

Concurrency and State in the BETA Programming Language

Ole Lehrmann Madsen
Aarhus University
The Alexandra Institute Ltd.
Mjølnir Informatics Ltd.

Purpose of this talk

Present

- Main elements of the BETA language
- Concurrency in BETA
- Work on state-machine support for BETA
- Experience with concurrent object-oriented modeling

Based on experience with object-oriented analysis & design using the Unified Modeling Language (UML)

Main benefits of object-orientation

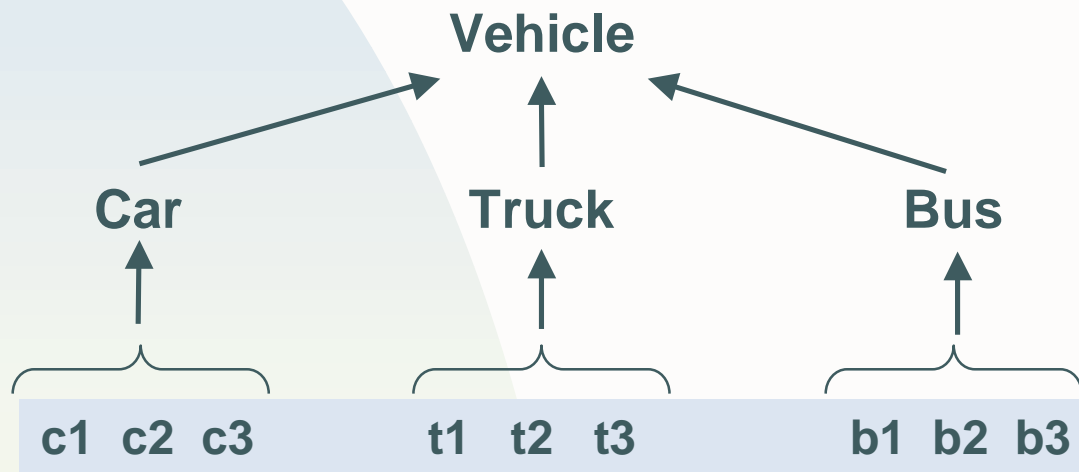
1. Good support for modeling
2. Good support for reusability and extensibility
3. Unifying perspective on most phases of the software life cycle

1. Modeling

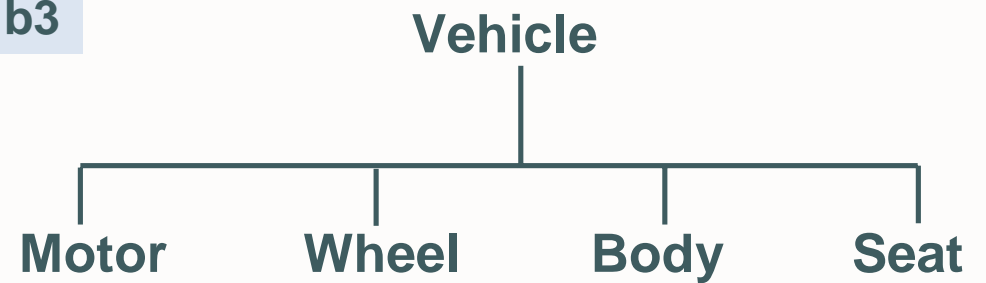
- Programs reflect reality in a *natural* way
- Objects, object attributes, classes, etc. are good means for building *physical* models of concepts and phenomena from the application domain
- A physical model is more *stable* than a model based upon the desired functionality of a system (Jackson)
- Rich *conceptual framework* for organizing knowledge about the problem domain
 - ◆ classification: generalization, specialization, clustering
 - ◆ composition: part/whole, reference composition, localization

Modeling means

Classification



Composition



2. Reuse and extensibility

- New possibilities for reuse and extensibility
- Constructs like class, subclass, virtual function, etc.
- Architectural principles like
 - ◆ Application frameworks
 - ◆ Binary components
 - ◆ Design patterns

3. Unifying Perspective

Object-orientation applies to:

- analysis
- design
- implementation
- data bases
- object-distribution

- Common conceptual framework
 - ◆ classification, generalization, etc.
- Common (abstract) language for central aspects
 - ◆ class, subclass, virtual procedure, etc.
- Formal language notation
 - ◆ textual syntax (programming languages)
 - ◆ diagrammatic syntax (design languages)

The object technology morass

- **Modeling languages for analysis and design**
 - ◆ UML - the PL/1 of modeling languages
- **CASE tools**
 - ◆ Code generation and reverse engineering problems
- **Programming languages**
 - ◆ C++, insecure and C-based
 - ◆ Java, the emperor's new clothes
- **Object-oriented databases**
 - ◆ Data definition languages (ODMG)
- **Components**
 - ◆ COM/OLE/ActiveX, Java/Beans
- **Distributed objects**
 - ◆ Interface language like CORBA/IDL, DCOM
- **Too many languages**

Object-orientation started with programming

- Object-orientation started with the development of Simula in the sixties
- Simula was created as a means for making simulation models
- Simula was a general purpose programming language
- Simula was used for analysis, design and implementation
- The real world is reflected in the programs
- Simula was a programming language with a built-in method
- No need for structured analysis/structured design

Based on the Scandinavian approach to object-orientation

- Experience with Simula
- Development and experience with the the Beta programming language
- The Mjølner project
- The Devise Centre, Aarhus University
- Centre for Object Technology
- Based on cooperation with many people ...
- Based on work adopted or adapted from others ...

Influence from Petri Net community

- Simula (Dahl & Nygaard)
- Delta (Nygaard et al.)

■ BETA

- ◆ Bent Bruun Kristensen
- ◆ Birger Møller-Pedersen
- ◆ Kristen Nygaard
- ◆ Ole Lehrmann Madsen

■ Mjølner

- ◆ Jørgen Lindskov Knudsen
- ◆ Elmer Sandvad
- ◆ Peter Andersen
- ◆ Boris Magnusson
- ◆ ...

■ Early Petri Net influence

- ◆ Antoni Mazurkiewicz
- ◆ Hartmann Genrich
- ◆ Kurt Lautenbach
- ◆ P.S. Thiagarajan
- ◆ Robert Shapiro

■ Epsilon

- ◆ Kurt Jensen
- ◆ Morten Kyng
- ◆ (Ole Lehrmann Madsen)

■ DAIMI CPN group

- ◆ Kurt Jensen
- ◆ Søren Christensen
- ◆ ...

Content

- 1. Introduction**
- 2. The BETA language**
- 3. Concurrency in BETA**
 - ◆ Basic primitives
 - ◆ Abstraction
 - ◆ Alternation
- 4. Object-oriented modeling**
 - ◆ State-machine support in BETA
 - ◆ Concurrent modeling

2 BETA

Goals of 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 abstraction and modularization
- Unified paradigm
 - ◆ Neither pure OO or multi-paradigm

Unified paradigm

- Based on object-orientation
- Supports imperative programming
- Some support for functional programming
 - ◆ Higher order functions and classes
 - ◆ Functions and classes as 'first class' values
 - ◆ 'New' functional notation
- Supports concurrency
- Some support for object-based programming
 - ◆ Singular objects that are not instances of classes
 - ◆ No delegation

Abstraction and modularization

Abstraction

- The right basic primitives
- Powerful abstraction mechanisms for building higher-level abstractions

logical
structure

Modularization

- Organization of program fragments into logic and manageable units
- Language independent mechanism based on the grammar
 - ◆ a module is a sentential form

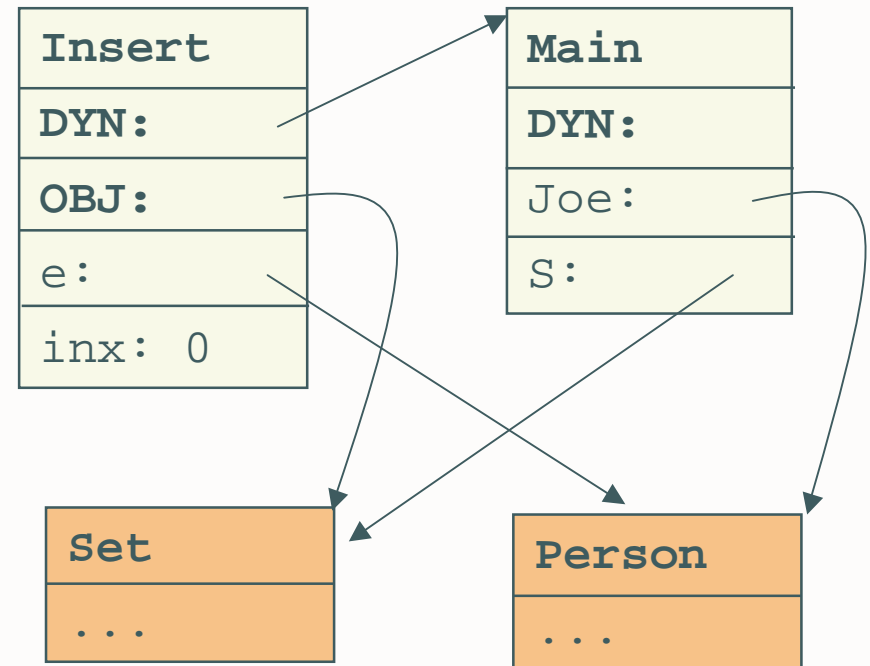
physical
structure

Unification of abstraction mechanisms

- Abstraction mechanism:
 - ◆ a pattern for creating instances
- Class:
 - ◆ a pattern for creating objects
- Procedure:
 - ◆ a pattern for creating action-sequences
- Function, interface, process type, generic class, exception type, ...
- In BETA: unified into the **pattern**

Unifying class and procedure

```
class Person: { ... };  
class Set: {  
  proc insert(e: ref object):  
    { inx: obj integer  
      do ...  
    };  
  ...  
};  
proc main():  
  { S: ref Set;  
    Joe: ref Person;  
    do S = new Set;  
      S.insert(Joe);  
    }
```



Objects and procedure activations

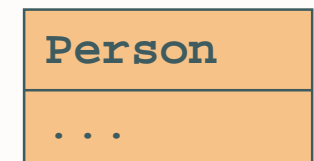
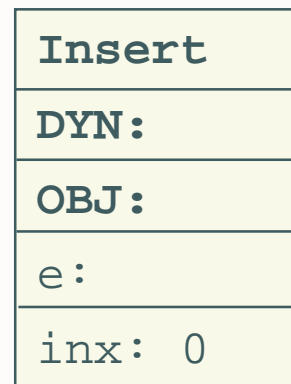
- Object
 - Data-items: ...
 - Static link: enclosing object/procedure
- Procedure-activation
 - Parameters & local variables
 - Dynamic link: caller
 - Static link: enclosing object/procedure

Procedure call

```
M: ref R.insert
```

```
M = new R.insert (Joe) ;
```

```
M.execute
```



Pattern: class & procedure

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

Static link: if nesting

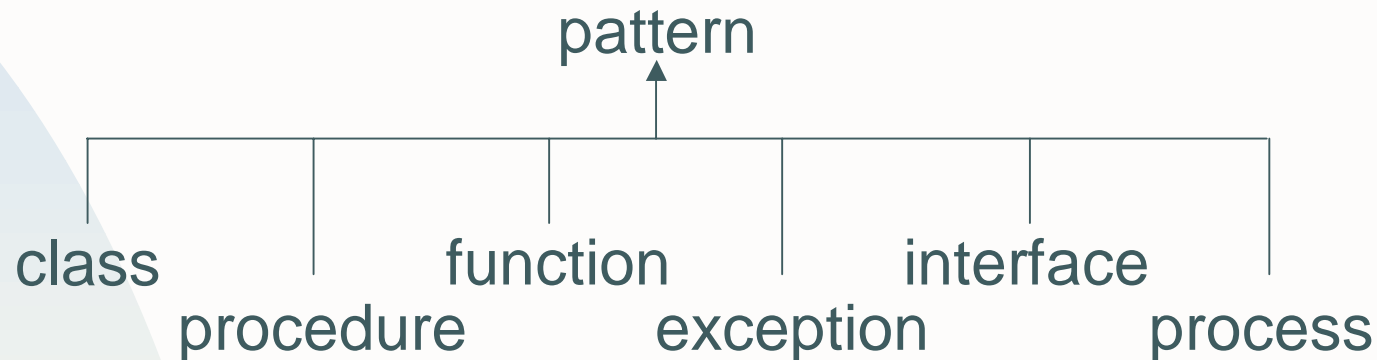
Dynamic link: if do-part

Parameters

Data-items

Object layout
STAT:
DYN:
parameters
data-items

Uniform abstraction mechanisms



The **Pattern**

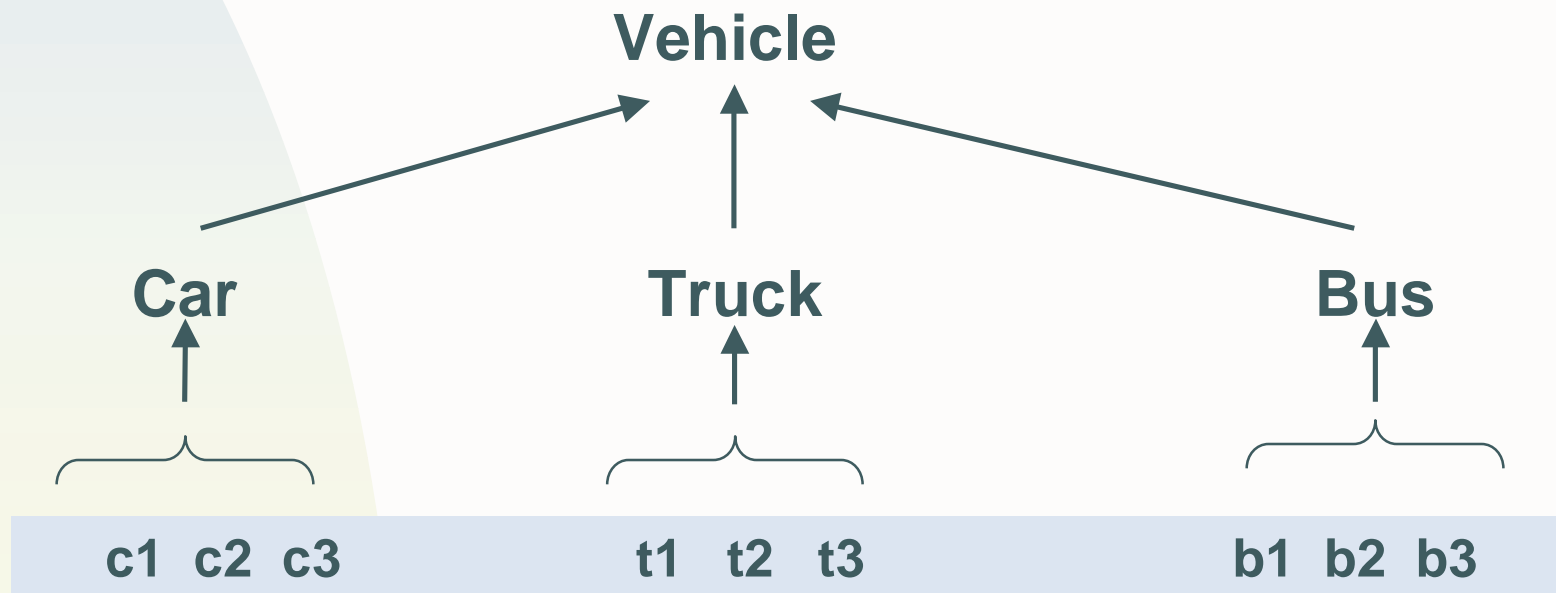
- The ultimate abstraction mechanism
- A generalization/unification of class, procedure, function, exception, interface, process, generic class,

Benefits of the unification

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

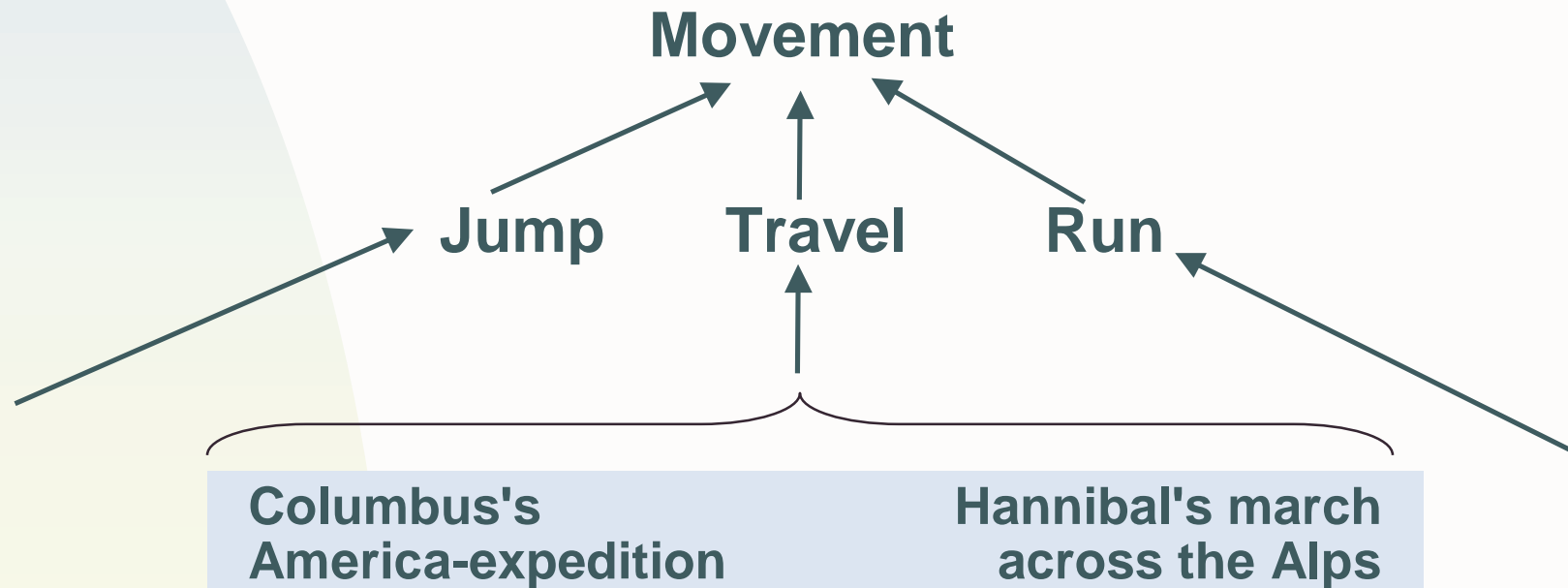
Subpattern: subclass

Class and subclass supports classification of objects



Subpattern: subprocedure

Procedure and subprocedure supports classification of action-sequences



Action sequence generalization

```
L: list;
M: Semaphore;
void Put(e: integer)
{ M.wait;
  L.put(e);
  M.signal
}
integer get()
{ e: integer;
  M.wait;
  e = L.get();
  M.signal
  return e
}
```

Put:



Get:



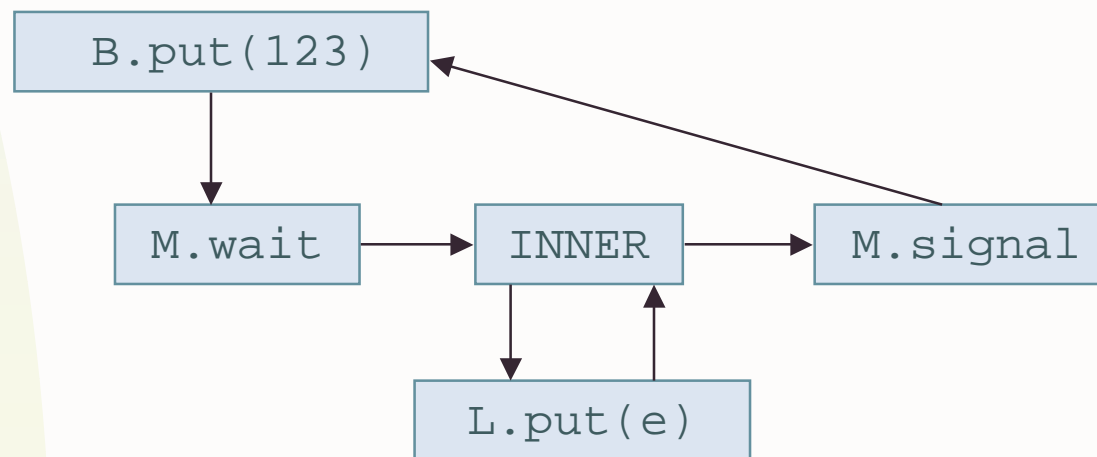
Generalized action sequence:



Procedure specialization

```
class Buffer: extends monitor
{ L: list;
  proc put(e: integer)
    extends entry
    { L.put(e)
    };
  ...
};
B: Buffer
```

```
class monitor:
{ M: semaphore;
  proc entry()
    { M.wait;
      INNER;
      M.signal
    };
  ...
};
```



Other benefits of the unification

- Virtual pattern:
 - ◆ virtual procedure
 - ◆ virtual class
 - ◆ virtual process
- Nested pattern:
 - ◆ Nested procedure
 - ◆ Nested class
 - ◆ Nested process
- Pattern variable:
 - ◆ Procedures as first class values
 - ◆ Classes as first class values
 - ◆ Processes as first class values

3 Concurrency in BETA

Issues

- Basic primitives for
 - ◆ Protection of shared objects
 - ◆ Communication
 - ◆ Synchronization
 - ◆ Scheduling
- Abstraction mechanisms
 - ◆ Definition of concurrency abstractions
- Modeling
 - ◆ Concurrency
 - ◆ Alternation, non-preemptive scheduling, non-determinism/guarded commands
 - ◆ Analysis/Design (UML)

Mainstream OO languages

- Simula
 - ◆ quasi-parallel systems
 - ◆ active objects
 - ◆ schedulers
- Concurrent Pascal
 - ◆ Class, process, monitor
- C++
 - ◆ No built-in mechanisms
- Java
 - ◆ Monitors

Concurrency

- Concurrency has not been a well integrated part of most main-stream OO languages: Smalltalk, CLOS, C++, (Eiffel), etc.
- Semi-concurrency was a central part of Simula: object are coroutines
- BETA is a further development of the Simula model for concurrency
- Concurrency should be a natural part of a general purpose programming language

Concurrency in Java?

- A step in the right direction?
- Part of the language
- Good for certain kinds of applications
- Perhaps not general enough
- Many research papers on concurrency problems with Java

In general to many papers on:

- How to add XXX in Java
- How to do YYY in Java
- How to avoid ZZZ in Java

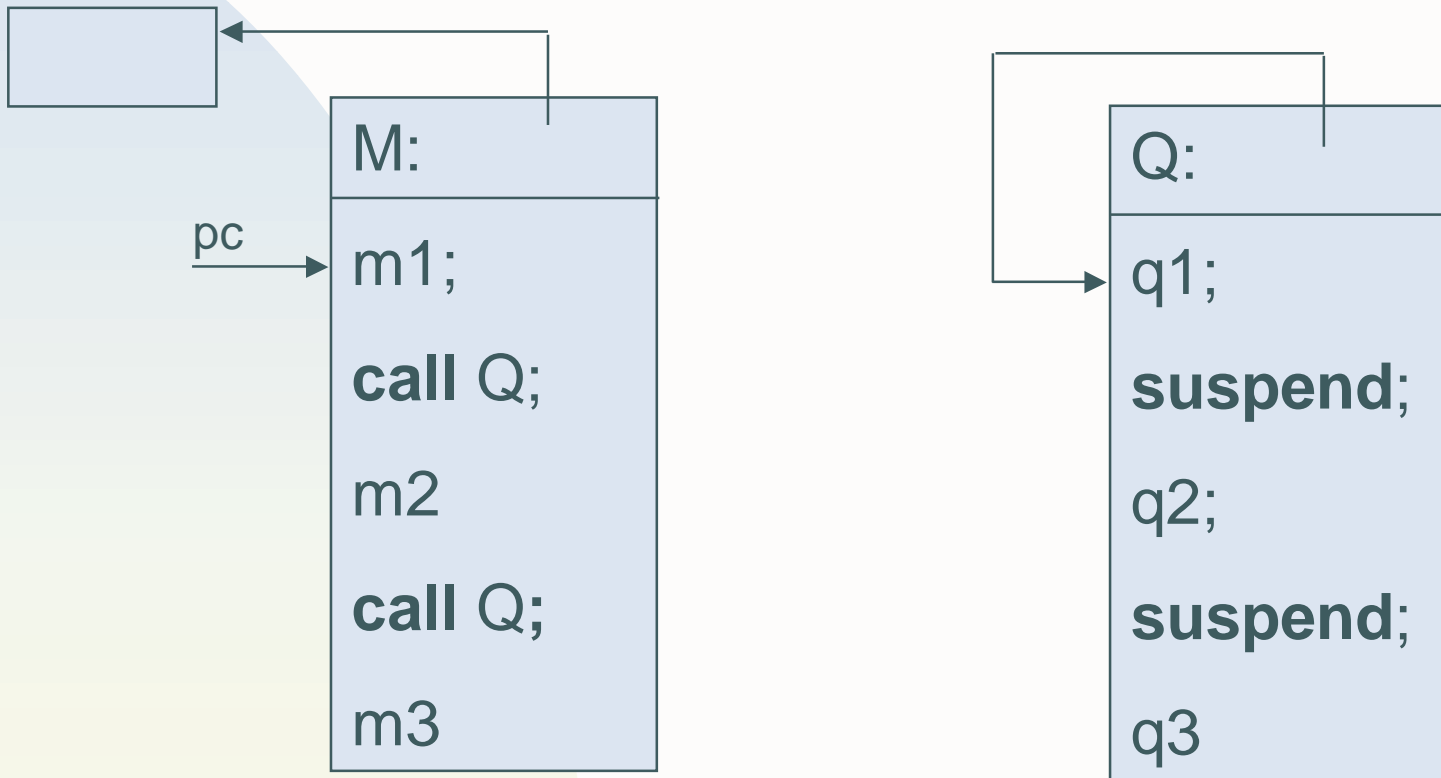
Brinch Hansen (Sigplan, April, 1999)

- Gosling [...] claims that Java uses monitors to synchronize threads. Unfortunately, a closer inspection reveals that Java does not support a monitor concept.
- The failure to give an adequate meaning to thread interaction is a very deep flaw of Java that vitiates the conceptual integrity of the monitor concept.
- It is astounding to me that Java's insecure parallelism is taken seriously by the programming community, a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.
- If programmers no longer see the need for interference control then I have apparently wasted my most creative years developing rigorous concepts which have now been compromised or abandoned by programmers.

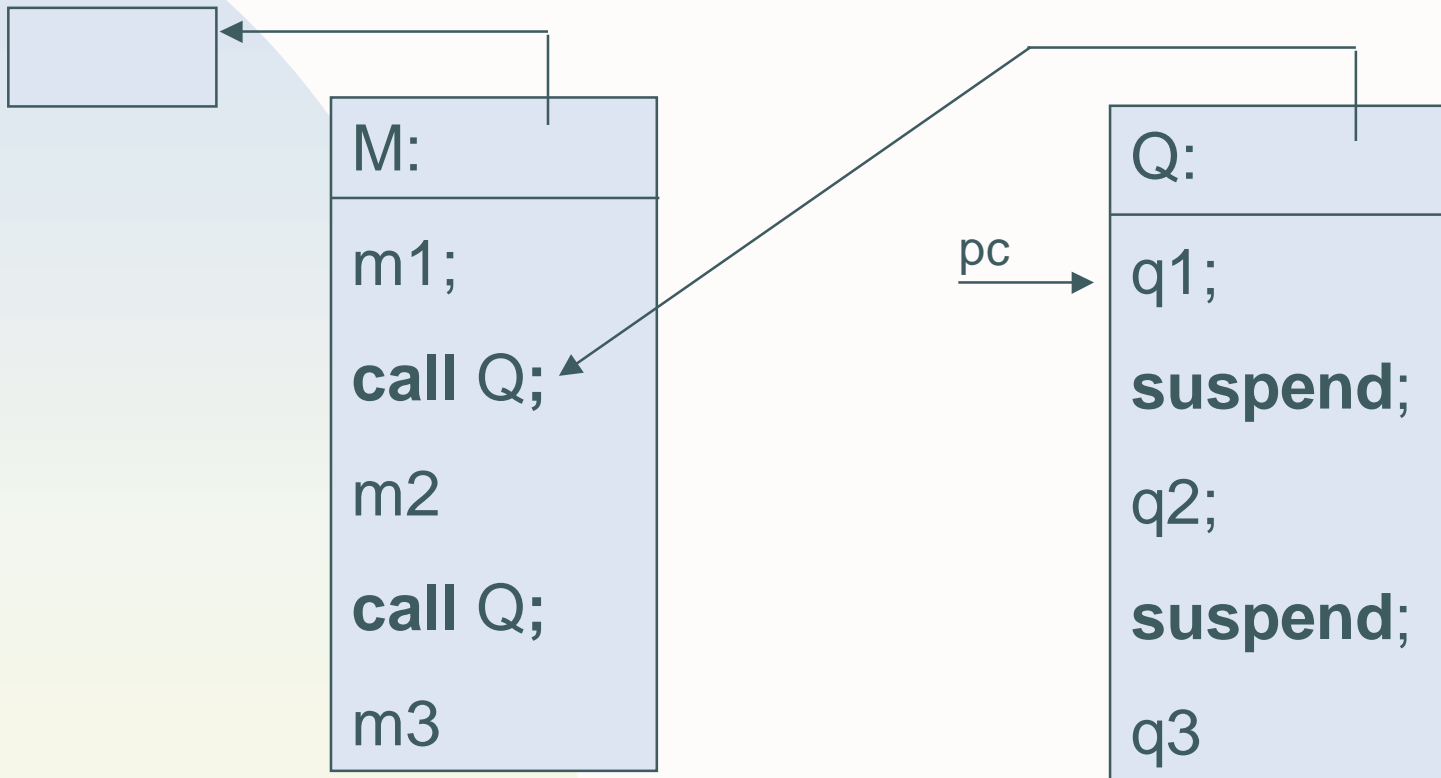
3.1 Simula/BETA : basic primitives

- **Active object**
 - ◆ An object may be the head of a thread
- **Object may execute as**
 - ◆ coroutines - alternating
 - ◆ concurrent
- **Synchronization by means of semaphores**
- **Good support for defining high-level concurrency abstractions**
 - ◆ Monitor, Ada-like rendezvous, etc.

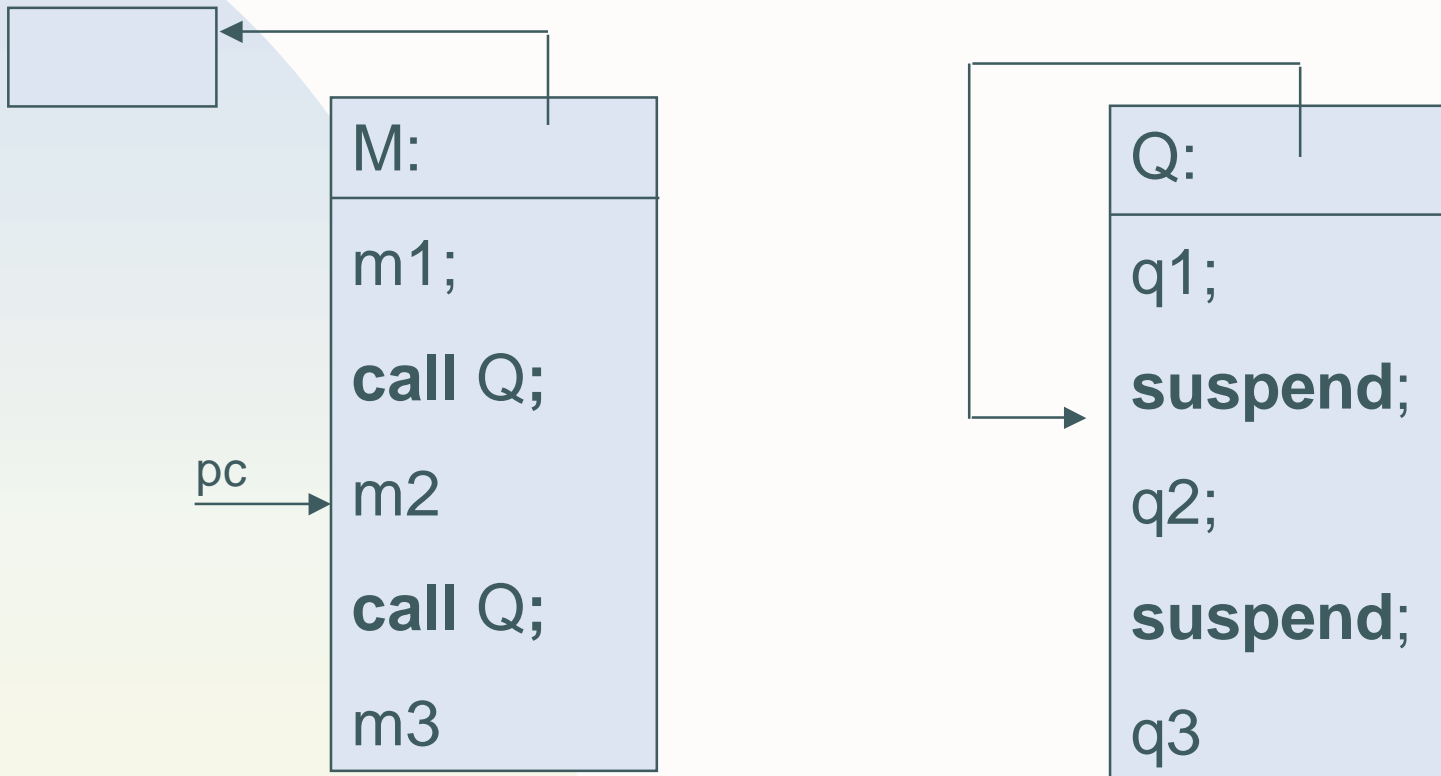
Semi-symmetric coroutine 1



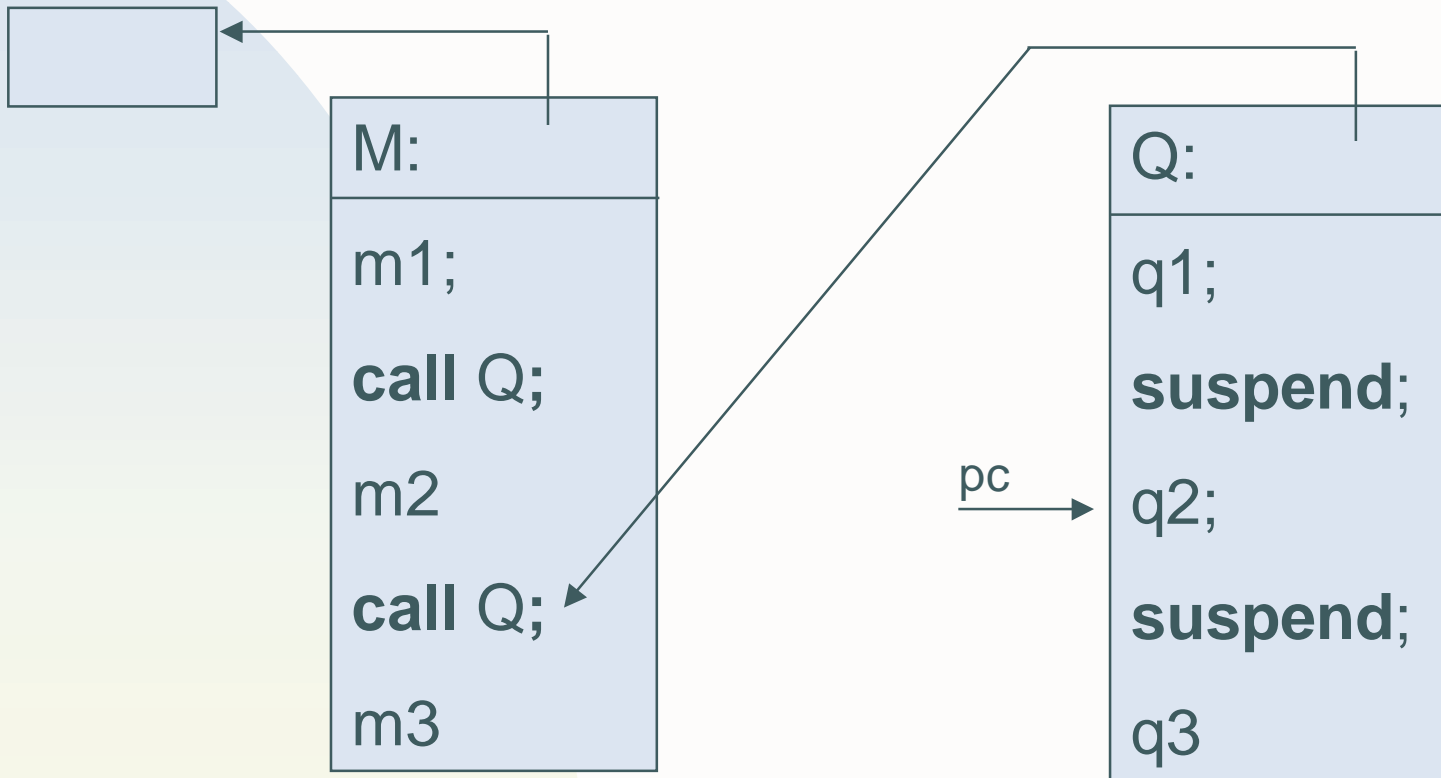
Semi-symmetric coroutine 12



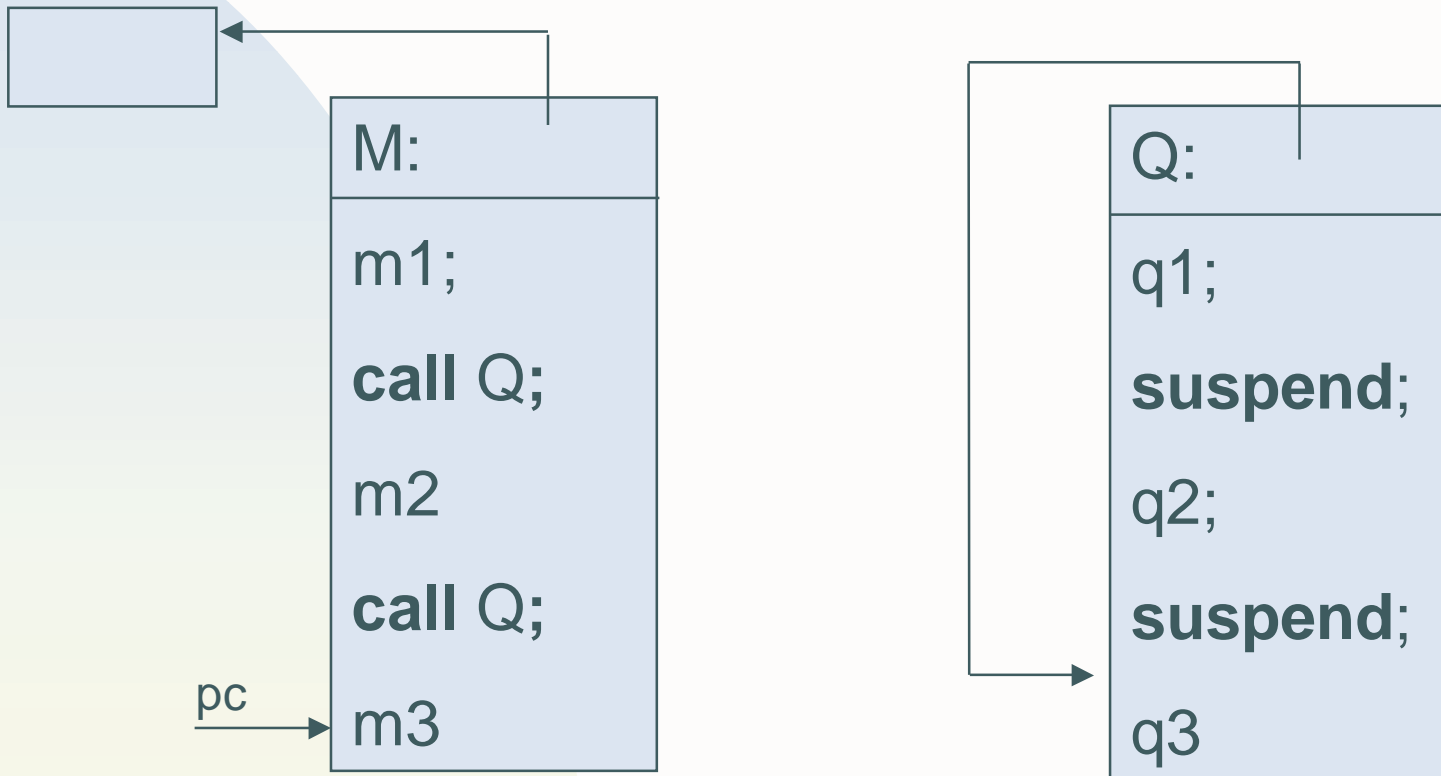
Semi-symmetric coroutine 123



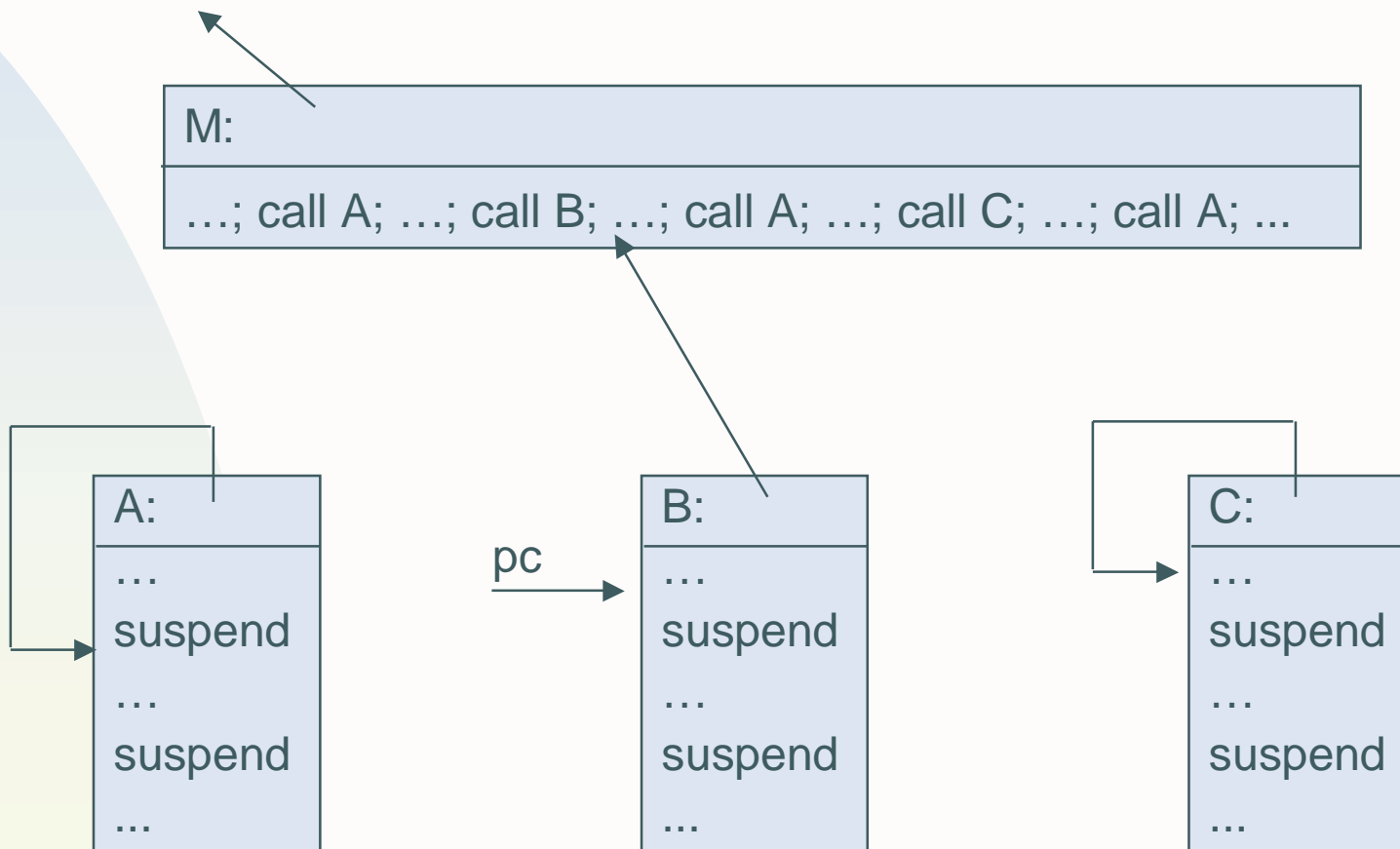
Semi-symmetric coroutine 1234



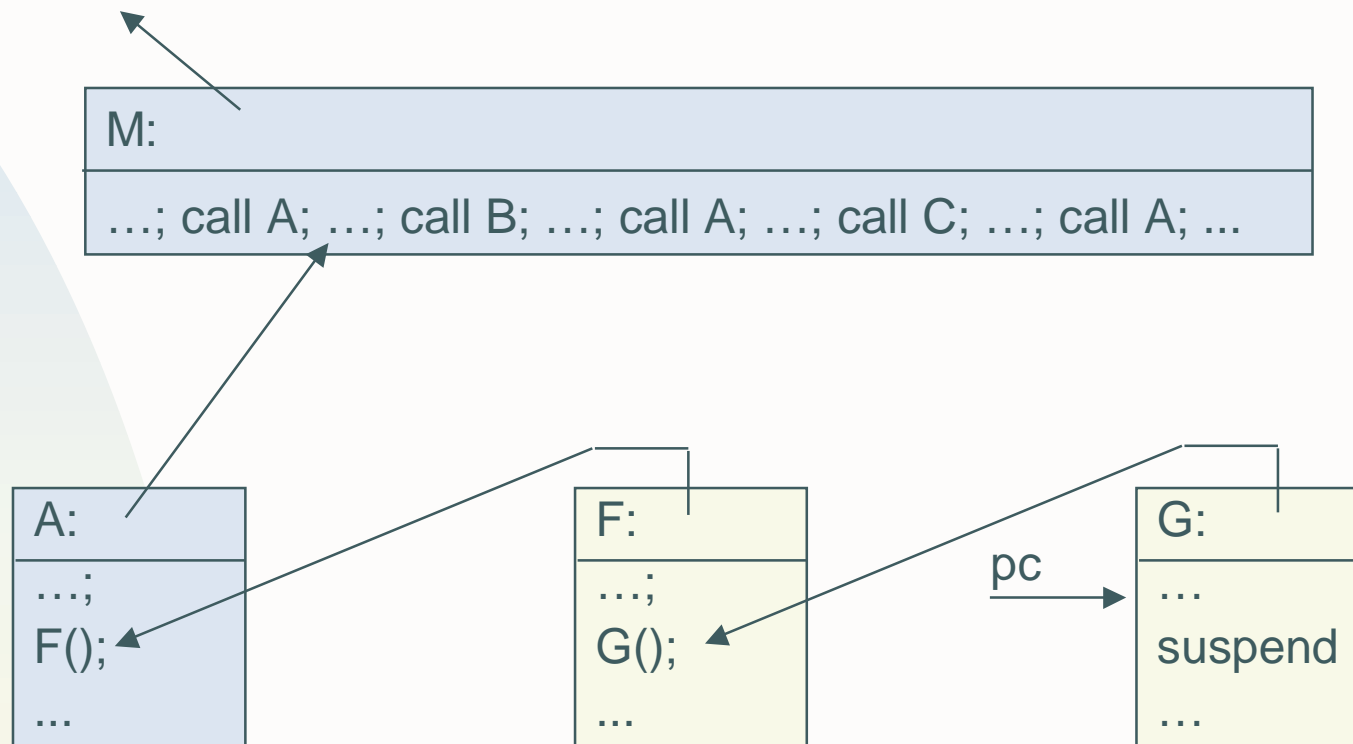
Semi-symmetric coroutine 12345



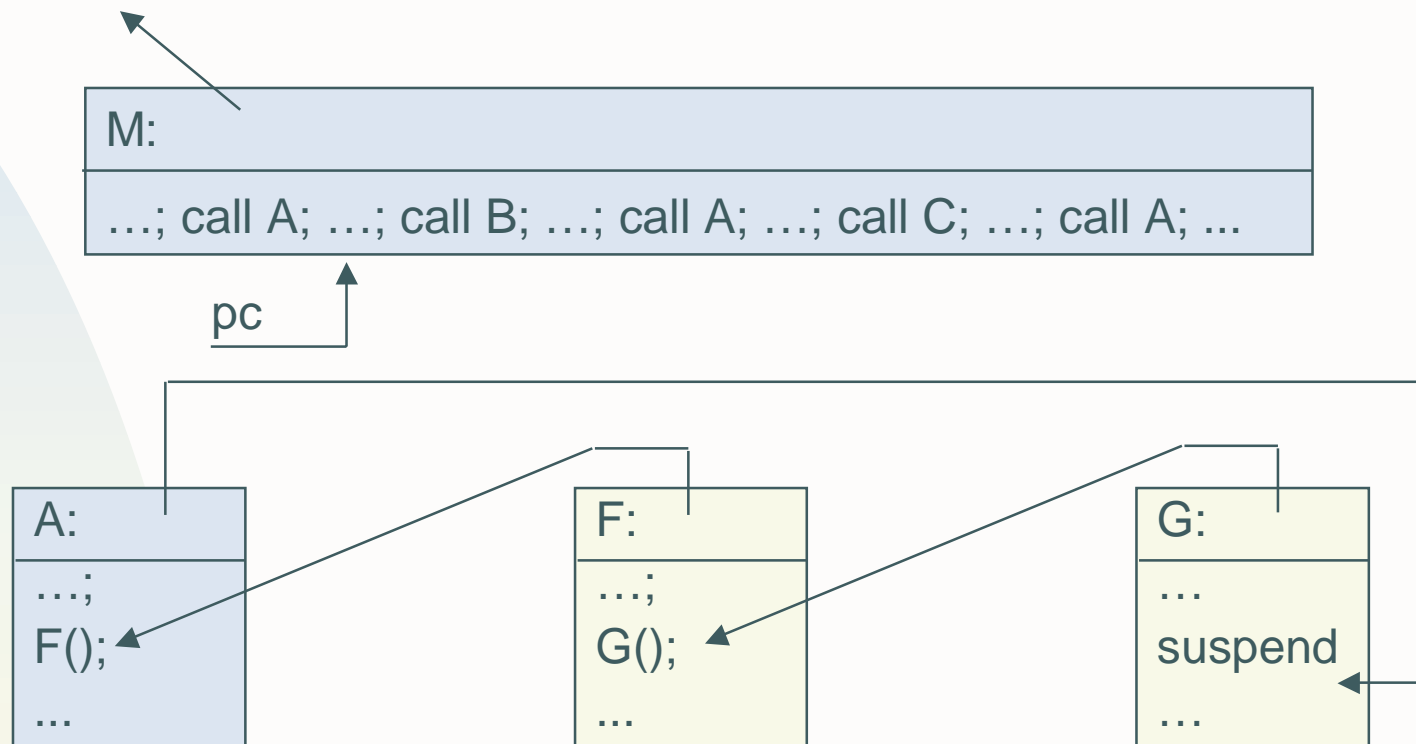
Semi-coroutines - in general



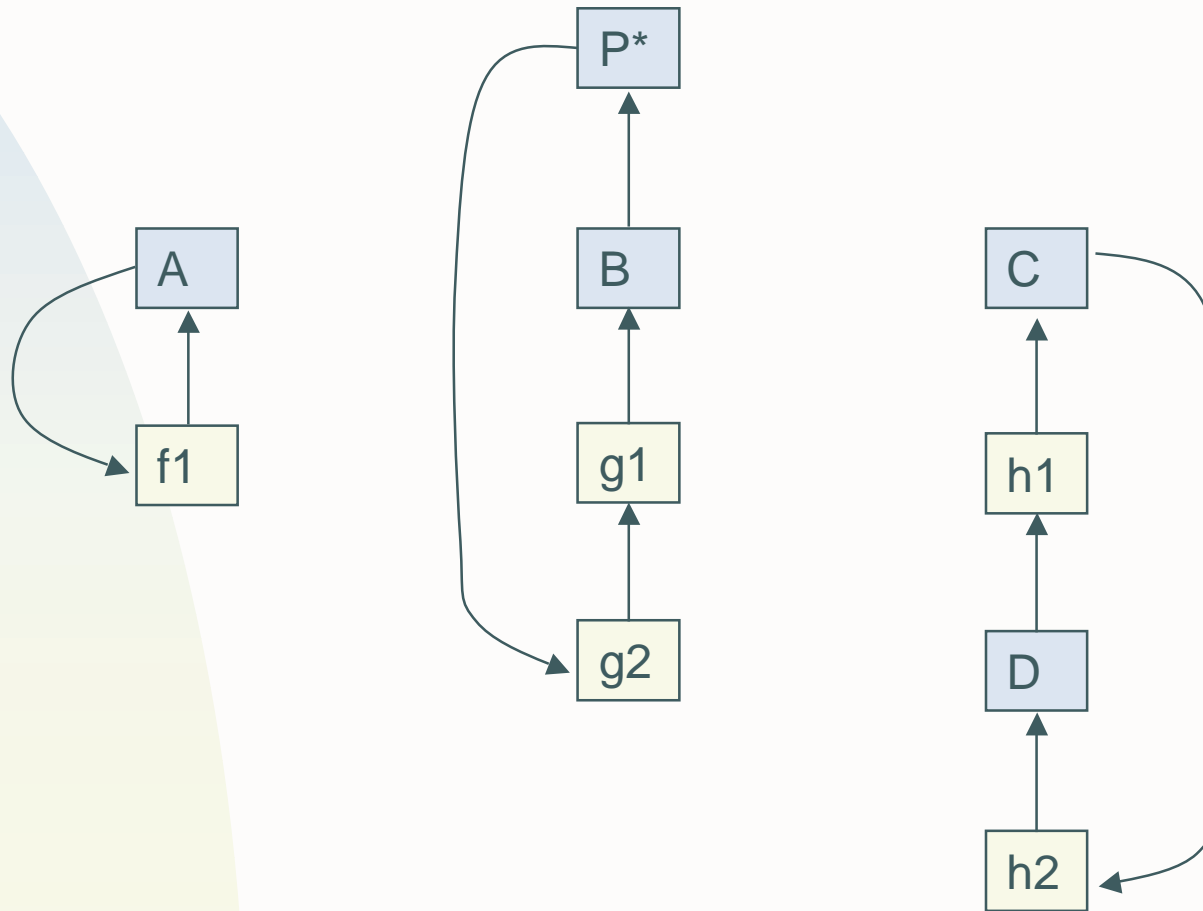
Object with activation stack



Object with activation stack



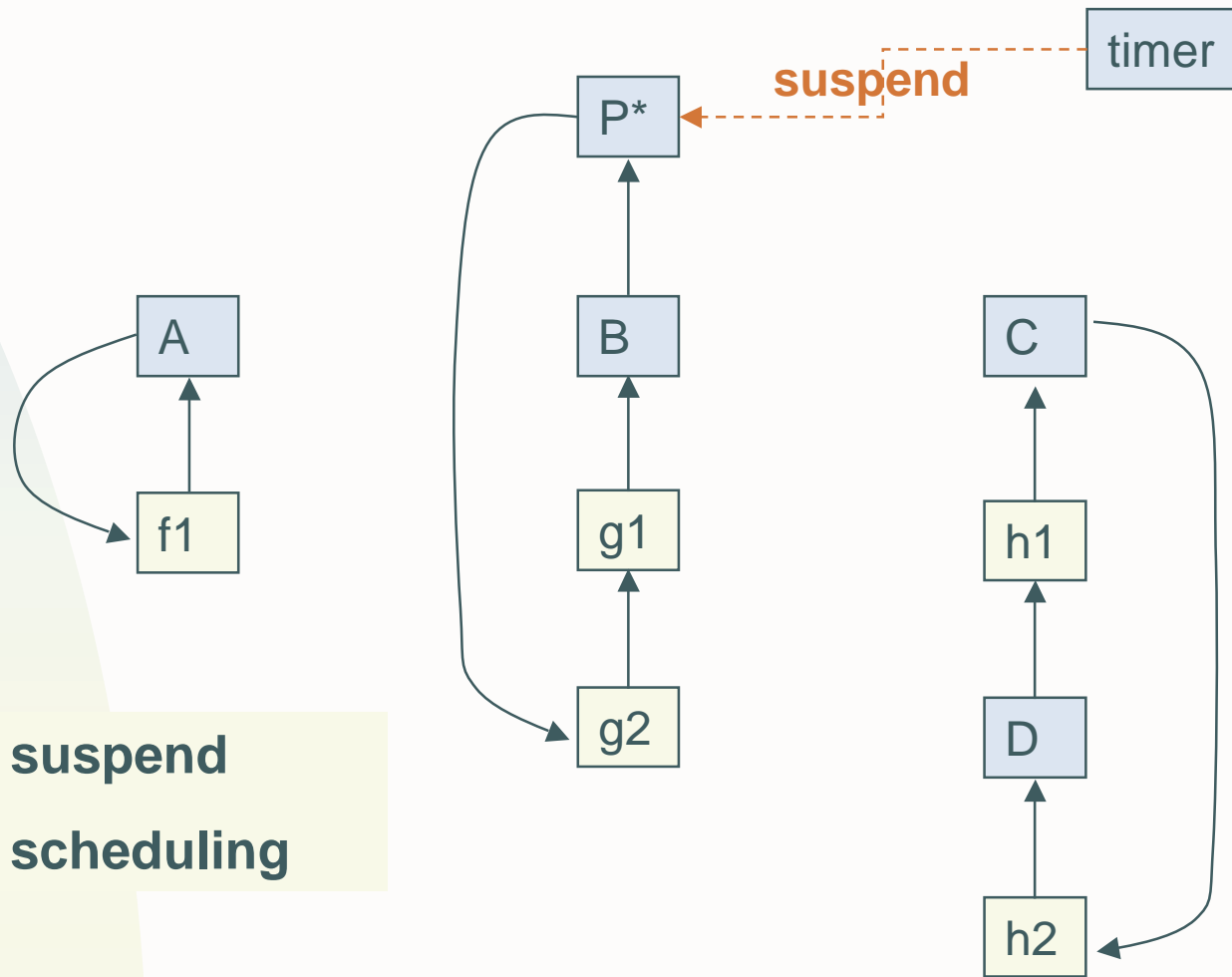
General picture



Use of symmetric coroutines

- **Basis for non-preemptive scheduling**
- **Simula: simulation framework**
 - ◆ The first application framework
 - ◆ Process concept
 - ◆ Scheduling
- **Generators (Icon)**
- **Symmetric coroutines (resume)**
- **Semi-concurrent algorithms**
 - ◆ merge of binary search trees

Concurrency



- Preemptive suspend
- Preemptive scheduling

Synchronization

- Basic primitive: semaphore
- Basis for building higher-level concurrency abstractions

```
M: obj semaphore
```

```
M.P;
```

```
M.V;
```

3.2 Abstraction

Monitor

```
class Buffer: extends monitor
{ L: list;
  proc put(e: integer)
    extends entry
    { L.put(e)
    };
  ...
};
B: Buffer
```

```
class monitor:
{ M: semaphore;
  proc entry()
    { M.wait;
    INNER;
    M.signal
    };
  ...
};
```

Ada-like rendezvous

```
S1: obj { do ...; R.foo; ...; R.bar; ... };
S2: obj { do ...; R.snorf; ... };
R: obj
  { P,Q: obj Port;
    foo: P.entry{ ... };
    bar: P.entry{ ... };
    snorf: Q.entry{ ... };
  do ...; P.accept;
    ...; Q.accept;
    ...; P.accept;
  }
```

Restricted ports

```
P: obj Port,  
P.accept
```

Accept all objects

```
Q: obj ObjectPort;  
Q.accept(anObject)
```

Accept only anObject

```
R: obj QualifiedPort;  
R.accept(aClass)
```

**Accept all instances of
aClass or its subclasses**

Other abstractions

- asynchronous communication
- futures
- multi-cast
- inheritance of synchronization constraints

Main point:

- No "winning" concurrency abstraction
- Provide
 - ◆ basic primitives
 - ◆ abstraction mechanisms for building concurrency abstractions

4 Concurrency and Alternation

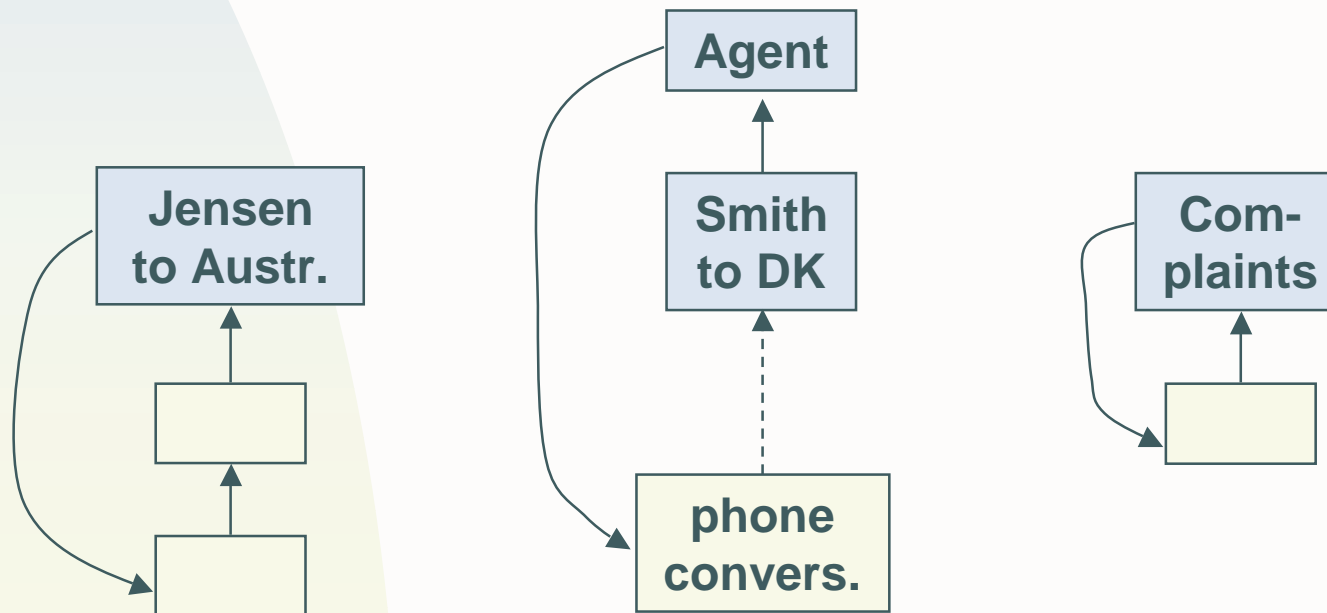
- Concurrency
 - ◆ Modeling of concurrent activities in the real-world
- Alternation
 - ◆ Application domain modeling of non-preemptive scheduling
 - ◆ Alternative model for non-deterministic guarded commands
 - ◆ Interrupt handling

3.3 Alternation

- An actor (person, machine, ...), may be engaged in several activities at the same time
- At most one activity at a given time
- Change of activity decided by the agent
 - ◆ The activity has reached some well defined state
 - ◆ External event

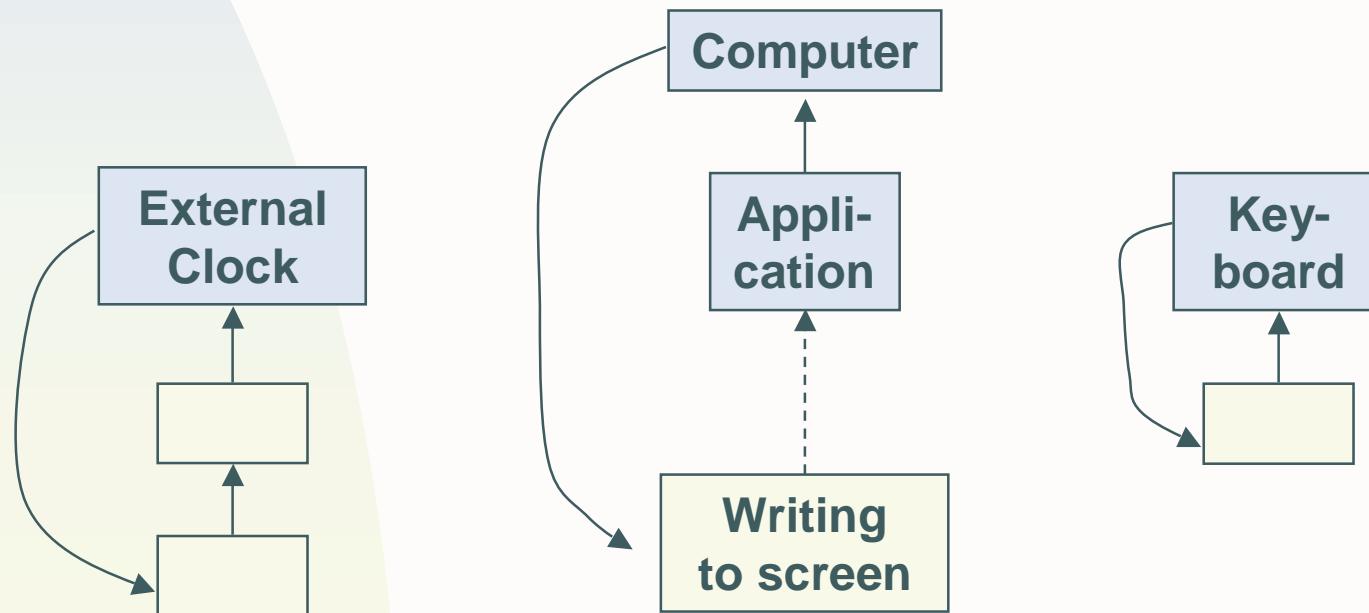
Alternating activities of a travel agent

- Booking a business trip to Australia for Mr. Jensen
- Booking a vacation trip for Mr. Smith to Denmark
- Handling complaints for customers



Alternating activities of a computer

- Handling keyboard events
- Handling external clock
- Executing application program



Non-deterministic guarded commands

- Accept input from $p1$ or $p2$

p1

S:

...

(if

// $p1?v$ then $i1$

// $p2?w$ then $i2$

if);

...

p2

Guarded input & output

- Accept input from $p1$
- or
- output to $p2$

$p1:$

```
S:  
...  
(if  
  // p1?v    then i1  
  // p2!e    then i2  
if):  
...
```

$p2:$

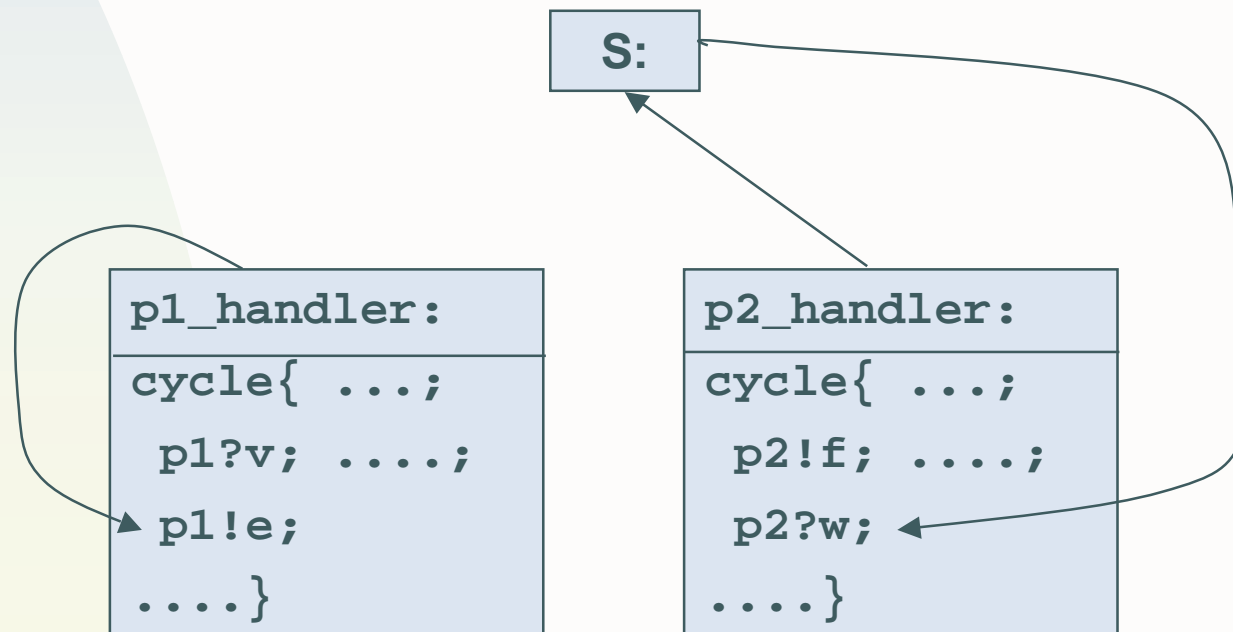
Logical communication sequences

- `cycle{ ... p1?v; ... p1!e; ... }`
- `cycle{ ... p2!f; ... p2?w; ... }`

```
S:  
...  
(if  
// p1?v and B1      then ...; B1:=false  
// p1!e and not B1 then ...; B1:=true  
// p2!f and B2      then ...; B2:=false  
// p2?w and not B2 then ...; B2:=true  
if):  
...
```

Alternating object

- A concurrent object may have internal alternating objects
 - ◆ one pr logical communication sequence

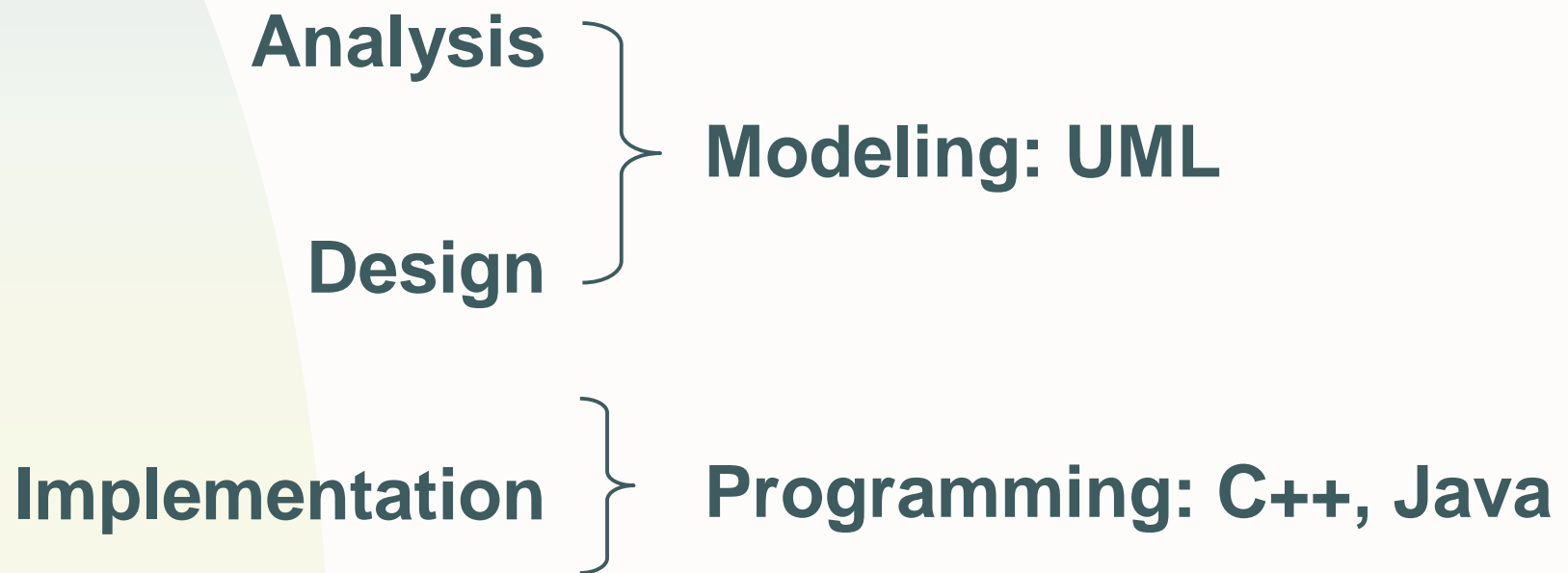


4 Object-oriented modeling

- **Object-oriented modeling versus programming**
- **Graphical modeling languages**
- **Early experience with Unified Modeling Language (UML)**

Modeling versus programming

- **Major benefit of object-orientation:**
 - ◆ **Unifying perspective on analysis, design, implementation**



Language overhead

- **The developers should be able to alternate freely between analysis, design and implementation**
- **Different languages are used analysis/design (UML) and implementation (C++, Java)**
- **Creates CASE gap and reverse engineering problems**
- **Difficult to alternate between design and implementation**
- **Too much discussion of actual syntax of diagrams**
- **Mental overhead in dealing with several notations**

One language for design and implementation

- The same abstract language can be used for expressing
 - ◆ class, subclass, virtual procedure, part objects, references, associations, etc.
- The design language can be a subset of the implementation language
- A diagrammatic syntax can be used for design diagrams
- A textual syntax can be used for programming
 - ◆ Possible to build CASE tool with no CASE gap:
 - ◆ Incremental code generation
 - ◆ Full round-trip and incremental reverse engineering

Modeling versus implementation

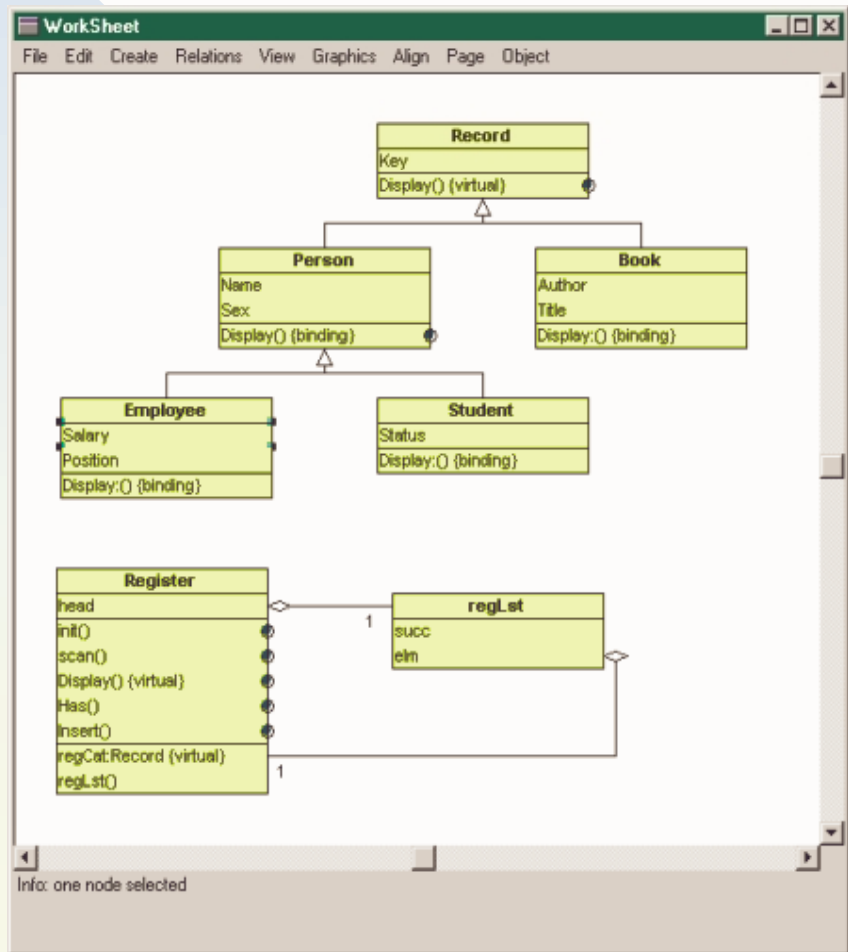
- Modeling:
Development of representative elements
- Implementation:
Programming non-representative elements

	graphical	textual
Modeling	+	+
Implementation	+	+

The Mjølnir CASE Tool

- BETA is used for design and implementation
- Graphical syntax for parts of BETA
 - ◆ Based on subset of UML with well-defined semantics
- No CASE gap:
 - ◆ Incremental generation of textual BETA
 - ◆ No annotations in the code
- Full incremental reverse engineering:
 - ◆ Immediate generation of diagrams from textual BETA
- Free alternation between design and implementation

Screen dump from Mjølner CASE tool



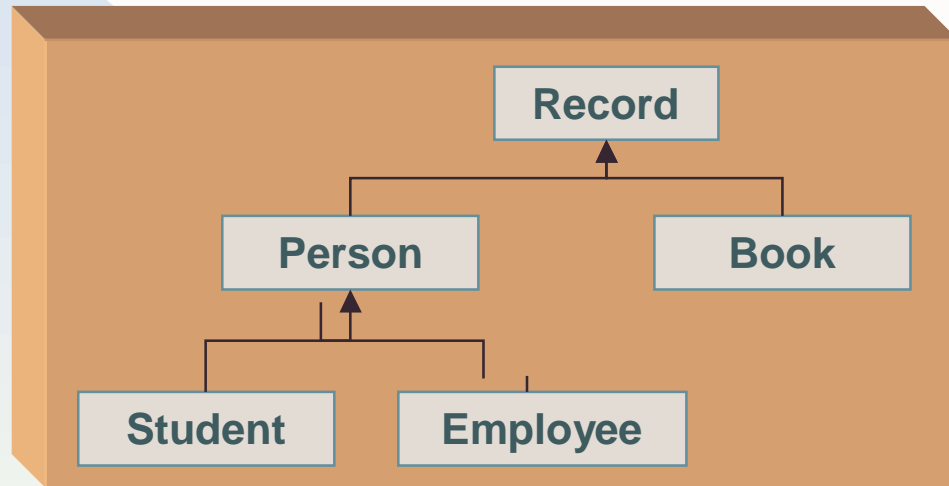
The window displays the following structure:

- Projects:** editor/, bifrost/, guiew/, Std. Libraries/, ~/ , recordlib*
- Groups:** recordlib
- Fragments:** ORIGIN '~beta/basiclib/betaenv/', lib: Attributes

```
(*)
Record: (*) ...;
Person: Record ...;
Employee: Person
  (# (*) Salary: @integer; Position: @text; Display::< ...; #);
Student: Person ...;
Book: Record ...;
docD: (*) ...;
NewRecord: (*) ...;
NewPerson: NewRecord (*) ...;
NewEmployee: NewPerson (*) ...;
NewStudent: NewPerson (*) ...;
NewBook: NewRecord (*) ...;
(*)
Register: (*) ...
```

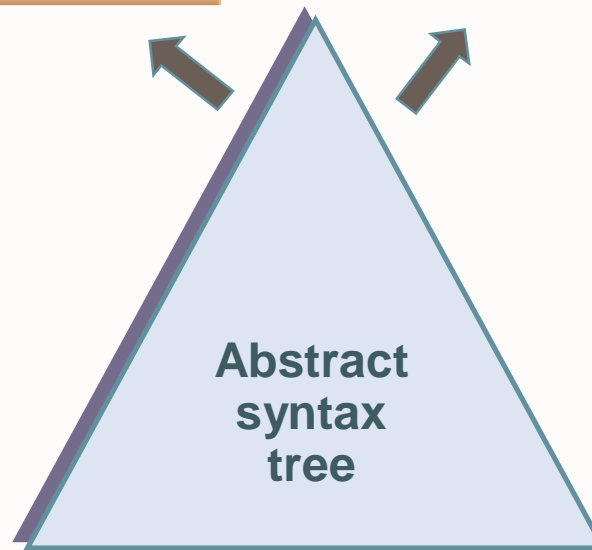
Location: c:\beta\v4.2\freja\demo\recordlib
Info:

Integration of design diagrams and BETA



```
Record: { ... };  
Person: Record { ... }  
Student: Person{ ... }  
Employee: Person{ ... }  
Book: Record{ ... }
```

Common
Representation



Summary on Mjølner CASE tool

Issues for discussion: associations

- Clear semantics for associations necessary
- In Mjølner a specific semantics of associations based on a set of *association classes* and *reference composition* has been defined

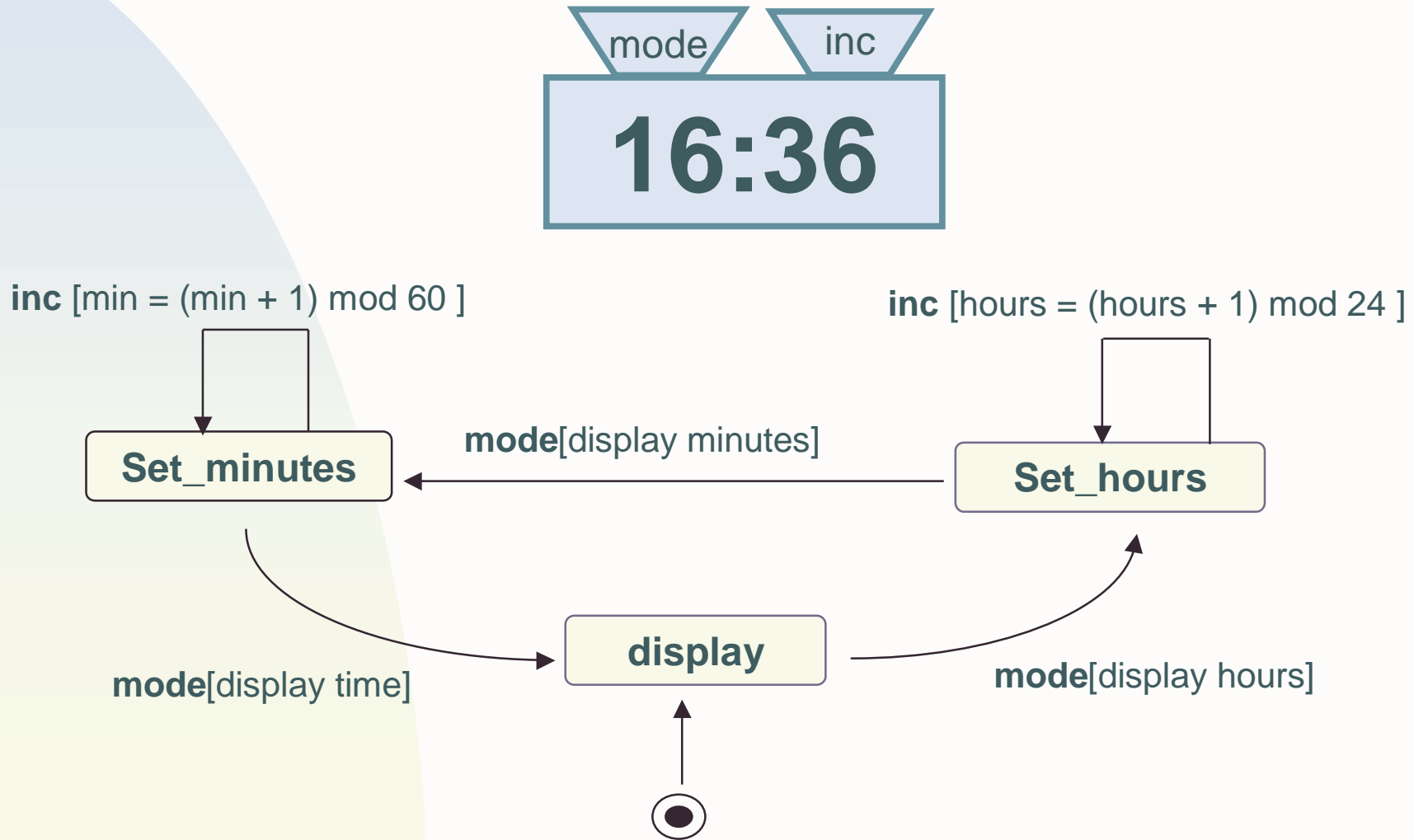
The Mjølner CASE tool supports the use of

- one language for analysis, design and implementation
- *for class diagrams*

Dynamic aspects - state machines

- Dynamic aspects
 - ◆ Sequence diagrams
 - ◆ Collaboration diagrams
 - ◆ State machines (state charts)
 - ◆ ...
- Useful, but a lot of time is often wasted to keep diagrams consistent and up-to-date
- Better integration with programming language is needed
 - ◆ Usual problems with code generation and reverse engineering
- Experiments with direct support for state machines in BETA

State machine for Watch



Implementation of state

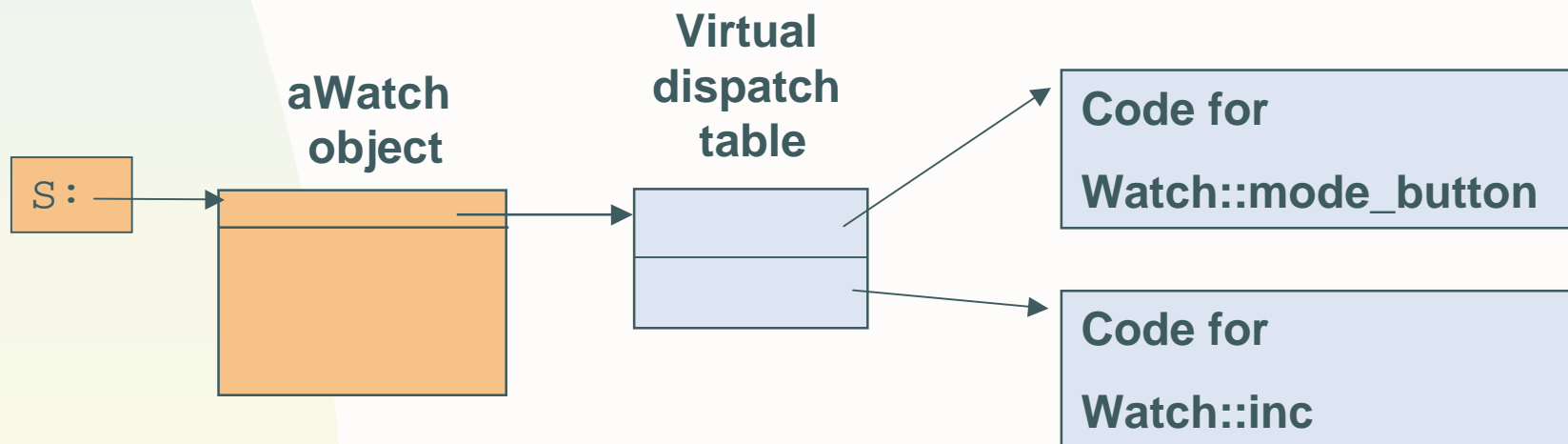
- Explicit state variable
 - ◆ switch on state in each operation
- State pattern
- Direct language support
 - ◆ Dynamic inheritance in Self
 - ◆ State support in BETA

Implementation of virtual functions

```
class Watch
{ proc mode_button():< {} ;
  proc inc():< {} ;
  ...
};
```

`S.mode_button()` call `state[0][0]`

`S.inc()` call `state[0][1]`



Class Watch with several states

```
class Watch: {
    LCD: ref DigitalDisplay;
    proc mode_button():<;
    proc inc():<;

    state Display: { ... };
    state Set_hours: { ... };
    state Set_Minutes: { ... };

    proc Watch(): {
        LCD.display_time;
        state = Display
    }
}
```

State implementations

```
state Display: {  
  mode_button()::< {  
    LCD.display_hours;  
    state = Set_Hours;  
  }  
}
```

```
state Set_hours: {  
  mode_button()::< {  
    LCD.display_minutes;  
    state = Set_minutes  
  }  
  inc()::< { LCD.inc_hours }  
}
```

```
state Set_Minutes: {  
  mode_button()::< {  
    LCD.display_time;  
    state = Display;  
  }  
  inc()::< { LCD.inc_minutes }  
}
```

Implementation

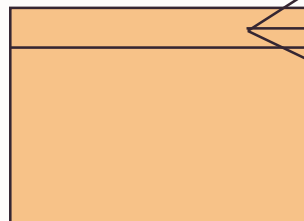
A state is represented as a virtual dispatch table (VDT)

State = Display

State = Set_hours;

State = Set_minutes;

aWatch object



Display VDT

Set_hours VDT

Set_minutes VDT

Extensions and status

- Further extensions
- Substates
 - ◆ Inheritance on state patterns
- Concurrent sub-states
 - ◆ Multiple use of state pattern

Status

- Preliminary implementation exist for BETA

Concurrency in UML

- UML has limited support for modeling concurrency
 - ◆ Active objects
 - ◆ State-machines
 - ◆ Activity diagrams
- We experience major problems with industry projects in real-time embedded system modeling concurrency issues at the UML-level
- UML implies main focus on object/class structures, internal state of objects
- Something more powerful than concurrent state-machines, and activity diagrams is needed.

Experiments with concurrent object-oriented modeling

- Centre for Object Technology
- Participants
 - ◆ **B&O, Danfoss**, Systematic, WM-data, Rambøll, Maersk Line, Odense Steel Shipyard, A.P. Møller
 - ◆ **Technological Institute**, Danish Maritime Institute
 - ◆ **Aarhus University**, University of Copenhagen, Southern Danish University
- Sponsored by
 - ◆ The Danish National Centre for IT Research
 - ◆ The Danish Ministry of Trade and Industries

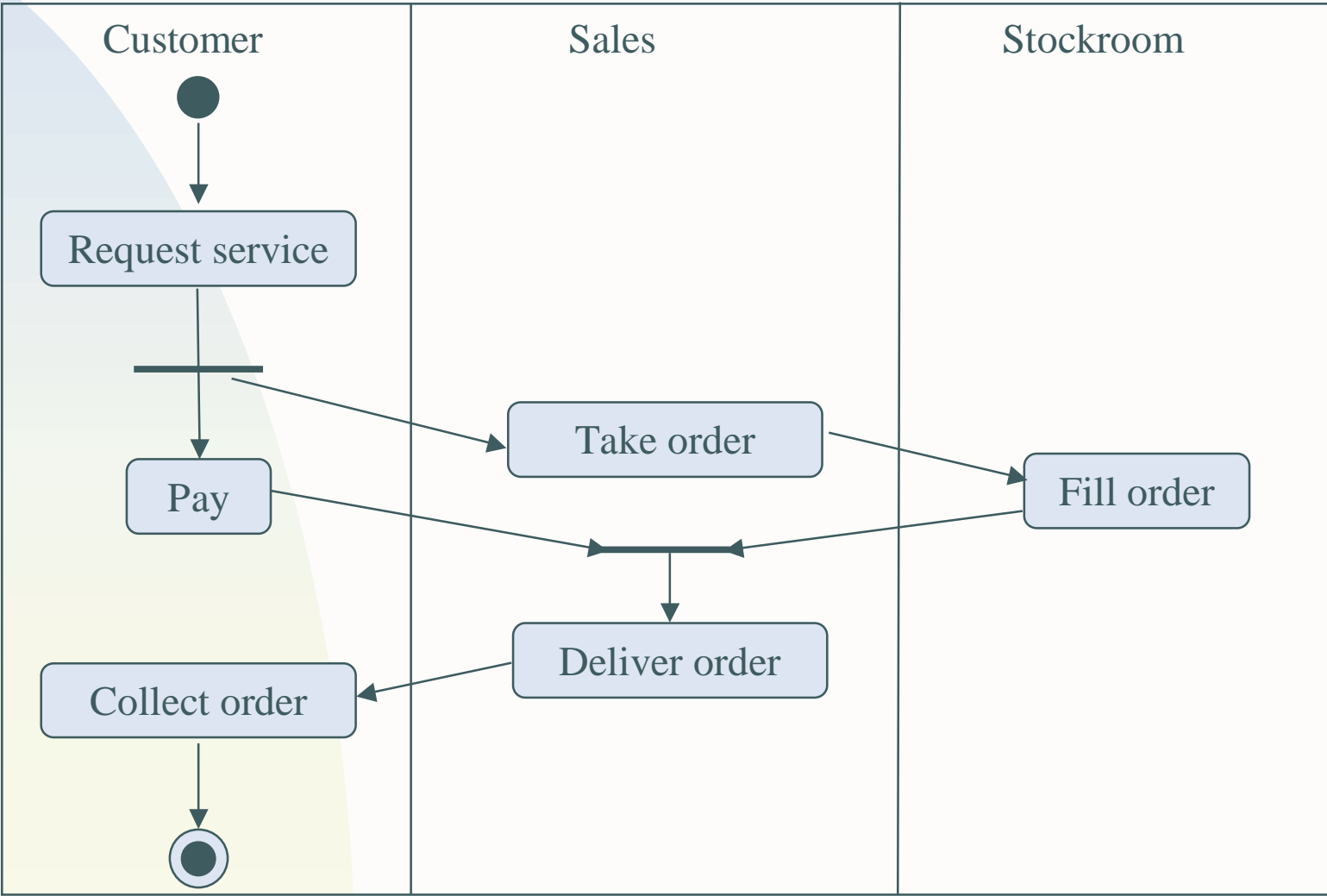
Graphical modeling languages for objects

- UML: Activity diagrams
- Use-case maps
- Petri Nets

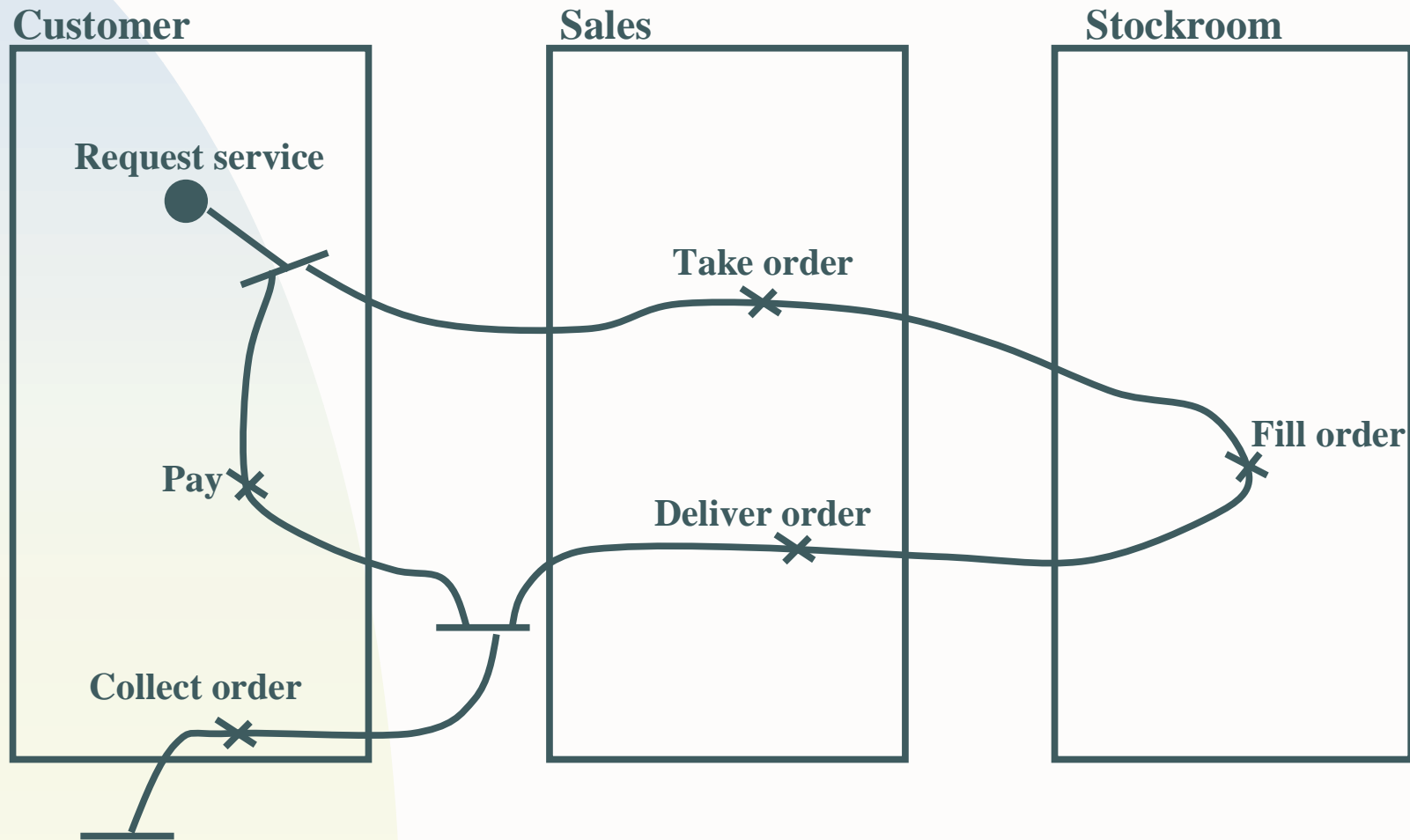
Activity diagrams

- A recent add-on to UML
 - ◆ Action state: description of an activity
 - ◆ Fork & join
 - ◆ Guarded conditions
 - ◆ etc.
- Often useful: better than nothing
 - ◆ Easy to draw an overview of concurrent activities of a system
- Difficult to find a precise semantic description
- Limited applicability

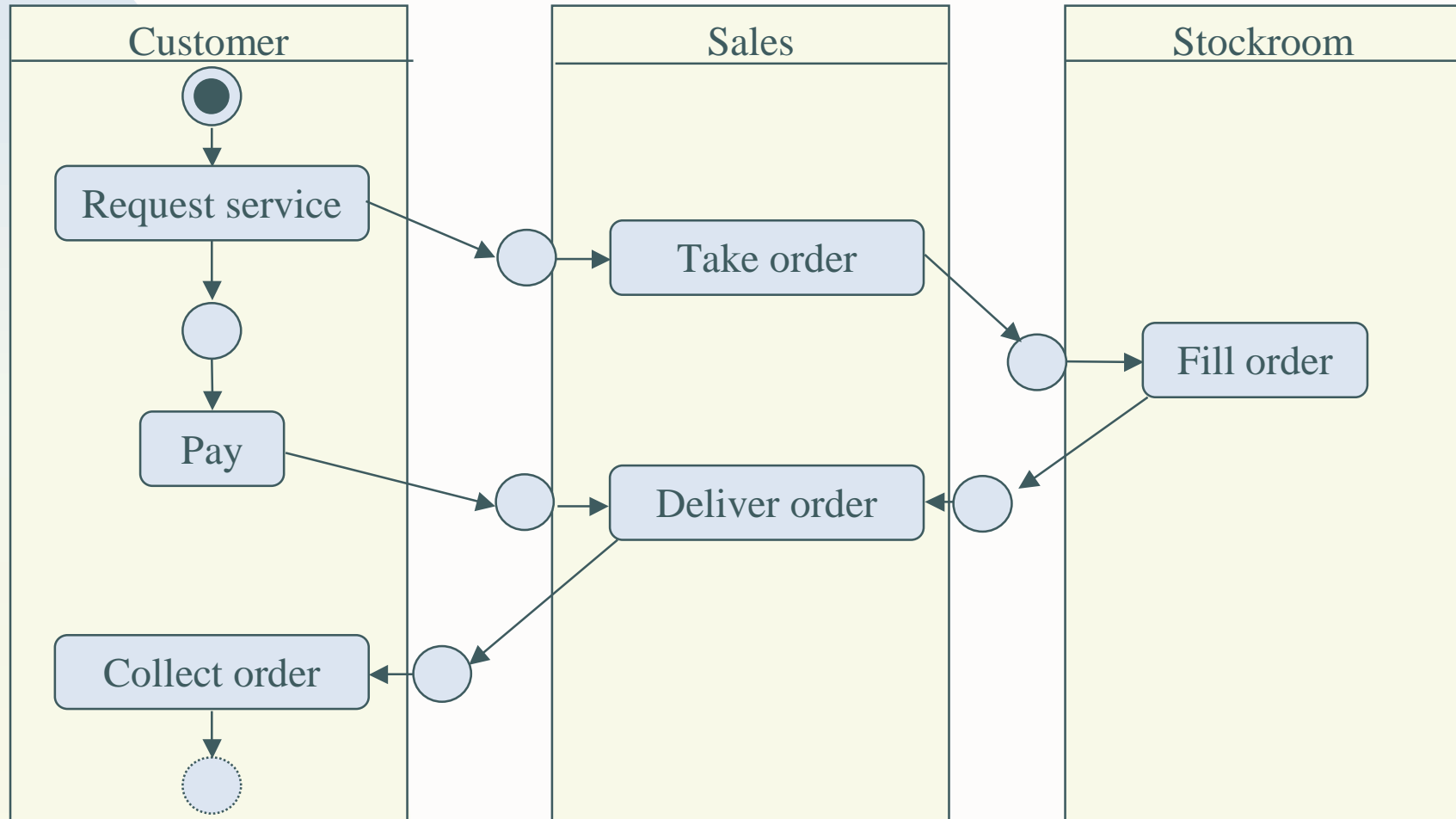
Activity diagram with swimlanes



Example of Use-case map



Petri Net in object diagram



Requirements for graphical

- Provide overview of concurrent activities
- Scenario-based as well as system-based
- Handle synchronization
- Easy to identify critical regions
- Smooth transition from high-level abstract descriptions to complete descriptions
- Dynamic replacement of objects
- Short learning curve
 - ◆ State-machines and activity diagrams are easy to learn
- Straight-forward mapping to (concurrent) object-oriented language

Distributed systems

- The Simula concepts have been a good foundation for OO programming systems
- For distributed systems a new set of concepts for handling location and time is needed
 - ◆ Kristen Nygaard: ECOOP'97

5. Summary

- **BETA**
 - ◆ pattern: unification of programming language abstraction mechanisms
 - ◆ basic primitives and abstraction mechanisms for building concurrency abstractions
 - ◆ emphasis on modeling as well as implementation
- **Issues**
 - ◆ better concurrency abstractions are needed
 - ◆ better support for concurrent object-oriented modeling at the "UML"-level

Contact information

- **Ole Lehrmann Madsen**
- **Computer Science Department**
- **Aarhus University**
- **Åbogade 34, DK-8200 Århus N**
- **DENMARK**
- **Phone: +45 89 42 56 70**
- **E-mail: `ole.l.madsen@daimi.au.dk`**