Egon Börger

Ambient Abstract State Machines

Visiting ETH Zürich, Department of Computer Science
on sabbatical leave from Università di Pisa, Italy

Joint work with Vincenzo Gervasi and Antonio Cisternino

ASMs and Temporal Logic: my interaction with Amir Pnueli

- Idea (1990) triggered by experience made with applying ASMs for
  - definition of ISO-Prolog semantics and WAM implementation
  - verification of Prolog2WAM compilation

Fact: Tarski structures (or algebras) as

states of ASMs evolve over time

Conclusion: a perfect fit should be to

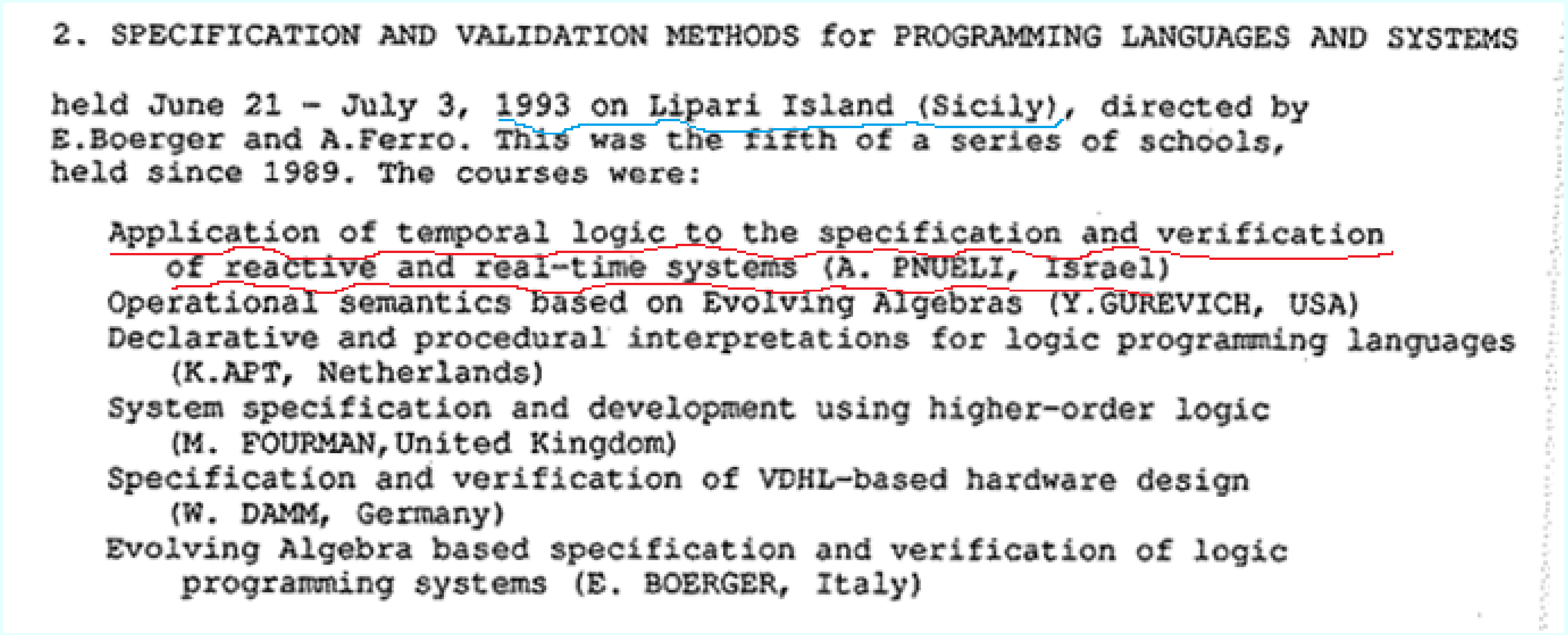
use first order temporal logic for ASM verifications & model checking

i.e. for mathematical verifications of ASM behaviors (proving or model-checking state-related runtime properties, not restricted to in-/output behavior) and their time-based logical analysis

- Invitation to lecture in Lipari School (Manna/Pnueli Books 1991/95)
  - 1993 Amir Pnueli: *Specification and Validation Methods* (OUP 1995)
  - 1997 Zohar Manna: *Architecture Design and Validation Methods*

Amir Pnueli at Lipari Summer School 1993: EATCS Report

From EACSL President in EATCS Bulletin 51, 1993, p.66



2. SPECIFICATION AND VALIDATION METHODS for PROGRAMMING LANGUAGES AND SYSTEMS

held June 21 - July 3, 1993 on Lipari Island (Sicily), directed by E.Boerger and A.Ferro. This was the fifth of a series of schools, held since 1989. The courses were:

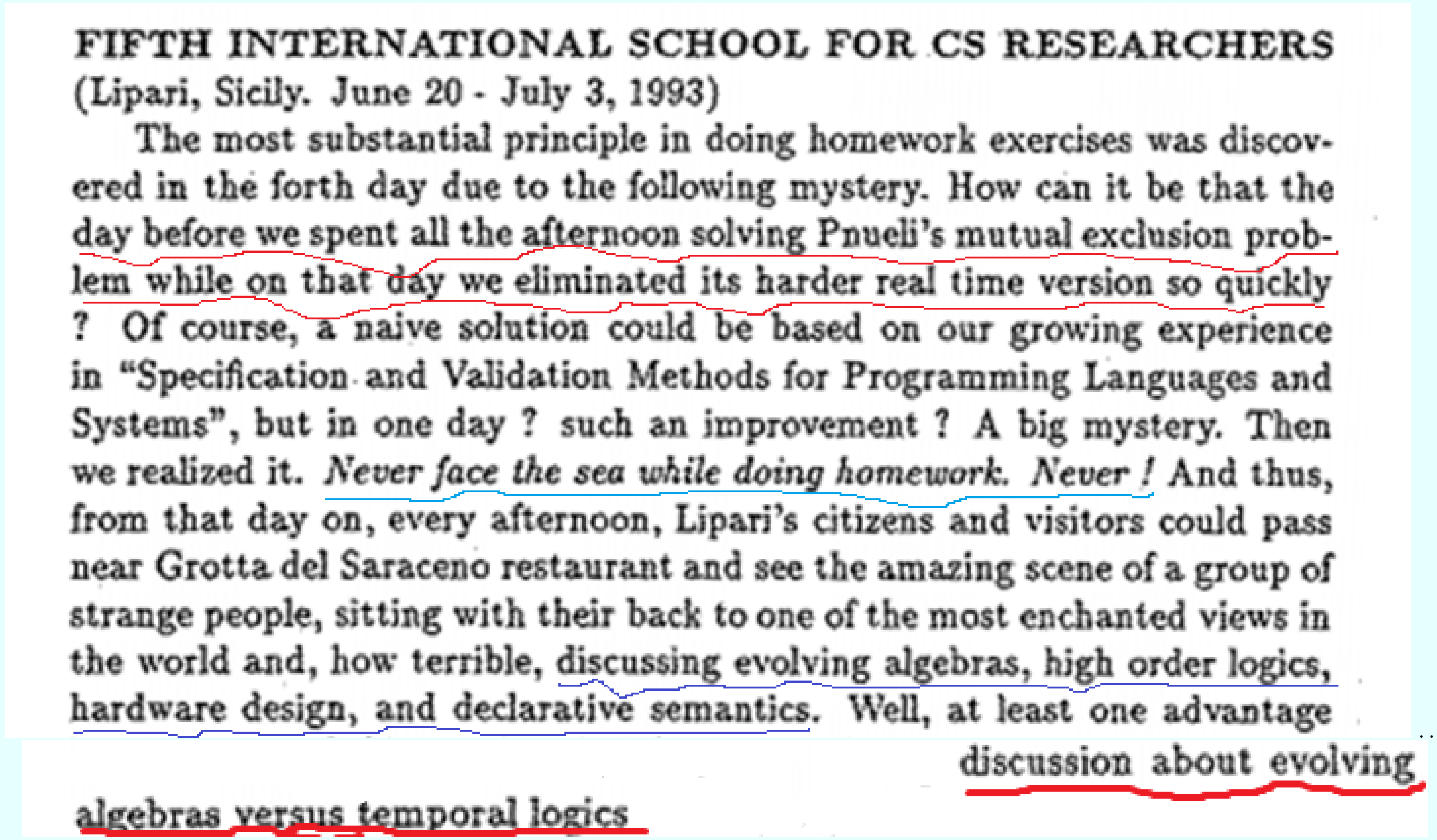
- Application of temporal logic to the specification and verification of reactive and real-time systems (A. PNUELI, Israel)
- Operational semantics based on Evolving Algebras (Y.GUREVICH, USA)
- Declarative and procedural interpretations for logic programming languages (K.APT, Netherlands)
- System specification and development using higher-order logic (M. FOURMAN,United Kingdom)
- Specification and verification of VDHL-based hardware design (W. DAMM, Germany)
- Evolving Algebra based specification and verification of logic programming systems (E. BOERGER, Italy)

From ACM Portal Consortium Swiss Academic Libraries: ETH Zurich

Zohar Manna and Amir Pnueli: Verification of parameterized programs
in: *Specification and Validation Methods* (Ed. E. Börger)
Oxford University Press, 1995, 167-230, ISBN:0-19-853854-5

Amir Pnueli at Lipari School 1993: a student's report

From Orna Bernholtz (CS, Technion, Haifa): EATCS Bull. 51, 66-68



FIFTH INTERNATIONAL SCHOOL FOR CS RESEARCHERS
(Lipari, Sicily. June 20 - July 3, 1993)

The most substantial principle in doing homework exercises was discovered in the forth day due to the following mystery. How can it be that the day before we spent all the afternoon solving Pnueli's mutual exclusion problem while on that day we eliminated its harder real time version so quickly ? Of course, a naive solution could be based on our growing experience in "Specification and Validation Methods for Programming Languages and Systems", but in one day ? such an improvement ? A big mystery. Then we realized it. *Never face the sea while doing homework. Never !* And thus, from that day on, every afternoon, Lipari's citizens and visitors could pass near Grotta del Saraceno restaurant and see the amazing scene of a group of strange people, sitting with their back to one of the most enchanted views in the world and, how terrible, discussing evolving algebras, high order logics, hardware design, and declarative semantics. Well, at least one advantage ...

discussion about evolving algebras versus temporal logics

Cf. Spec meths which combine transition systems with temporal logic

NB. Vincenzo Gervasi a PhD student of the 1997 Lipari school

Context and Goal of Ambient ASMs

- Work on a high-level model for client/server WEB systems
  - for a comparative (experimental and mathematical) analysis of major current WEB application architectures
- This goal implies the need to define a general ambient concept which is flexible enough to support
  - current system modeling and programming practice
    - to *isolate states* of agents concurrently executing in heterogeneous environments
      - statically: scope, module, package, library, etc.
      - dynamically: process instances, threads, executing agents, etc.
    - to speak and reason about *mobility features* (concerning places where agents perform actions)
  - *modularization* of specifications and proofs of their properties

Approach: Parameterization of ASMs

Use ASM (= FSM where states are Tarski structures) framework to:

- achieve desired *generality* (via Gurevich's ASM Thesis 2000)
- permit combination of experimental *validation* (by machine executions) and mathematical *verification* of properties of interest
- exploit simplicity of semantical foundation of *parameterization*

  $$f(x) = f(params, x)$$

  in particular when used with implicit (hidden) parameters

  – Idea: introduce implicit parameter **curamb** expressing a context for evaluation of terms and execution of machines

  – Executions of M in ambient exp can then be described by

  $$\mathbf{amb}\ exp\ \mathbf{in}\ M$$

  through binding **curamb** to exp

  - supporting conventional implicit oo parameterization

    $$this.f(x) = f(x)$$

Transforming ambient ASMs into standard ASMs

- For *location* symbol f:

  $f(t_1, \dots, t_n)^* = f(\mathbf{curamb}, t_1^*, \dots, t_n^*)$
- For dot-terms: $t \,.\, f(s_1, \dots, s_n))^* = f(t^*, s_1^*, \dots, s_n^*)$
- For logical variable, rule name, *ambient independent* fct symbol f:

  $f(t_1, \dots, t_n)^* = f(t_1^*, \dots, t_n^*)$

- For rules:

  $(f(t_1, \dots, t_n) := t)^* = (f(t_1, \dots, t_n)^* := t^*)$ // location symb f

  $(\mathbf{amb}\ t\ \mathbf{in}\ R)^* = (\mathbf{let\ curamb} = t^*\ \mathbf{in}\ R^*)$

The rest goes by induction

$$\mathbf{skip}, \mathbf{par}, \mathbf{if\ then\ else}, \mathbf{forall}, \mathbf{choose}, \mathbf{let}, \mathbf{seq}, \dots$$

Looking for Applications to Test the Definition

- Static naming disciplines: *isolation of states*
- Dynamic naming disciplines: isolation *of computations*
  Exls: Multi-Threading, Process Instantiation
  – MultiThreadJavaInterpreter
  – ThreadPoolExecutor task management in J2SE 5.0
- Memory sharing disciplines: model for Visitor pattern
- Cardelli's and Gordon's calculus for mobility of agents
- Characteristic oo programming patterns
  – *Delegation* (capturing conventional patterns Template, Responsibility, Proxy, Strategy, State, Bridge)
  – *Incremental refinement*: Decorator
  – *Encapsulation*: Memento
  – *Views*: Publish-Subscribe

Extending SingleThreadJavaInterpreter for Concurrency

- mono-core involves *thread context saving/restoring upon rescheduling*
- synchronization involves
  - active threads being put to wait when needed locks are not available
  - notifications about availability of locks

One can simplify (and generalize for multi-core archs) by abstracting from rescheduling details via providing context to Run via **curamb**

```
MultiThreadJavaInterpreter =
    let $q = schedule(\{t \in Thread \mid Runnable(t)\})$
        //requested locks if become available must be acquired
    HandleLockAcquisition($q$)  seq  Run($q$)
  where Run($q$) =
    if $Active(q)$ and $q = executingThread$ then
        amb $q$ in SingleThreadJavaInterpreter // JBook
```

JBook Submachines for Lock Acquisition

The unique *executingThread* (mono-core) may not be *Active* because waiting for lock availability (synchronizing or notified, but now runnable).

```
$\textsc{HandleLockAcquisition}(q) =$
  **if** $q = executingThread$ **then**
    **if not** $Active(q)$ **then** $\textsc{AcquireLocks}(q)$
  **else** $\textsc{MakeExecutingActive}(q)$
$\textsc{MakeExecutingActive}(q) =$
  $Active(q) := true$
  $executingThread := q$
  $\textsc{AcquireLocks}(q)$
$\textsc{AcquireLocks}(q) =$
  **if** $Synchronizing(q)$ **then** $\textsc{Synchronize}(q)$
  **if** $Notified(q)$ **then** $\textsc{WakeUp}(q)$
```

Thread Pool Management (J2SE 5.0 ExI)

- Goal: separate $\textsc{Run}$ning an application from thread management
  - *assignment* of threads to tasks upon $\textsc{TaskEntry}$
  - *decoupling* of threads from tasks upon $\textsc{TaskCompletion}$
  - *creation* of threads
  - *suspension* of threads
    - making them idle to possibly $\textsc{RunTaskFromQueue}$
  - *deletion* of threads
    - if one cannot any more $\textsc{RunTaskFromQueue}$ so that the thread has to $\textsc{Exit}$

$$
\begin{aligned}
&\textsc{ThreadPoolExecutor} = \\
&\quad \textsc{TaskEntry} \\
&\quad \textsc{TaskCompletion} \\
&\quad \textsc{TaskFromQueueOrExit}
\end{aligned}
$$

$\textsc{TaskEntry}(task) = $ **if** $Enters(task)$ **then**

```
if $| CreatedThread | < corePoolSize$ then // fill $corePoolSize$
  let $t$ = new $(CreatedThread)$ in $\textsc{Run}(t, task)$
elseif $| CreatedThread | < maxPoolSize$ then // use $Idle$ threads
  if forsome $t \in CreatedThread\ Idle(t)$ then
    choose $t \in \{t \in CreatedThread \mid Idle(t)\}$ $\textsc{Run}(t, task)$
  else
    if $BlockingFreePlaceable(task, queue)$ then
      $\textsc{Insert}(task, queue)$ // first fill $queue$ before creating threads
    else let $t$ = new $(CreatedThread)$ in $\textsc{Run}(t, task)$
else
  if forall $t \in CreatedThread\ Running(t)$ then
    if $| queue | < maxQueuesize$ then $\textsc{Insert}(task, queue)$
      else $\textsc{Reject}(task)$
```

Decoupling thread from task upon completion

```
$\textsc{TaskCompletion}(task, thread) =$
    **if** $thread \in CreatedThread$ **and** $Completed(task, thread)$
        **and** $Running(thread)$ **then**
            **if** $queue \neq empty$ **then** $\textsc{RunTaskFromQueue}(thread)$
            **else**
                $Idle(thread) := true$
                $completionTime(thread) := now$
```

Reassign idle thread or delete it upon timeout

```
$\textsc{TaskFromQueueOrExit}(thread) =$
  if $Idle(thread)$ and $thread \in CreatedThread$ then
    if $now - completionTime(thread) \leq keepAliveTime(thread)$
      and $queue \neq empty$
    then $\textsc{RunTaskFromQueue}(thread)$
    elseif $\mid CreatedThread \mid > corePoolSize$ then
      $\textsc{Delete}(thread, CreatedThread)$
```

Run: application logic interface to thread management

```
$\textsc{Run}(thread, task) =$
    $program(thread) :=$
        $\textbf{amb } task \textbf{ in } \textsc{Execute}(thread)$
    $Running(thread) := true$

$\textsc{RunTaskFromQueue}(thread) =$
    $\textbf{let } task = next(queue)$
        $\textsc{Run}(thread, task)$
        $\textsc{Delete}(task, queue)$
```

Ambient separation in behavioral interfaces supports modular verifns:

- ASM-based analysis of *C# thread model* (LNCS 3052, TCS 343)
- Proofs for conservative theory extensions corresponding to incremental model extensions in Batory/Börger: *Modularizing Theorems for Software Product Lines*: The Jbook Case Study. J.UCS 2008

Mobile Agents (Cardelli & Gordon)

- ambient processes $n[P]$ interpreted as process P located to run at n
- $n[P]$ definable in ASM framework by $\mathbf{amb}\ n\ \mathbf{in}\ P$
- tree structure induced by the nesting of ambients:
  - $ambName$, element of a domain $AmbName$, considered as root of the tree induced by $\mathbf{amb}\ n\ \mathbf{in}\ P$, which is also identified with n
  - $locAg(n)$: (possibly empty) dynamic set of (non-ambient) processes, say $P_1, \ldots, P_p$, called local agents of the ambient process and viewed as running at n
  - $subAmb(n)$: (possibly empty) dynamic set of subambients, say $\mathbf{amb}\ m_1\ \mathbf{in}\ Q_1$, …, $\mathbf{amb}\ m_q\ \mathbf{in}\ Q_q$
  - $ambBody(n) = P$ in $\mathbf{amb}\ n\ \mathbf{in}\ P$ is interpreted as parallel composition of the elements of $subtrees(n)$

$$P = P_1 \mid \ldots \mid P_p \mid \mathbf{amb}\ m_1\ \mathbf{in}\ Q_1 \ldots \mid \mathbf{amb}\ m_q\ \mathbf{in}\ Q_q$$

ASM Interpreter for ambient changing operations

- ambient process change by three actions: Entry, Exit, Open
- can be viewed as tree operations performed on derived location $curAmbProc$ (read: a set of nodes equipped with tree structure)

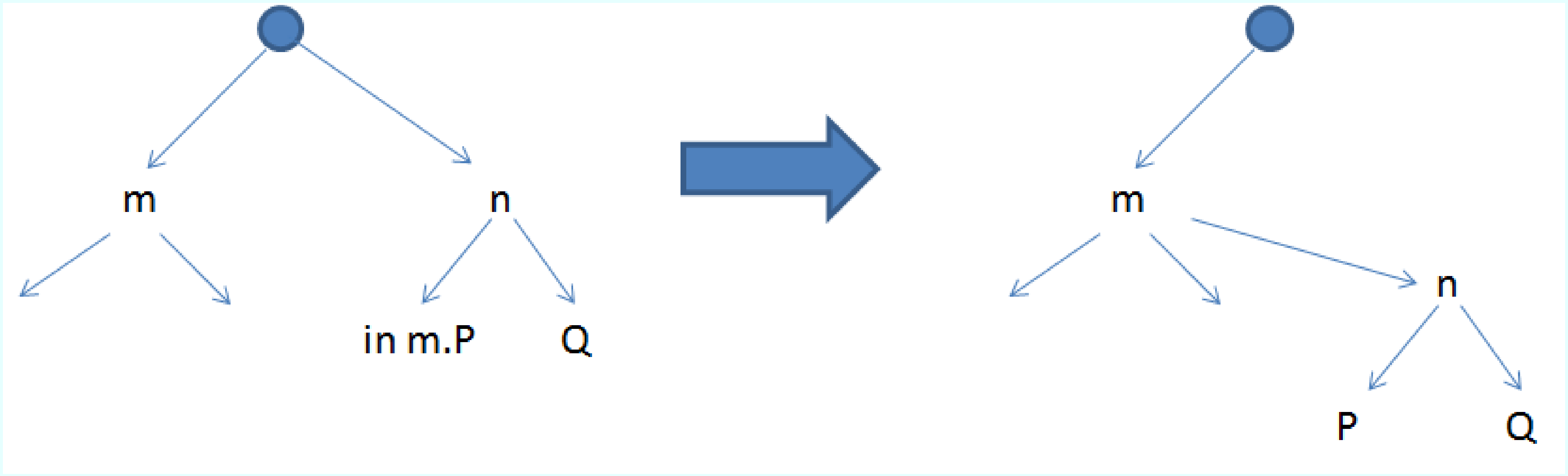
```
$\textsc{MobileAgentsInterpreter} =$
    $\textbf{choose}\ R \in \{\textsc{Entry}, \textsc{Exit}, \textsc{Open}\}\ \textbf{in}$
        $R$
```

- restriction operator definable: $(\nu n)P = P(n/new(AmbName))$
- none of the remaining 17 structural congruence rules of Cardelli & Gordon needed

ENTRY into an ambient as subambient

Entry from where?

- Sibling ambient chosen as neighbourhood from where to enter into an ambient m
- $S = n[in\ m.P \mid Q]$ becomes $n[P \mid Q] \in subtrees(m)$
  – if $sibling(S)$ contains a process with $ambName\ m$



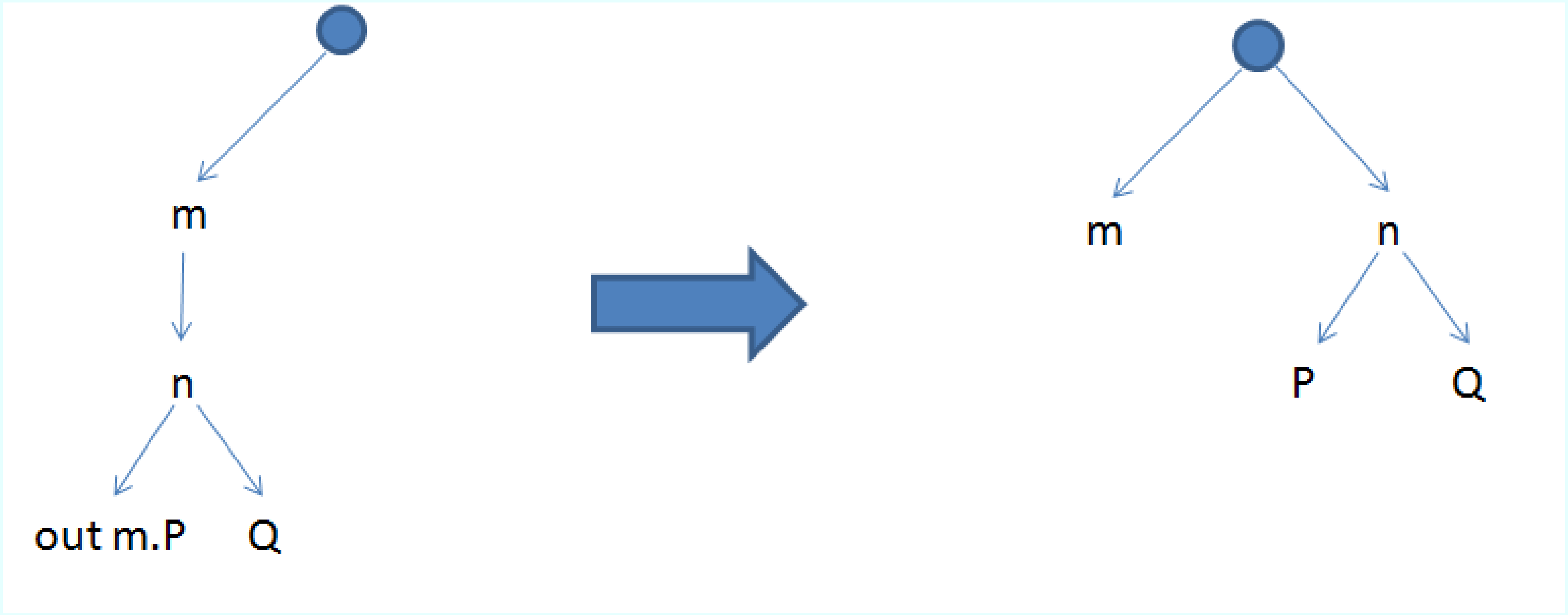
Entry

```
$=$ **if** $EntryAction(curAmbProc) \neq \emptyset$ // there is some entry action
**then choose**
  $S =$ **amb** $n$ **in** $((in\ m.P) \mid Q) \in EntryAction(curAmbProc)$
  **if** $sibling(S)$ contains a process with ambient name $m$ **then**
    **choose amb** $m$ **in** $R \in sibling(S)$
      $\textsc{Delete}(S, subtrees(parent(m)))$
        // $n$ disappears as sibling of target ambient $m$
      $\textsc{Insert}($**amb** $n$ **in** $(P \mid Q), subtrees(m))$
        // modified $n$ becomes subambient of $m$
**where**
  $EntryAction(curAmbProc) =$
    $\{n \in curAmbProc \mid ambBody(n) = (in\ m.P) \mid Q\}$
```

Exit of a subambient

Exit to where?

- Sibling ambient chosen as neighbourhood where to exit as subambient of an ambient m



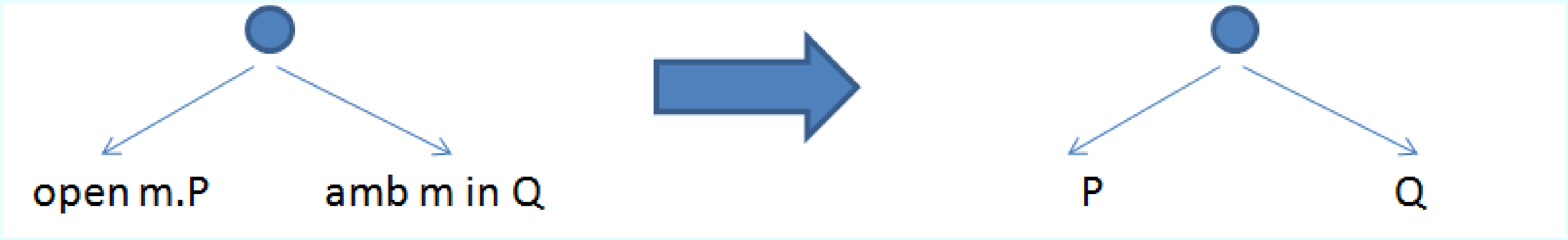
Exit

```
EXIT =
  if $ExitAction(curAmbProc) \neq \emptyset$ // there is some exit action
  then choose
    $S = \textbf{amb}\ n\ \textbf{in}\ ((out\ m.P) \mid Q) \in ExitAction(curAmbProc)$
    if $parent(n) = m$ then
      $\textsc{Delete}(S, subtrees(m))$ // $n$ disappears as subambient of $m$
      $\textsc{Insert}(\textbf{amb}\ n\ \textbf{in}\ (P \mid Q), subtrees(parent(m)))$
        // modified $n$ becomes sibling ambient of $m$
where $ExitAction(curAmbProc) =$
  $\{n \in curAmbProc \mid ambBody(n) = (out\ m.P) \mid Q\}$
```

Ambient dissolving action OPEN

Operating at which level upon a process to open its ambient?

- dissolving the boundary of an ambient named m "located at the same level"
- sibling ambient chosen as neighbourhood
- replaces a subtree pair $(open\ m.P, \mathbf{amb}\ m\ \mathbf{in}\ Q)$ of siblings in $curAmbProc$ by the new siblings pair (P, Q)



Open

```
$= \textbf{if}\ AmbDissolvAction(curAmbProc) \neq \emptyset$
    // there is some ambient dissolving action
$\textbf{then choose}\ S_1 = open\ m.P_1 \in AmbDissolvAction(curAmbProc)$
$\textbf{if}\ sibling(S_1)$ contains a process with ambient $m$ $\textbf{then}$
    $\textbf{choose}\ S_2 = \textbf{amb}\ m\ \textbf{in}\ P_2 \in sibling(S_1) \quad \textbf{let}\ p = parent(S_1)$
    $\textbf{forall}\ i \in \{1, 2\}$
        $\textsc{Delete}(S_i, subtrees(p))$
        $\textsc{Insert}(P_i, subtrees(p))$
```

where

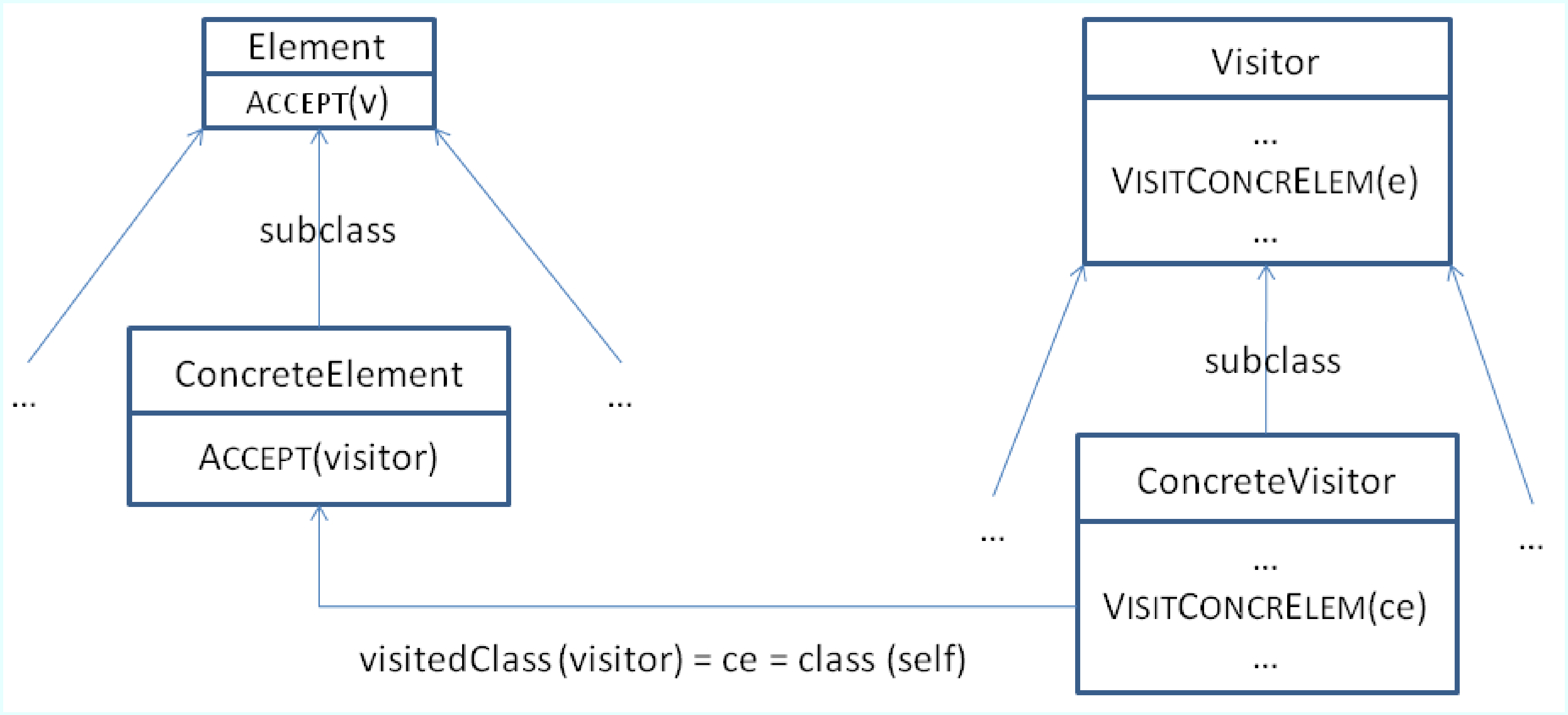
$AmbDissolvAction(curAmbProc) =$

$\{open\ m.P_1 \mid open\ m.P_1 \in curAmbProc\}$

X contains a process with ambient m $=$

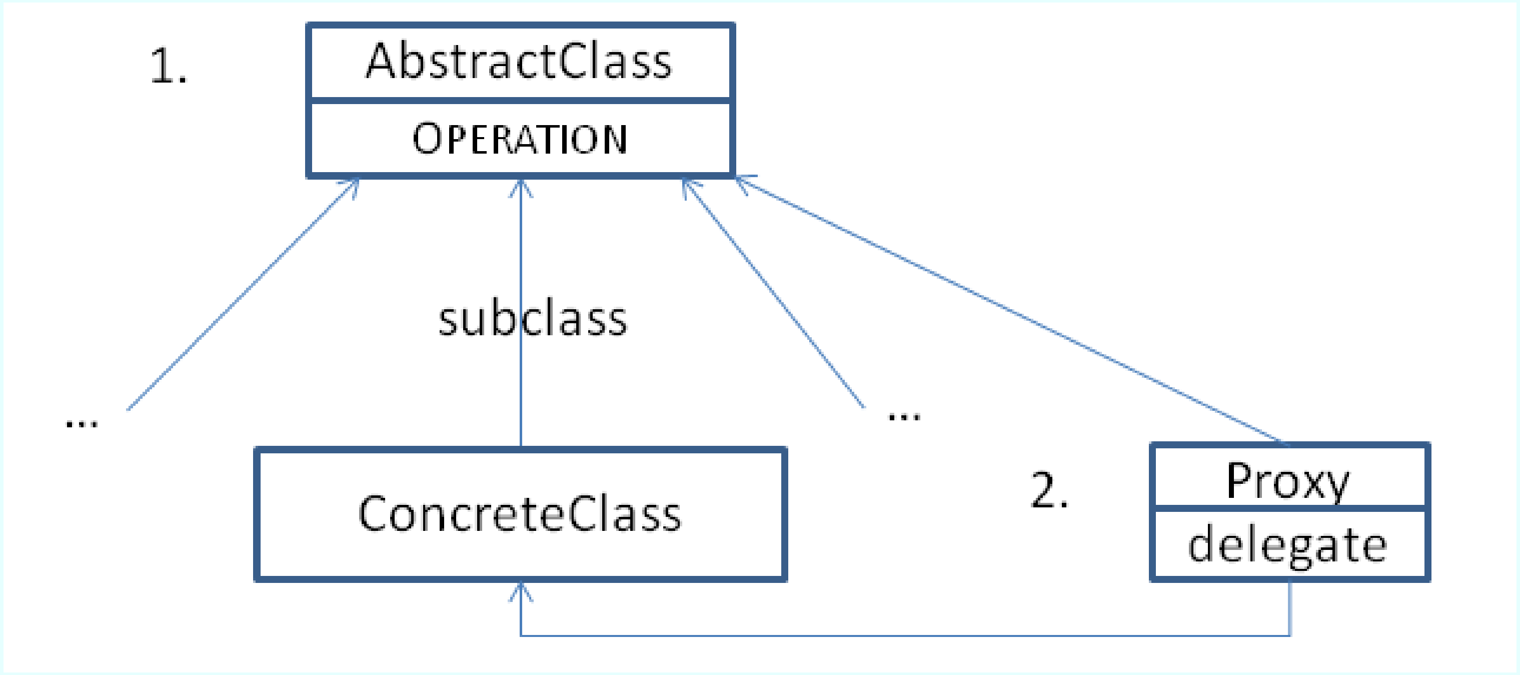
$\textbf{forsome}\ Q \quad \textbf{amb}\ m\ \textbf{in}\ Q \in X$

Sharing memory (Exl: Visitor pattern)



- operation on concrete element ce as $\textsc{VisitConcrElem}$ of a $visitor$ accepted by ce so that $visitor$ can appropriately access ce's state to execute $\textsc{VisitConcrElem}(ce)$
- $\textsc{Accept}(visitor) = \ visitor\ .\ \textsc{VisitConcrElem}(\mathbf{self})$

oo Patterns: Delegation Scheme (1) to subclasses



Delegation equation for $\textsc{Operation}$ calls for specific $Request$

$$\textsc{Delegate}(\textsc{Operation}, delegate)(Request) =$$
$$\textbf{amb } delegate \textbf{ in } \textsc{Operation}_{classOf(delegate)}(Request)$$

Instantiating Delegation by definition of $delegate$

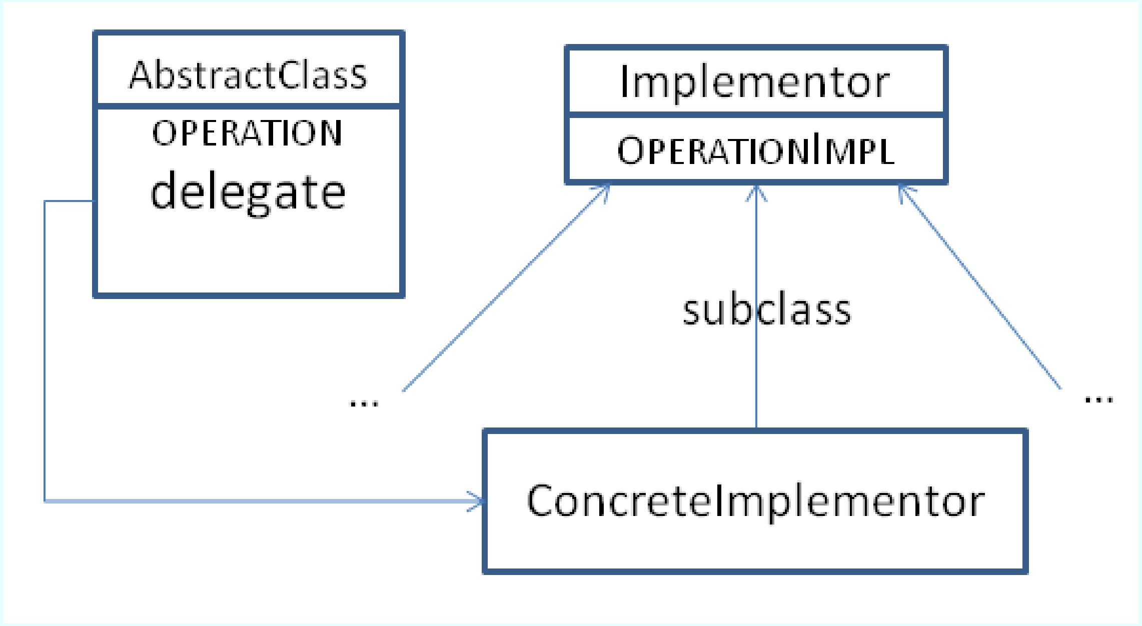
- *external* definition as part of the signature
  - *static*ally determined by:
    - class structure: $Template$ = "skeleton of an algorithm ... deferring some steps to subclasses", i.e. define $delegate = ConcreteClass$
    - data-structure fct: chain traversal in $ChainOfResponsibility$ where (see below) $select = first_{chain}$
  - *dynamic*: $Responsibility$ = "giving more than one object a chance to handle the request", i.e. define

    $$delegate = select(\{o \in ReceivingObj(Request) \mid CanHandle(o, \textsc{Operation})(Request)\})$$
- *internal* definition by a location $delegate$
  - in a dedicated class: $Proxy$ to 'provide a placeholder for another object' so that $delegate$ is 'the real object that the proxy represents'. Dto for $Strategy$, $State$: interchangeable/state dependent impls
  - in $AbstractClass$: $Bridge$

Delegation (2) to 'outsourced' classes (Bridge)

"both the abstractions and their implementations should be extensible by subclassing": implementations become run-time configurable/assignable

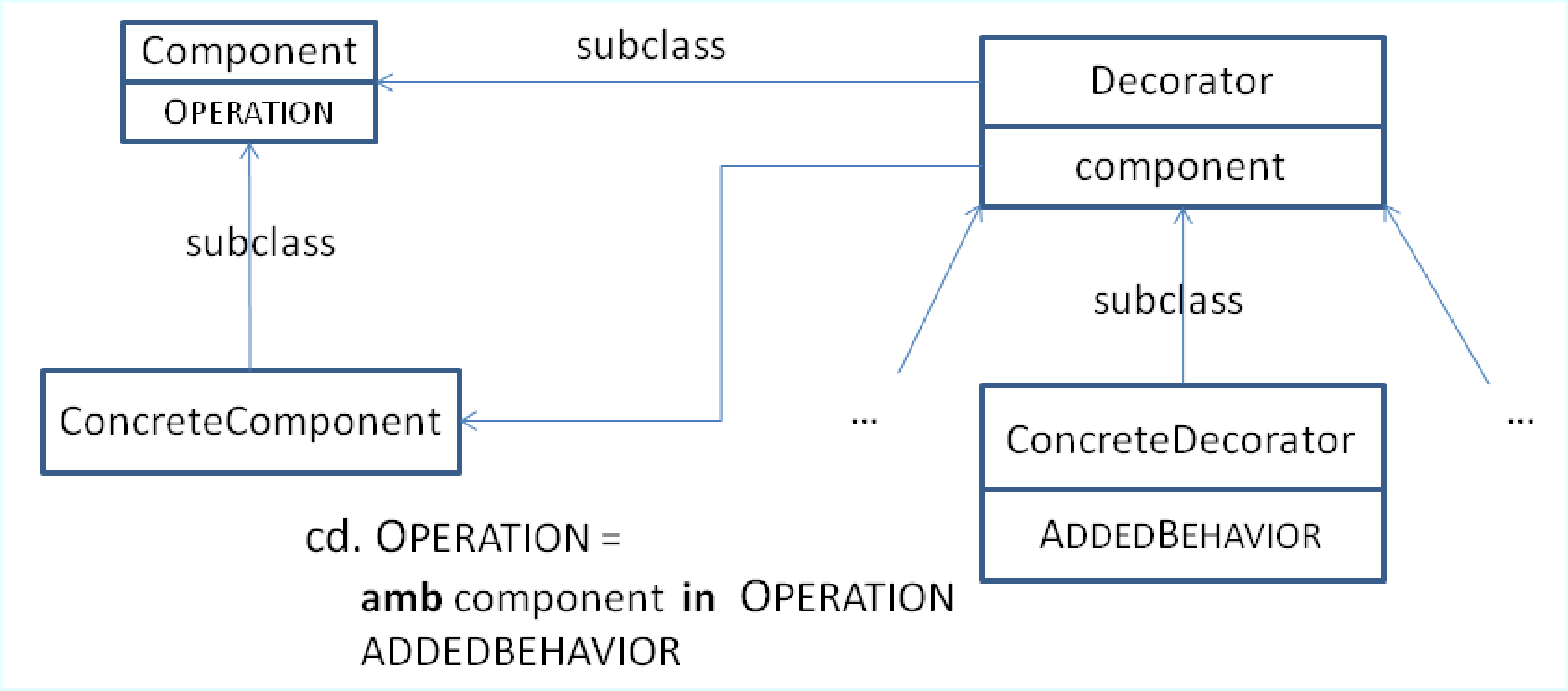
- $delegate$ is an $AbstractClass$ location
- $ConcreteImplementor$ is *subclass of another class* $Implementor$



'Typically the *Implementor* interface provides only primitive operations ... *AbstractClass* defines higher-level operations based on these primitives'

Incremental Refinement (Decorator Pattern)

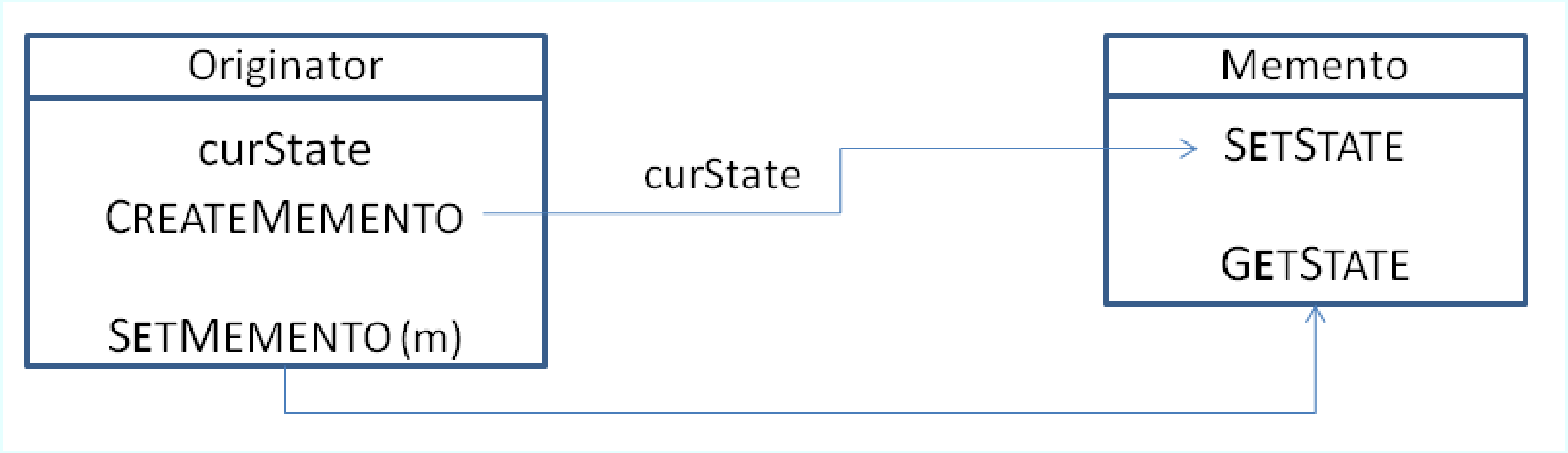
'attach additional responsibilities to an object dynamically' as 'a flexible alternative to subclassing for extending functionality'



Variation: ADDEDBEHAVIOR executed in the *component* ambient

Encapsulation (Memento Pattern)

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



$\textsc{CreateMemento} = \textbf{let } m = \textbf{new } (\mathit{Memento}) \textbf{ in}$

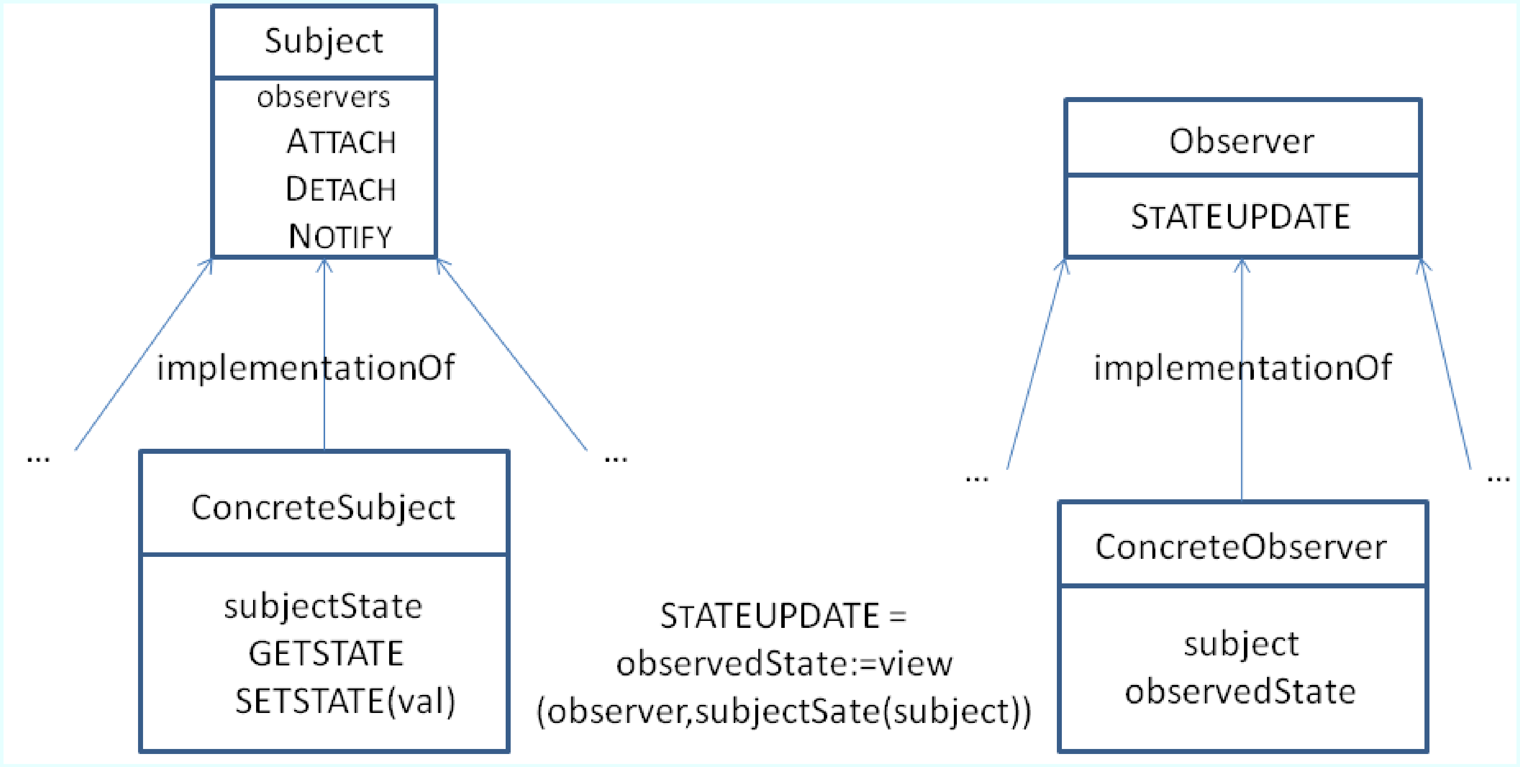
$\quad \textbf{amb } m \textbf{ in } \textsc{SetState}(\mathit{curState})$

$\quad \textsc{Return}\ m$ // ambient can be an entire internal state!

$\textsc{SetMemento}(m) = \textsc{Return}\ \textbf{amb } m \textbf{ in } \textsc{GetState}$

Views (Publish-Subscribe Pattern)

define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated



$$\textsc{StateUpdate} =$$
$$\quad \mathit{observedState} := \mathit{view}(\mathbf{amb}\ \mathit{subject}\ \mathbf{in}\ \ \textsc{GetState})$$

Unfolding Publish-Subscribe Memory Sharing

$\textsc{Notify} = \textbf{forall } o \in \mathit{observers} \ \textbf{ amb } o \textbf{ in } \ \textsc{StateUpdate}$

$\textsc{StateUpdate} =$

$\quad \mathit{observedState} := \mathit{view}(\textbf{amb } \mathit{subject} \textbf{ in } \ \textsc{GetState})$

$\textsc{GetState} = \ \textsc{Return} \ \mathit{subjectState}$

$\textsc{SetState}(\mathit{val}) = (\mathit{subjectState} := \mathit{val})$

Unfolding the definitions shows the intended *memory sharing*:

$\textbf{amb } o \textbf{ in } \ \textsc{StateUpdate}$ // evaluate for $\textbf{curamb} = o$

$= \quad \mathit{observedState}(o) := \mathit{view}(o,$

$\qquad \textbf{let curamb} = \mathit{subject}(o) \textbf{ in } \ (\textsc{GetState})^*)$

$= \quad \mathit{observedState}(o) := \mathit{view}(o, \textbf{let curamb} = \mathit{subject}(o) \textbf{ in}$

$\qquad \textsc{Return} \ \mathit{subjectState}(\textbf{curamb}))$

$= \quad \mathit{observedState}(o) := \mathit{view}(o, \mathit{subjectState}(\mathit{subject}(o)))$

NB. $o, \mathit{subjectState}$ are ambient independent

Conclusion and Outlook: ad maiora

The defined ambient concept seems to support the practice of

- system development (from modeling to programming)
- system verification

by modular (also refinement driven) design and proof techniques.

We are currently using it to

- develop a high-level model for client/server WEB systems
- analyze current WEB application architectures using this model for
  - experiments (testing or model checking runtime properties)
  - mathematical verification (proving runtime properties)

  two good reasons to

advocate encore using temporal logics for analyzing ASMs

building upon dynamic logics for ASMs in ASM-theories in KIV, LTL in ASM-theories in PVS, ASM-logic by Stärk (2001-2005) and Nanchen (Diss ETHZ 2007) and Wang (Diss Kiel 2010)

References

- L. Cardelli and A. D. Gordon: *Mobile Ambients*, LNCS 1378 (1998)
- S. Oaks and H. Wong: *Java Threads*, O'Reilly 2004
- E. Gamma and R. Helm and R. Johnson and J. Vlissides: *Design Patterns*, Addison-Wesley 1994
- R. Stärk and J. Schmid and E. Börger, *Java and the JVM*. Definition, Verification, Validation. Springer 2001
- D. Batory and E. Börger: *Modularizing Theorems for Software Product Lines*: The Jbook Case Study. J.UCS 14.12 (2008) 2059-2082
- R. Stärk and E. Börger, *An ASM specification of C# threads and the .NET memory model*. LNCS 3052 (2004) pp. 38-60
- R. Stärk: *Formal specification and verification of the C# thread model*. TCS 343 (2005)
- E. Börger and R. Stärk, *Abstract State Machines*. A Method for High-Level System Design and Analysis. Springer 2003