VIRTUAL CLASSES

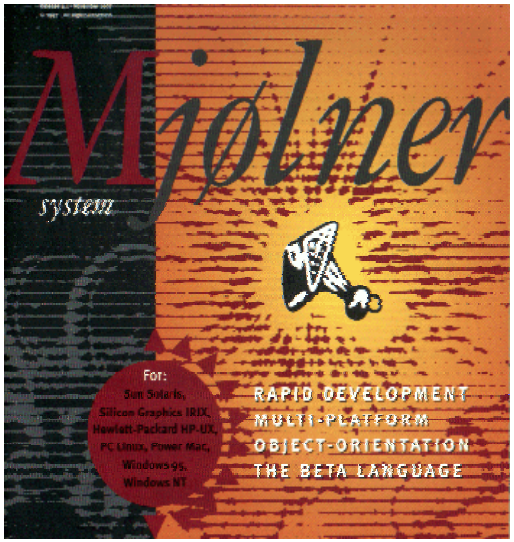
and their Implementation

Ole Lehrmann Madsen
Aarhus University
Alexandra Institute A/S
Mj¿lner Informatics A/S

Contents

Origins of BETA

- Based on the Scandinavian approach to object-orientation
  - Successor to Simula
- The Mj¿lner System
  - Development environment for BETA
- Beta was originally developed by:
  - Bent Bruun Kristensen
    Ole Lehrmann Madsen
    Birger M¿ller-Pedersen
    Kristen Nygaard
- Important contributors to BETA and the Mj¿lner System
  - J¿rgen Lindskov Knudsen, Elmer Sandvad, Peter Andersen and many more



Design goals for BETA

- One language for design and implementation
  - Good modeling language
  - Efficient programming language
- Simple and general
  - Scheme, Smalltalk, Self
- Statically typed
  - As much as possible
- Good support for structuring and abstraction
- Unified paradigm
  - Neither pure OO or multi-paradigm

Abstraction in BETA

- **Conceptual framework**
  - **The conceptual means for organizing and understanding phenomena and concepts**
  - **Independent of formal languages**
  - **Should be richer than the formal languages**
- **BETA pattern**
  - **Unification of class, procedure, type, exception, ...**
  - **Towards the ultimate abstraction mechanism**
  - **Concepts represented as patterns**
  - **Phenomena represented as objects**

Benefits of object-orientation

- **Unifying perspective on major elements of software development**
  - **analysis, design, implementation**
  - **data bases, object distribution**
- **Good support for modeling**
  - **programs reflect reality in a natural way**
- **Good support for extensibility & reuse**
  - **class, subclass, virtual, ...**
  - **frameworks, binary components, design-, architectural patterns, ...**

2. Unification of abstraction mechanisms

- **Abstraction mechanism:**
  - **A pattern for creating instances**
- **Class :**
  - **A pattern for creating objects**
- **Procedure:**
  - **A pattern for creating event-sequences**
- **Function, interface, process type, generic class, exception type,** É
  - **Patterns for creating various kinds of instances**

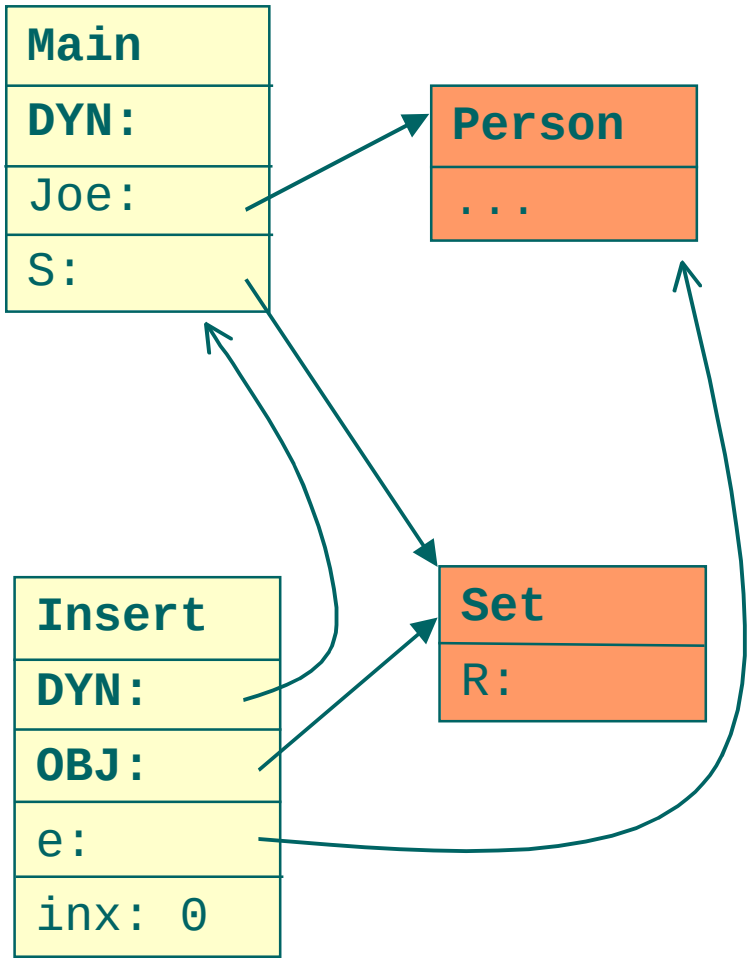
The pattern

In BETA, class, procedure, and other abstraction mechanisms have been unified into

- The Pattern
  - Uniform treatment of all abstraction mechanisms:
    - Procedure, class, ...
  - Class-inheritance, procedure-inheritance, ...
  - Virtual procedure, virtual class, É
  - Nested procedure, nested class, É
  - Procedure variable, class variable, ...

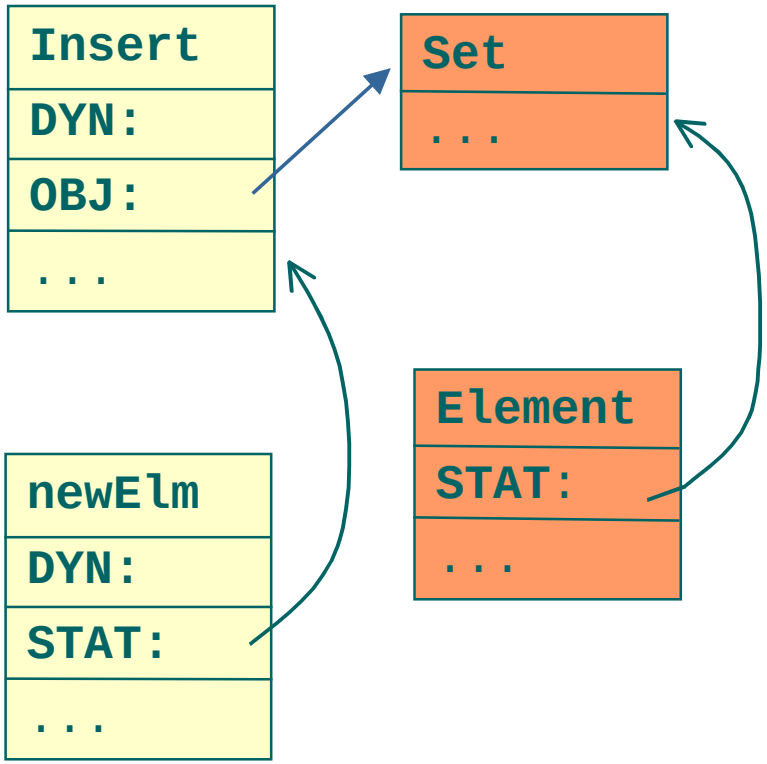
Procedure activations & objects

```
class Set: { ...
  proc insert(e: ref object):
    { inx: obj integer
    do É; e →R[inx]; É;
    };
  R: array of object
 };
proc main():
  { Joe: ref Person;
    S: ref Set;
  do new Person → Joe;
     new Set → S;
     S.insert(Joe);
  }
```



Nested classes & procedures

```
class Set: { ...
   class Element: {
      elm: ref object;
      next: ref Element;
    }
   proc insert(e: ref object):
    { proc newElm(e:É): { É }
    do É
    };
};
```



OBJ: object-reference for procedure activation

STAT: enclosing object for object or activation

Objects and procedure activations

Object:

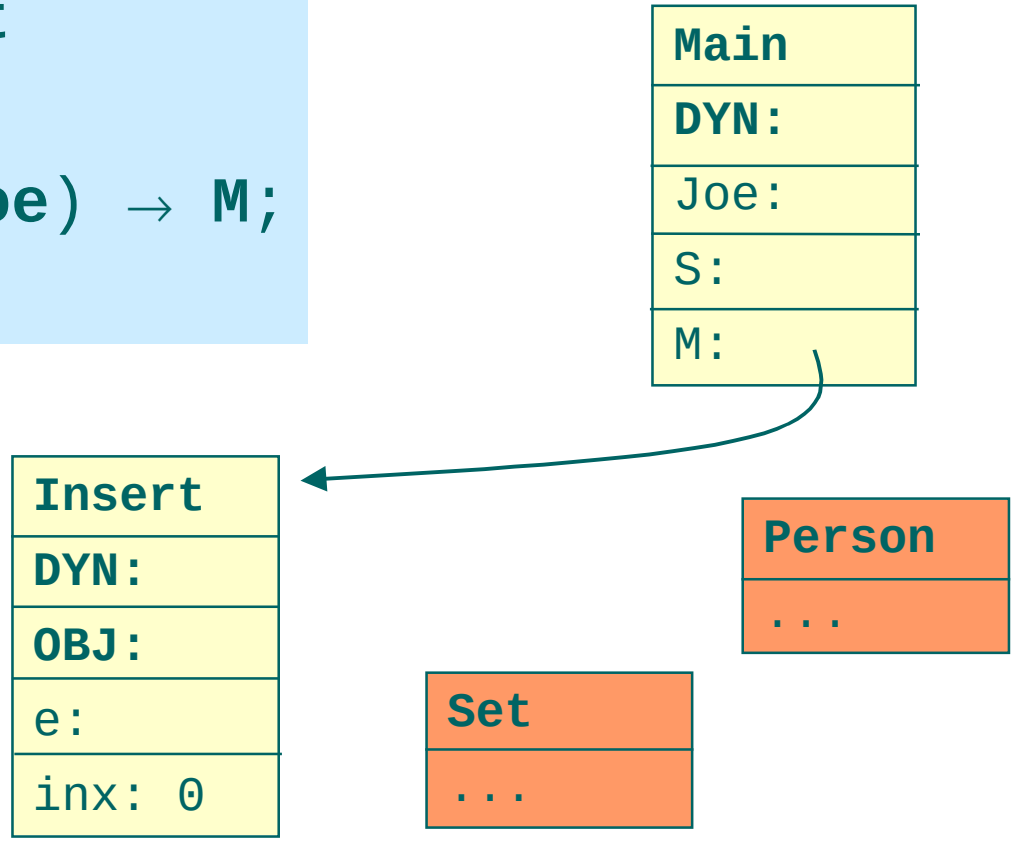
- **Data-items: ...**
- **Static link: enclosing object/procedure**

Procedure-activation:

- **Data-items:**
  - Parameters
  - Local variables
- **Static link: enclosing object/procedure**
- **Dynamic link: caller**

Procedure call

```
M: ref R.insert

new R.insert(Joe) → M;
M.execute
```



Pattern: class & procedure

```
Person: { É };
Set: {
   insert(e: ref object):
       { É do É };
   É
 };
main():
  { Joe: ref Person;
    S: ref Set;
  do new Person → Joe ;
     new Set → S ;
     S.insert(Joe);
  }
```

Static link: if nesting

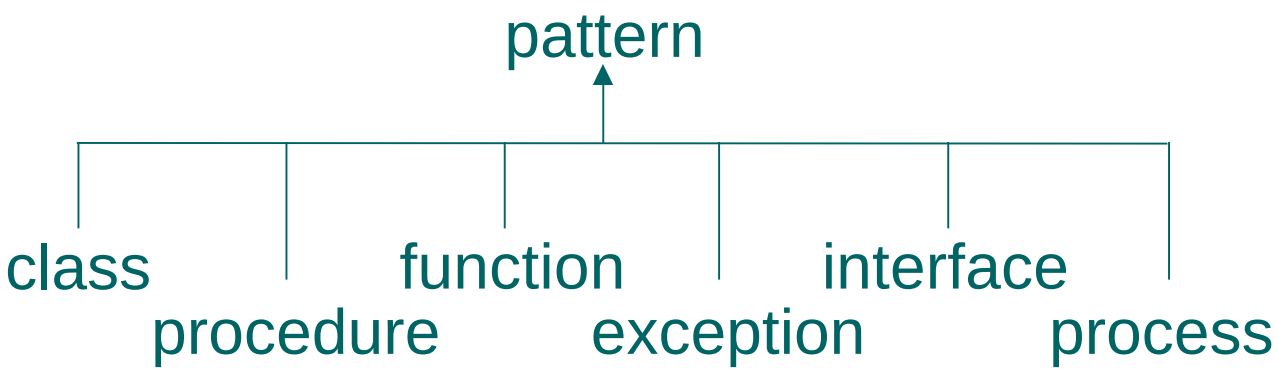
Dynamic link: if do-part

Parameters

Data-items

Object layout
STAT:
DYN:
parameters
data-items

Uniform abstraction mechanisms



The **Pattern**

- The ultimate abstraction mechanism
- A generalization/unification of class, procedure,function, exception, interface, process, generic class, ...

Benefits of the unification

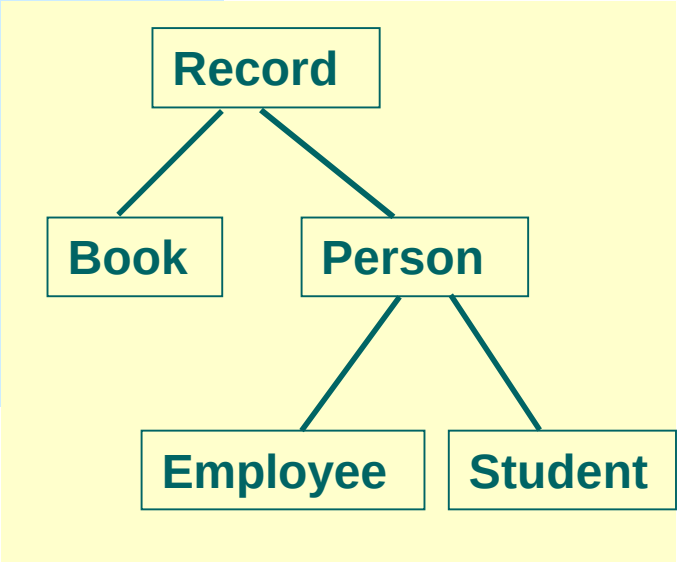
	class	procedure	exception	process	....
Pattern	+	+	+	+	+
Subpattern	+	+	+	+	+
Virtual pattern	+	+	+	+	+
Pattern variable	+	+	+	+	+
Nested pattern	+	+	+	+	+

Examples of new possibilities

- Subpattern **: classification of actions - inheritance for procedures using** inner
- Virtual pattern**: OO genericity**
- Pattern variable**: procedures & classes as first class values; higher order procedures & classes**
- Nested pattern**: enhanced modeling capability; object with several interfaces**

Subclass: classification of objects

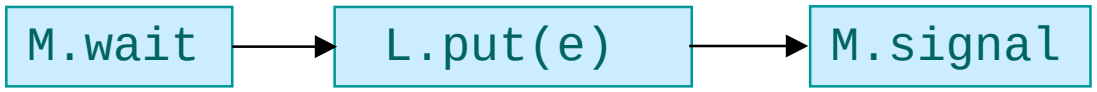
```
Record: { Key: ref Identification };

Book: Record
  { Author: ref Person; Title: ref Text };

Person:  Record
  { Name: ref Text; Sex: ref SexType };

Employee: Person
  { Salary: obj Integer;
     Position: ref PositionType
  };

Student: Person
  { Status: ref StatusType };
```



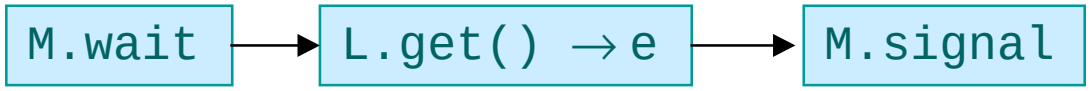
Action sequence generalization

```
L: obj Set;
M: obj Semaphore;

Put(e: obj integer):
   {
   do M.wait;
      L.put(e);
      M.signal
   };

integer Get():
   {
   do M.wait;
      L.get() → e;
      M.signal
      return e
   }
```

Put:



Get:

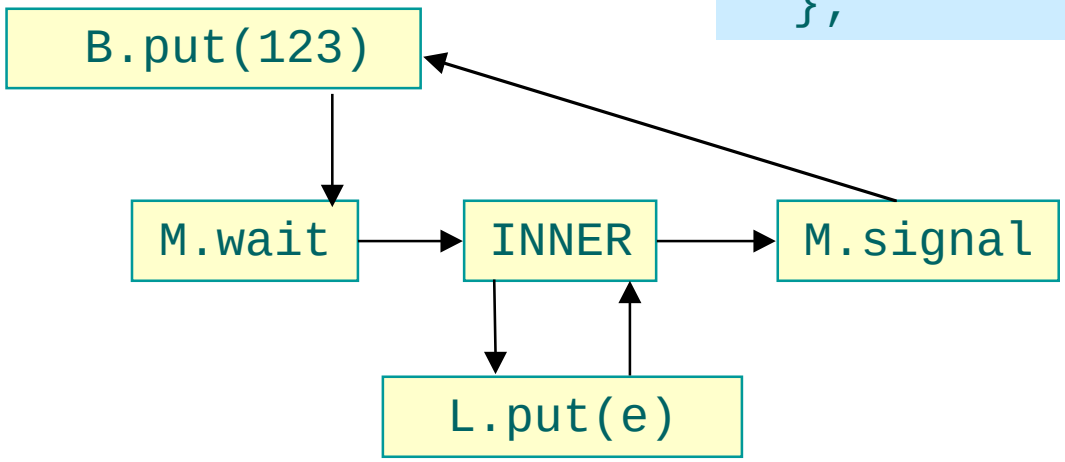


Generalized event sequence:

Procedure specialization

```
Buffer: monitor
  { L: obj Set;
     put(e: obj integer):
       entry
       {
       do L.put(e)
       };
      ...
   };
B: obj Buffer
```

```
Monitor:
  { M: obj semaphore;
     entry:
        {
        do M.wait;
           INNER;
           M.signal
        };
     ...
  };
```



Inheritance characterization

P. Wegner, S. Zdonik:

Relation between a subclass and its superclass

- **Name compatible**
- **Signature compatible**
- **Structurally compatible ← BETA**
- **Behaviorally compatible**

3. Virtual pattern

- Virtual procedure
- Virtual class

Virtual class:

- A language construct for defining generic classes
- Alternative to:
  - Parameterized class
    - Template class in C++
    - ÒGenericÓclass in Eiffel

Set of objects

```
class Set:
 { proc insert(E: ref object):
      {
      do top+1 → top;
         E → R[top]
      };
   proc display:< {};
   R: array[100] ref object;
   top: obj integer
 }
```

:<
describes that **display** is a virtual procedure

Virtual procedure binding

```
class StudentSet: Set
 { display()::<
    { current: ref Object
    do (for i: top repeat
            R[i] → current;
            (Student)current.print()
       for)
    };
}
```

::<
virtual procedure binding of **display**

Cast to **Student**

Generic class Set

:<
describes that **element** is a virtual class

```
class Set:
 { class element:< object;
   proc insert(E: ref element): {...};
   proc display():< {};
   R: array[100] ref element;
   top: obj integer
 }
```

StudentSet: subclass of Set

```
class StudentSet: Set
 { element::< Student;
   display()::<
    { current: ref element
    do (for i: top repeat
            R[i] → current;
            current.print()
       for)
    };
}
```

`::<` binding of **`element`** to **`Student`**

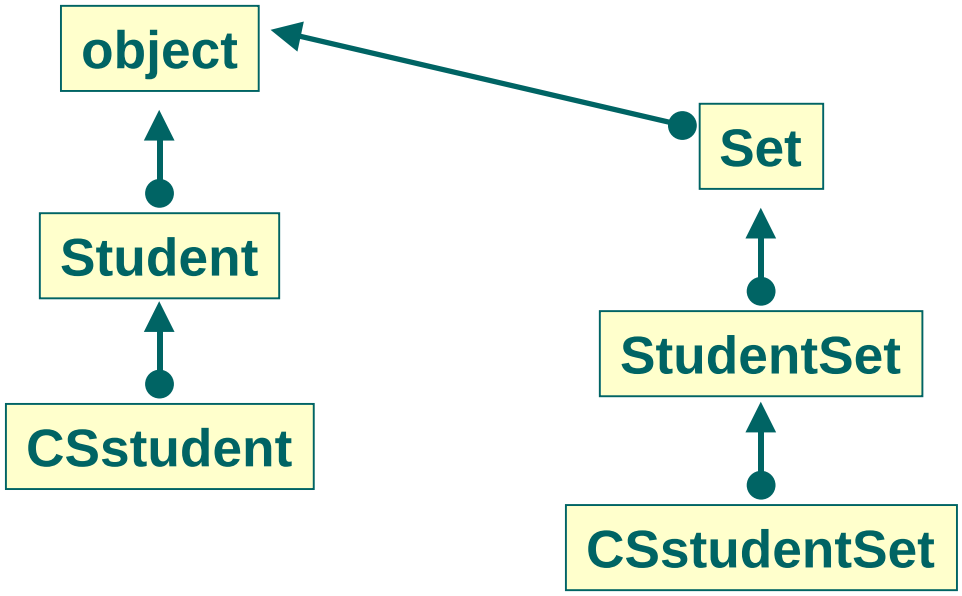
`::<` virtual procedure binding of **`display`**

`current` is of type **`element`**

No cast:
`current: Student`
has a **`print`** operation

Further specialization

```
class CSstudentSet: StudentSet
 { element::< CSstudent;
 }
```



Local virtual class pattern

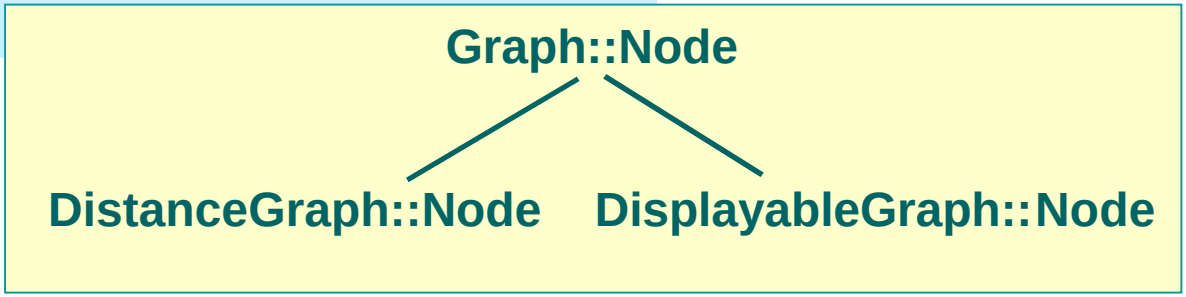
```
Graph:
 { Node:< { connected: obj boolean };
   Link:< { source,dest: ref Node };
   Connect(S,D: ref Node ):<
      {L: ref Link
      do new Link → L;
         S → L.source;
         D → L.dest;
         true → S.connected → D.connected;
         inner
      }
 }
```

Local virtual class pattern bindings

```
DisplayableGraph: Graph
  { Node::< { dispSymb: ref DisplaySymbol };
    Link::< { dispLine: ref DisplayLine };
    Connect(DL: ref DisplayLine)::<
       {
       do DL → L.dispLine; INNER
       };
    Display:< { ... }
   }
```

Extending virtual class patterns

```
DistanceGraph: Graph
  { Node::< { name: ref text };
    Link::< { distance: obj integer };
    Connect(dist: obj integer)::<
        {
        do dist → L.distance; INNER
        };
  }
```



Virtual class versus parameterized class

- **Can both express the same basic elements**
- **Virtual class:**
  - **A simple extension to OO languages**
  - **Analogous to virtual procedure**
  - **Class = type**
  - **A generic class is a ÓrealÓ class**
  - **StudentSet is a subclass of Set**
  - **Covariance + final bindings**
  - **Mutually recursive classes**

Summary of virtual patterns

```
A:
 { V:< Q;
   W:< P{ ... };
 };

AA: A
 { V::< QQ;
   W::< { ... }
 };

AAA: AA
 { V:: QQQ
   W:: { ... };
 }
```

Virtual declarations

Further bindings

Final bindings

Virtual procedure & class patterns

```
A:
 { V:< Q;
   W:< P{ ... };
 do V(e1,e2,e3) → x;
    W(e1,e2,e3) → x;
 };
```

Virtual procedure pattern:

Instances are created and executed

```
A:
 { V:< Q;
   W:< P{ ... };
   S: ref V;
   R: ref W
 do new V → S;
    new W → R
 };
```

Virtual class pattern:

Instances are created and assigned to variables

4. Typing issues

- **Covariance**
- **Reverse assignment**
- **Static typing**

Covariance

```
Point:
 { ThisClass:< Point;
    x,y: obj integer;

    equal(p: ref ThisClass):<
     { eq: obj boolean
     do (x = p.x) and (y = p.y) → eq;
        inner;
        return eq
     }
 }
```

Emulation of thisClass using virtual pattern

ColorPoint

```
ColorPoint: Point
  { ThisClass::< ColorPoint;
     c: obj Color;

     equal::<
       {
       do (eq and (c = p.c)) → eq;
          inner
       }
  }
```

Further binding of thisClass

Type cases

Using subtype substitutable variables (dynamic references)

```
p1,p2: ref Point;  c1,c2: ref ColorPoint;

new Point → p1; new ColorPoint → c1;
new Point → p2; new ColorPoint → c2
```

1.	p2.equal(p1)	OK	run time
2.	c2.equal(c1)	OK	run time
3.	p1.equal(c1)	OK	run time
4.	c1.equal(p1)	NON OK	run time

Reverse assignment

```
p1: ref Point; c1: ref ColorPoint; e: ref c1.equal
```

c1 → p1 OK - compile time
p1 → c1 OK? - run time - reverse assignment

```
c1.equal(p1)
```

```
new c1.equal → e;
p1 → e.p;
e();
```

```
Point:
  { ThisClass:< Point;
    equal(p: ref ThisClass):<
       {
       do ...
          return eq
       };
      ...
   }
```

Desirable properties

At most 2 possible:

1. **Subtype substitutability**
2. **Covariance**
3. **Static type checking**

4. **Class = type**

   subclass = subtype
5. **Mutable state**
6. **Message send**
7. **Modular type checking**

Cases for static typing

- Type guards
- Exact types
- Final bindings
- No invocation of virtual procedure with virtual arguments
- Nested virtual class patterns
  - class families

Further work on virtual classes

- Erik Ernst
- Kresten Krab Thorup
- Mads Torgersen
- A. Igarashi
- Benjamin Pierce
- David Shang
- Kim Bruce
- Martin Odersky
- Philip Wadler
- And others

- Virtual types in Java
- Unifying genericity
- GBETA
  - a generalization of BETA
- Alternatives to virtual classes
- Mathematical foundation
- Block structure

- ECOOP
- OOPSLA
- FOOL
- ...

5. Implementation issues

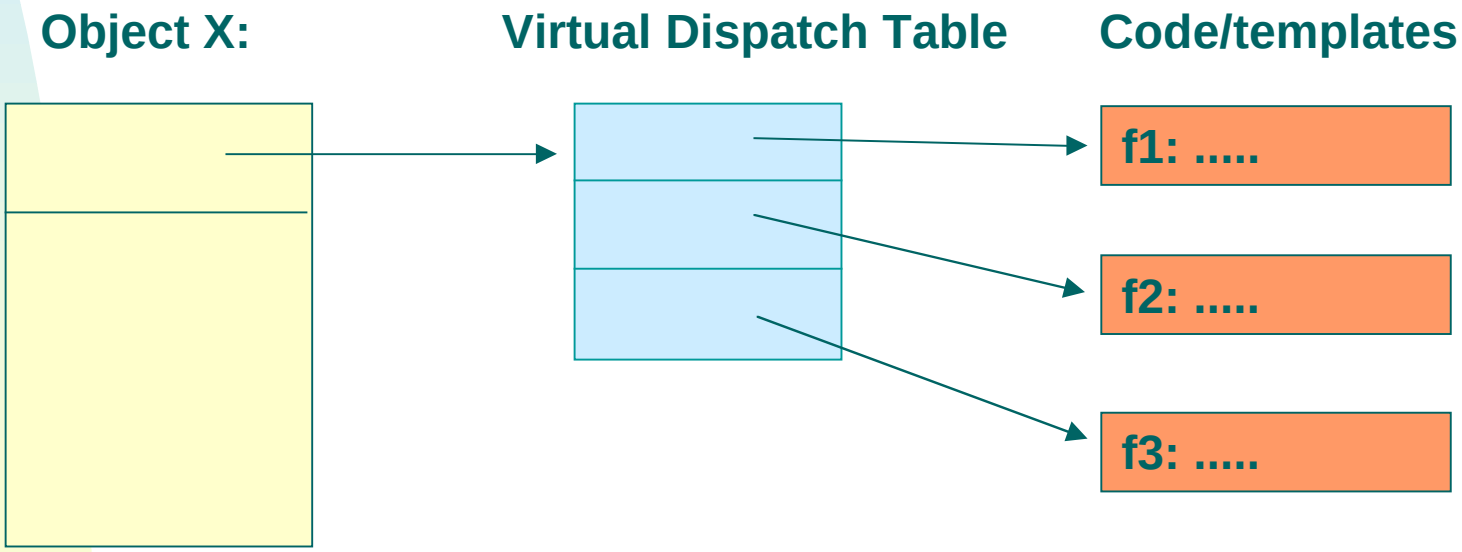
- **Run-time organization**
  - **Allocation of objects**
  - **Virtual patterns**
- **Semantic analysis**

Allocation of objects

- Procedure activations are first-class objects
- A reference can be obtained to a procedure-activation
- Requires heap-allocation - in general
- Room for optimization
- Stack allocation of procedure activations

Virtual class implementation

A virtual pattern is instantiated through a virtual dispatch table



The problem: virtual procedure binding

Declaration:

```
proc V():< ...
```

Binding 1:

```
VÕ()::< ...
```

Binding 2:

```
VÕÕ()::< ...
```

Virtual procedure application:

```
V();
```

- **Dynamic binding: the actual instance of V generated at run-time**
- **Static binding: the binding of V known on compile-time;**
- **The static binding is needed for type checking of parameters**

The problem: virtual class binding

Declaration:

```
class V:< ...
```

Binding 1:

```
VÕ: :< ...
```

Binding 2:

```
VÕÕ: :< ...
```

Variable declaration:

```
X: ref V
```

Use of instance of virtual class:

```
X.f()
```

- **Dynamic binding: the actual instance of V generated at run-time**
- **Static binding: the binding of V known at compile-time;**
  - **For X this is the *type* of X**
- **Static binding is needed for type checking**

Partial problems in semantic analysis

Easy:

- Nested procedures — no virtuals
- Nested classes — no virtuals

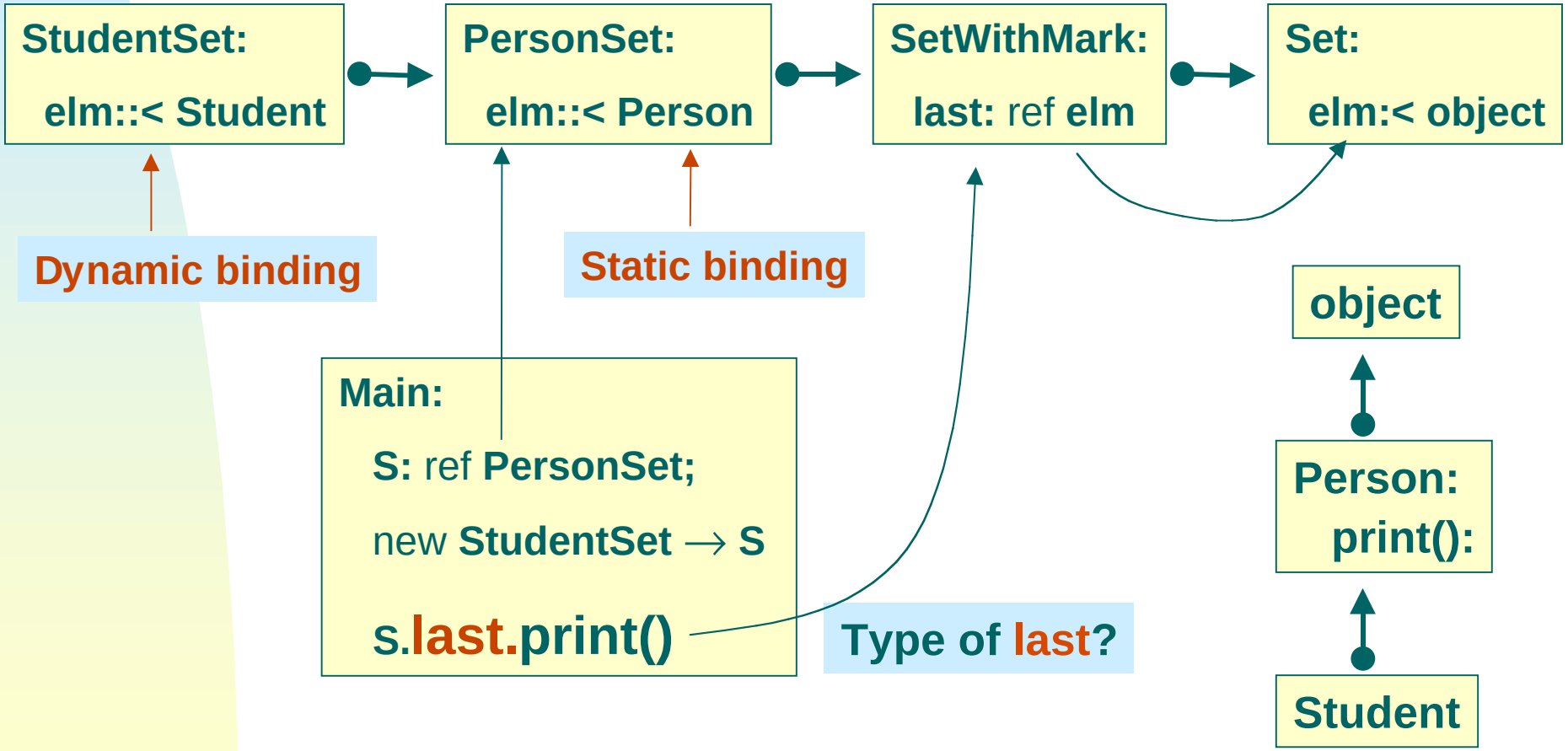
Easy:

- Virtual procedure binding — no nesting
- Virtual class binding — no nesting
- Virtual procedure binding — with nesting

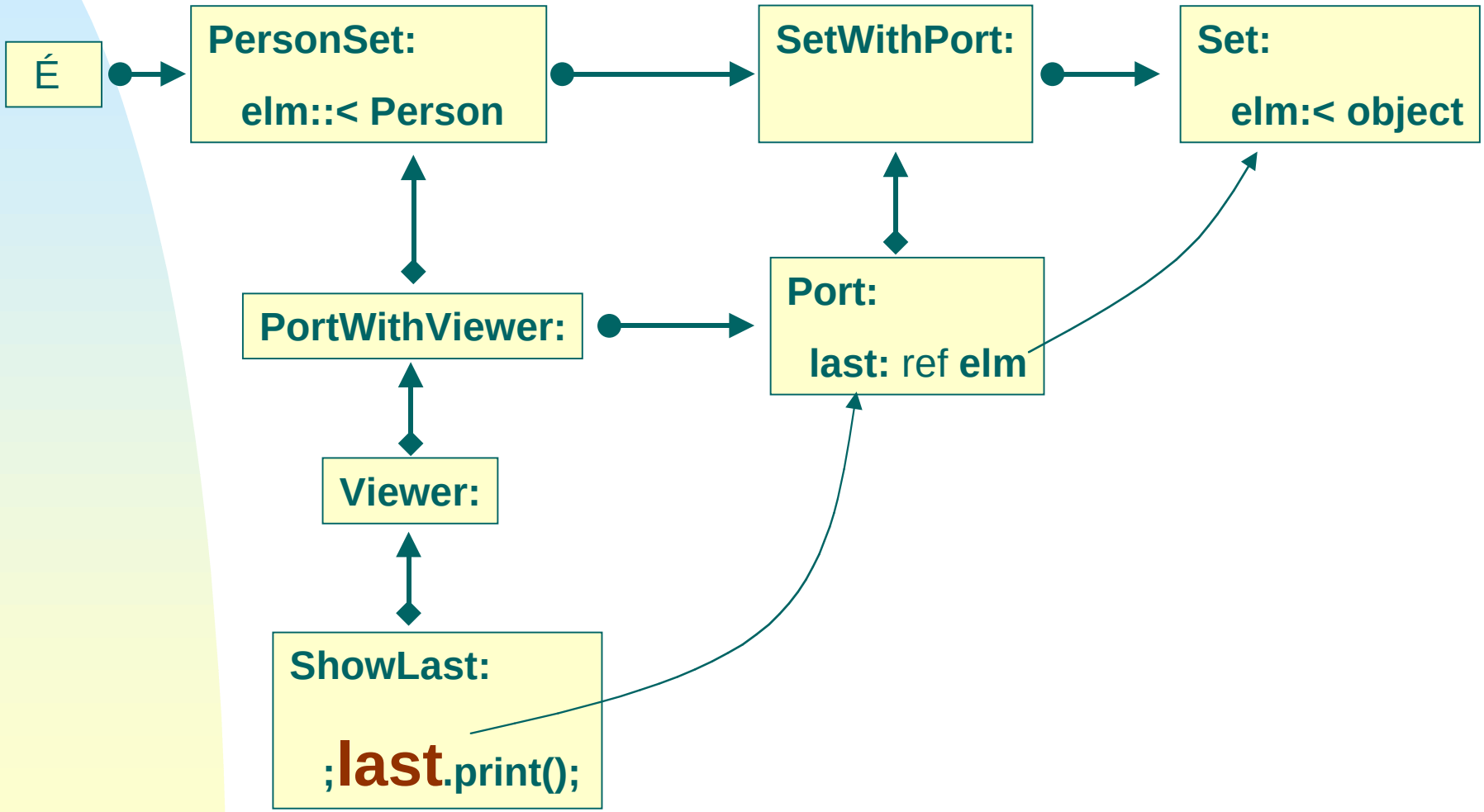
Hard:

- Virtual class binding — with nesting

Simple virtual class binding



General example



6. Summary

- Simple and general language
  - Based on a rich conceptual framework
  - For modeling and implementation
- The pattern is a powerful abstraction mechanism
- Virtual patterns
  - A simple generalization of virtual procedures
  - Powerful means for defining generic classes
- General nesting of patterns
- Patterns as first class values
- Classless objects
- Concurrency
- Unified paradigm

The Mj¿lner System

- **An industrial standard mature environment for development of BETA programs**
- **For an example of a large application developed in BETA, see the demo of**
- **CPN/Tools:**
  - **A Tool for Editing and Simulating Coloured Petri Nets**
  - M. Beaudouin-Lafon, W. Mackay, M. Jensen, P. Andersen, P. Janecek, M. Lassen, K. Lund, K. Mortensen, S. Munck, A. Ratzer, K. Ravn, S. Christensen, K. Jensen (Aarhus University)
  - **Thursday 5/4 — 15.00**

Contact address

Ole Lehrmann Madsen

Computer Science Department

Aarhus University

Aabogade 34, DK-8200 Aarhus N

DENMARK

e-mail: olmadsen@daimi.au.dk

Get a free Mj¿lner System

- **e-mail: sales@mjolner.com**
- **http://www.mjolner.com**