The Zonnon Object Model: A Structured Approach to Composability & Concurrency

Jürg Gutknecht
ETH Zürich
September 2005

The Pascal Language Family

- Guiding Principle: „Make it as simple as possible but not simpler“

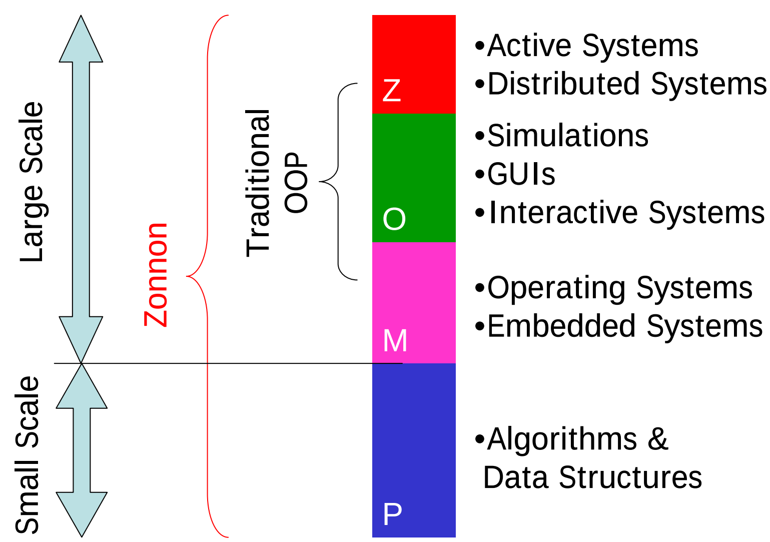
	Language	New Feature	Concept
1970	Pascal	Pointer	A&D
1980	Modula	Module	Systems
1990	Oberon	Type Extension	OOP
2005	Zonnon	Activity	Concurrency

- The Zonnon project has emerged from MS Project 7/7+ initiative
  http://zonnon.ethz.ch

"With a new computer language, one not only learns a new vocabulary and grammar but one opens oneself to an new world of thought"

Niklaus Wirth

Spectrum of Programming



The Object Model

- Modular
- Compositional
- Active

The Object Model

- Modular
- Compositional
- Active

A Simple Interactive Module

```
• module Weekday imports Zeller;
    procedure { public } Compute;
      var d, m, y: integer;
    begin
      read(d);
      while d > 0 do
        readln(m, y);
        writeln(Zeller.Weekday(
          y div 100, y mod 100, m – 2, d);
        read(d)
      end
    end Compute;
  end Weekday.
```

A Simple Interactive Module

```
• module Weekday imports Zeller;
    procedure { public } Compute;
      var d, m, y: integer;
    begin
      read(d);
      while d > 0 do
        readln(m, y);
        writeln(Zeller.Weekday(
          y div 100, y mod 100, m – 2, d);
        read(d)
      end
    end Compute;
  end Weekday.
```


no classes,
no objects,
no inheritance,
no virtual methods,
no overriding,
no static fields

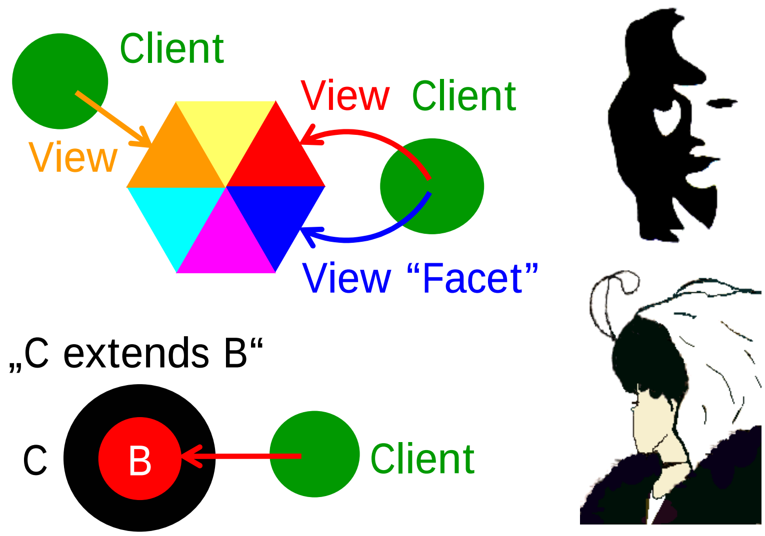
A Simple A&D Module

```
• module Zeller;
    var wd: array 7 of string;
    procedure { public } WeekDay
      (c, y, m, d: integer): string;
      var n: integer;
    begin
      n := entier(2.62*m – 0.2)
        + d + y + y div 4 + c div 4 – 2*c;
      return wd[n mod 7]
    end WeekDay;
  begin
    wd[0] := "Sunday"; wd[1] := "Monday";
    wd[2] := "Tuesday"; wd[3] := "Wednesday";
    wd[4] := "Thursday"; wd[5] := "Friday";
    wd[6] := "Saturday"
  end Zeller.
```

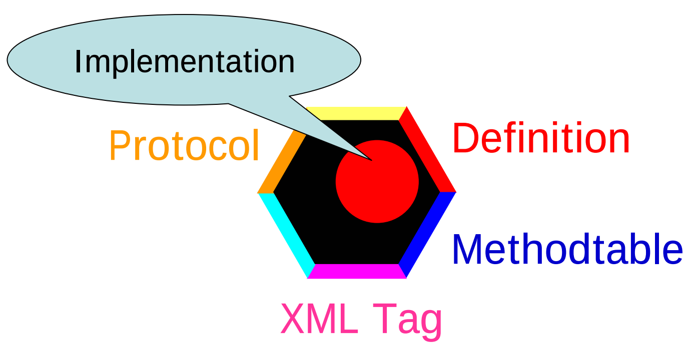
The Object Model

- Modular
- Compositional
- Active

Compositional vs. Hierarchical



Definition vs. Implementation



Example Jukebox (1)

```
definition Store;(*view*)
  procedure Clear;
  procedure Add (s: Songs.Song);
end Store.

implementation Store;
  var rep: Songs.Song;
  procedure Clear;
  begin rep := NIL
  end Clear;
  procedure Add (s: Songs.Song);
  begin s.next := rep; rep := s
  end Add;
begin Clear
end Store.
```

Example Jukebox (2)

```
definition Player;
  var cur: Songs.Song;
  procedure Play (s: Songs.Song);
  procedure Stop;
end Player.

object JukeBox implements Store, Player;
  (*aggregates implementation Store*)
  procedure Play (s: Songs.Song) implements
    Player.Play;
  begin ...
  end Play;
  procedure Stop implements Player.Stop;
  begin ...
  end Stop;
end JukeBox.
```

The Object Model

- Modular
- Compositional
- Active

The Challenge of Concurrency

- Moore's Law
  - ➢Double performance each 1.5 years
- Achieved via
  - ➢Until now: # FLOPS
    - 10 MHz → 100 MHz → 1 GHz → 3.2 GHz
    - Power, Heat ⇒ Stop at ≈ 3.5 GHz
  - ➢From now: Multi CPU cores
    - 1 CPU → 2 CPU → 8 CPU →
- Challenge of exploiting multiple CPU
  - ➢Support needed from programming language

Some Language Design Goals ...

- Integrate concurrency with OOP
- Replace library calls with language constructs
- Abstract from deployment details (central or distributed)
- Present active objects as self-contained units with programming-language independent interfaces

Two New Constructs

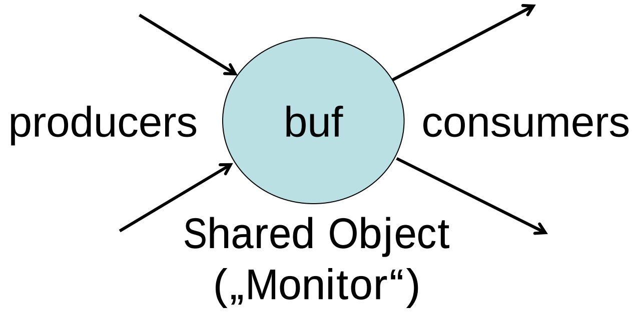
- The *await* statement
- Activities

The *await* Statement

- Used in shared objects for waiting on a local condition to be established by other activities
- Replaces the method of signalling by an autonomous concept

Example: Finite Buffer

- Scenario: consumers and producers communicating via finite buffer
- m = number of free slots in buffer
- n = number of occupied slots in buffer



Finite Buffer with Signals

```
• public void Put (object x) {
    lock(this) {
      while (m == 0) { Monitor.Wait(this); }
      m--; buf[tail] = x;
      tail = (tail + 1) % size;
      n++; Monitor.PulseAll(this);
    }
  }
• public object Get () {
    lock(this) {
      while (n == 0) { Monitor.Wait(this); }
      n--; object x = buf[head];
      head = (head + 1) % size;
      m++; Monitor.PulseAll(this); return x;
    }
  }
```

Cost: unnecessary context switches

Finite Buffer with *await*

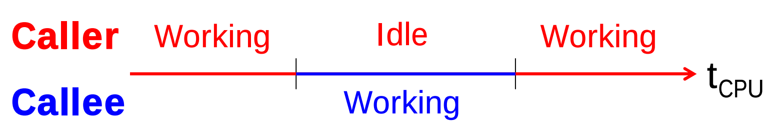
```
• procedure Put (var object x);
  begin
    await (m # 0);
    dec(m); buf[tail] := x;
    tail := (tail + 1) mod size;
    inc(n);
  end Put;
• procedure Get (): object;
    var x: object;
  begin
    await (n # 0);
    dec(n); x := buf[head];
    head := (head + 1) mod size;
    inc(m); return x
  end Get;
```

Activities

- Generalization of the procedural paradigm
- Used for multiple purposes
  - Run independent statements concurrently
  - Run intrinsic activities encapsulated in objects
  - Carriers of comunications

Procedural Paradigm

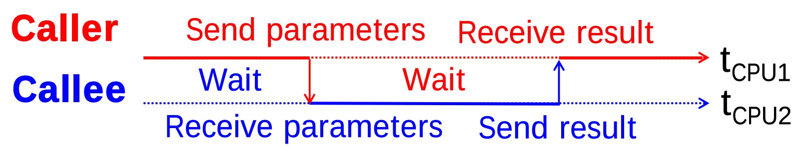
- Directed at single CPU configurations



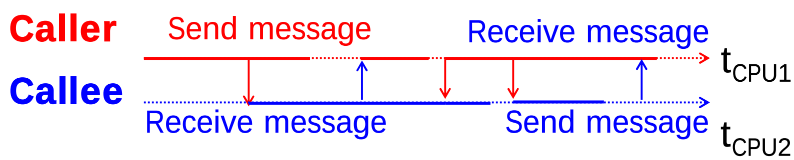
- The same paradigm used in different cases
  - Local procedure call
  - Method call
  - Remote procedure call

New Concept: Activities

- Procedure call as dialog



- Activities as generalized procedures



Scenario 1: Independent Actions

```
• activity A ( … );
    var …
  begin …
  end A;
• activity B ( … );
    var …
  begin …
  end B;
• begin { barrier }
    new A(…); new B(…)
  end
```

Example 1: Quicksort

```
• activity Sort (l, h: integer)
    var i, j: integer;
  begin { barrier }
    … (*partition array l, j & i, h*)
    if l < j then new Sort(l, j) end;
    if i < h then new Sort(i, h) end
  end Sort;
• (*start Quicksort*)
  begin
    new Sort(1, N)
  end
```

Example 2: Active Objects

```
• type X = object
    activity A (…);
      … (*intrinsic behavior*)
    end A;
    activity B (…);
      … (*intrinsic behavior*)
    end B;
    procedure new (…);
      … (*constructor*)
    end new;
  begin { barrier }
    new A(…); new B(…)
  end X;
```

Passive vs. Active

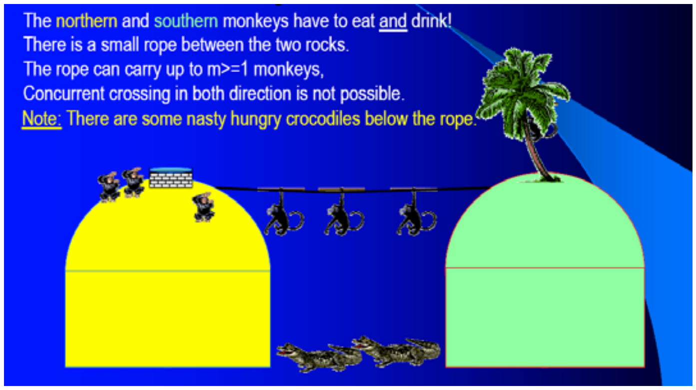
put item
get item

set time
get time

Scenario 2: Object Dialogs

- ```
  type Y = object (*callee*)
    activity D (…): …;
      var t, u: T;
    begin (*dialog*)
      … return t; … u := *; …
    end
  end Y;
  ```
- ```
  var y: Y; d: Y.D; t, u: T;
  begin (*caller*)
    y := new Y;
    d := new y.D; (*active link*)
    t := d(…); … d(u); …
  end
  ```

Example 1: World of Monkeys



The Rope as Shared Resource

```
• module { shared } Rope; (*global view*)
    type
      Monkey = object; (*active*)
      MonkeyMsg = (claim, release);
    var cur, i: integer;
      (*number of monkeys on rope
        > 0 South-North traversal
        < 0 North-South traversal*)
    activity MonkeyDialog (): MonkeyMsg;
  begin
    for i := 0 to 99 do new Monkey () end
  end Rope;
```

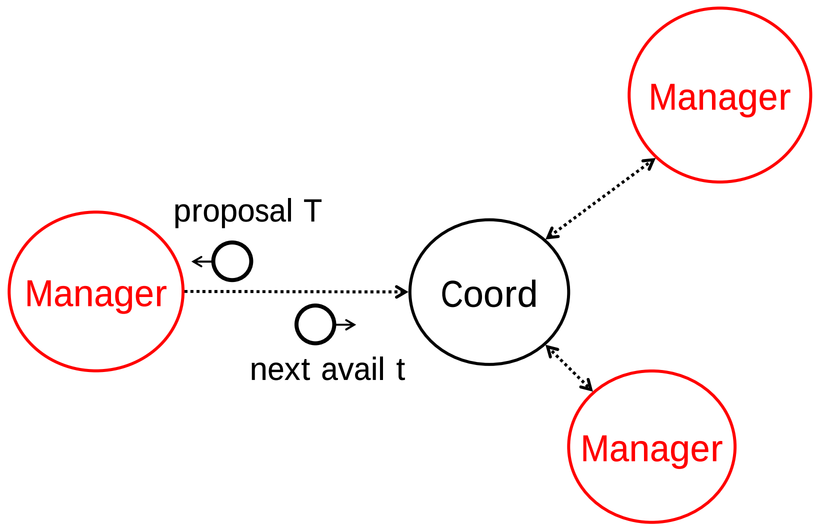
Monkeys as Active Objects

```
• type Monkey = object
    activity LiveOnTheRocks ();
      var res: MonkeyMsg;
        d: Rope.MonkeyDialog;
    begin (*story of life*)
      d := new Rope.MonkeyDialog;
      loop
   ⇨    passivate(Random.Next()); (*eat/dr*)
        res := d(MonkeyMsg.claim);
      end
    end LiveOnTheRocks;
  begin { barrier }
    new LiveOnTheRocks()
  end Monkey;
```

The Monkey Dialog Activity

```
• activity MonkeyDialog (): MonkeyMsg;
    var req: MonkeyMsg;
  begin
    loop
      req := *; (*South-North request*)
 ⇨    await (0 <= cur) & (cur < m);
      inc(cur); passivate(100);
      dec(cur); return MonkeyMsg.release;
      req := *; (*North-South request*)
      await (0 >= cur) & (cur > -m);
      dec(cur); passivate(100);
      inc(cur); return MonkeyMsg.release
    end
  end MonkeyDialog;
```

Example 2: Next Meeting Time



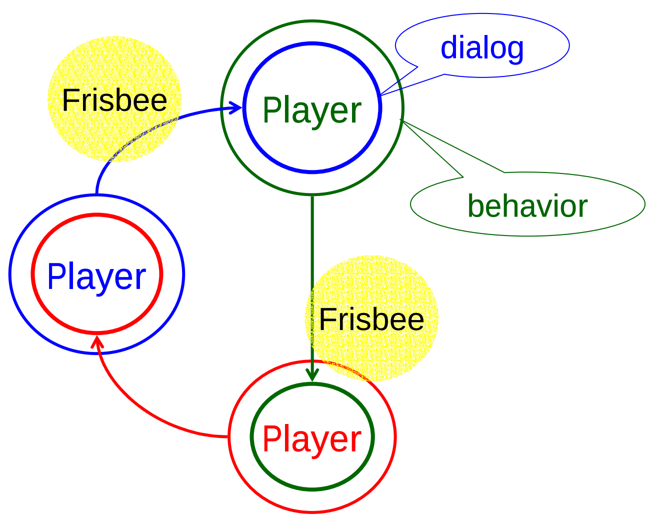
The Coordinator

- ```
  module { shared } Coordinator;
    type Manager = object; (*active*)
    var T, i: integer;
    activity ManagerDialog ();
      var next: integer;
    begin
      loop
        return T; t := *;
        if t > T then T := t end;
        await T > t;
      end
    end ManagerDialog;
  begin T := 0;
    for i := 0 to 9 do new Manager() end
  end Coordinator.
  ```
  ```

Managers as Active Objects

```
• type Manager = object
    activity Check ();
      var t: integer;
        d: Coordinator.ManagerDialog;
    begin
      d := new Coordinator.ManagerDialog;
      loop
        t := d();
        (*check agenda and update t*)
        d(t)
      end
    end
  begin new Check()
  end Manager;
```

Example 3: Frisbee Fun



Starting the Game

- ```
  module Game;
    type Player = object; (*active*)
    var i: integer; p, q, last: Player;
  begin
    last := new Player(); q := last;
    for i := 0 to 9 do
      p := new Player ();
      p.Init(q, Random.Next() mod 2);
      q := p
    end;
    last.Init(q, 0)
  end Game.
  ```

Player as Dual Activity Object

- ```
  type Player = object { shared }
    FrisbeeMsg = (request, catch);
    var nofFrisbees: integer;
      d: Player.FrisbeeDialog;
    procedure Init (q: Player; f: integer);
    begin
      d = new q.FrisbeeDialog;
      nofFrisbees = f;
    end Init;
    activity Play ();
    activity FrisbeeDialog (): FrisbeeMsg;
  begin { barrier }
    new Play ()
  end Player;
  ```

The Playing Activity

```
• activity Play ();
    var msg: FrisbeeMsg;
  begin
    d := new Player.FrisbeeDialog;
    loop
      await nofFrisbees # 0;
      msg := d(); d(FrisbeeMsg.catch);
      nofFrisbees := 0
    end
  end Play;
```

The Frisbee Dialog Activity

```
• activity FrisbeeDialog ();
    var msg: FrisbeeMsg;
  begin
    loop
      await nofFrisbees = 0;
      return FrisbeeMsg.request;
      msg := *;
      nofFrisbees := 1
    end
  end FrisbeeDialog;
```

Example 4: Santa Claus

- Invented by John Trono in „J. A. Trono. A new exercise in concurrency. SIGCSE Bulletin, 1994"
- Discussed and solved later by Ben-Ari with Rendez-Vous (in Ada95) and monitors (in Java)

The Original Story

- Santa Claus sleeps at the North pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He performs one of two indivisible actions:
  - If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation.
  - If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys.
- A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santa's time is extremely valuable, marshalling the reindeer or elves into a group must not be done by Santa.

Semaphore Approach (1)

```
loop
  P(Santa);
  if All_Reindeer_Ready then
    -- Deliver
  else -- All_Elves_Ready
    -- Consult
```

```
-- Consult
for All_Waiting_Elves loop
  V(Elf_Wait);
end loop;
for All_Elves loop
  V(Invite_In);
end loop;
Consult;
for All_Elves loop
  V(Show_Out);
end loop;
```

```
-- Deliver
for All_Waiting_Reindeer loop
  V(Reindeer_Wait);
end loop;
for All_Reindeer loop
  V(Harness);
end loop;
Deliver_Toys;
for All_Reindeer loop
  V(Unharness);
end loop;
```

Semaphore Approach (2)

One for every Reindeer and Elf (correspondingl)

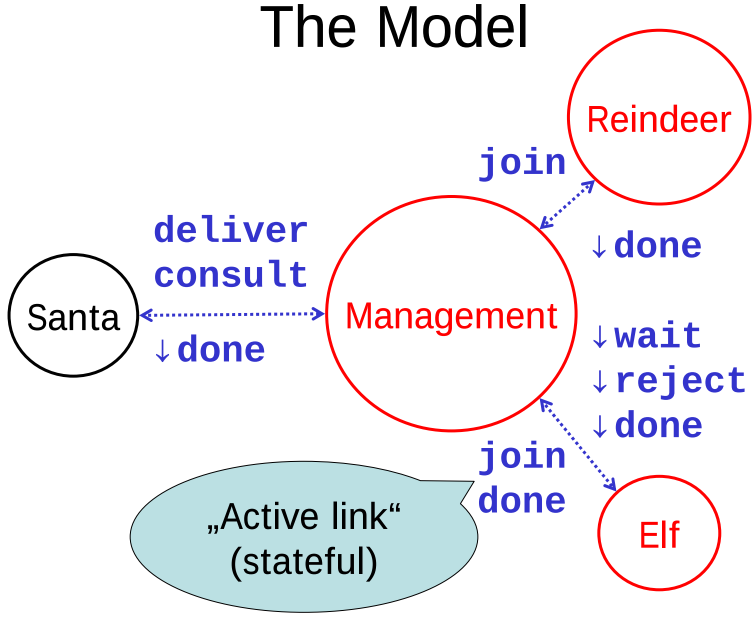
```
loop
  if Is_Last_Reindeer then
    V(Santa);
  else
    P(Reindeer_Wait);
  end if;
  P(Harness);
  Deliver_Toys;
  P(Unharness);
end loop;
```

Our Extension: Negotiation

- Before joining, elves should be informed about the expected waiting time and be given the opportunity to withdraw
- Dialog as formal syntax in EBNF
  - ➢ Messages from callee to caller marked " ↓ "

```
HandleElf = join ( Negotiate  | ↓reject ).
Negotiate = [ ↓wait join ] ↓done |
 ↓wait  done.
```

The Model



Management as Active Server

```
• module { shared } Management;
    type
      ElfMsg = (join, reject, delay, done);
      ReindeerMsg = (join, done);
      SantaMsg = (deliver, consult, done);
      Elf = object; (*active*)
      Reindeer = object; (*active*)
      Santa = object;
    var r0, r, R, e0, e, E: integer;
      santa: Santa;
    activity Work ();
    activity ElfDialog (): ElfMsg;
    activity ReindeerDialog (): ReindeerMsg;
  begin { barrier }
    new Work ()
  end Management.
```

Manager as Active Server

```
• activity Work ();
    var res: SantaMsg;
      d: Santa.Dialog;
  begin
    d := new santa.Dialog;
    loop
      await (r > r0) & (e > e0);
      if r > r0 then
        res := d(SantaMsg.deliver); inc(r0)
      else
        res := d(SantaMsg.consult); inc(e0)
      end
    end
  end Work;
```

The Elf Handling Activity

```
• activity ElfDialog ();
    var myGroup: integer; req: ElfMsg;
  begin
    loop req := *;
      if (*too soon*) then return ElfMsg.reject
      else
        if e0 < e then
          return ElfMsg.wait; req := *
        end;
        if req = ElfMsg.join then
          myGroup = e; inc(E);
          if E = 3 then E := 0; inc(e) end;
          await e0 > myGroup;
          return ElfMsg.done
        end
      end
    end
  end ElfDialog;
```

Elves as Active Objects

```
• type Elf = object
    activity Work ();
      var res: ElfMsg; d: Manager.ElfDialog;
    begin
      d := new Manager.ElfDialog;
      loop
        passivate(Random.Next());
        res := d(ElfMsg.join);
        if res = ElfMsg.wait then
          if (*impatient*) then d(ElfMsg.done)
            else res := d(ElfMsg.join)
          end
        end
      end
    end Work;
  begin { barrier } new Work()
  end Elf;
```

Santa Dialog Controlled

```
• type Santa = object
    activity Dialog (): SantaMsg;
      var req: SantaMsg;
    begin
      loop
        req := *;
        if req = SantaMsg.deliver then
          passivate(10000)
        else (*consult*) passivate(500)
        end;
        return SantaMsg.done
      end
    end Dialog;
  end Santa.
```

Example 5: Rental System

```
module { shared } Rental;
  const N = 100;
  type
    Message = (accept, return);
    Client = object;
  var nofFree, i: integer;
    free: array N of boolean;
  activity Negotiate (): Message;
  procedure Next (obj: integer);
  procedure Release (obj: integer);
begin
  for i := 0 to N - 1 do free[i] := true end;
  nofFree := N
end Rental.
```

Rental System: Negotiate

```
activity Negotiate (): Message;
  var msg: Message; obj: integer;
begin
  obj := -1;
  repeat
    obj = Next(obj); return obj;
    msg := *;
    if msg # Message.Accept then
      Release(obj)
    end
  until msg = Message.Accept;
  msg := * (*return*)
  Release(obj)
end Negotiate;
```

Rental System: Find & Release

```
procedure Next (obj: integer);
begin
  await nofFree > 0;
  while ~free[obj] do
    obj := (obj + 1) mod N
  end;
  free[obj] := false; dec(nofFree);
  return obj;
end Next;

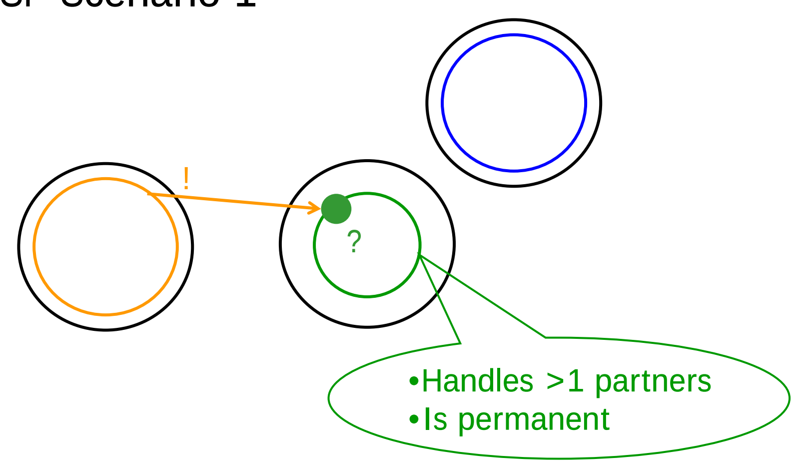
procedure Release (obj: integer);
begin
  free[obj] := true; inc(nofFree)
end Release;
```

Rental System: Client

```
type Client = object
  activity Rent ();
    var res: Message; d: Rental.Negotiate;
      suitable: boolean;
  begin d := new Rental.Negotiate;
    repeat
      obj := d(); suitable := Check(obj);
      if ~suitable then d(Message.Return) end
    until suitable;
    d(Message.Accept); (*now use the object*)
    d(Message.Return)
  end Rent;
  begin { barrier } Rent()
  end Client;
```

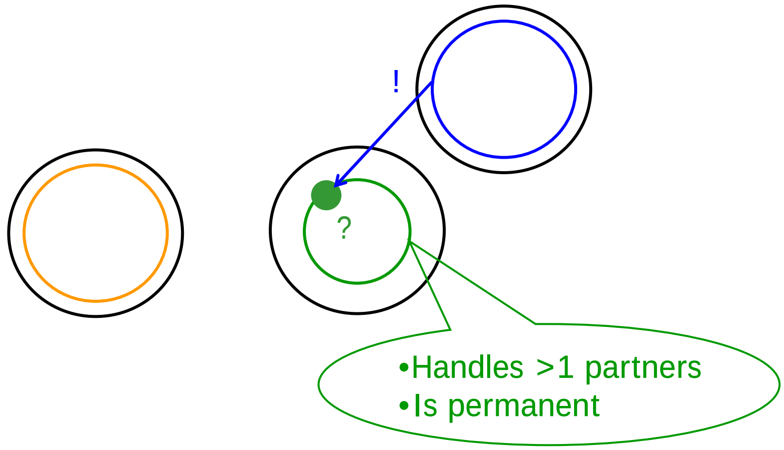
Active Links vs. CSP

- CSP Scenario 1



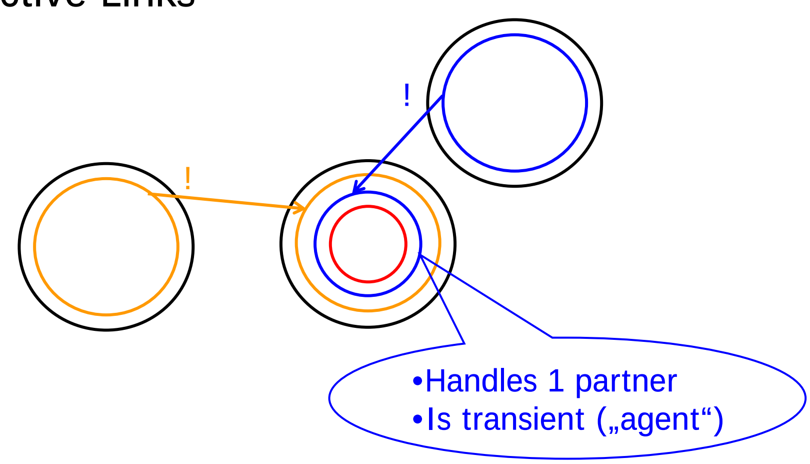
Active Links vs. CSP

- CSP Scenario 2



Active Links vs. CSP

- Active Links



Summary (1)

- The presented *await* construct
  - ➢Adds autonomity to objects
  - ➢Contributes to scalability
  - ➢Delegates condition scheduling to the runtime system (compare with garbage collection)

Summary (2)

- The presented concept of activity upgrades the ordinary object-oriented model in three respects by adding
  - ➢An option of orchestrating multiple concurrent activities according to programmed „launch logic“
  - ➢Optional intrinsic encapsulated behavior of objects
  - ➢A new way of dialog-oriented and stateful interoperability based on „active links“

Summary (3)

- The Zonnon concurrency model is an object-oriented combination of a shared-memory model and a message-passing model

Summary (4)

- The concept of activity has proved its suitability in several case studies and in implementations
  - ➢ The model of active objects underlies the *Aos Active Oberon* operating system
  - ➢ Active objects and dialogs have been implemented in the *Active C# ROTOR* compiler available from http://www.avocado.ethz.ch/ActiveCSharp/
  - ➢ Activities are currently being implemented in the *Zonnon for .NET* compiler, see http://zonnon.ethz.ch