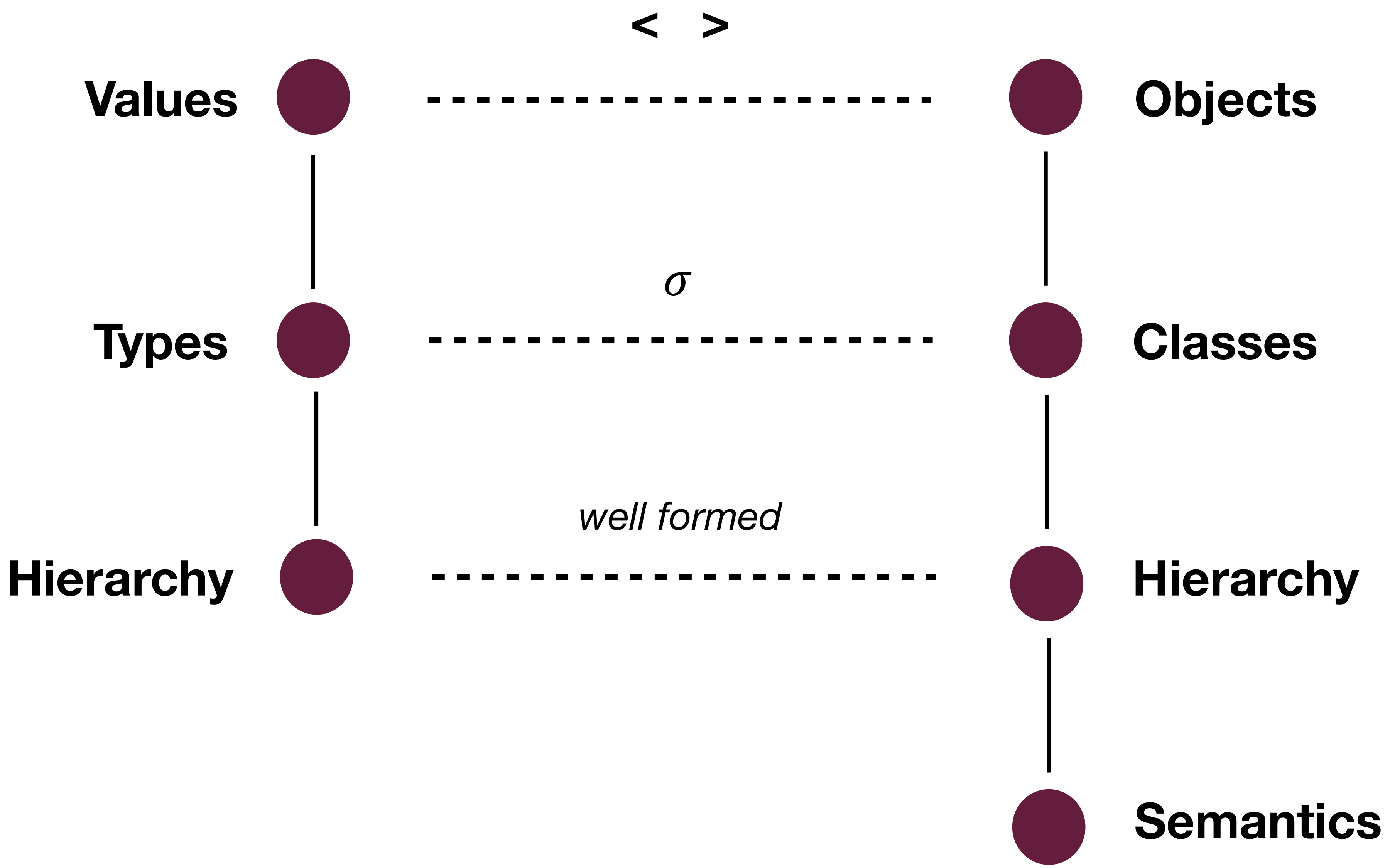
- $\forall\, t : t \leq$ **any** ,
- $c < k \;\Rightarrow\; c \leq k$,
- $(\forall\, i \in [1, n], n \leq m : t_i \leq t'_i) \;\Rightarrow\; [a_1 : t_1 , \ldots , a_m : t_m] \leq [a_1 : t'_1 , \ldots , a_n : t'_n]$,
- $t \leq t' \;\Rightarrow\; \{ t \} \leq \{ t' \}$.

Well formed structure

The class hierarchy $< C, \sigma, < >$ is called to be of a well formed structure if for any given pair of classes c and k

$$c < k \Rightarrow \sigma(c) \leq \sigma(k)$$


< >
Values
Objects
σ
Types
Classes
well formed
Hierarchy
Hierarchy


<   >
Values
Objects
σ
Types
Classes
well formed
Hierarchy
Hierarchy
Semantics

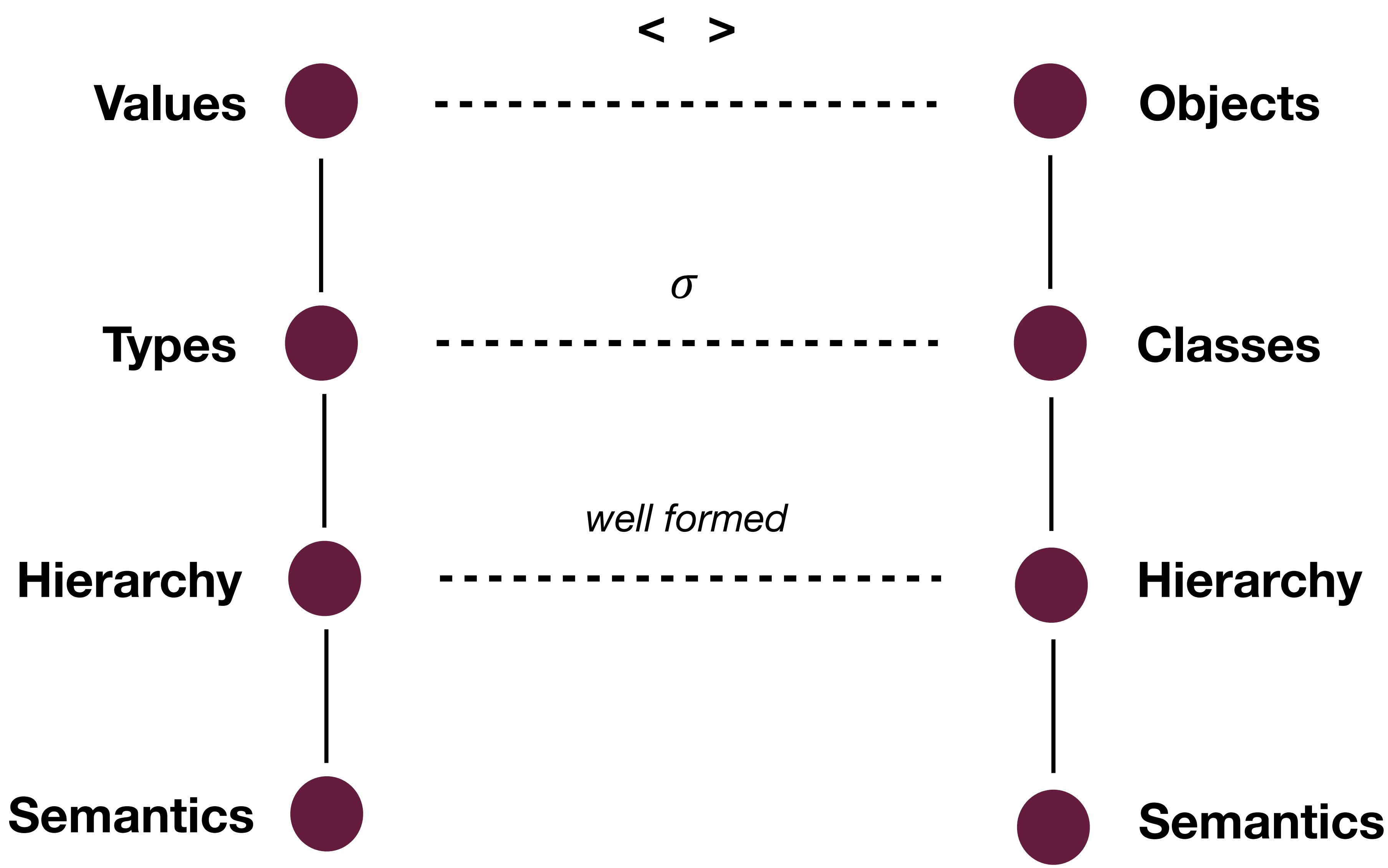
Semantics of the classes

Let $< C, \sigma, < >$ be a class hierarchy (of the well formed structure). *Oid assignment* is a function π which for every element of C assigns a particular set of object identifiers from **oid**.

Therefore $\pi(c)$ is called a *proper extent* of the class c.

The *extent* of the class c (denoted by $\pi^*(c)$) is a set

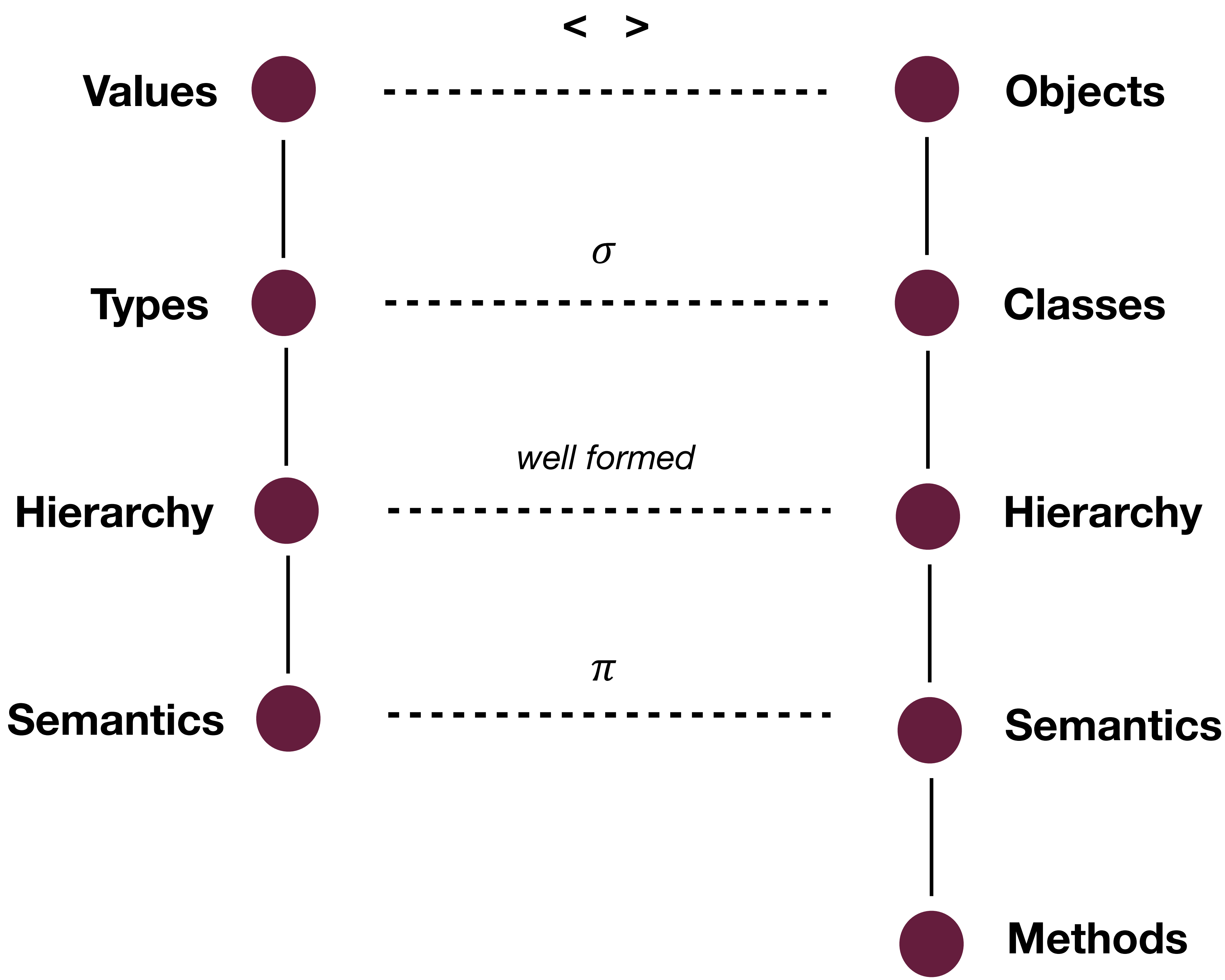
$$\pi^*(c) = \bigcup_k \{ \pi(k) : k = c \vee k < c \}$$


< >
Values
Objects
σ
Types
Classes
well formed
Hierarchy
Hierarchy
Semantics
Semantics

Semantics of the types

Let $< C, \sigma, < >$ be a class hierarchy and $O = \bigcup\{ \pi^*(k) : k \in C \}$. Then we can derive that $O = \pi^*(\boldsymbol{any})$. And then the *type interpretation* **dom**(t) of the type t is defined by:

- **dom**(**any**) = **val**(O)
- for every atomic type t, **dom**(t) is it's „usual" interpretation
- $\forall\, c \in C$: **dom**(c) = $\pi^*(c) \cup \{nil\}$,
- **dom**($\{t\}$) = $\{ \{v_1 , \ldots , v_n\} \mid v_i \in$ **dom**(t) $\}$
- **dom**($[a_1 : t_1 , \ldots , a_n : t_n]$) = $\{ [a_1 : v_1 , \ldots , a_n : v_n] \mid v_i \in$ **dom**(t_i) $\}$


< >
Values
Objects
σ
Types
Classes
well formed
Hierarchy
Hierarchy
π
Semantics
Semantics
Methods

Methods

A method has 3 parts:

- name
- signature
- implementation

Given the method name $m \in$ **meth**, its signature is

$$m : c \times t_1 \times \ldots \times t_n \rightarrow t_{out}$$

where $c \in C$ ($< C, \sigma, < >$ being a class hierarchy) and t_i are the types over C (that is, $t_i \in$ **types**(C)).

Inheritance

Given two classes c and *k* such that

- method *m* is defined in the class *c*
- $k < c$
- does not exists such a class *p* that $k < p < c$,

then it is said that class *k* inherits the method *m* from the class *c*.

Inheritance

Given two methods

$$m : c \times t_1 \times \ldots \times t_n \rightarrow t_{out}$$

and

$$m : k \times t'_1 \times \ldots \times t'_k \rightarrow t'_{out}$$

where $k < c$, the following rules must be followed:

1. *Consistency*. If $k < c$ and $k < p$ without any sub-class relationship between p and c, and method m is defined in both classes p and c, method m must be explicitly defined in the class k as well.
2. *Covariation*. It must be $t'_i \leq t_i$ for every i, and $t'_{out} \leq t_{out}$ as well.

Database scheme

Database scheme is a quintuplet $\mathbf{S} = < C, \sigma, \prec, M, G >$, where:

- $< C, \sigma, \prec >$ is a class hierarchy
- M is a set of method signatures
- G is a set of names, such that $G \cap C = \varnothing$
- $\sigma : C \cup G \rightarrow \textbf{types}(C)$

Object Definition Language

Short introduction

Types

```
<type> ::= <atomic_type> |
           <class_name> |
           <collection_type> |
           <tuple_type> |
           <enumerative_type>

<atomic_type> ::= long | short | unsigned long |
                  unsigned short | float | double |
                  boolean | octet | char | string |
                  date | interval | time | timestamp |
                  void

<class_name> ::= <name>
```

Collection types

```
<collection_type> ::= <collection_constructor> < <type> > |
                      <array>
<collection_constructor> ::= set | bag | list
<array> ::= <type> [<size>][{, [<size>]}]
```

Examples

```
set <Person>

octet [3][3];

char [256]
```

Tuple types

```
<tuple_type> ::= struct <name>
                 {
                   <type> <name>
                   [{; <type> <name>}]
                 }
```

Examples

```
struct Date {
  octet day;
  octet month;
  unsigned short year;
};

struct Student {
  string name;
  string surname;
  short grades [10];
}
```

Enumerative types

```
<enumeratgive_type> ::= enum <name>
                        {
                          <name>
                          [{, <name>}]
                        }
```

Examples

```
enum Colours {
  Red, Green, Blue, Yellow, Black, White, Green, Purple, NonDescriptive
};

enum WorkingDays {
  Monday, Tuesday, Wednesday, Thursday, Friday
}
```

Type definition

```
<type_definition> ::= typedef <type> <name>
```

Examples

```
typedef char[256] Stack

typedef unsigned short SimpleNumber
```

Classes

```
<class> ::= interface <class_name>
                      [: <superclass> [{, <superclass>}]]
            {
              <property> {;<property>}
            };

<class_name> ::= <name>

<superclass> ::= <name>

<property> ::= <attribute> |
               <association> |
               <method>
```

Attributes

```
<attribute> ::= attribute <type> <name>
```

Examples

```
interface Employee : Person
{
  attribute string name;
  attribute string surname;
  attribute struct TypeAddress
                    { string city;
                      string street;
                      short house;
                      short flat;
                    } address;
  attribute Person[5] children;
}
```

Associations

```
<association> ::= relationship <type> <name>
                      [inverse <class_name> :: <association_name>]
<class_name> ::= <name>
<association_name> ::= <name>
```

Examples

```
interface Person {
   relationship Flat lives_in inverse Flat::resident;
};

interface Flat {
   relationship Person resident inverse Person::lives_in;
};

interface person {
   relationship Set<Person> parents
                             inverse Person::children;
   relationship List<Person> children
                             inverse Person::parents;
};
```

Methods

```
<method> ::= <type> <method_name> ( <argument> {, <argument>} )

<method_name> ::= <name>

<argument> ::= <argument_qualifier> <type> <name>

<argument_qualifier> ::= in | out | inout
```

Examples

```
interface Person {
   attribute String name;
   attribute String surname;
   attribute Person spouse;

   void mariage ( in Person whomToMarry );
}
```

Object Query Language

Syntax

Example database

Classes (with attributes and methods):

Person: name, surname, birthDate, address, age()

Student: studentId, courses

Employee: department, salary()

Lecturer: title, courses

Course: title, id, lecturers, students

Address: city, street, house, flat

Storage roots:

People: Bag <Person>

Students: Set <Student>

Employees: Set <Employee>

Lecturers: Set <Lecturer>

Courses: List <Course>

Dean: Lecturer

Data access

Any name returns its value(-s):

```
Dean; // some object of the class Lecturer

Employees; // a set of Employee objects

1 + 1; // the result is … 2 :)
```

Unary path expressions

Like in any object oriented programming language:

```
Dean.name;

Dean.address.street;

Dean.salary();
```

Constructors

One can create objects/values *ad hock*:

```
struct ( street: “Lokio g–vė”,
         city: “Meškai” );

Address ( house: 5,
           flat: 14,
         street: “Partizanų”,
           city: “Joniškėlis” );

set (1, 3, 5, 7, 9);

array (1, 3, 5, 7, 9);

list (1, 2, 3, 4, 5, 6, 7, 8, 9);

list (1..9);
```

Iterators

SELECT is just a sort of the query, meaning iteration:

```
Select s
  from s in Students
 where s.name = “Sigitas”

Select struct ( name: s.name,
                age: age() )
  from s in Students

Select c.lecturer.address.city
  from c in Courses
 where c.title = “Object Databases”

Select s.name
  from s in Students,
       l in Lecturers
 where s.name = l.name
```

N-nary path expressions

```
select l.surname
  from l in Lecturers,
       c in l.courses,
       s in c.students
 where s.name = “Sigitas”
```

Pointer join

```
select struct (student: s.surname,
               lecturer: l.surname)
  from l in Lecturers,
       c in l.courses,
       s in c.students
```

Methods

It can be used anywhere:

```
Dean.salary();

select p.name
  from p in Persons
 where p.age() > 21
```

Collections

It can be used anywhere as well:

```
select c.title
  from c in Dean.courses

select struct ( lecturer: l.surname,
                DB_courses: select c
                              from c in l.courses
                             where c.title like "*DB*"
              )
  from l in Lecturers
 where count(select c
               from c in l.courses
              where c.title like "*DB*") > 0
```

Sorting

```
select s
  from s in Students
order by s.age() asc,
         s.name
```

Alternative syntax, seen in some DBMSes:

```
sort s in Students
  by s.age() asc, s.name
```

Grouping

```
select e
  from e in Employees
group by      rich: e.salary() > 2000,
          moderate: e.salary() > 500 and e.salary() <= 2000,
              poor: e.salary() <= 500
having avg( select p.salary() from p in partition ) > 1111
```

The result of this query is like this:

```
{
  [ rich: false, moderate: false, poor: true,  partition: { e11,e12,...} ],
  [ rich: false, moderate: true,  poor: false, partition: { e21,e22,...} ],
  [ rich: true,  moderate: false, poor: false, partition: { e31,e32,...} ],
}
```

Agreagation etc.

```
max ( select e.salary()
        from e in Employees )


element ( select c
            from c in Courses
           where c.title like "*DB*" and
                 c.id = 101 );

first ( element( select c
                   from c in Courses
                  where c.title like "*DB*" and
                        c.id = 101
                ).lecturers
      );

listtoset (Dean.courses);

flatten ( select l.courses
            from l in Lecturers )
```

Set operations

```
Persons except Students;

Students union set(Dean);

Courses || list ( Course( title: “ODB”,
                          id: 13,
                          lecturers: set(Dean),
                          students: Students )
                )
```

Quantums

```
select s.name
  from s in Students
 where for all c in s.courses : c.title like "*DB*";

select s.name
  from s in Students
 where exists c in s.courses :
            (exists l in c.lecturers :
               l.name = "Janeliūnas")
```

Named queries

```
Define Sigitai as
  select distinct s
    from s in Students
   where s.name = “Sigitas”
```

```
Select ss.address.city
  from ss in Sigitai
```

```
Undefine Sigitai
```

Summary

1. **c** - any constant;
2. **n** - names (data storage roots);
3. **x** - iteration variables;
4. constructors **struct, set, list, bag, array** :

```
struct (name1 : expr1 , ... , namen : exprn)
   set (expr1 , ... , exprn)
  list (expr1 , ... , exprn)
  list (expr1 .. expr2)
   bag (expr1 , ... , exprn)
 array (expr1 , ... , exprn)
```

Summary

5. Operations:

numerical: `+ , – , * , /,` **`mod`**`,` **`abs`**`(expr)`

logical: **`not, and, or`**

structural: `.` (attribute extractor)

set: **`except, union, intersect, flatten, element, distinct, count,`**
`sum, avg, min, max, count`

list: **`||, first, last, listtoset`** and set operations

bag: **`distinct`** and set operations

object: `.` (message sending)

Summary

6. predicates:

`expr`$_1$ Θ `expr`$_2$, $\Theta \in$ { **=** , **!=**, **<**, **>**, **<=**, **>=** }

for all x **in** col: boolexpr(x)

exists x **in** col: boolexpr(x)

Summary

7. iterations:

```
select [distinct] expr1(x1 , ... , xn),
                    . . .
                  exprm(x1 , ... , xn)
  from x1 in col1 ,
          . . .
       xn in coln
[where boolexpr (x1 , ... , xn)]
[group by name1 : expr1 , ... , namen : exprm ]
[having boolexpr(name1 , ... , namen) ]
[order by expr1 [asc, desc], ... , exprq  [asc, desc]]
```

Summary

8. naming:

```
define name as expr

undefine name
```

9. comments:

```
// single line comments
/* block comments,
   having as many lines
   as you decide is necessary */
```

Object Query Language

Typing

Name resolution

Name resolution order:

1. variable;
2. property;
3. query name;
4. name from the schema.

```
class Person {
 . . .
};

class Car {
 person: Person;
 . . .
};

name Persons : set <Person>;

define person as . . . ;

select . . .
  from person in Persons,
       auto in MyCVars
 where auto.person = . . .
```

Typing rules

Rules are read like this:

The following is true if the condition holds

$$\frac{\texttt{q} :: \{t\}}{\texttt{element(q)} :: t}$$

Rule examples

Casting:

$$\frac{\texttt{q} :: c' \,,\, c \leq c' \text{ arba } c \geq c'}{(\texttt{c})\,\texttt{q} :: c}$$

Property extraction:

$$\frac{\texttt{q} :: t \,,\, t \leq [\texttt{a} : t']}{\texttt{q.a} :: t'}$$

Sum:

$$\frac{\texttt{q}_1 :: int \,,\, \texttt{q}_2 :: int}{\texttt{q}_1 + \texttt{q}_2 :: int}$$

Rule examples

Another sum:

$$\frac{\mathtt{q}_1 :: t_1\ ,\ \mathtt{q}_2 :: t_2\ ,\{real\} \subseteq \{t_1\} \cup \{t_2\} \subseteq \{int, real\}}{\mathtt{q}_1 + \mathtt{q}_2 :: real}$$

Calling a method:

$$\frac{\mathtt{q} :: c\ ,\quad \mathtt{m} : (c, t_1, \ldots, t_n) \rightarrow t\ ,\quad \forall i = \overline{1,n}\quad \mathtt{q}_\mathtt{i} :: t'_i\ ,\ t'_i \leq t_i}{\mathtt{q.m}(\mathtt{q}_1, \ldots, \mathtt{q}_\mathtt{n}) :: t}$$

Structure constructor:

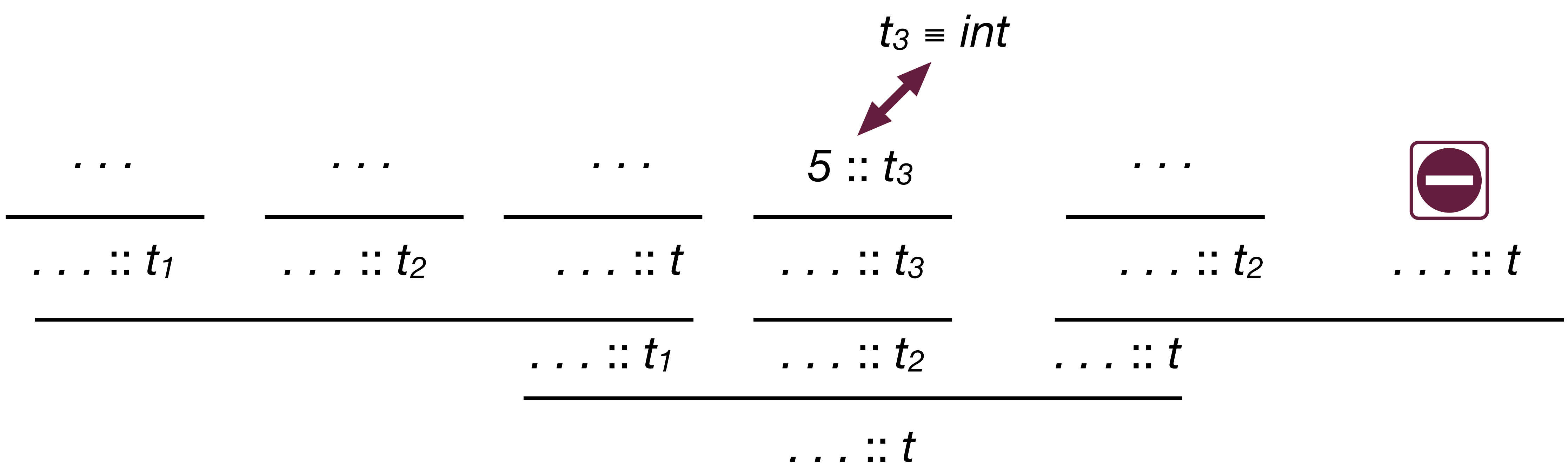
$$\frac{\forall i = \overline{1,n}\quad \mathtt{q}_\mathtt{i} :: t_i}{\mathtt{struct}(\mathtt{a}_1 : \mathtt{q}_1, \ldots, \mathtt{a}_\mathtt{n} : \mathtt{q}_\mathtt{n}) :: [\mathtt{a}_1 : t_1, \ldots, \mathtt{a}_\mathtt{n} : t_n]}$$

Rule examples

Iterator:

$$\frac{\mathtt{q}_1 :: col(t_1),\ \mathtt{q}_2 :: [t_1] \to col(t_2),\ \ldots,\ \mathtt{q}_n :: [t_1,\ldots,t_{n-1}] \to col(t_n),\quad \mathtt{p} :: [t_1,\ldots,t_n] \to bool,\ \mathtt{q} :: [t_1,\ldots,t_n] \to t}{\mathtt{x}_1 :: t_1,\ldots,\mathtt{x}_n :: t_n,\ \left(\begin{array}{l}\mathtt{select\ q}[\mathtt{x}_1,\ldots,\mathtt{x}_n] \\ \quad \mathtt{from\ x}_1\ \mathtt{in\ q}_1,\ \mathtt{x}_2\ \mathtt{in\ q}_2[\mathtt{x}_1],\ldots,\mathtt{x}_n\ \mathtt{in\ q}_n[\mathtt{x}_1,\ldots,\mathtt{x}_{n-1}] \\ \ \mathtt{where\ p}[\mathtt{x}_1,\ldots,\mathtt{x}_n]\end{array}\right) :: \{\{t\}\}}$$

Typing tree



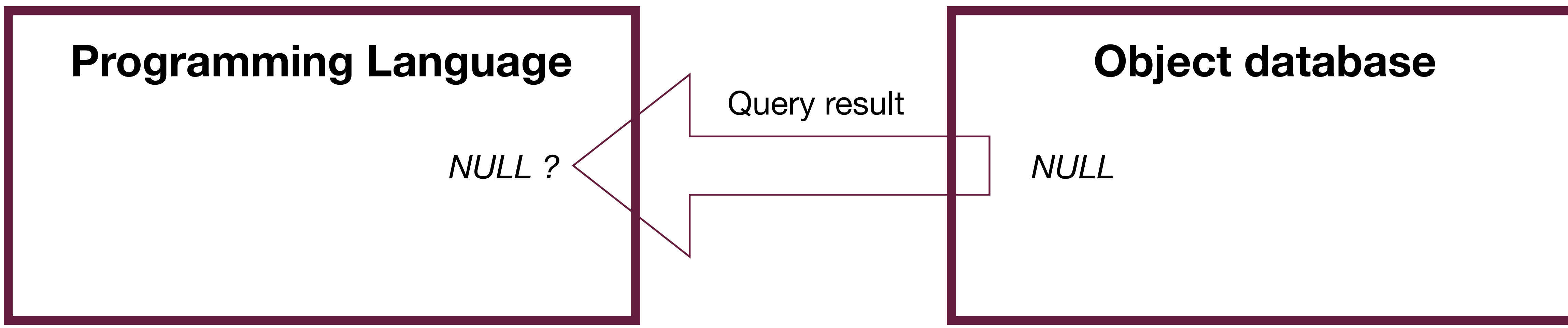
Run-time typing errors

Regardless query typing, run-time errors still exists:

- operations **min**, **max**, **avg**, **sum**, etc with empty set;
- operation **element** with a set having more than one element;
- division by zero;
- index out of bounds with arrays, lists, strings…;
- wrong casting;
- accessing properties of *nil*.

NULL value issue

Is there a *NULL* value in the Programming Language (*NULL* != *nil*)



Attribute specialisation issue

$$(\forall i \in [1, n], n \leq m : t_i \leq t'_i) \Rightarrow [a_1 : t_1, \ldots, a_m : t_m] \leq [a_1 : t'_1, \ldots, a_n : t'_n]$$

```
class A {
  x: { myBool: boolean }
};

method set_x_to_true:boolean in class A {
    if(this.x.myBool == False) {
        this.x = {myBool:true};
        return true;
    }
    else return false;
};

class B extends A {
  x: { myBool: boolean, myInt: int }
};


method read_x_int:integer in class B {
    return this.x.myInt
};
```

All is well here…

… but this query breaks it all:

```
select b.read_x_int()
  from b in Some-B-Objects-Set
 where b.set_x_to_true();
```

So no attribute specialisation is allowed in practice

Covariation vs. Contrvariation

```
class Point {
 x: real,
 y: real
};

class ColorPoint extends Point {
 c: string  // x and y inherited from Point
};

method equal (p:Point):boolean
                    in class Point
{
  return ((this.x == p.x) && (this.y == p.y));
};

method equal (p:ColorPoint):boolean
                    in class ColorPoint
{
    return ( (this.x == p.x) &&
             (this.y == p.y) &&
             (this.c == p.c) )
};
```

All is well here…

… but then let's add this:

```
method break_it (p:Point):boolean
                        in class Point
{
   return p.equal (this)
};
```

… and write a query:

```
(new Point()).break_it(new ColorPoint())
```

Covariation vs. Contravariation

Class **C**	method	$\mathbf{m : A \to B}$
$\vee$		
Class **c**	method	$\mathbf{m : a \to b}$

It is safe to use method **c:m** instead of **C:m**, if:

- **c:m** can accept same arguments as **C:m** ($\mathbf{a} \geq \mathbf{A}$)
- any code expecting results from **C:m** will accept results from **c:m** ($\mathbf{B} \geq \mathbf{b}$)

Class **C**	method	$\mathbf{m : A \to B}$	**that's**
$\vee$		$\wedge\!\mid$ $\quad$ $\mid\!\vee$	***Contravariation***
Class **c**	method	$\mathbf{m : a \to b}$	**rule**

Covariation vs. Contravariation

	Class **C**	method	**m : A → B**	
type-safe	$\vee$		$\wedge\!\!\mid$ $\quad$ $\mid\!\!\vee$	***Contravariation***
	Class **c**	method	**m : a → b**	

	Class **C**	method	**m : A → B**	
practical	$\vee$		$\mid\!\!\vee$ $\quad$ $\mid\!\!\vee$	***Covariation***
	Class **c**	method	**m : a → b**	

OQL Semantics

Let’s talk algebra

Matematikos
ir informatikos
fakultetas

What is *algebra*?

Simply put, algebra consists of two parts:

- *Data* - what kind of elements we do consider?
- *Operations* - what operations we do perform on this data and what properties those operations have?

What is our *data*?

… or in other words:

So, what kind of elements we will manipulate in OQL algebra?

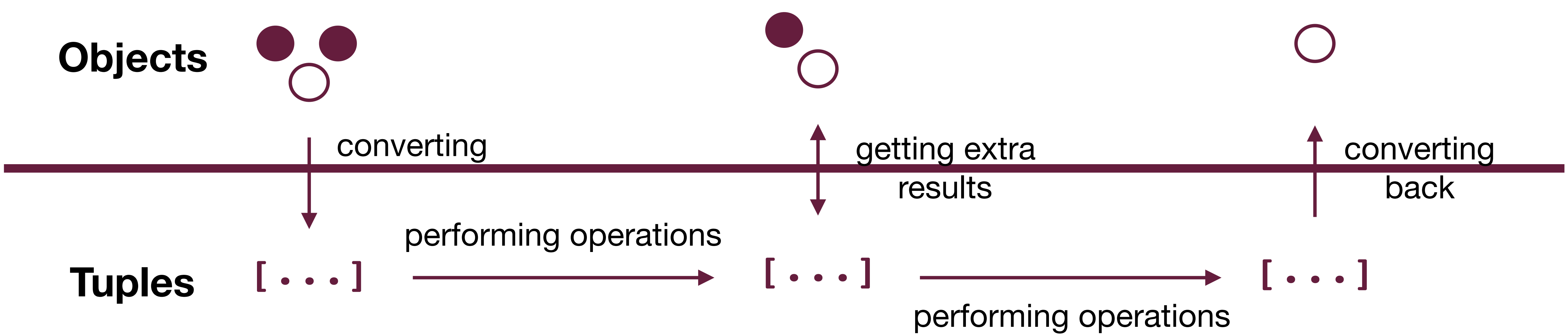
Let it be tuples of the form:

```
[y:some_object, x:some_value, . . .]
```

No methods in algebra…

Principal overview

OQL



Algebra

MAP

```
expr[a] = { [a:x] | x∈expr }
```

```
People[p]
```

gives us

```
{ [p:obj1], [p:obj2], … }
```

MAP

$$MAP_{a:expr2}(expr1) = \{ x \otimes [a:expr2(x)] \mid x \in expr1 \}$$

means „concatenate"

$MAP_{n:p.name}(People[p])$

gives us

{ [p:obj1,n:"Adam"], [p:obj2,n:"Eve"], … }

MAP

Calling methods:

$\mathtt{MAP}_{\mathtt{a:p.address.getStreet()}}\mathtt{(People[p])}$

Adding sub-queries:

$\mathtt{MAP}_{\mathtt{pc:}\ \sigma_{\mathtt{o=p}}\mathtt{(MAP}_{\mathtt{o:c.owner}}\mathtt{(Cars[c]))}}\mathtt{(People[p])}$

$\sigma_{\mathtt{o=p}}\mathtt{(MAP}_{\mathtt{o:c.owner}}\mathtt{(Cars[c]))}$

MAP

$$\mathtt{MAP}_{\mathtt{expr2}}\mathtt{(expr1) = \{ expr2(x) \mid x \in expr1 \}}$$

$$\mathtt{MAP}_{\mathtt{n}}\mathtt{(MAP}_{\mathtt{n:p.name}}\mathtt{(People[p]))}$$

gives us

`{ "Adam", "Eve", … }`

$\mathtt{MAP}_{\mathtt{p.name}}\mathtt{(People[p])}$
would be quicker, don't you find?

Filtration (selection)

$\sigma_{p}(\texttt{expr}) = \{\ x \mid x \in \texttt{expr} \land p(x)=\texttt{true}\}$

$\sigma_{n=\text{„Arunas“}}(\texttt{MAP}_{n:\texttt{c.owner.name}}(\texttt{Cars[c]}))$

gives us

{ [c:obj1,n:“Arunas“], [c:obj2,n:“Arunas“], … }

Join

$$expr1 \bowtie_{p} expr2 = \{ x1 \otimes x2 \mid x1 \in expr1 \wedge x2 \in expr2 \wedge p(x1,x2)=true \}$$

$$(MAP_{n:p.name}(People[p])) \bowtie_{on=n} (MAP_{on:c.owner.name}(Cars[c]))$$

gives us

`{ [c:obj1,p:obj2,n:“Arunas“,on:“Arunas“], … }`

Matematikos
ir informatikos
fakultetas

Dependent Join

```
expr1<expr2> { x1⊗x2 | x1∈expr1 ∧ x2∈expr2(x2)}
```

```
People[p]<p.cars[c]>
```

gives us

```
{ [p:obj1,c:obj11],[p:obj2,c:obj21],[p:obj2,c:obj22], … }
```

Sorting

$$\mathtt{Sort}_{A,\theta}(\mathtt{expr}) = \{\ x_1,\ldots,x_n \mid x_i \in \mathtt{expr} \wedge x_i.A_k\ \theta_k\ x_{i+1}.A_k\ \}$$

$$\mathtt{Sort}_{\{n,a\},\{<,<\}}(\mathtt{MAP}_{a:p.getAge()}(\mathtt{MAP}_{n:p.name}(\mathtt{People[p]})))$$

Grouping

$$\Gamma_{g,A,\theta,f}(\mathrm{expr}) = \{\ x.A \otimes [g{:}\mathrm{group}]\ \mid\ x \in \mathrm{expr}\ \wedge\ \mathrm{group} = f(\{y \mid y \in \mathrm{expr}\ \wedge\ y_i.A_k\ \theta_k\ x_i.A_k\ \}$$

$$\Gamma_{\mathrm{partition},\{n\},\{=\},\mathrm{Id}}(\mathrm{MAP}_{n:p.\mathrm{name}}(\mathrm{People}[p]))$$

gives us

```
{ [n:"Arunas",partition:{[p:obj2,n:"Arunas"],…}],
  [n:"Sigitas",partition:{[p:obj7,n:"Sigitas"],…}],
   … }
```

Query translation

General query form for the „iteration query“:

```
select s
  from x1 in f1, ..., xn in fn
 where p
group by a1:c1, ... , am:cm
having q
order by o1, ... , ok
```

Step 1

```
select s
  from x₁ in f₁, ..., xₙ in fₙ
 where p
group by a₁:c₁, ... , aₘ:cₘ
having q
order by o₁, ... , oₖ
```

Step 1: FROM

$$F = f_1[x_1] \langle f_2[x_2] \rangle \ldots \langle f_n[x_n] \rangle$$

Here we use either Dependent Joins or simple Joins depending on the expressions f_i

Step 2

```
select s
     from x₁ in f₁, ..., xₙ in fₙ
    where p
 group by a₁:c₁, ... , aₘ:cₘ
   having q
 order by o₁, ... , oₖ
```

Step 2: WHERE

$$W = \sigma_{p(v_1,...,v_w)}(MAP_{v_1:m_1,..,v_w:m_w}(F))$$

First we map all sub-queries results as additional attributes v_i and then filter the output set by the predicate p

Step 3

```
select s
     from x₁ in f₁, ..., xₙ in fₙ
    where p
 group by a₁:c₁, ... , aₘ:cₘ
   having q
 order by o₁, ... , oₖ
```

Step 3: GROUP BY

$$G = \Gamma_{\texttt{partition},\{a_1,\dots,a_w\},\{=,\dots,=\},\texttt{Id}}(\texttt{MAP}_{a_1:c_1,\dots,a_w:c_w}(W))$$

First we map all sub-queries results as additional attributes a_i and then filter the output set by the predicate p

Step 4

```
select s
     from x₁ in f₁, ..., xₙ in fₙ
    where p
 group by a₁:c₁, ... , aₘ:cₘ
   having q
 order by o₁, ... , oₖ
```

Step 4: HAVING

$$H = \sigma_{q(h_1,...,h_m)}(MAP_{h_1:g_1,..,h_m:g_m}(G))$$

Once again, first we map all needed sub-queries as attributes h_i and then filter the output set by the predicate q

Step 5

```
select s
     from x1 in f1, ..., xn in fn
    where p
group by a1:c1, ... , am:cm
   having q
order by o1, ... , ok
```

Step 5: ORDER BY

$$S = \mathrm{Sort}_{\{o1,\ldots,ok\},\{\ldots\}}(\mathrm{MAP}_{o1:s1,\ldots,ok:sk}(H))$$

Step 6

```
select s
  from x₁ in f₁, ..., xₙ in fₙ
 where p
group by a₁:c₁, ... , aₘ:cₘ
having q
order by o₁, ... , oₖ
```

Step 5: SELECT

$$\texttt{Result = MAP}_{\texttt{S}}\texttt{(S)}$$

Final `MAP` operation is designed to get the desired set of query result

Query translation

Even more straightforward translation for simple queries.

For example:

```
BigBoss.address.getCity()
```

may be translated simply as

$\mathtt{MAP}_{\mathtt{bb.address.getCity()}}\mathtt{(BigBoss[bb])}$

Complex example

```
select struct ( age: a,
                cnt: count(partition)
              )
  from l in Lecturers,
       c in l.courses
 where c.title = “ODB”
 group by a:l.getAge()
```

Step 1: FROM

```
F = Lecturers[l] <l.courses[c]>
```

gives us

```
{ [l:obj1,c:obj11],[l:obj2,c:obj21],[l:obj2,c:obj22], … }
```

Step 2: WHERE

$$W = \sigma_{t=\text{„ODB“}}(MAP_{t:c.title}(F))$$

gives us

{ [l:obj1,c:obj11,t:“ODB“],[l:obj2,c:obj22,t:“ODB“], … }

Step 3: GROUP BY

$$G = \Gamma_{\texttt{partition},\{\texttt{a}\},\{=\},\texttt{Id}}(\texttt{MAP}_{\texttt{a:l.getAge()}}(\texttt{W}))$$

gives us

```
{  [a:42, partition:{[l:obj1,c:obj11,t:"ODB",a:42],…} ,
   [a:53, partition:{[l:obj9,c:obj91,t:"ODB",a:53],…}
   …
}
```

Step 4: SELECT

$$\texttt{Result} = \texttt{MAP}_{\texttt{[age:a,cnt:count(partition)]}}\texttt{(G)}$$

gives us

```
{
  [age:42,cnt:2],
  [age:53,cnt:3],
  …
}
```

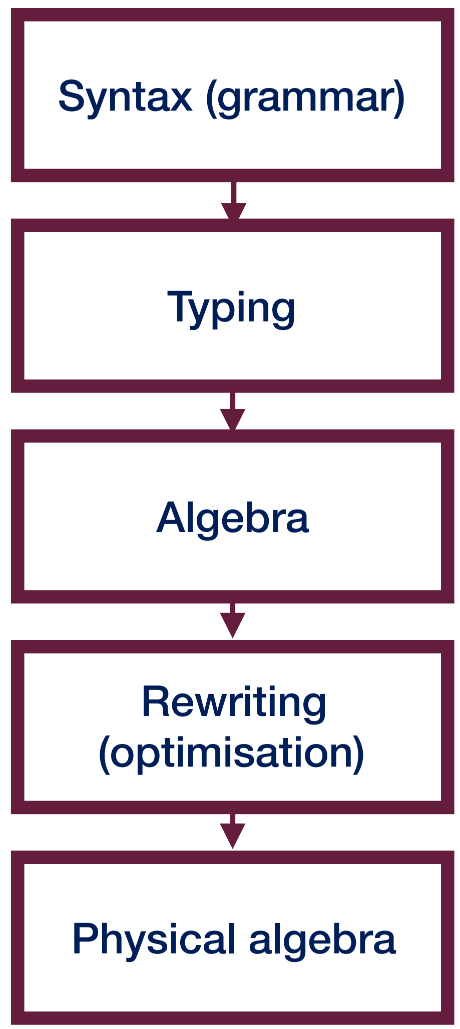
OQL Physical Algebra

Getting down to the libs

Matematikos
ir informatikos
fakultetas

Arūnas Janeliūnas

Behind the query



Is the query well written?

Will the query output its result?

What result query will output?

Can it be done faster?

How it will be done?

Matematikos ir informatikos fakultetas

O_2 physical algebra

- Just an example
- C programming language

get

```
get (monoid, extent_name, range_variable, predicate)
```

implements

$\sigma_{\texttt{predicate}}(\texttt{extent\_name[range\_variable]})$

Actually, it's

```
select *
  from range_variable in extent_name
 where predicate
```

reduce

reduce (monoid, expr, variable, head, predicate)

implements

$$\mathtt{MAP}_{\mathtt{variable:head}}(\sigma_{\mathtt{predicate}}(\mathtt{expr}))$$

join

join (monoid, left, right, predicate, keep)

implements

$$\texttt{left} \bowtie_{\texttt{predicate}} \texttt{right}$$

Is it the outer join?

```
keep = left
keep = right
keep = none
```

unnest

```
unnest (monoid, expr, variable, path, predicate, keep)
```

implements

$\sigma_{\texttt{predicate}}(\texttt{expr[path\_root]<path[variable]>})$

Path may be of the form *path_root.path_links*

```
l.courses
```

Is it the outer d-join?

```
keep = true
```

```
keep = false
```

nest

`**nest** (monoid, expr, var, head, groupby, nestvars, predicate)`

implements

$$\mathtt{MAP}_{\mathtt{nestvars:nestvars}}(\Gamma_{\mathtt{var,\ groupby,\ \{=,=,\ldots\},\ head}}(\mathtt{expr}))$$

Here the attribute `var` which is added to every combination of `groupby` attributes is

`var = **reduce** (monoid, expr, var, head, predicate)`

map

map (monoid, expr, variable, function)

implements

$\mathrm{MAP}_{\mathtt{variable:function}}(\mathtt{expr})$

merge

`merge` `(monoid, left, right)`

implements an union of the two collections: `left` **and** `right`

Example

```
select struct ( age: a,
                cnt: count(partition)
              )
  from l in Lecturers,
       c in d.courses
 where c.title = "ODBS"
group by a: l.age()
```

```
get    : set ([d: Destytjas])
unnest : bag ([d: Destytjas, k: Kursas])
unnest : bag ([d: Destytjas, k: Kursas, a: integer])
nest   : bag ([a: integer, partirion: bag(Destytojas)])
reduce : set ([amzius: integer, kiekis: integer])
```

```
reduce ( set,
         nest ( bag,
                unnest ( bag
                         unnest ( bag,
                                  get ( set,
                                        Lecturers,
                                        l,
                                        and()
                                      ),
                                  c,
                                  l.courses,
                                  and( c.title="ODBS" )
                                ),
                         a,
                         l.age(),
                         and()
                       )
                partition,
                d,
                vars(a),
                vars(),
                and()
              )
         result,
         struct( age:a, cnt:count(partition) ),
         and()
       )
```

Matematikos
ir informatikos
fakultetas

Database Integrity

How do we control methods?

Database integrity

Ensure that the data in the database is correct at any given moment

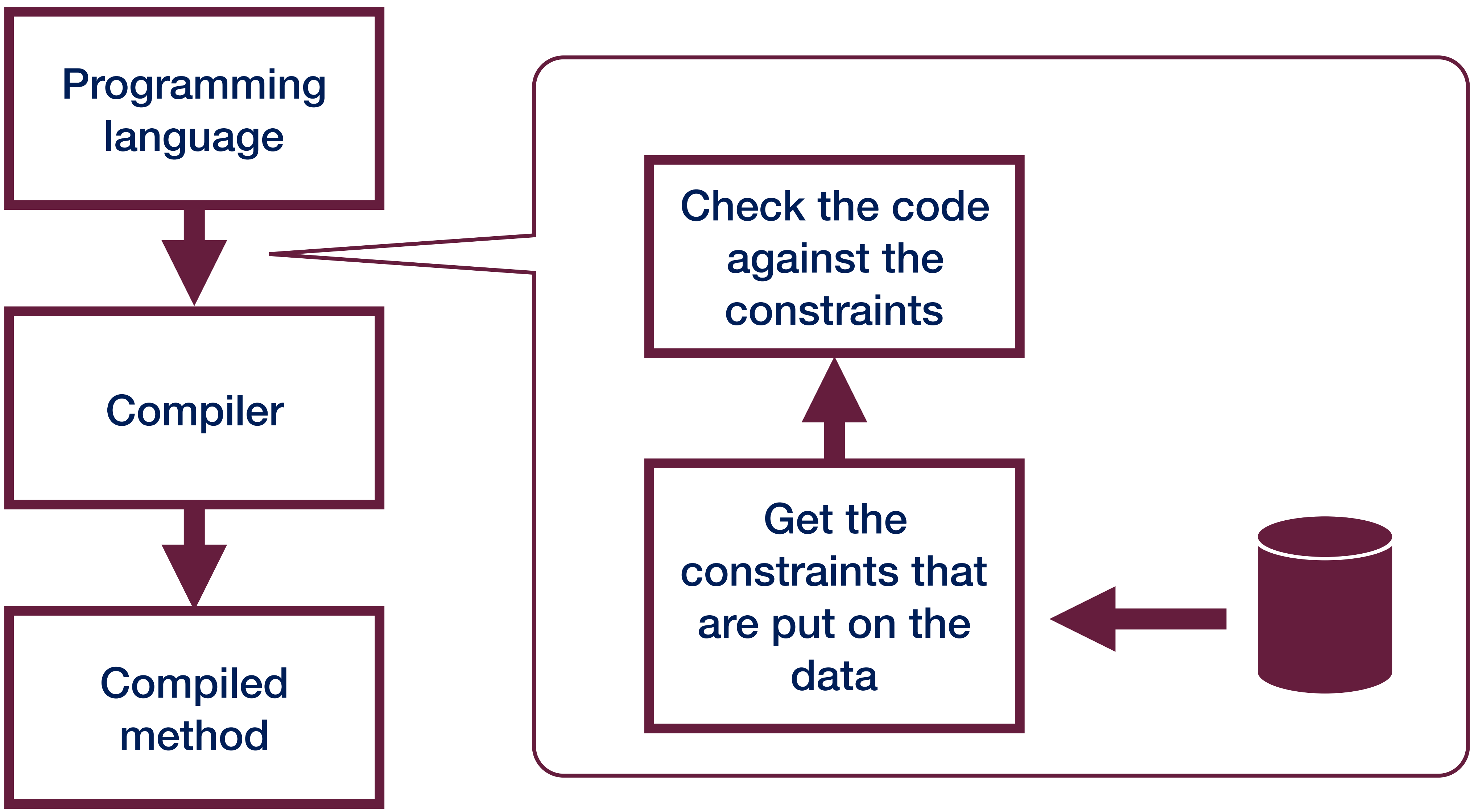
Monitoring data changes

Object methods are doing changes to the database

Monitoring methods, how are they changing the data

How we are supposed to do this?

Idea


Programming language
Compiler
Compiled method
Check the code against the constraints
Get the constraints that are put on the data

„Any“ programming language

Simplified object-oriented programming language:

```
expression ::=
    variable.attribute := variable
  | expression ; expression
  | { expression }
  | if condition then expression
  | forall variable where condition do expression
  | forone variable where condition do expression
```

Now one only need to translate his PL into the one above…

What `forone` does?

```
a.spouse.spouse := b.spouse
```

translates into…

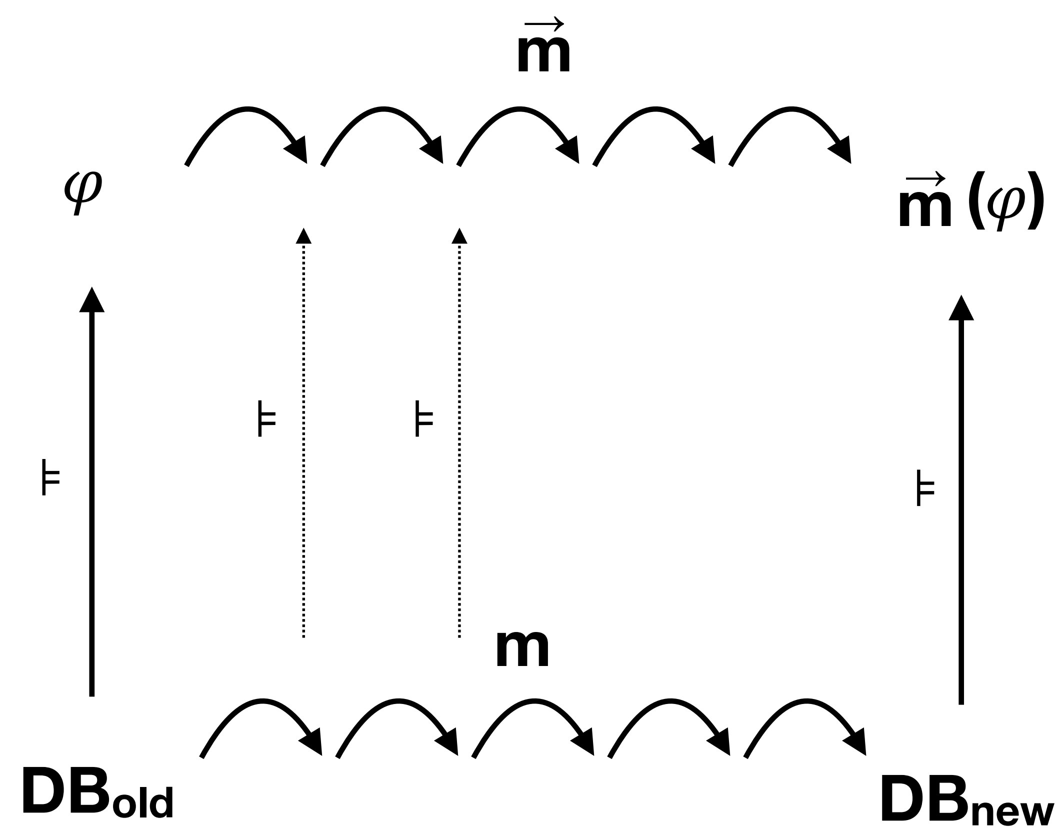
```
forone o₁ where a.spouse = o₁ do
  forone o₂ where b.spouse = o₂ do
    o₁.spouse := o₂
```

Constraints language

1st level logic language

```
exists x in Persons: x.spouse = nil          ∧
forall x in Persons: x.spouse ≠ x            ∧
forall x in Persons: x.spouse ∉ x.children   ∧
. . .
```

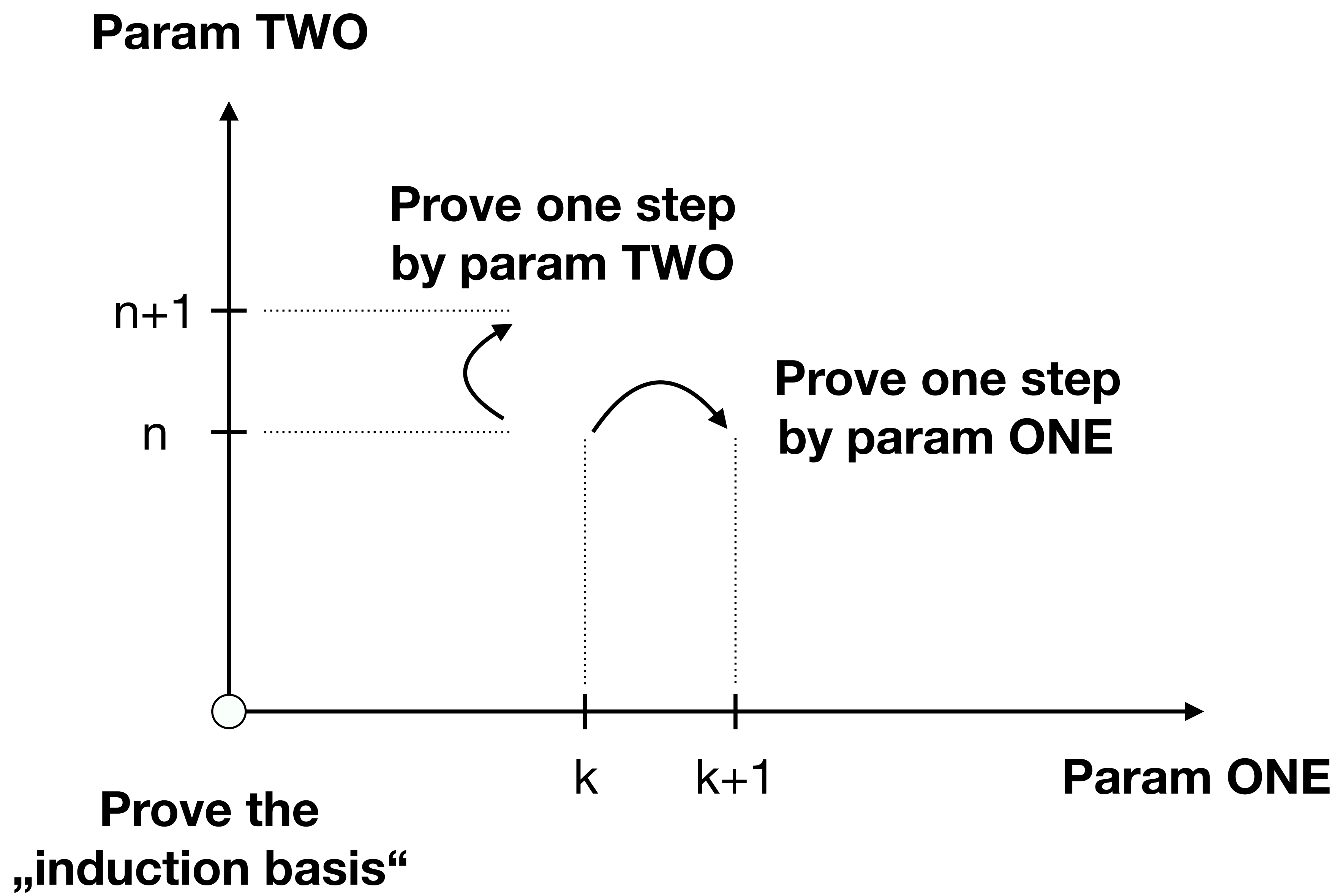
Forward predicate transformation



$$\vec{m}(\varphi) \Rightarrow \varphi$$

If TRUE then method is holding the constraints

Two-dimensional induction



Forward predicate transformation

Induction by the complexity of the constraint formula

1. $\vec{m}((\varphi)) \equiv \vec{m}(\varphi)$
2. $\vec{m}(\varphi \wedge \psi) \equiv \vec{m}(\varphi) \wedge \vec{m}(\psi)$
3. $\vec{m}(\varphi \vee \psi) \equiv \vec{m}(\varphi) \vee \vec{m}(\psi)$
4. $\vec{m}(\texttt{forall}\ \mathbf{x}\texttt{:}\ \varphi(\mathbf{x})) \equiv \texttt{forall}\ \mathbf{x}\texttt{:}\ \vec{m}(\varphi(\mathbf{x}))$
5. $\vec{m}(\texttt{exists}\ \mathbf{x}\texttt{:}\ \varphi(\mathbf{x})) \equiv \texttt{exists}\ \mathbf{x}\texttt{:}\ \vec{m}(\varphi(\mathbf{x}))$

Forward predicate transformation

Induction by the complexity of the method:

6. If $m \equiv$ `u.a:=v` and φ is an atomic formula:

   — If $\varphi \equiv (x.a = y)$, then $\vec{m}(\varphi) \equiv (u = x \wedge u.a = v) \vee$
   $(u \neq x \wedge u.a = v \wedge x.a = y)$

   — If $\varphi \equiv (x.a \neq y)$, then $\vec{m}(\varphi) \equiv (u = x \wedge u.a = v) \vee$
   $(u \neq x \wedge u.a = v \wedge x.a \neq y)$

   — Otherwise $\vec{m}(\varphi) \equiv \varphi \wedge u.a = v$

7. If $m \equiv i_1 \,;\, i_2$, then $\vec{m}(\varphi) \equiv \vec{i}_2(\vec{i}_1(\varphi))$

8. If $m \equiv$ `{`i`}`, then $\vec{m}(\varphi) \equiv \vec{i}(\varphi)$

9. If $m \equiv$ `if` ψ `then` i, then $\vec{m}(\varphi) \equiv \vec{i}(\psi \wedge \varphi) \vee (\neg\psi \wedge \varphi)$

10. If $m \equiv$ `forone` v `where` $\psi(v)$ `do` i, then
    $\vec{m}(\varphi) \equiv$ `exists` $v : \vec{i}(\psi(v) \wedge \varphi)$

Forward predicate transformation

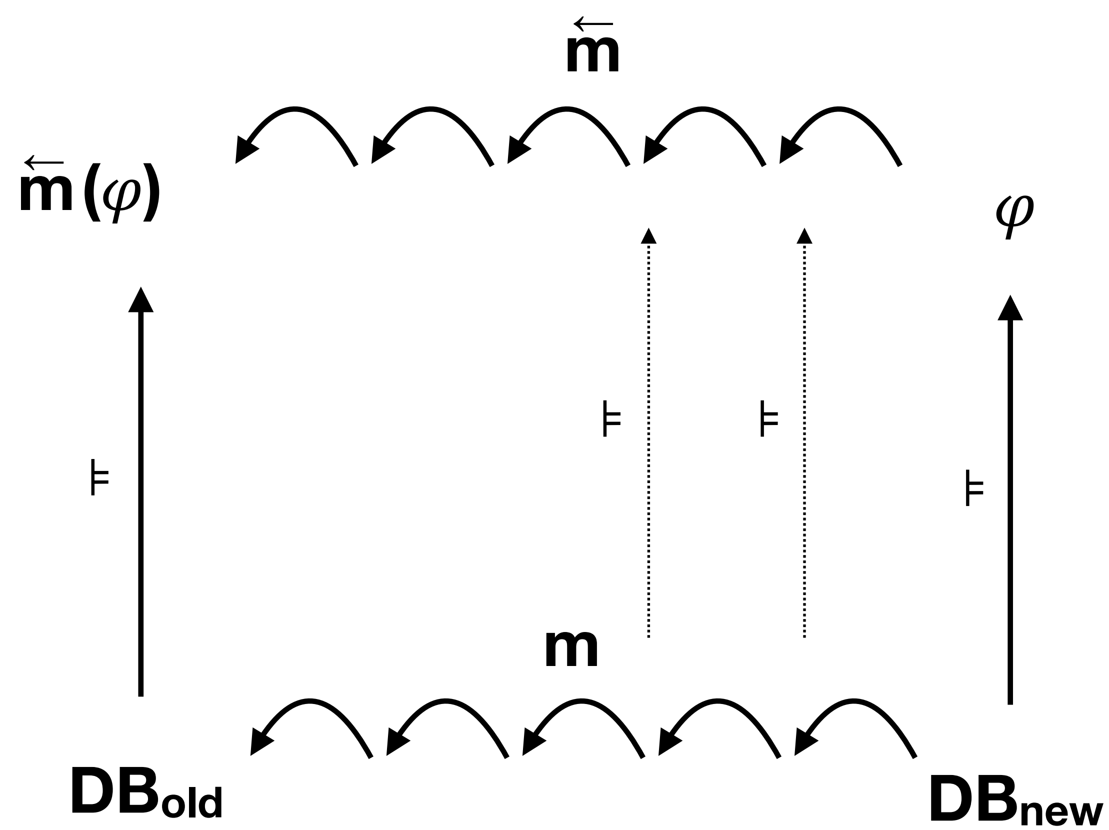
Say the method m is an instruction of the form

```
forall variable where condition do i
```

And say C is a full constraint formula. Then for any formula φ

$$\vec{\mathbf{m}}(\varphi) = \begin{cases} C & \textbf{if} \quad \varphi \Rightarrow C \ \textbf{ and } \ \vec{\mathbf{i}}(C) \Rightarrow C \\ true & \textbf{otherwise} \end{cases}$$

Backward predicate transformation



$\overleftarrow{\mathbf{m}}(\varphi) \Rightarrow \varphi$

If TRUE then method is holding the constraints

Backward predicate transformation

Induction by the complexity of the constraint formula

1. $\overleftarrow{m}((\varphi)) \equiv \overleftarrow{m}(\varphi)$
2. $\overleftarrow{m}(\varphi \wedge \psi) \equiv \overleftarrow{m}(\varphi) \wedge \overleftarrow{m}(\psi)$
3. $\overleftarrow{m}(\varphi \vee \psi) \equiv \overleftarrow{m}(\varphi) \vee \overleftarrow{m}(\psi)$
4. $\overleftarrow{m}(\texttt{forall x: } \varphi(x)) \equiv \texttt{forall x: } \overleftarrow{m}(\varphi(x))$
5. $\overleftarrow{m}(\texttt{exists x: } \varphi(x)) \equiv \texttt{exists x: } \overleftarrow{m}(\varphi(x))$

Backward predicate transformation

Induction by the complexity of the method:

6. If $m \equiv$ `u.a:=v` and φ is an atomic formula:
   — If $\varphi \equiv (x.a = y)$, then $\overleftarrow{m}(\varphi) \equiv (u = x \wedge v = y) \vee (u \neq x \wedge x.a = y)$
   — If $\varphi \equiv (x.a \neq y)$, then $\overleftarrow{m}(\varphi) \equiv (u = x \wedge v \neq y) \vee (u \neq x \wedge x.a \neq y)$
   — Otherwise $\overleftarrow{m}(\varphi) \equiv \varphi$

7. If $m \equiv i_1 ; i_2$, then $\overleftarrow{m}(\varphi) \equiv \overleftarrow{i_1}(\overleftarrow{i_2}(\varphi))$

8. If $m \equiv \{i\}$, then $\overleftarrow{m}(\varphi) \equiv \overleftarrow{i}(\varphi)$

9. If $m \equiv$ `if` ψ `then` i, then $\overleftarrow{m}(\varphi) \equiv (\psi \wedge \overleftarrow{i}(\varphi)) \vee (\neg\psi \wedge \varphi)$

10. If $m \equiv$ `forone` v `where` $\psi(v)$ `do` i,
    then $\overleftarrow{m}(\varphi) \equiv ($`exists` $v : \psi(v)) \wedge \overleftarrow{i}(\varphi)$

Backward predicate transformation

Good news - it allows method auto-correction

method *m* (params) **in class** *K*
{
method_body
}

method *m* (params) **in class** *K*
{
if($\overleftarrow{m}(C)$ **)**
method_body
}

Database Architecture

Including but not limited to…

Data saving

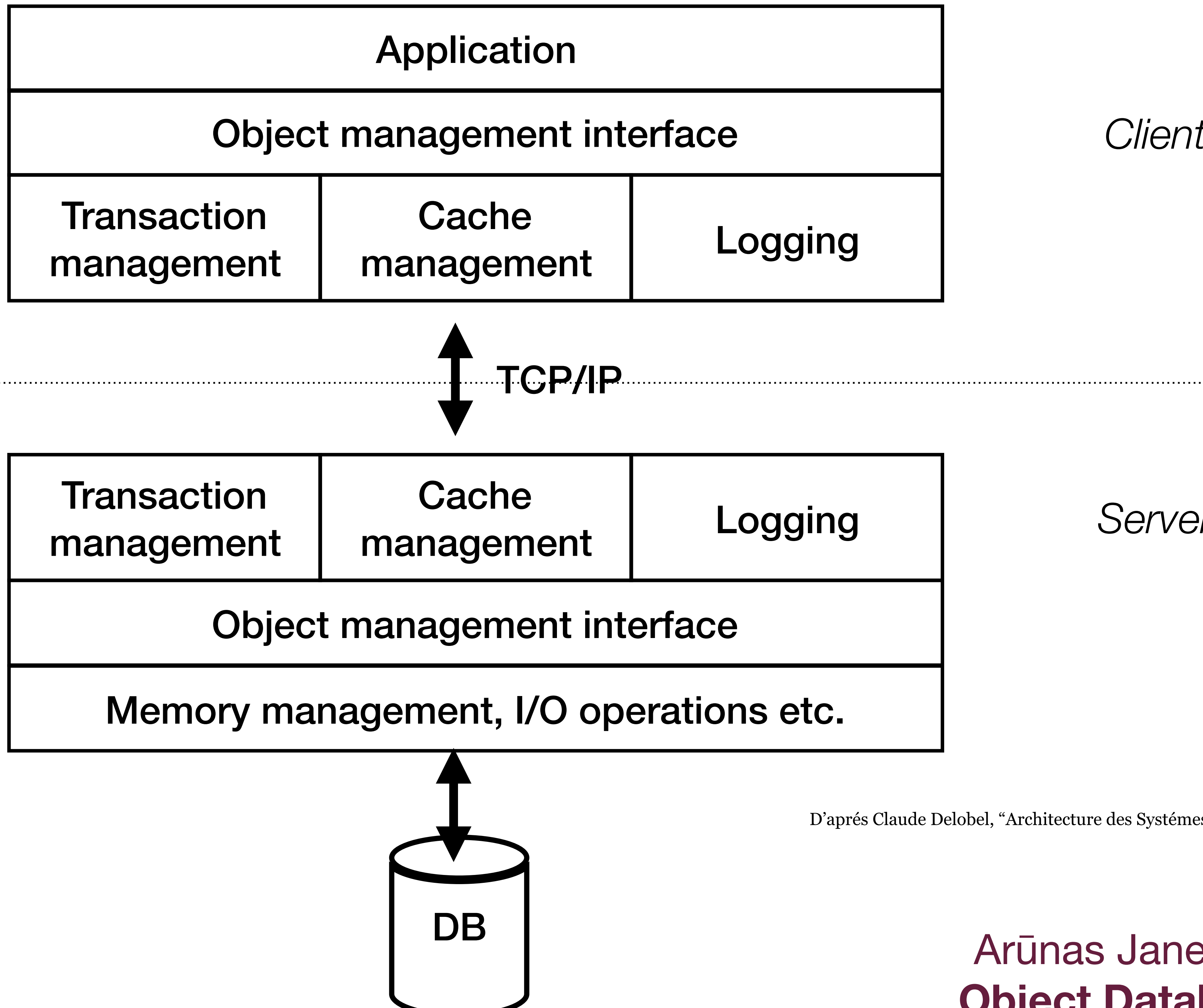
- ODB is an „extension“ to an Object Oriented Programming Language, providing it with data preserving capabilities.
- Then some objects in the program are of „temporal“ nature (to be dismissed after program ends) and some are to be preserved.
- How to know which object is which?

Data saving

3 data saving models:

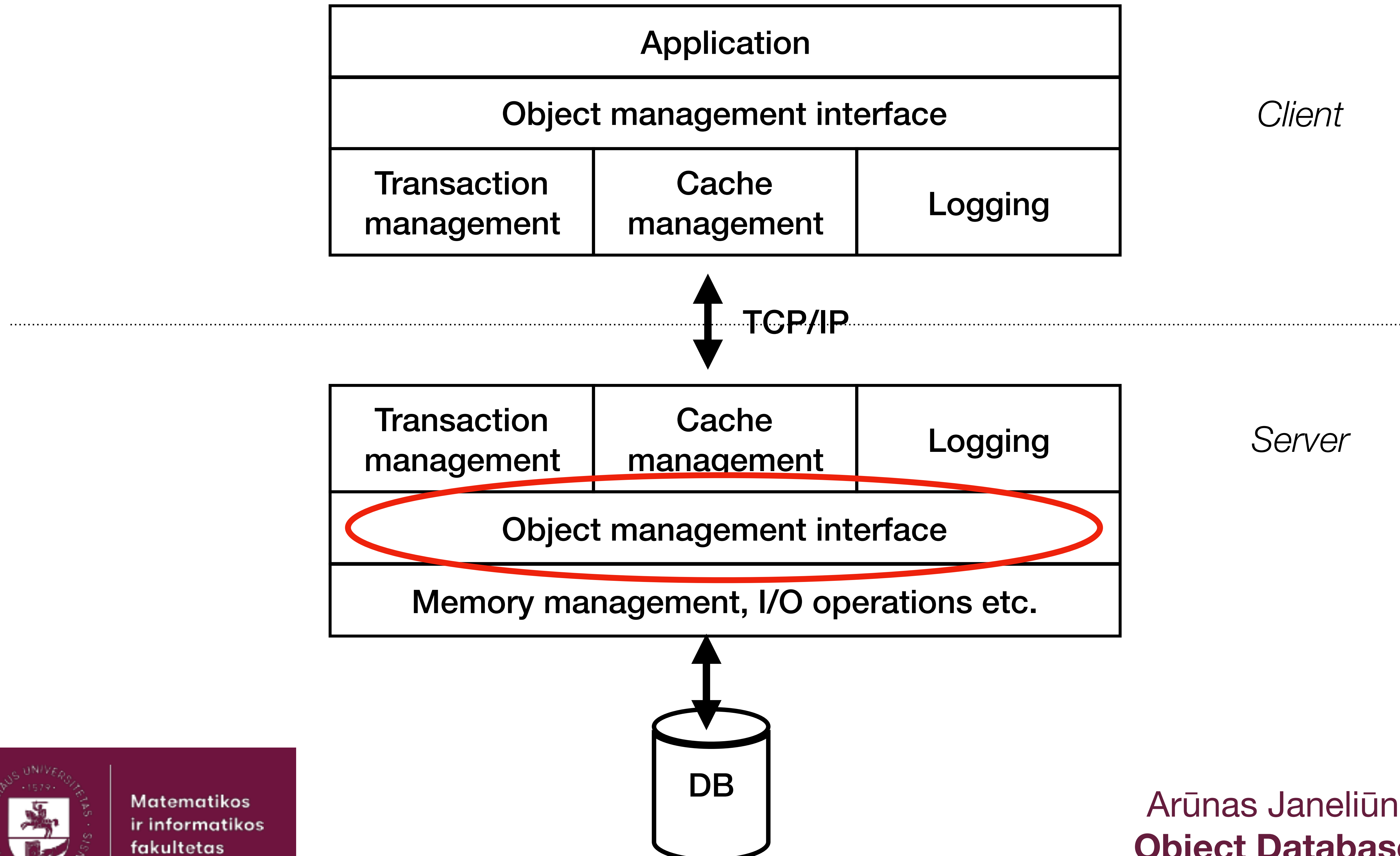
- **Persistent classes.** Some classes are declared to be persistent and every object of that class persists.
- **Persistent *new*.** Objects that are to be stored in the database are created with specific *persistent new* operator.
- **Persistence by accessibility.** We call a „save“ method to the object and then every other object accessible by associations to that object is stored as well automatically.

Client-Server architecture



D'aprés Claude Delobel, "Architecture des Systémes a Objets", 1997

Server knowledge levels



Server knowledge levels

Low knowledge level

Server side knows only the size of an object and it's ID. It regards objects just as identifiable byte arrays.

PROS

- Easily built
- May be applied to various data models

CONS

- Data interpretation may be done only on Client
- Data navigation (and associative access models) cannot be done on server

Server knowledge levels

Medium knowledge level

Server side knows objects structure, but still cannot execute methods on server side.

PROS

- Data navigation (and associative access models) can be done on server
- Query predicates can be evaluated on server
- Query optimisation is possible

CONS

- Query predicates and other sub-queries involving methods can be calculated only on client-side

Server knowledge levels

High knowledge level

Server side knows everything about objects structure and can execute.

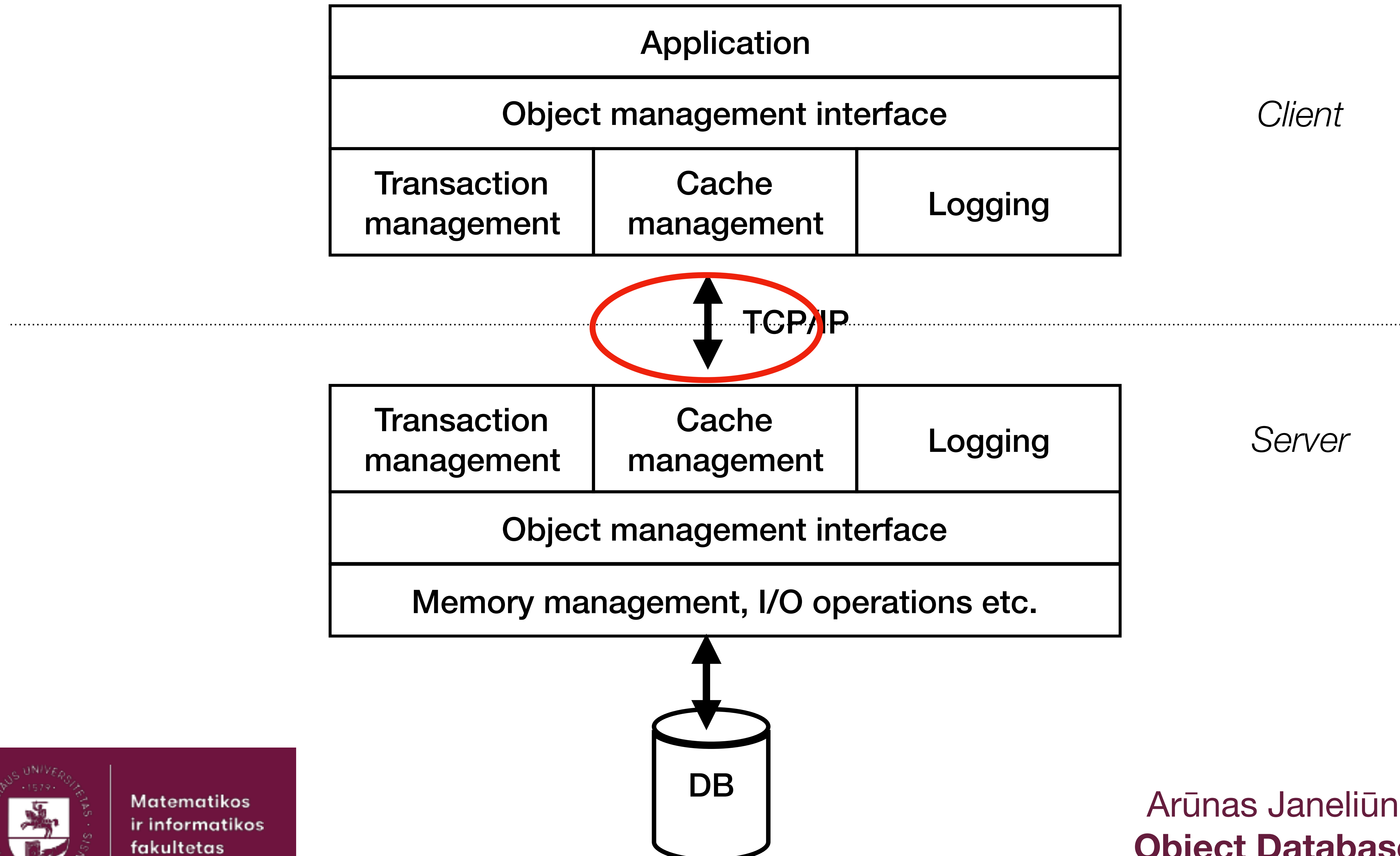
PROS

- Maximum server capabilities
- Possible detection of commutative operations (concurrency control)
- you name it…

CONS

- Hard to implement
- Server „weight“

Server output

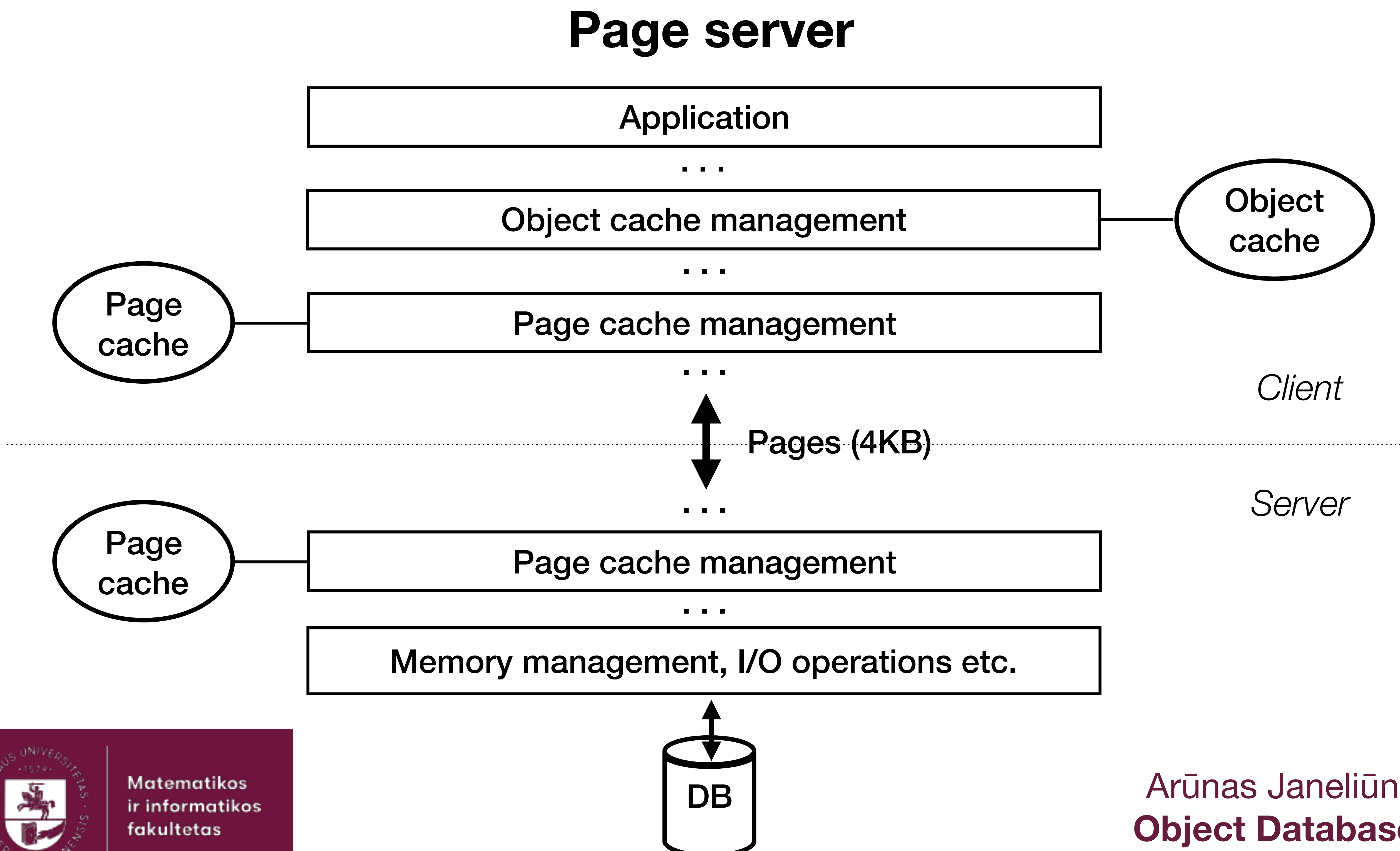


Server output

Page server

Server sends out pages (4KB).

Server output



Server output

Page server

Server sends out pages (4KB).

PROS

- Easier server architecture
- Easier communication
- Good usage of objects grouping and associative access techniques

CONS

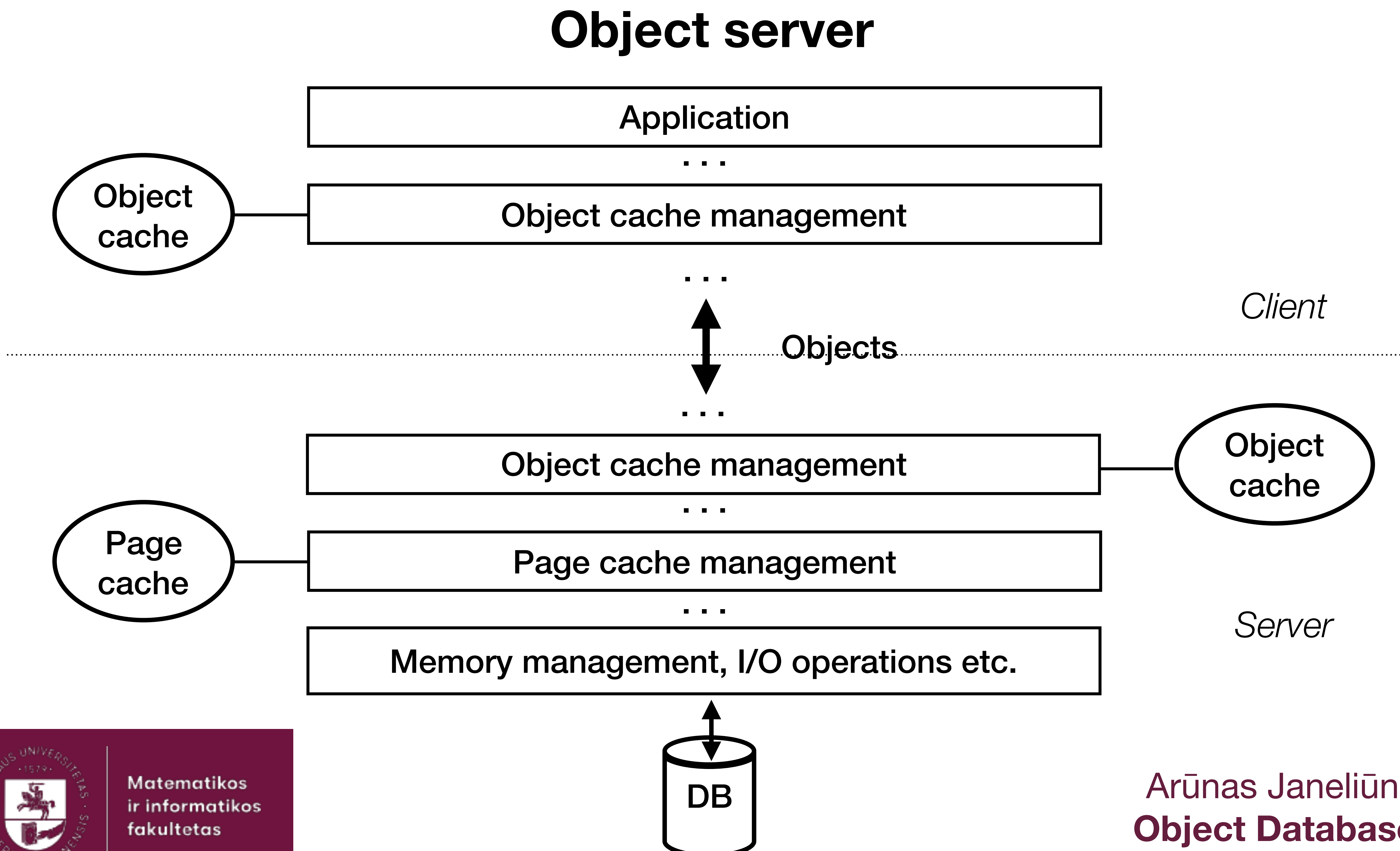
- Busy communication while sending many small objects
- Concurrent access control is set to pages (and every data on the same page, regardless whether it is occupied ATM or not)

Server output

Object server

Server sends out objects.

Server output



Server output

Object server

Server sends out objects.

PROS

- Concurrency control is more sophisticated

CONS

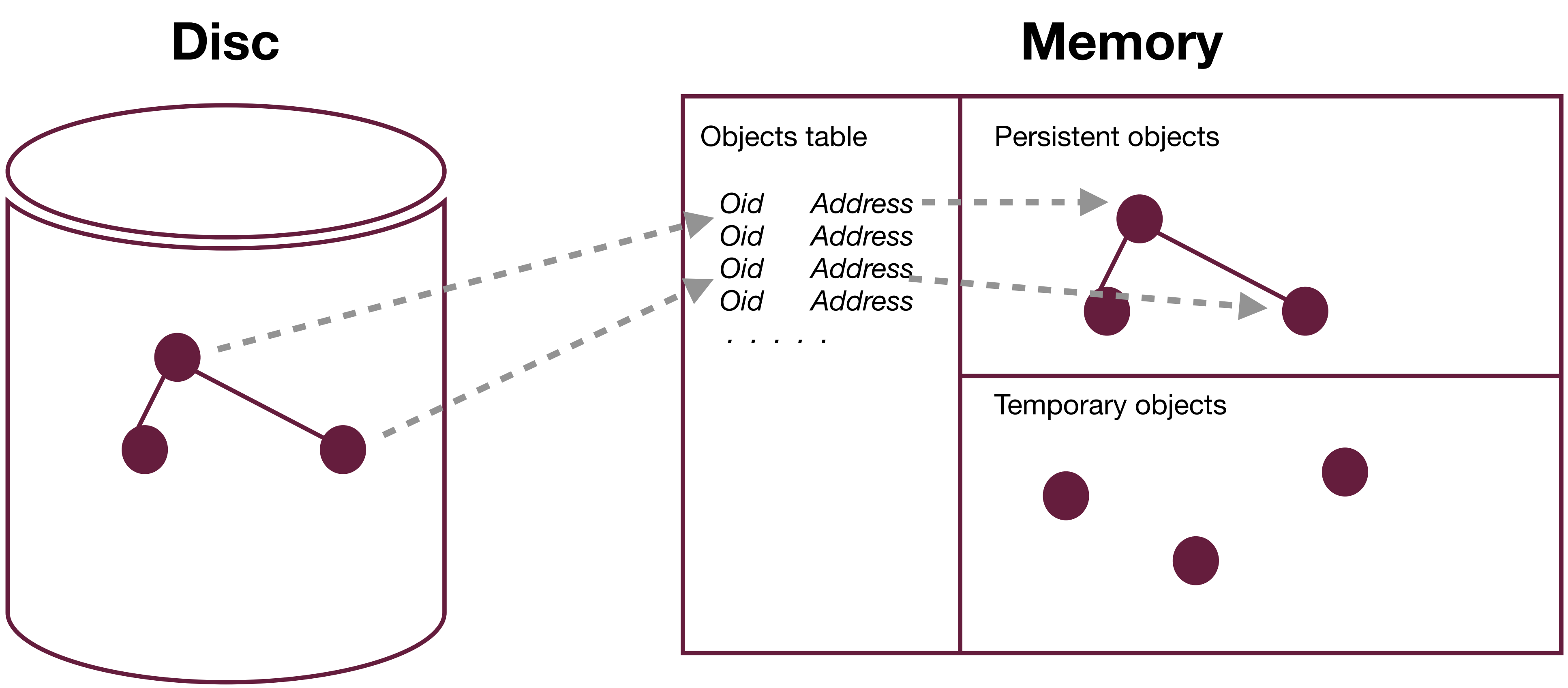
- More communication on checking data size which is sent over
- Ineffective communication when sending small objects
- More complex to implement

Object identification

- Usually object IDs contain information (address) where the object is to be found (in the memory)
- If some objects „live“ both on the server (disc) and application (memory), how their IDs are constructed and supported thren?

Object identification

Disc-oriented addresses



Object identification

Disc-oriented addresses

Each time a persistent object is created:

- the space in the disc is allocated
- *oid* is created for the object
- the entry in the Objects table is created
- only then the object is placed in the memory

Object identification

Disc-oriented addresses

PROS

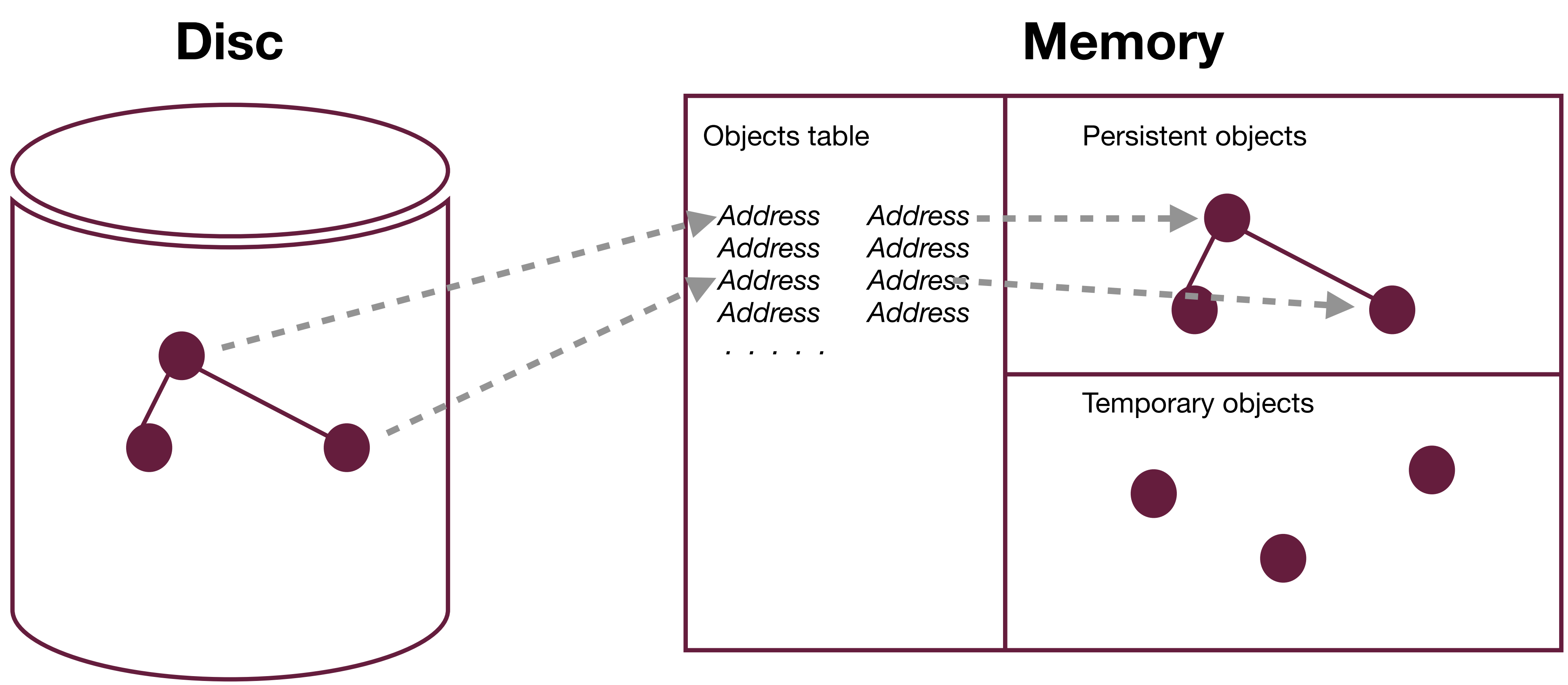
- The storage space in the disc is controlled
- No need to convert *oid*s on copying the object from/to database to the memory

CONS

- The link between objects in the memory is always indirect: you’ve got the *oid* of the other object, then you go to the Objects table, find this *oid* and only then you know where this object is located

Object identification

Two-levels addressing



Object identification

Two-levels addressing

The object addressing is always converted on moving it from/to the memory. Objects in the disc are addressed directly by the address in the disc. If it is copied in the memory, then it stats being addresses by its address in the disc.

But then it means, if you copy the object from the disc to the memory, you should fill in all its links to the other objects by their correct address in the memory. For that you should copy the linked object to the memory to get a „memory address“ for it. And so on, and so on… Possible chain reaction

Object identification

Disc-oriented addresses

PROS

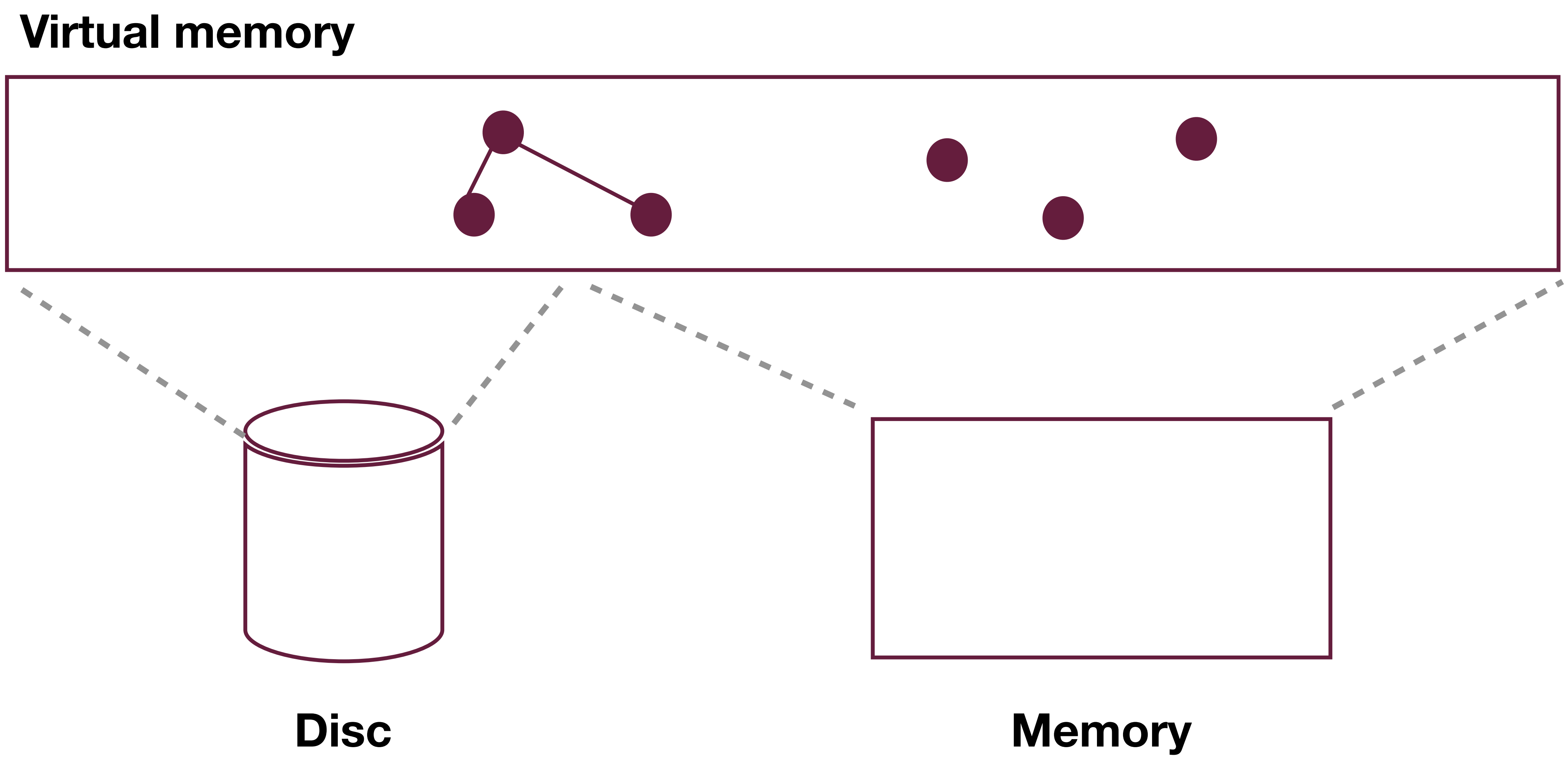
- The storage space in the disc is controlled
- Programming language can have no idea where objects are comming from, they all are linked the same.
- The link between objects is quick.

CONS

- Costly copying objects from the disc to the memory and *vice versa*.

Object identification

Single-level addressing



Object identification

Single-level addressing

All objects, regardless their „physical" location, are operating in a single Virtual Memory. And the mapping function behind it is responsible to maintain this virtuality.

How temporary objects are to be distinguished in such a Virtual Memory?

What about Virtual Memory Fragmentation?

Objektai su rolėmis

Kas tai yra?

```
john := new Person(“John”,“Smith”,1987)

johnAsAthlete := in Athlete (john,
             {sport:“tennis”, code:123})
```

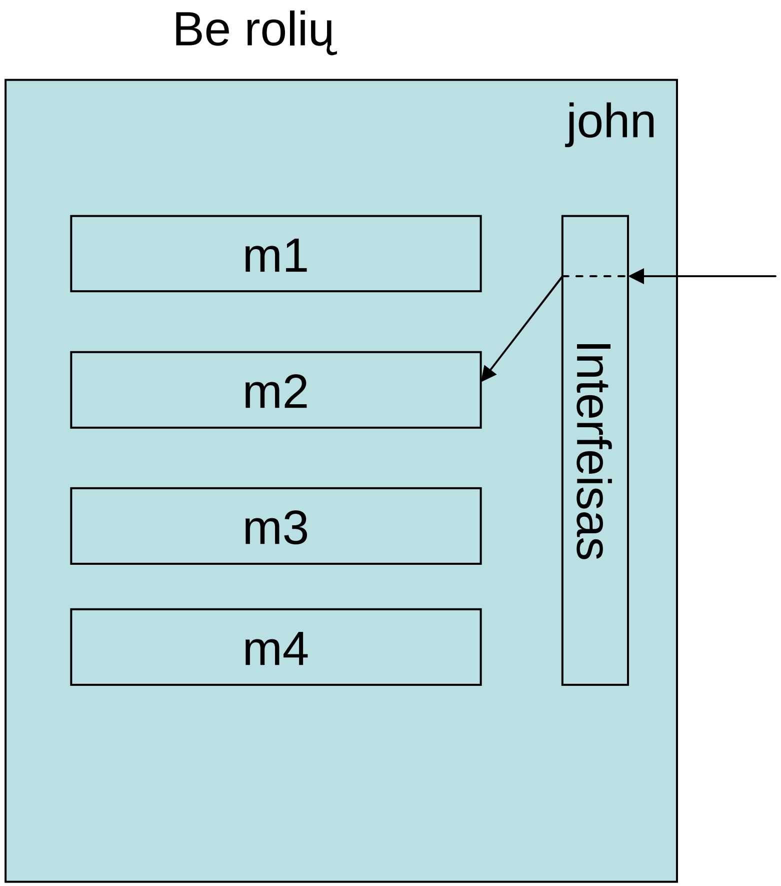
dabar:

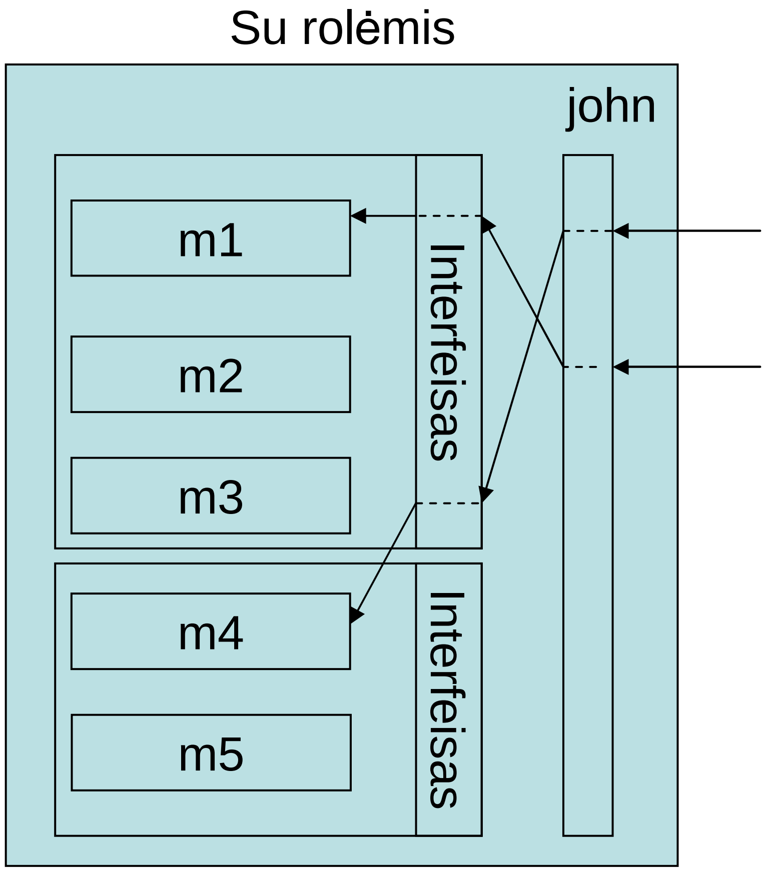
```
john.introduceMyself()
```

VIENODI!

```
johnAsAthlete.introduceMyslf()
```

Kaip tai daroma?





Kas kaip atsako?

```
johnAsStudent := in Student (john,
      {faculty:“Science”, code:”333”})


johnAsStudent.code    - string

johnAsAthlete.code    - int

john.introduceMyself() -  “I’m Student”
```

Kaip pasiekti norimą rolę?

`john!introduceMyself()`

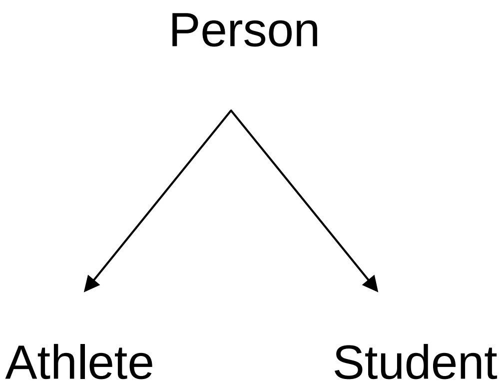
Downward lookup – ieškoma nuo objekto klasės **žemyn** iki tinkamos rolės

Upward lookup – ieškoma aukštyn klasių hierarchijoje, kol randama pirmoji implementacija

. yra *double lookup (downward,* po to *upward)* operatorius

! yra *upward lookup* operatorius

Kas esu AŠ?




```
toString() {
   self.introduceMyself();
}
```

```
john.toString()              “Studentas”
john!toString()              “Asmuo”
johnAsAthlete.toString()     “Atletas”
```

Ištriname rolę

```
drop Student (john)
```

```
myCar := new Car (johnAsStudent, “Ferrari”);
drop Sudent(john);
myCar.owner    ?????
```

Rezultatas priklausys nuo realizacijos

Virtualūs objektai

Kas tai yra?

Tai objektai sukurti ne naudojant klasę, o iš kito objekto, padedant 4 operatoriams:

- **`project`**
- **`rename`**
- **`extend`**
- **`times`**

Kas tai yra?

```
johnView := john project (name, address, introduceMyself);

johnView.surname                                        error
johnView.name = john.name      jie ir keičiasi kartu!
```

Kas tai yra?

```
johnView := john rename (introduceMyself()=>whoAreYou());

johnView.whoAreYou() = john.introduceMyself();  //true
```

Kas tai yra?

```
johnView := john extend (spouse: Person);


Gali būti naudojamas ir perrašyti savybes:

johnItaliano := john extend (
                  introduceMyself := meth():string {
                      return “Me chiamo ” + me.Name;
                  }
                )
```

Kas tai yra?

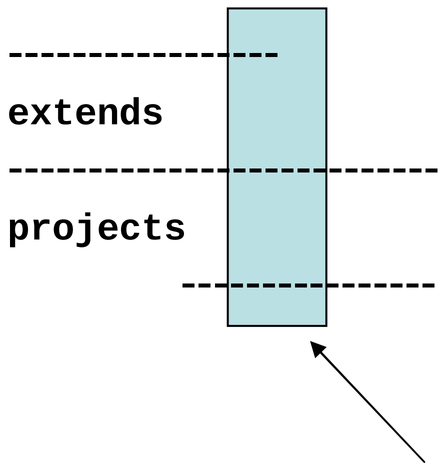
```
johnAndCompany := john times company;
```

Gausime objektą, turintį ir Asmens, ir Darbovietės savybių.

Jeigu yra atributų/metodų vienodu vardu – vieną kurį prieš tai reikia pervadinti (**rename**)

Kas gaunasi?

Gauto objekto klasė NEBŪTINAI yra pirminio objekto klasės poklasis:



Bendra dalis tik tokia

Kas esu AŠ?

`me` ir **`self`**

`self` yra rolė, kuri PRIIMA pranešimą

`me` yra rolė, kurioje jis parašytas

Kas esu AŠ?

```
johnView := john extend (
                newMethod := meth:void {
                   return me.age;
                }
           ) project (introduceMyself(), newMethod())
```

Naudoti **`self`** čia būtų klaida

O dabar dar viską suderinkit su metodo iškvietimo operatoriumi **`!`** ☺

Virtualios klasės

Kas tai yra?

```
classview Adults in Persons
      where ( CurrentDate.Year() – this.BirthYear > 17 )
      store ( char* Phone := “” )
      compute ( newMethod := meth:void {
                  return this.Phone;
                } )
      import ( Name )
```

`where` – uždeda tam tikrą aprobojimą.

`store` – sukuria naują atributą, nesantį klasėje **`Persons`**.

`compute` – sukuria naują metodą, nesantį klasėje **`Persons`**

`import` – iš klaės **`Persons`** “importuojami” atributai/metodai

Sąsajos su virtualiais obj.

Visa tai galime padaryti prieš tai matytais “virtualių objektų operatoriais” **`extend`**.
ir **`project`**

Kas tai yra?

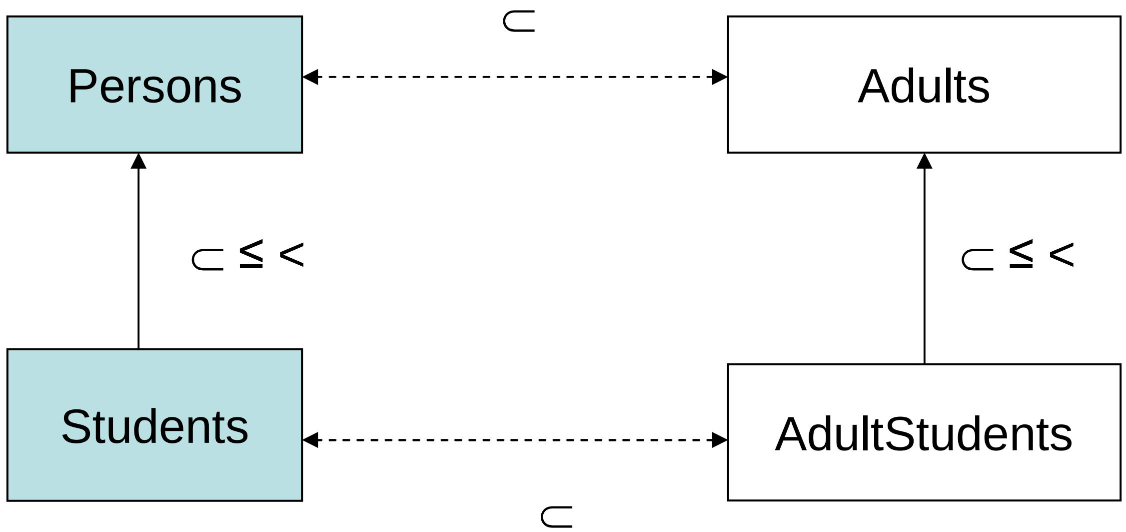
```
classview AdultStudents:Adults in Persons
       where ...
       store ...
```

Virtuali klasė `AdultStudents` yra virtualios klasės `Adults` poklasis ir yra padaryta iš (*based on*) realios klasės `Students`.

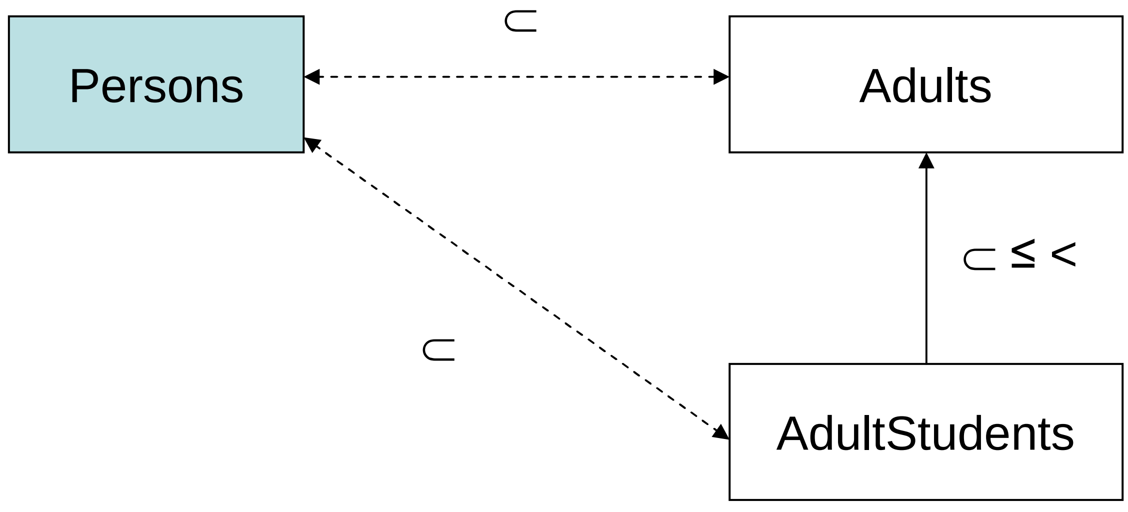
`AdultStudents` paveldi visus **`where`**, **`store`**, **`compute`** ir **`import`** veiksmus iš virtualios klasės `Adults`.

Kuo skiriasi ryšiai

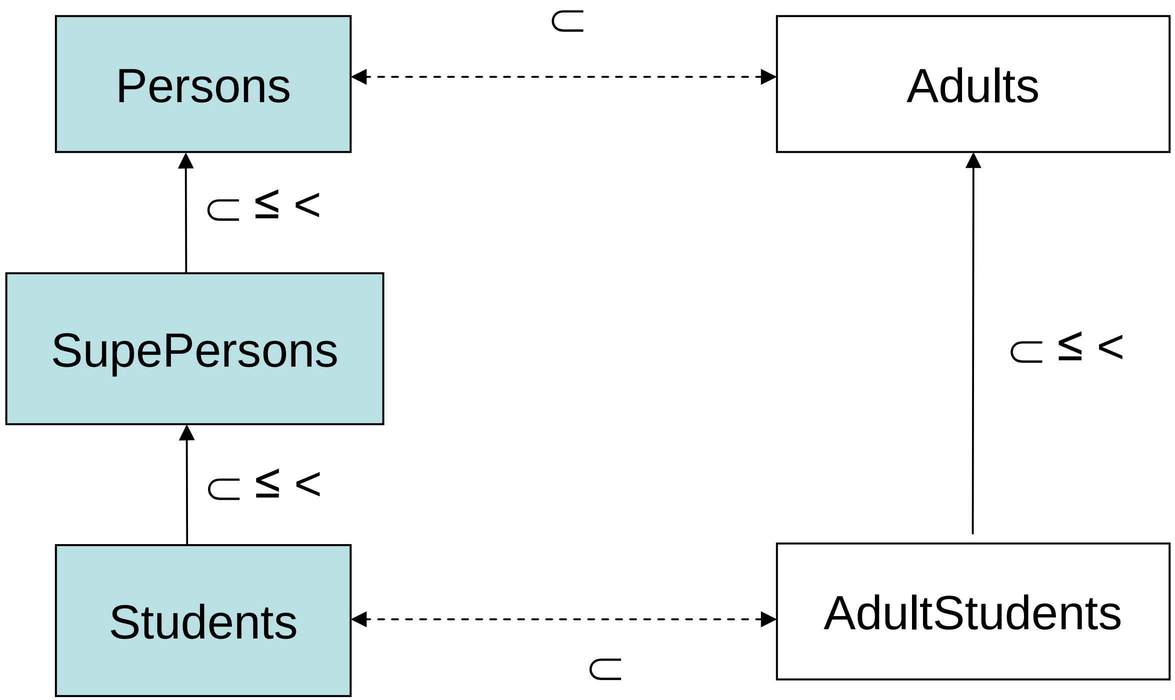
Ryšys *based on* yra tik “aibės/poaibio” ryšys. O virtualių klasių/poklasių ryšys yra ir aibės/poaibio, ir tipo/potipio ir klases/poklasio ryšys.



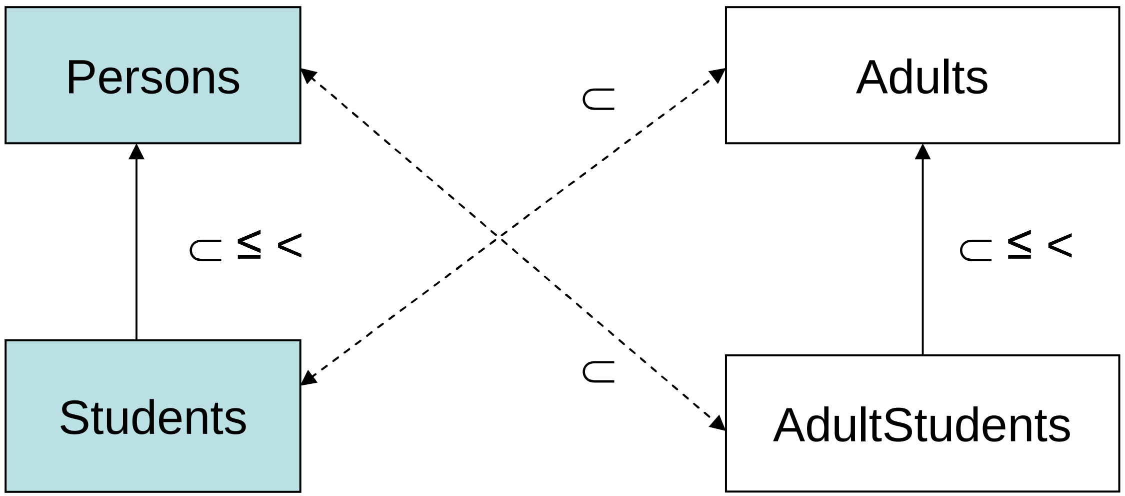
Kaip dar gali būti?



Kaip dar gali būti?



Kaip dar gali būti?



Kaip dar gali būti?