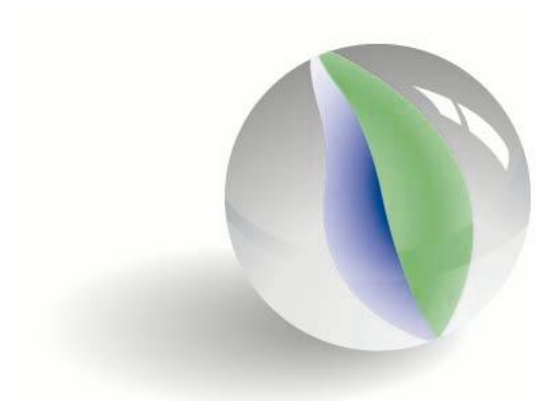
The Concepts of Zonnon

A language for systems engineering with Modules, Objects and Concurrency

Brian Kirk, David Lightfoot and
Prof Jurg Gutknecht



A Project of ETH Zürich
Institute for Computer Systems

Overview

- The purpose of this presentation is to introduce the concepts which underpin the Zonnon language design and its implementation.
- Zonnon is a general purpose language in the Pascal, Modula-2 and Oberon family
- Its conceptual model is based on objects, definitions, implementations and modules
- Its computing model is concurrent, based on active objects which interact via syntax controlled dialogs

The Ethos of the Project

The three golden rules -

Eliminate the superfluous

Emphasise the comfortable and

Acknowledge the elegance of the uncomplicated

Giorgio Armani

There are only two classes – first class and no class

David O Selznick

Good composers don't imitate, they steal

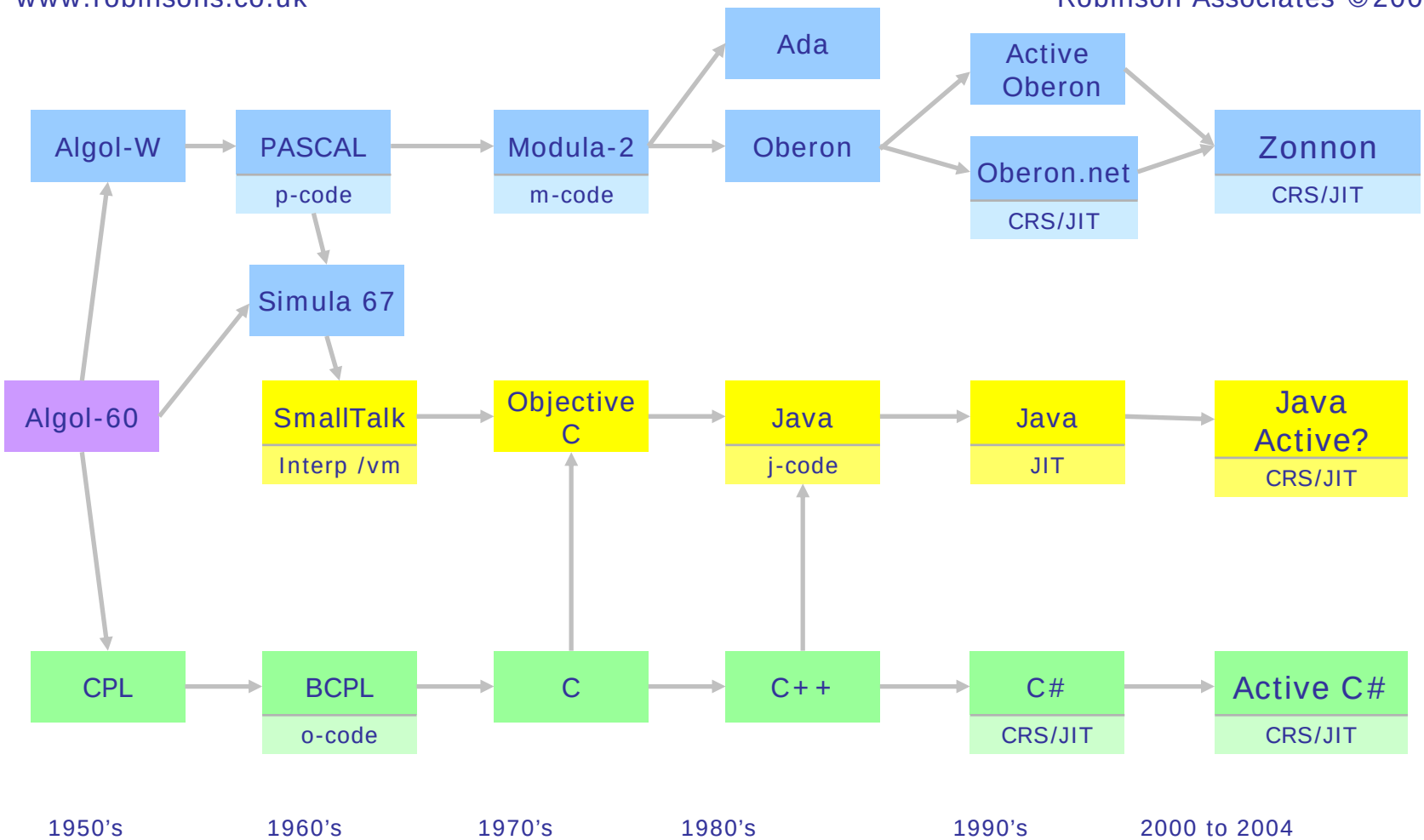
Igor Stravinsky

Goals of the Zonnon Project

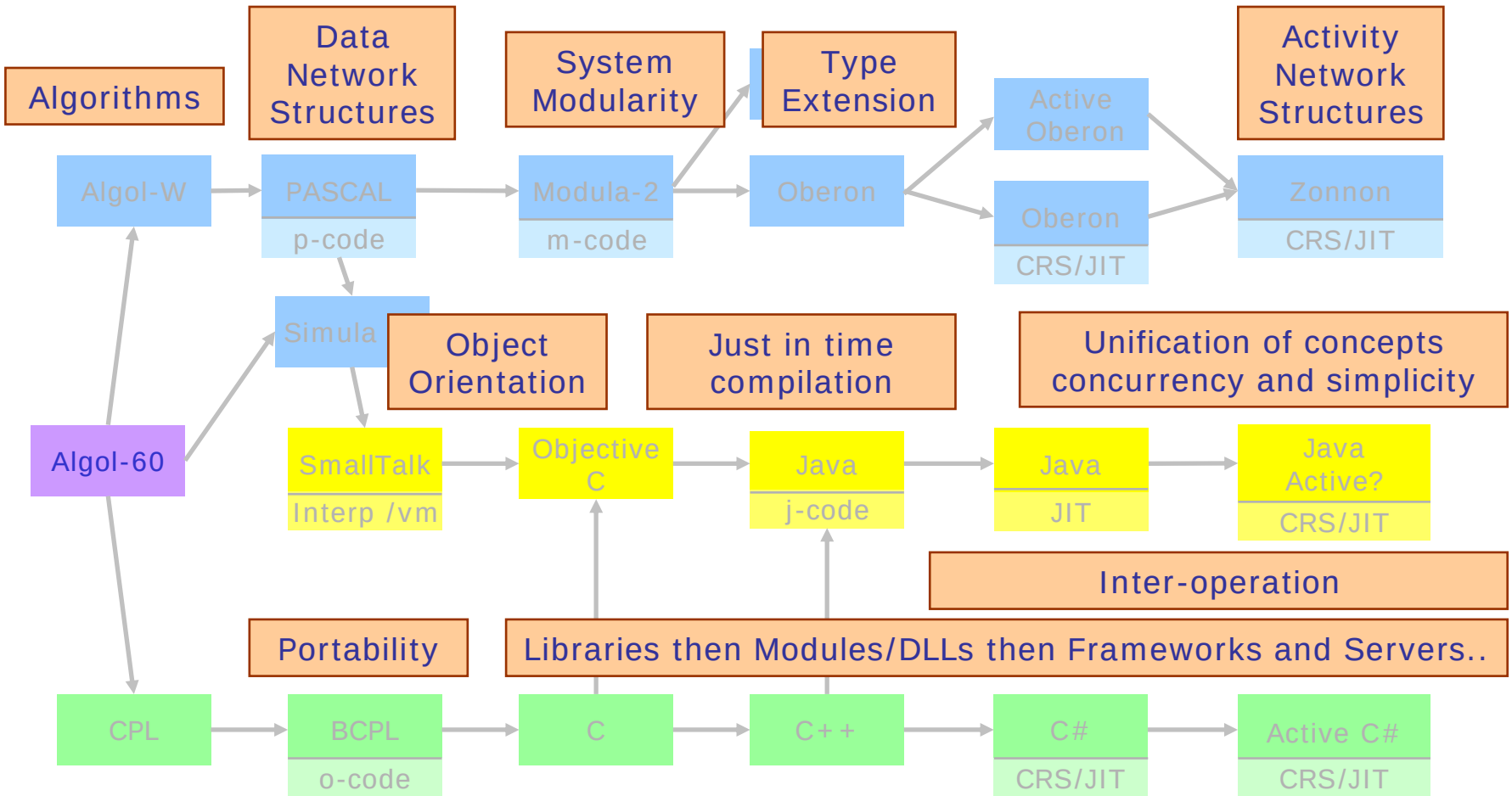
- design support for ‘programming in the large’
- a modern object model that supports concurrency: activities in objects replacing passive method calls
- formalised interaction between activities (dialogs)
- good for structured *and* OO programming styles for a wide range of applications ... OS to Business
- retain the simplicity of PASCAL, Modula-2, Oberon family of languages

- support multi-processor and distributed systems
- support inter-operation with other languages and their libraries
- produce a .NET Compiler, IDE, Test Suite and concise Language Report (40 Pages)

Evolution of some languages ...



Evolution of key concepts ...



Straight–jackets and orthodoxy!

Programs are composed of a hierarchy of classes based on inheritance relations for flexibility/reuse

Objects are activated by method calls, scheduling of system activity is by ‘event driven calling’

At design time

At run time

There is another way of designing programs

The Zonnon Approach …

- Use self-contained active objects
- The objects interact via syntactic dialogs
- Objects are composed by aggregating pre-programmed fragments
- Design the partitioning of the program into parts right from the start to
  - Represent abstractions as directly as possible
  - Make it easier to understand
  - Promote reuse systematically
  - Provide flexibility via appropriate mechanisms

Different kinds of program parts

Module

Definition

Object

Implementation

Their **purpose** is to represent abstractions as well as supporting program composition, separate compilation and dynamic loading of program parts

The Module ...

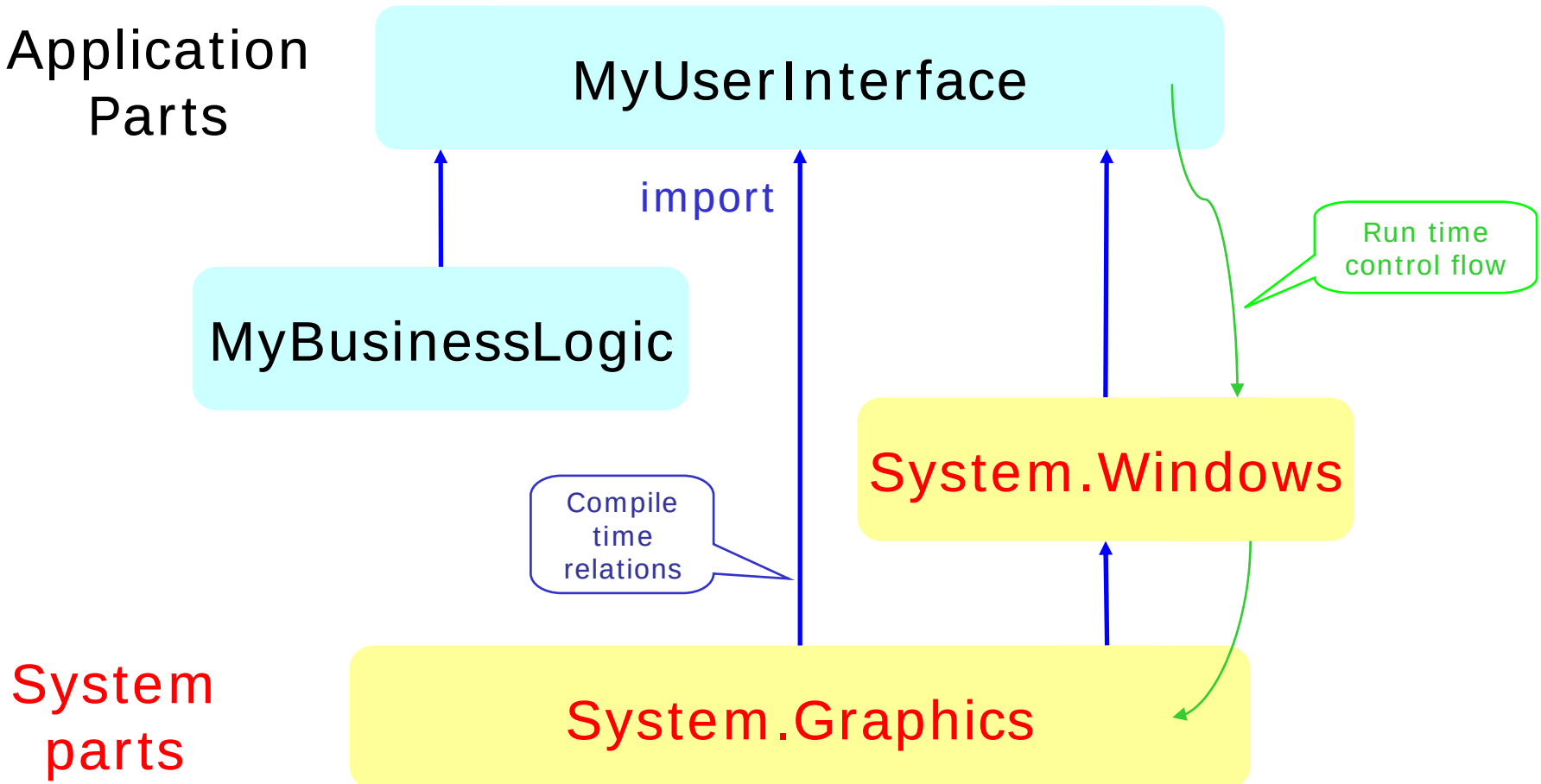
At design and compile time a module

- is the declaration of part of a program
- enables separation of concerns, encapsulation and data hiding
- is a container for logically cohesive program declarations
- defines dependency relations via IMPORT

At run-time the module is

- loaded to and unloaded from memory by the system as a unit when needed
- a static object instantiated by the system

Sample module hierarchy



An example of a simple module

namespace

```
MODULE RandomNumbers;
  VAR z: INTEGER {32}; (*global variable*)
  PROCEDURE {PUBLIC} Next( ): REAL;
    CONST a = 16807; m = 2147483647;
          q = m DIV a; r = m MOD a;
    VAR g: INTEGER {32};
  BEGIN (*calculate next number*)
    g := a*(z MOD q) - r*(z DIV q);
    IF g > 0 THEN z := g ELSE z := g + m END;
    RETURN z*(1.0/m)
  END Next;
BEGIN
  z := 31459 (* initialise the seed on loading*)
END RandomNumbers.
```

The Object ...

- Is a type template for defining objects
- It contains
  - fields to represent state
  - methods for its functionality
  - activities for its concurrent behaviour
- It has one or more interfaces ...
  - an intrinsic interface, the set of all the elements made public, not private
  - and definitions which may selectively expose *facets* of the objects services to its clients

The Definition

Is used at design and compile time to

- declare a facet of an object which is
- an abstract interface consisting of
  - declarations (types, vars etc) and
  - method signatures
- refine an object by
  omitting, adding or modifying its services
- multiple definitions can form
  a network of related types (abstractions)
  not just a hierarchy

Abstractions: why must one view be dominant ?

Sax player
Face or Logo?

JukeBox: Player or Store?

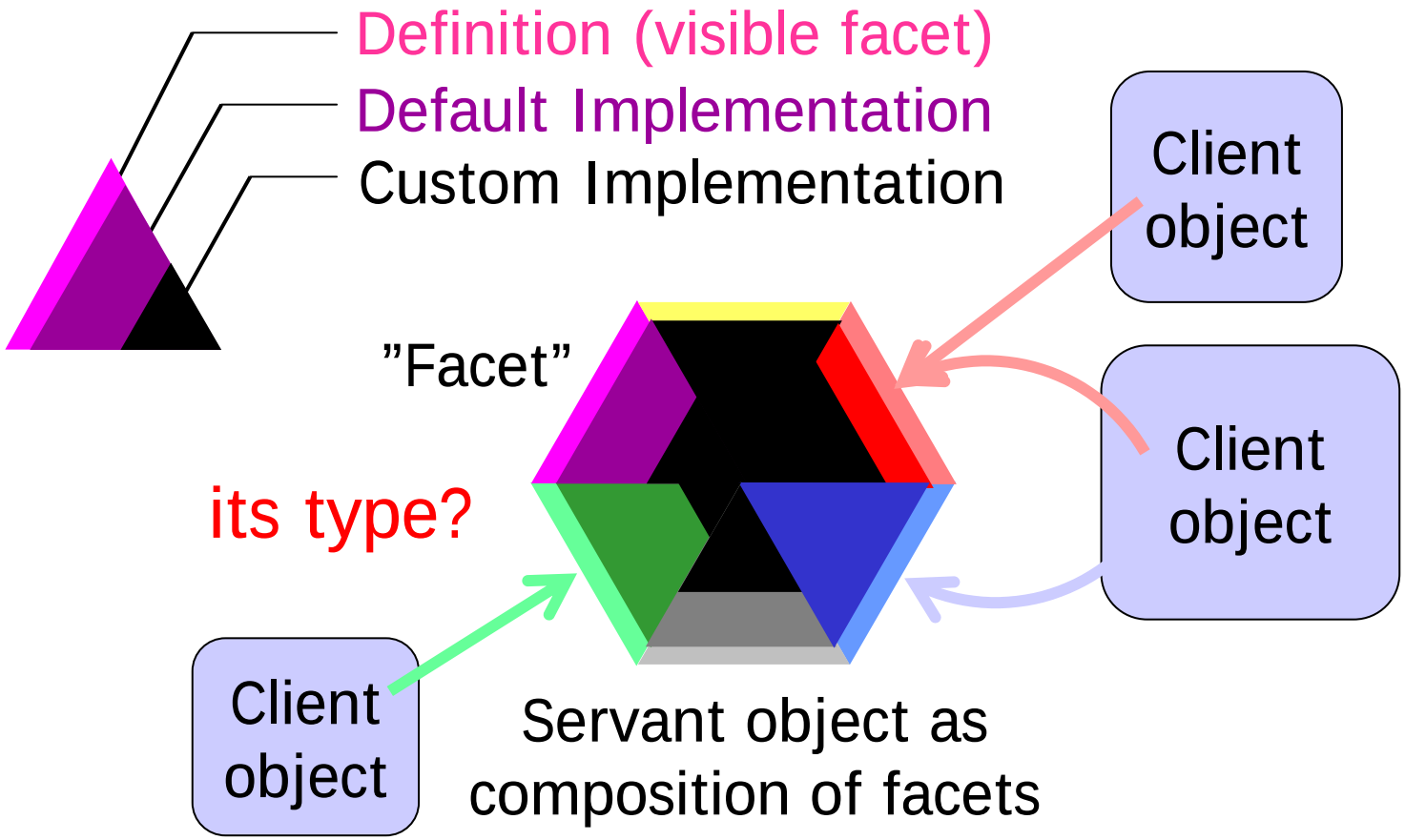
```
class JukeBox: Player, Store
{
}
```

Truck: Container or Vehicle?

```
class Truck: Vehicle, Container
{ ...
}
```

An object may need to reveal several different interfaces to different clients

A unifying abstraction concept



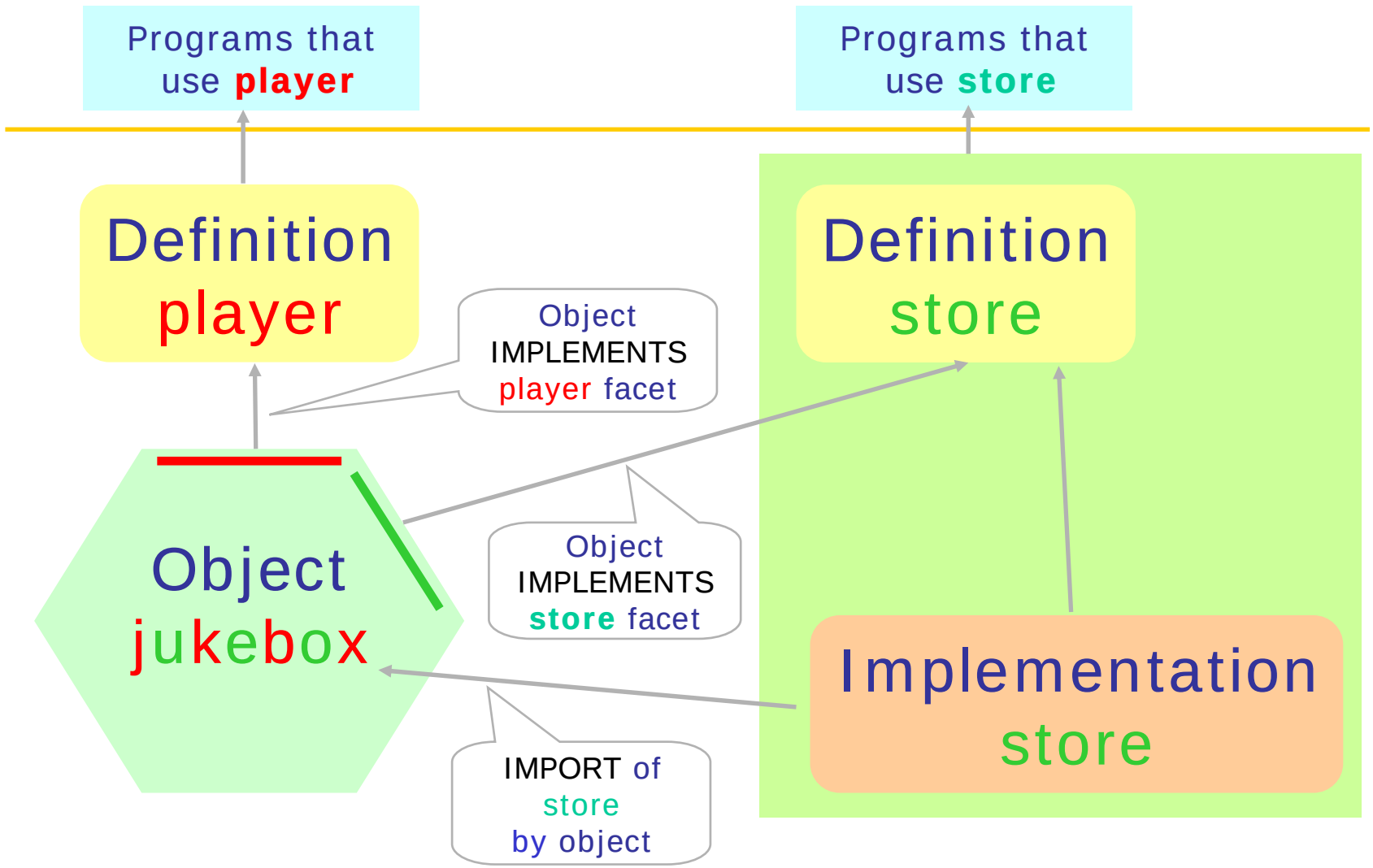
The Implementation

Is used at design and compile time as

- a library concept
- a container to aggregate field and method program fragments
- a compile time tool for managing a unit of program fragments intended for re-use

- Special case: an implementation with the same name as a definition is presumed to be its whole implementation

A jukebox (iPod) object design



The Jukebox as an example

Namespace

```
DEFINITION Music.Player;
  VAR cur: Song;
  PROCEDURE Play (s: Song);
  PROCEDURE Stop;
END Player.
```

```
DEFINITION Music.Store;
  PROCEDURE Clear;
  PROCEDURE Add (s: Song);
END Store.
```

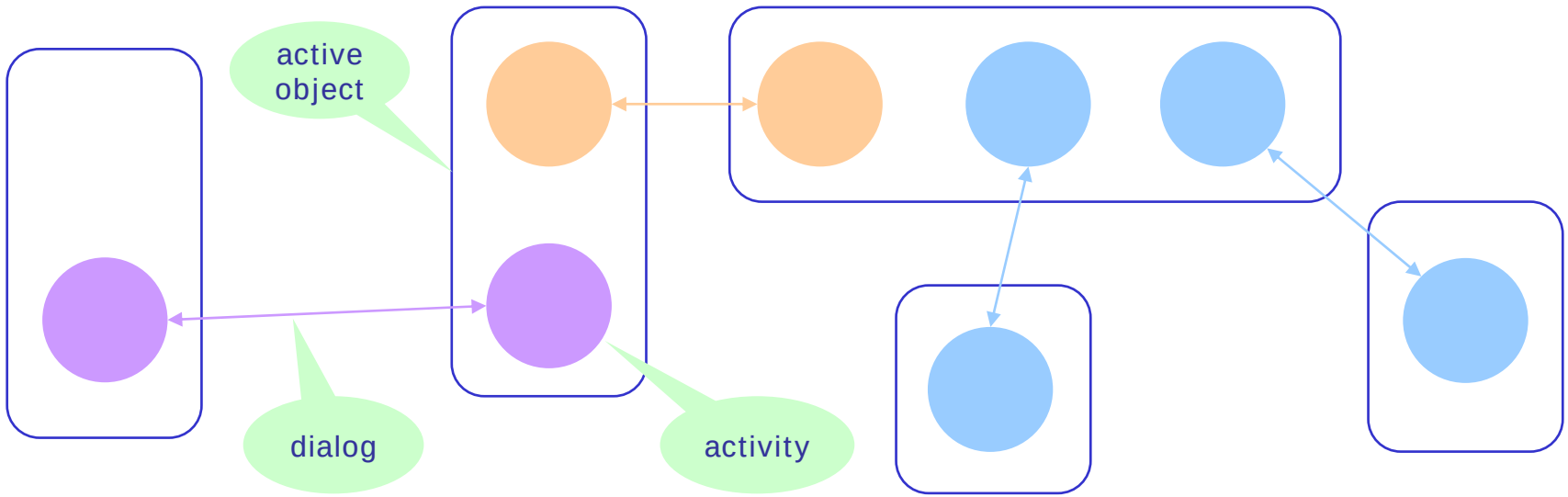
```
OBJECT Music.JukeBox
  IMPLEMENTS Player, Store;
    IMPORT Store; (* aggregate *)
  PROCEDURE Play (s: Song);
      IMPLEMENTS Player.Play;
  PROCEDURE Stop;
      IMPLEMENTS Player.Stop;
END JukeBox.
```

```
IMPLEMENTATION Music.Store;
  VAR rep: Lib.Song;
  PROCEDURE Clear;
    BEGIN loop := NIL; …
  END Clear;
  PROCEDURE Add (s: Song);
    BEGIN s.next := rep; rep := s …
  END Add;
BEGIN Clear (*initialisation*)
END Store.
```

Concurrency

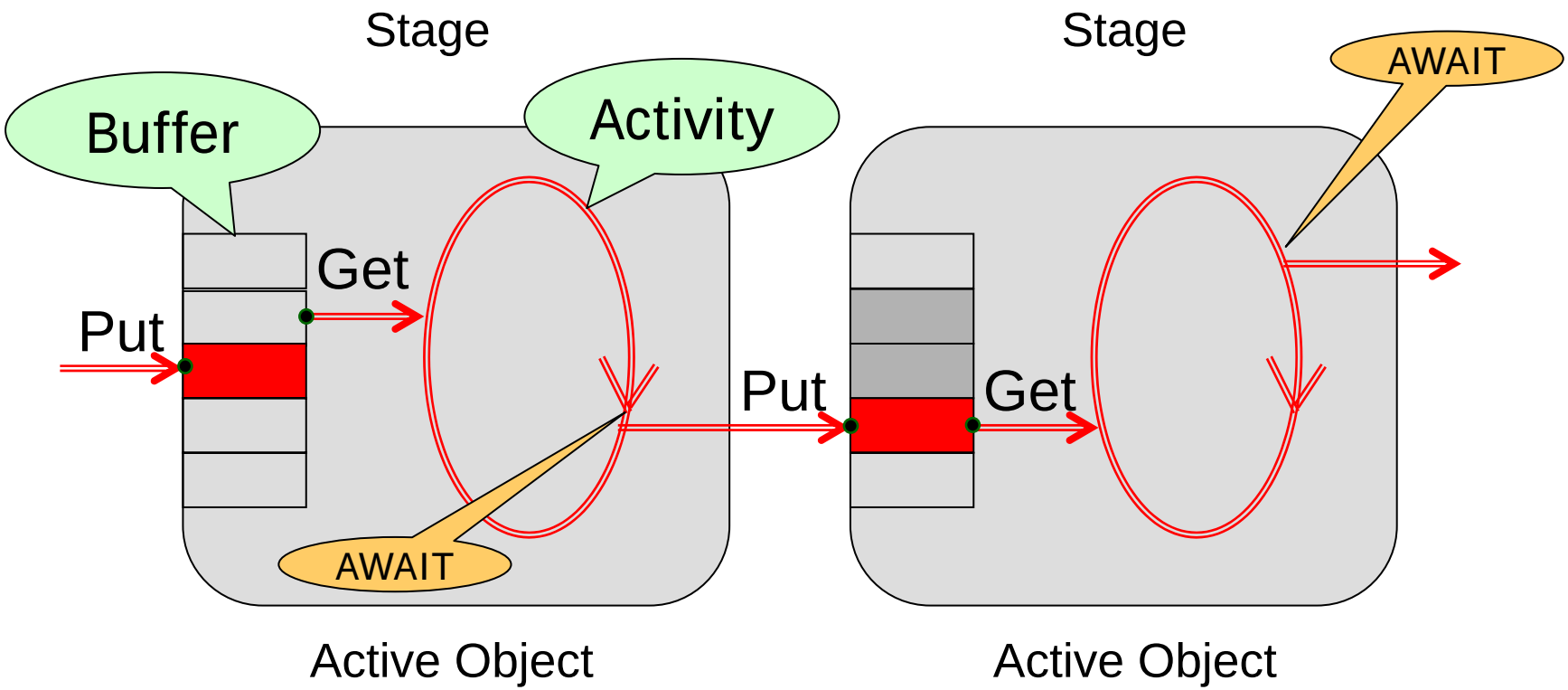
- Mainstream approach (C#, Java ...)
  - System.Threading library
  - General lock objects
  - Wait/Pulse for threads self-management
- Zonnon approach ...
  - Activities built into objects, none or more
  - Syntax based dialogs governing interaction
  - Object level monitor locks via blocks
  - Thread management by the system via **await** on pre-condition for continuation (transparently on single or multiple processors)
  - Statement level concurency also available

Program level concurrency model



- Arbitrary topology (network) of object's activities which interact according to formalised dialogs
- Dynamically scheduled by the system based on validity of *await* preconditions
- *send*, *receive* and *accept* dialog elements

Example:
A Pipeline with Active Objects



System-wide activity is scheduled by evaluating the set of all AWAIT preconditions

An active object

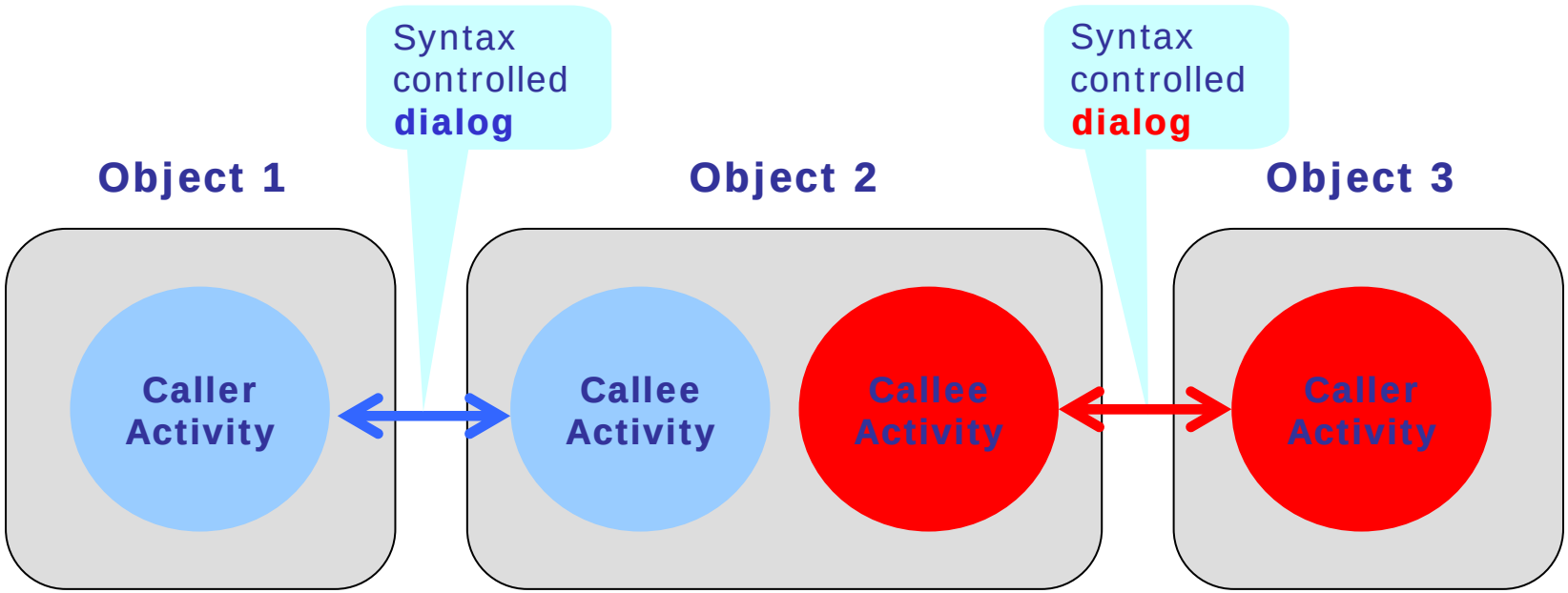

```
OBJECT Stage (next: Stage);
   VAR { PRIVATE } n, in, out: INTEGER;
      buf: ARRAY N OF OBJECT; (*used as a circular buffer*)
      PROCEDURE  { PRIVATE }  Get (VAR x: OBJECT);
      BEGIN { LOCKED } (*mutual exclusion*)
         AWAIT (n # 0);  (* precondition to continue*)
         DEC(n); x := buf[out]; out := (out + 1) MOD N
      END Get;
      PROCEDURE { PUBLIC } Put (x: OBJECT);
      BEGIN { LOCKED } AWAIT (n # N); (*mutual exclusion*)
         INC(b); buf[in] := x; in := (in + 1) MOD N
      END Put;
      ACTIVITY Processing; VAR x: OBJECT;
      BEGIN LOOP Get(x); (*process x;*) next.Put(x) END
      END
BEGIN (*initialise this new object instance*)
    n := 0; in := 0; out := 0; NEW(Processing)
END Stage;
```

Wait whilst the buffer is empty

Wait whilst the buffer is full

each activity has a separate thread

Formal dialogs binding activities



1. the caller first creates an activity in the callee object
2. this implicitly binds the activities with a dialog
3. a formal syntax is associated with the dialog
4. caller and callee interaction is specified by the dialog syntax
5. interaction between the caller and callee activities is asymmetrical, the caller knows the callee by its name, but it is anonymous to the callee
6. the dialog is formally defined in EBNF notation

Example of a dialog syntax


```
DEFINITION Fighter
   (*full example in the Zonnon Language Report*)
   (* this type is used to create Fighter.karate activities*)
   DIALOG  Karate
     (* syntax of the dialog*)
     { fight = { attack ( { defense  attack } | RUNAWAY
             [ ?CHASE] | KO | fight ) }.
     attack = ATTACK strike.
     defense = DEFENSE strike.
     strike = bodypart [ strength ].
     bodypart = LEG | NECK | HEAD.
     strength = integer. }
     (*enumeration of the dialog elements to be exchanged*)
    = (RUNAWAY, CHASE, KO, ATTACK, DEFENSE,
          LEG, NECK, HEAD);

END Fighter.
```

Statement level concurrency

- The block defines the scope of concurrency

```
BEGIN {CONCURRENT}
  statement a; (* statements a, b and c are *)
  statement b; (* executed concurrently     *)
  statement c   (* and non-deterministically*)
END
```

- The **launch** statement can optionally be used to sequence the start of execution of concurrent statements in a defined sequence within a block

In summary ...

- Abstraction is a key principle of language design particularly to support design of large programs
- The fundamental concepts are few and simple
- Zonnon introduces a new computing model based on concurrency, retaining type safety
- An effective language need not be complex
- The language specification doesn't have to be 100's of pages long, in fact it's only 40 pages!
- Please see www.zonnon.ethz.ch for details *free* downloads and progress updates

References

1 Revised Report on the Algorithmic Language Algol 60 by J.W. Backus, F.L. Bauer, J.Green, C. Katz, J. McCarthy,P. Naur, A.J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois J.H. Wegstein, A. van Wijngaarden, M. Woodger

2 Simula Begin (Simula 67) by G Birtwistle, Ole-Johan Dahl Bjorn Myhrhaug and Kristen Nygaard
Auerbach 1973

3 Pascal User Manual and Report by K Jensen and N Wirth,
Springer Verlag ISBN-0-387-90144-2, 1974

4 Programming in Modula-2 by N Wirth,
Springer Verlag, ISBN 0-387-15078-1, 1982

5 Project Oberon: the design of an operating system and compiler by Niklaus Wirth and Jurg Gutknecht,
Addison Wesley, ISBN 0-201-54428-8, 1992

6 On the Criteria to be used in Decomposing Systems into Modules by D Parnas,
Communications of the ACM, December 1972

7 Designing systems with objects, processes and modules by B R Kirk
Microprocessors and Microsystems Vol 18 No3 April 1994

8 Do The Fish Really Need Remote Control , a proposal for self active objects, by J Gutknecht,
LNCS Modular Programming Languages, JMLC 1997, Springer ISSN 0302-9743

9 A Multiprocessor Kernel for Active Object Based Systems by Pieter Muller, Lecture Notes in Computer Science, Modular Programming Languages, JMLC 2000, Springer ISSN 0302-9743

10 Standard ECMA-335:Common Language Infrastructure (CLI), see section on Common Type System (CTS)
http://www.ecma.ch/ecma1/STAND/ecma-355.html

11 Zonnon Language Experiment, or How to Implement a Non-Conventional Object Model for .NET
Jürg Gutknecht, Eugene A. Zueff, Computer Systems Institute, ETH Zürich, Switzerland.
Extended Abstract, OOPSLA'2002 Conference, November 2002, Seattle.

12 Zonnon: A .NET Language Beyond C#
http://research.microsoft.com/collaboration/university/europe/events/RCC/Russia/content.aspx?11
Jürg Gutknecht, Institute for Computer Systems, ETH Zurich.
Microsoft Conference, June 15-17, 2003, Moscow.

13 Zonnon for .NET, A Language and Compiler Experiment
Jürg Gutknecht, Eugene A. Zueff, Computer Systems Institute, ETH Zürich, Switzerland.
JMLC Conference, August 2003.

14 The Zonnon Project web site is at www.zonnon.ethz.ch

Acknowledgements

Prof Jürg Gutknecht, ETH Zürich
Eugene Zueff, ETH Compiler Writer
Vladimir Romanoff, ETH Test Suite
Microsoft Research (.NET CCI Framework)

Prof N Wirth, ETH (constructive feedback)
Al Freed, NASA (mathematics feedback)
and many more who have given feedback ...

Spare slides for possible use

Spare Slides follow ...

Programming-in-the-large needs building blocks and relations

They key concepts are

- Separation into parts, each with a purpose
  - specifications (in cohesive intelligible pieces)
  - functionality
  - loadable program parts (practical to deploy)
- The relations between them for combination
  - Refines: adds to, omits from or changes
  - Aggregates: collects various items as a unit
  - Implements: provides fragments of program defined elsewhere
  - Imports: reuses predefined declarations

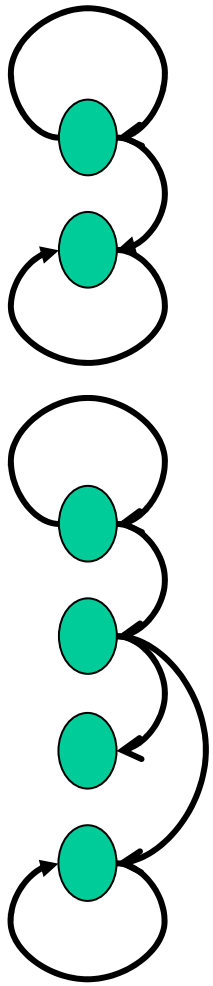
(C# offers extends, implements, inherits)

Programming-in-the-large needs building blocks and relations

- **C#**
  - Interfaces
  - Classes
- **Zonnon**
  - Definitions
  - Implementations
  - Object Types
  - Modules

compile

run



extends :n

implements :n

inherits :1

refines :1

implements :1

aggregates :n

imports :n

The Zonnon type system

- Inter-operability with other languages in a system important
- ECMA Common Type System used as the model (for compatibility with .NET compliant languages)
- The {modifier} for flexiblility
- integer {32}, char {8}, char {16} …
- Modifiers have default values
- {ref} {value} {public} {private} …

Zonnon's basic types

- OBJECT generic base for all object types
- BOOLEAN (true,false)
- SET of elements
- CARDINAL unsigned whole numbers
- INTEGER signed whole numbers
- REAL signed floating point numbers
- FIXED signed fixed precision numbers
- CHAR 8bit (UTF8) and 16bit (Unicode)
- STRING immutable, as ECMA

Programmer defined types

- ENUMERATION a set of named values
- ARRAY a number of elements of the same type
- OBJECT containing fields, method and activities
- RECORD containing fields and procedure signatures
- PROCEDURES for static and dynamic use