

Mitsubishi Electric (Control Software) Corporation
Workshop on Object Oriented Technologies

Workshop
on
Object Oriented Technologies

July 22 - 26, 1996

Kobe, Japan

organized

by

Takayuki Dan Kimura
Wolfgang Pree
Douglas C. Schmidt

Goals

- Introduction to **fundamental concepts** in OO technologies.
- Demonstration of OO **advantages** over traditional software engineering technologies.
- Overview of practical OO **applications**.
- Hands-on **experiences** in OO design and OO programming

Instructors

Dr. Wolfgang Pree
Associate Professor
University of Linz



<http://www.swe.uni-linz.ac.at/staff/wolf/wolfgangPree.html>

Dr. Douglas C. Schmidt
Assistant Professor
Washington University



Dr. Takayuki Dan Kimura
Professor
Washington University



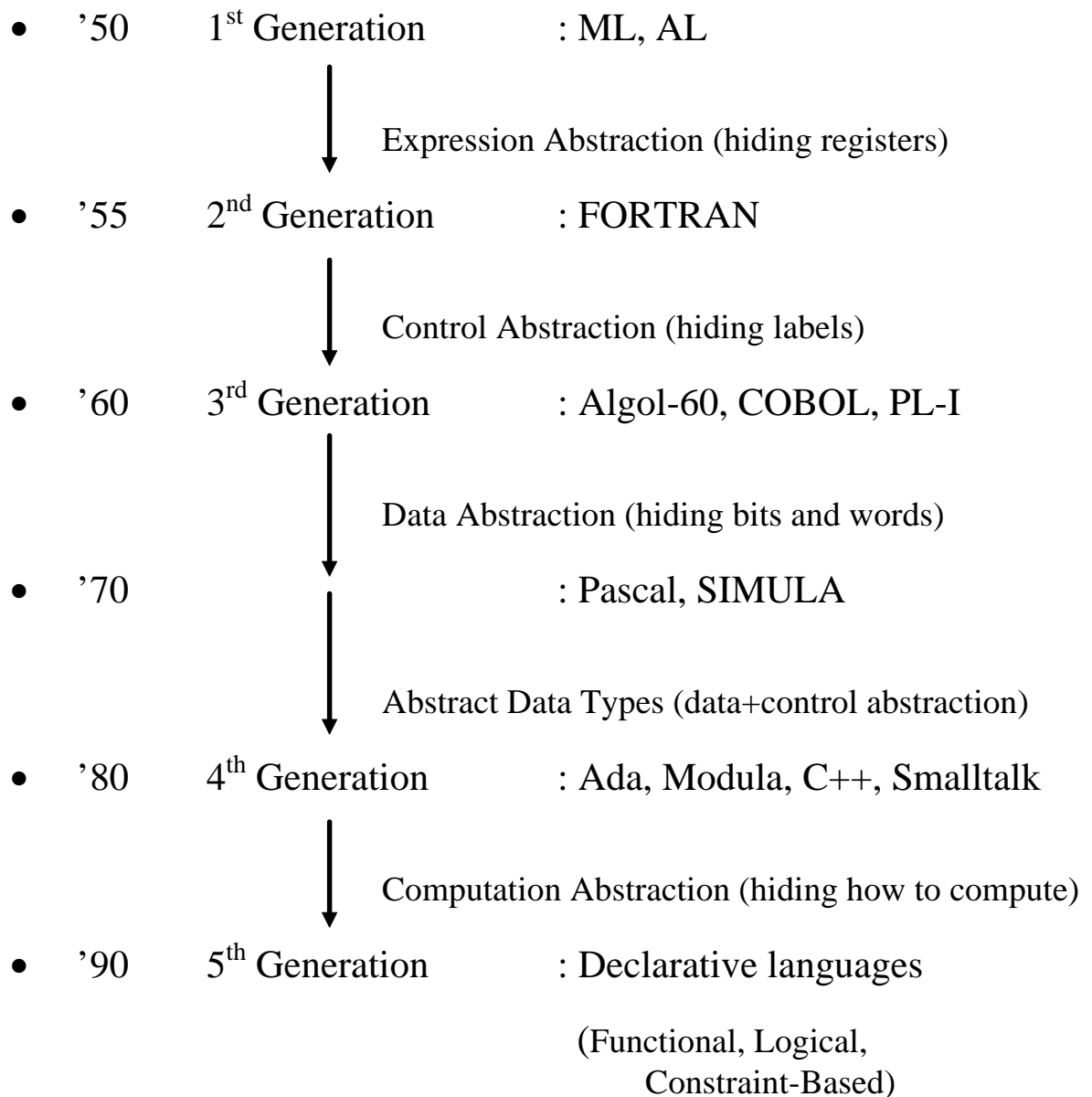
<http://www.cs.wustl.edu/cs/department.html>

Schedule

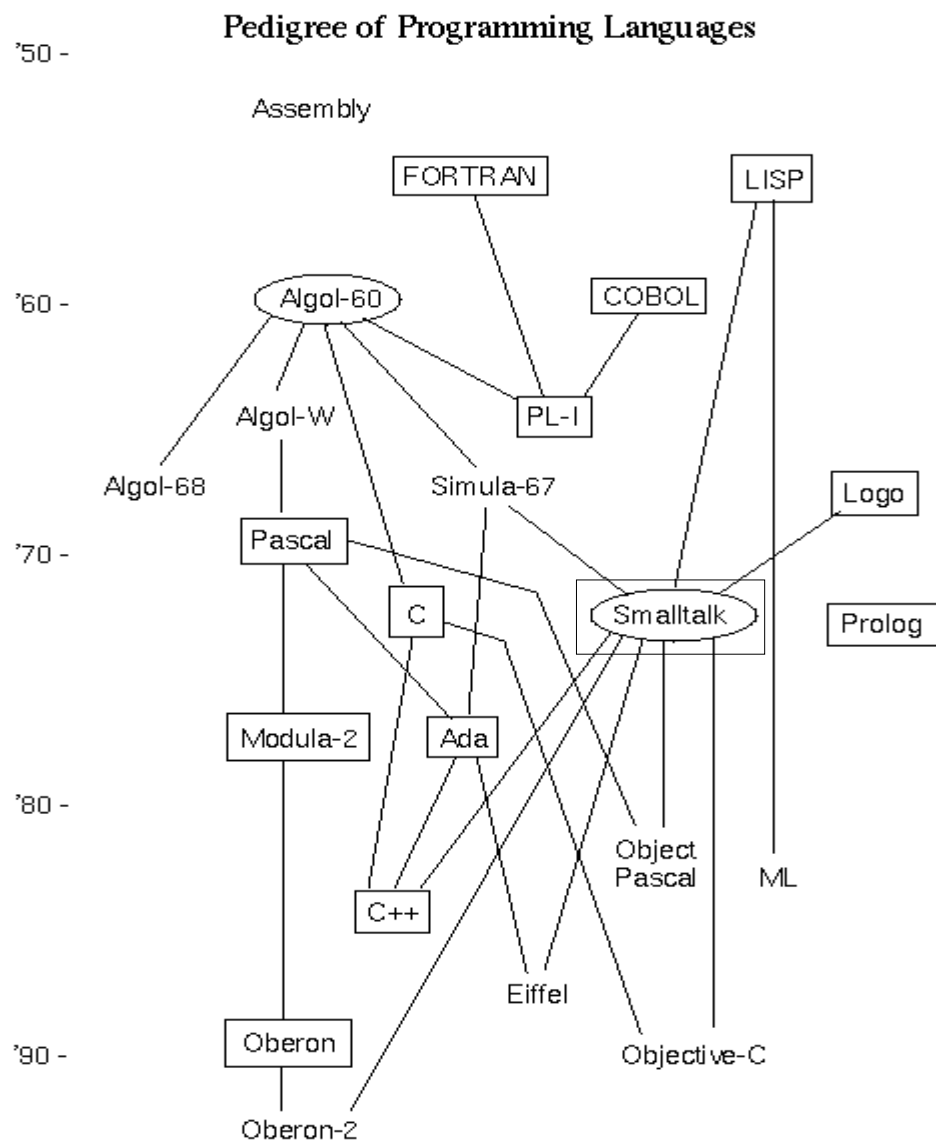
	7/22	7/23	7/24	7/25	7/26
	Monday	Tuesday	Wednesday	Thursday	Friday
Theme	Language	Analysis & Design	Design Mechanisms	Quality Management	Evaluation
9:00 - 10:30	OO Programming Concepts (TDK)	OO SE Concepts (DCS)	Class Libraries (WP)	Testing & Metrics (DCS)	OO Applications (DCS)
10:30 - 10:45					
10:45 - 12:15	C++ Language I (TDK)	OO Modeling (DCS)	Frameworks (WP)	Real-time Programming (DCS)	Java Programming (TDK)
12:15 - 2:00					
2:00 - 3:30	C++ Language II (TDK)	OO Design (DCS)	Case Study I: Document Embedding (WP)	Project Management (WP)	Summary (TDK)
3:30 - 3:45					
3:45 - 5:15	Programming Exercises (WP)	Design Exercises (WP)	Design Patterns (DCS)	Case Study II: Energy Management (WP)	Free Discussions (TDK/DCS/WP)

Software Technology

Concepts of Abstraction (Complexity Management)



History



Modular Programming

Modular = constructed with standardized units or dimensions for flexibility and variety in use (Webster).

Modular Programming

= a software engineering technique (decomposition method) in which no component in a complex system should depend on the internal details of any other components (Dan Ingalls)

i.e., black-box approach for software components.

Encapsulation

= building a protection wall for access control

Abstraction

= information hiding

- *control abstraction* (active)
procedure, process, coroutine
(how actions are executed is hidden)
- *data abstraction* (passive)
user definable data types
(how data are structured is hidden)

Modularization

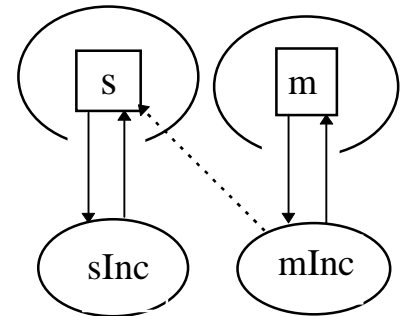
= reduction of interdependence (loosely-coupled-ness)

Motivations

- Reliability (visibility control)

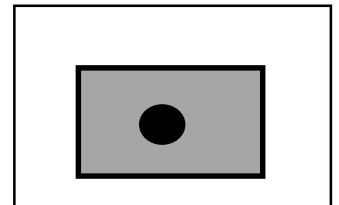
“Global variables considered harmful”

```
int      s, m;  
void sInc(x) { s = s + x; };  
void mInc(x) { s = m + x; };  
  
void main( )  
{ sInc(100); mInc(200); };
```



- Error Propagation (run time)

ex. Exception handling capabilities



- Reusability (software chip)

A general purpose module can be replaced by another module of the same functionality and the same interface (but different implementation and structure)

ex. Set module, sorting module, window module

ex. Ada generic units, C++ templates

The Concept of Type

- Type (Logic: B. Russell, The Theory of Types 1908)
= the range of significance of a propositional function

ex1. $P(x) = x \text{ loves Jane}$

The type defined by $P(x)$

= humans and animals but not numbers and symbols

“The number 3 loves Jane” is

syntactically correct, but

semantically incorrect (type violation!)

ex2. “This sentence is false” is true?

- Type (Computer Science: C.A.R. Hoare, Notes 1972)

= the set of values + set of operations

ex. $I = \{ 0, 1, 2, \dots \} + \{ +, *, <, >, =, \dots \}$

$2 + 3$

OK, meaningful (significant)

$2 \& 3$

No, nonsense (non-significant)

2 eats 3

No, type violation!

Why Type?

- Reliability

Elimination of *syntactically correct* but *semantically incorrect* expressions

i.e., *semantic error detection* by compiler

i.e., elimination of *inconsistent usage* of symbolism

```
ex.  int  birthMonth;
      ....
```

```
birthMonth = 13;
birthMonth = thisMonth * summerMonth;
```

- Data Abstraction

[illegible]

⇒ Hiding implementation details : understandability

⇒ Description of Object : readability

```
ex.  enum summerTime { June = 6, July, August };
      enum grade { fail, pass, average, good, excellent };
      struct student { char name[30]; grade status; };
```

```
summerTime      boatingMonth = July;
summerTime      vacationMonth = August;
student         Kimura;
```

```
if ( Kimura.status = good ) { ... }
```

SIMULA 67

(Dahl, Myhrhaug, Nygaard)

= Algol-60 based language for discrete event simulation

Class =

a **procedure** which is capable of giving rise to block instances (objects) which survive its call

⇒ Activation record → Object

⇒ Heap-based memory allocation

⇒ Coroutine

⇒ Reference type

⇒ Remote identifier

x.a = attribute **a** of object **x**

Prefixing (Program concatenation = Inheritance)

= a merging of the attributes of both component classes and the composition of their actions

class A(a1, a2, ...) begin D1; D2; ... ; S1; S2; ... end;

A class B(b1, b2, ...) begin *D1; D2; ... ; S1; S2; ...* end;

= class B(a1, a2, ..., b1, b2, ...)
begin D1; D2; ... ; *D1; D2; ... ; S1; S2; ... ; S1; S2; ...* end;

A is the prefix class of B

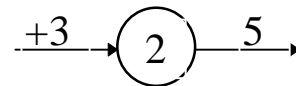
B is a subclass of A

SmallTalk

(Kay, Ingalls, Tesler)

= Lisp-based language for schoolchildren with graphic interface

Object = an active entity that interacts with each other through
messages

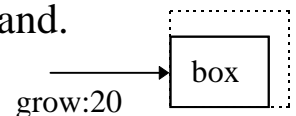


1. Object has internal state.

Object owns other objects (local objects, **instance variables**)

2. Object has a set of messages it can understand.

Message represents a service request.



3. Associated with each message, there is an action (**method**) it takes.

Method involves interaction with other objects.

4. Objects can be classified into a **class** by the set of shared attributes.

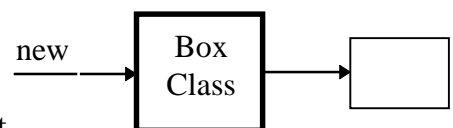
Local objects + behavior (message, method)

5. Each object belongs to exactly one class.

Hierarchical classification (No multiple inheritance).

6. Class is also an object.

new message for creating an object.



C++ (Bjarne Stroustrup)

- Previous Works (PhD at Cambridge University 1978)
 - = System Software for distributed systems
 - * Simulator construction using SIMULA
 - ⇒ too slow, but class and coroutine
 - * Simulator construction using BCPL
 - ⇒ fast, but no type checking
- C++ Goals: *Good support for*
 - * Program Organization (SIMULA)
 - class hierarchy
 - concurrency
 - static type checking
 - * Efficient Programs (BCPL)
 - fast run-time support
 - separate compilation
 - simple linkage
 - * Portability (C)
 - simple run-time system
 - limited integration with host OS
- + efficient, portable, OO concepts
 - heritage of C, complex, hybrid

Oberon-2

(Wirth, Moessenboeck at ETH)

- Wirth Stable

* 1966	Algol-W	record, reference, case, while
* 1970	Pascal	repeat, data types
* 1975	Modula	multiprogramming, module
* 1982	Modula-2	compilation unit
* 1986	Oberon	type extension
* 1993	Oberon-2	method

Make it as simple as possible, but not simpler (Einstein)

- Type Extension

= to derive a new record type from an existing one
by adding new fields but preserving compatibility

```
type person = record  name : array 32 of character; birth : date end;
```

```
    pilot = record ( person )  hoursInflight : integer end;
```

```
    clerk = record ( person )  jobCode : integer end;
```

```
procedure ProcessPerson( var p : Person );
```

```
begin
```

```
...
```

```
if p is pilot then    with p : pilot do (* process pilot data *) end
```

```
elseif p is clerk then with p : clerk do (* process clerk data *) end
```

```
end ... end
```