UNIVERSITY OF MANCHESTER INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTATION

Object Oriented Specification, Design and Implementation

Lecture 7: Object Oriented Programming Languages

Introduction

This lecture examines object oriented programming languages. It is not within the scope of this lecture course to provide a comprehensive treatment of each of the languages discussed here[1] - instead, this lecture seeks to enable the reader to reason about present and future approaches to designing and implementing object oriented (and beyond object oriented) languages.

This lecture describes, first, how an "Algol-based" language called Simula provided arguably the "first" object oriented language. It is instructive to reflect (as Computer Scientists) how such landmark developments in language design still influence the way in which "today's" languages are organised! Secondly, this lecture examines an entirely different style of object oriented language, i.e. Smalltalk. Thirdly, the Eiffel language is considered, not least, because its design demonstrates how fundamental and general notions which underpin the design of secure and robust programming languages (strong typing, an emphasis on static type checking, etc) provide a basis for an elegant "compilable" language. Finally, after a brief consideration of some other languages (included for completeness), the "model" object oriented programming language from earlier lectures is considered in more detail.

A Classification of Object Oriented Languages

An object oriented language must support at least two fundamental features:-

? A facility for *object creation*

? A facility for *message passing* (i.e. for invoking *methods*)

Additionally, and equally importantly, many languages provide a facility for *defining classes* and combine this with support for *inheritance*.

Languages may be classified into the following general categories:-

? Hybrid e.g. C++, CLOS, Eiffel, Object Pascal, Object Prolog, etc

? Frame-based e.g. FRL, KRL, KEE, LOOPS, etc

? Distributed, concurrent or actor-based, e.g. (the language) Hybrid, Concurrent Smalltalk, POOL-T, etc

? Other object models, e.g. (the language) Self*[2]*

Hybrid languages have developed from existing languages, usually via object oriented extensions. The hope is that their augmented structure is somehow "easier" for developers who are familiar with the original language to learn. Frame-based languages support a specific kind of application, i.e. knowledge-based "reasoning" and embody the notions of a *frame* composed of *slots*. Slots define a *relation* to other frames or contain a *value* of an *attribute* of a given frame. Distributed, concurrent or actor-based support the notion of parallel processing. Other object models, e.g. prototypes, delegation, in which the distinction between objects and classes is removed, e.g. by having a "class-less" language.

We can also distinguish between *object oriented* and *object based* languages. In order to do so we must first of all remove a "confusion" which has arisen about language support for the notions of an object and a class, and the notions of a class and inheritance. An object oriented language provides support for each of the following notions:-

? Objects

Embody the notion of *encapsulation* as a means of developing modular software systems.

- **?** Types

  Whether statically or dynamically typed (or a combination of both) the need to ensure that an operation is <u>only</u> applied to an object of an appropriate type is paramount.

- **?** Delegation

  The term *delegation* is intended to convey a *mechanism* for dynamically redirecting control, e.g. messages may be delegated to some objects ancestors "automatically" by means of a *message dispatching* mechanism.

- **?** Abstraction.

  A means of separating concerns relating to the *interface* provided by some object from concerns relating to how the operations defined in the interface are *implemented*.

An *object-based* language provides a subset of these features, and may also combine other features.

Simula67

Simula(67) was intended to provide direct support for the development of simulations[3] but is, in fact, a general purpose programming language. It is an object oriented extension of Algol 60[4],[5]. Simula is a *classical* language[6] in the sense that it embodies the notion of a "main" program containing a collection of "sub"-programs (routines or classes). As in languages which are much later descendants, entities of non-basic ("pre-defined" or "built-in") types denote references to class instances thus, entities are declared as:

```
REF(someentity)
```

and operations upon them use specific symbols:-

```
:-      assignment
==      equality,
=/=     inequality
```

Class instances, again as in later language descendants, are created explicitly, i.e. by evaluating expressions containing the "new" operator for example:-

```
REF(someentity) somevariable;
.
.
somevariable:-NEW someentity
```

Evaluation of the new expression first creates an instance of `someentity` then returns a reference to it. A class may have arguments, e.g.

```
CLASS someentity(x, y); integer x, y
BEGIN
END
```

and the associated `NEW` expression includes actual arguments:-

```
somevariable:-NEW someentity(1, 42)
```

A class may contain operations, attributes and a class body, i.e. a statement list which is executed as a result of a call to `NEW` . Single inheritance is supported by declarations of the form:-

```
superclass CLASS subclass;
BEGIN
.
.
END
```

The meaning of a declaration may be redefined in an inheriting class simply by providing a further declaration. More recent implementations of Simula provide support for encapsulation and information hiding via PROTECTED (unavailable to "clients") and HIDDEN (unavailable to proper descendants). Virtual routines provide a means of deferring the definition of arguments, for example:-

```
CLASS defer_example;
  VIRTUAL: PROCEDURE someargs;
BEGIN
...
END
```

Given the above definition, a "descendent" class may provide variable numbers of arguments of some type for the procedure `someargs`. Given the definitions:-

```
superclass CLASS subclass;
```

```
REF(superclass ) s1;
REF(subclass   ) s2;
```

then the assignment:-

```
s1:=s2;
```

is type safe and provides support for polymorphism. Binding is thus static for all routines except virtual routines. Dynamic binding may be "forced" by using the QUA construct or, alternatively, the INSPECT construct[7].

One of the most interesting features of Simula is its support for co-routines[8]. In Simula co-routines are represented by class instances not least, because co-routines usually need persistent data and as such represent not simply processes but also abstract data type implementations. The body of a co-routine is usually an iteration (loop) of the form:-

```
WHILE condition DO
BEGIN
  statement;
  statement;
  ...
  RESUME co-routine-name;
  statement;
  statement;
  ...
END;
```

A more detailed consideration of Simula's primitives for co-routines is beyond the scope of this lecture. The references in Appendix 2 provide a starting point for those students who wish to further their understanding of the Simula language including features not considered here, e.g. implementation support for *garbage collection*.[9]

Perhaps not surprisingly Simula still has a healthy user-community who continue to develop the language's definition[10].

Smalltalk

Chronologically, Smalltalk[11] is the "next" object oriented language after Simula, not least, because it originated in the early 1970's from a consideration of the support provided by a Simula compiler. Smalltalk combines such support ncepts with a "type-less" style associated with the Lisp language[12].

What differentiates Smalltalk from languages like Simula and Eiffel is its exclusive use of *dynamic binding*, i.e. no (static) type checking is performed in the style associated with Simula and its derivatives - instead all "routine" are checked at run-time to determine if it may be applied to an object[13].

Another fundamental difference between Smalltalk and Simula (and its derivatives) is the lack of a clear distinction between a class and a class instance (object), i.e. in a Smalltalk environment (implementation) everything is an object including classes (control structures are simply operations of the appropriate class, see below). Whilst this might, at first, seem a rather confusing notion it in fact provides a rather clean conceptual means of reasoning about object-orientation, i.e.

? A single concept, that of an *object*, is applicable to all Smalltalk notions

? The notion of a *class* has a meaning in the context of a "run-time" *object*

? A class viewed in the context of an *instance* of a more abstract "meta-class" enables *methods* to be defined on classes rather than on class instances.

Fundamental to the Smalltalk ethos is the notion of a *message* and the underlying message passing paradigm. Messages may be unary, keyword or binary.

The expression oriented nature of the language ensures that expressions yield values and also may produce side effects (an expression may be evaluated solely for its side effect), the result of an expression is always a reference to an object, thus, the arguments passed to an operation and the result returned are also references[14].

? Unary Messages

bankAccount balanceOf

This example denotes sending the message *balanceof* to the object associated with *bankaccount* and corresponds to the dot notation of Simula (and its derivatives), e.g. bankaccount.balanceof.

Note how the subexpression (*bankAccount*) is evaluated to determine the "receiver" object which is requested to perform the operation (*balanceOf*).

? Keyword Messages

Where a message has arguments, e.g.

bankAccount credit: amount

bankAccount debit: 200

? Binary messages

Binary messages provide direct support for the notion of "everything" being an object in Smalltalk with conventional arithmetic notations for arithmetic. Smalltalk supports the notion of a conventional infix expression, e.g. 2 + 3 as a synonym for:-

2 addMeTo: 3

i.e. the conventional arithmetic operators are treated as the keywords for binary messages and parentheses are used to establish the usual precedence.

There are no "inbuilt" control structures in Smalltalk, instead, a "block" (function abstraction) is used. In the example below, a parameterless block is shown:-

```
counter <- [n <- n + 1]
```

This assigns a block to counter such that a subsequent call to this block, e.g.

```
counter value
```

forces the block body (`n <- n + 1`) to be evaluated incrementing the variable `n` and also yielding the incremented value.

Smalltalk provides a rich collection of predefined object classes including:-

? Primitive objects, e.g. Boolean, Integer, Float and Character

? Composite objects grouped into classes, e.g. Set, Bag, LinkedList, Array, String, etc.

An object class is defined by its *name*, the name of its *superclass*, the names of *any hidden variables* and the names and definitions of the *operations* (*methods*) associated with the class. All object classes are arranged in a hierarchy (of sub- and super-classes) and operations (methods) associated with a class are "automatically' inherited by its subclasses. Unlike Simula (and its derivatives) a class in Smalltalk may only export operations (methods), thus, an *attribute* may only be exported through a *function* that gives access to its *value*.

Eiffel

Eiffel provides an example of a particularly rigorously designed language. The conceptual basis for its design is that an object-oriented "software system" is a *structured collection of abstract data type implementations* with the following properties:-

? A *software system* is structured according to the *classes* of *objects* it manipulates and not according to the operations it performs on such objects. *Reuse* is constrained to whole data structures (abstract data types) and not via subcomponents, e.g. individual procedures.

? *Objects* are described as instances of *abstract data types* which may only be accessed via a well-defined interface.

? A *class* describes an *implementation* of an abstract data type or a group of implementations of abstract data types and not the abstract data type itself.

? A term *collection* denotes a group of classes which may be exploited by one or more *software systems*

? The term *structured* reflects the existence of a *relationship* between classes, in particular, a *multiple inheritance relation*.

In Eiffel, a *class* that represents an implementation of an abstract data type describes a *set* of potential "run-time" *objects* in terms of the operations available on those objects and the properties of those operations. Such objects are termed *instances* of the class and there is a distinction between the term *class* which is a compile-time notion and the term *object* which denotes an instance of the class that exists only at run-time.

An *entity* declared of a *class type* may refer at any time during execution to an *object* such that the *object* is an *instance* of the corresponding type. An *entity* which does not refer to any *object* is *void*, and by default, *entities* are void at *initialisation*, thus, *objects* must be created explicitly, i.e. by instruction.

Examples:

```
CLASS unnamed;
  ...
  an_account: ACCOUNT
```

introduce an *entity* `an_account` into the class `unnamed` and declare it to be of the class type `ACCOUNT`

```
   an_account.Create
```

explicitly create an *object* and associate it with the entity `an_account`

```
   an_account.open("Chris  Harrison");
```

apply the *feature* `open` to the entity `an_account` with an appropriate argument `"Chris Harrison"`

```
  print(an_annount.balance)
...
END
```

Applications of a *feature* employ the "dot" notation, e.g. `an_account.Create,` and *features* can be either *routines* or *attributes*. *Routines* in common with conventional high-level block-structured programming languages may be either *procedures* or *functions*. Note how, in the above example, class `unnamed` does not does not differentiate between attributes and functions, for example, `balance` may be either an *attribute* or a *function* in the class `ACCOUNT`. As the class `unnamed` is a *client* of the class `ACCOUNT` it does not need to know how a `balance` is obtained. A definition of the class `ACCOUNT` is shown below:-

[pic]

Features (attributes and routines) which are not exported are said to be "secret". Routines have a `is..do..end` clause. Given the above definition, `balance` is in fact realised as an attribute and is automatically initialised to zero after the call to `create`. The routines are always applied to a certain object specified by a client invoking the routine, e.g. `an_account.open("Chris  Harrison")`, denotes the object corresponding to the entity `an_account.`

Assertions provide a means of defining formal properties of classes and may be included as:-

? Preconditions which must be satisfied every time a routine is called

? Postconditions which are guaranteed to be true when a routine returns control, provided that the associated precondition was satisfied when the routine was entered.

? Invariants on classes which must be satisfied by objects of the class at all times

In addition to the above language features, Eiffel provides support for systematic *exception handling* (when assertions are violated at run-time), support for *generic classes* and support for *multiple inheritance*.

A detailed treatment of the language's design philosophy and implementation is given in Meyer's book "Object-Oriented Software Construction", Prentice-Hall. Students are recommended to read its initial chapters.

C++[15]

The very widespread use of C as a programming language has led to the adoption of C++ as a language for developing software systems. The design of C++ evidences an attempt to design a "better" C, i.e. to move away from an "architecture independent pseudo-high level assembly language" towards a language which incorporates notions drawn from "classical" approaches to language design. The extensions to C include *strong typing* (cf earlier lectures), support for *data abstraction* (again cf earlier lectures) and *object oriented* features.

Appendix 5 gives an example of a simple ADT realised in C++ for those students unfamiliar with the language.

Appendix 6 contains a reasoned critique and comparison of the Eiffel language and C++.

A Comparison of Smalltalk, Eiffel and C++

Eliens[Eliens95] provides a comparison of these three languages in terms of a number of "characteristics":-

? Uniformity[16]

Smalltalk	Eiffel	C++
Each data type is a class including simple data types, and control structures	Distinction between elementary and user defined classes do not behave as objects	Elementary data types and simple data structures

? Documentation value[17]

Smalltalk	Eiffel	C++
Consistent style (everything is an object)	Explicit support for assertions	Arguably a "terse" syntax

? Reliability

Smalltalk	Eiffel	C++
Dynamic typing means all type checking at run-time	Static type checking leads to high-reliability	Static type checking is weak across module boundaries

? Inheritance

Smalltalk	Eiffel	C++
Only single inheritance	Single and Multiple inheritance	Single and Multiple inheritance

? Efficiency[18]

Smalltalk	Eiffel	C++
Interpreted language so arguably "slower"	Compilation then dynamic binding with optimisations (by compiler) ensures adequate "efficiency"	Supposedly "designed" with efficiency "in mind"

? Complexity[19]

Smalltalk	Eiffel	C++
Regular structure (again everything is an object) leads to conceptual simplicity	Designed specifically as an object oriented language and does not support other "impure" features	Generally regarded as a "complex" language

"Object-Pascal"

A number of products are available which provide object oriented extensions to modular Pascal implementations[20]. Typically, these products provide:-

? A "class" type constructor

? Single inheritance only

? No support for type parameterisation.

? A mixture of block-structured imperative and object-oriented styles

? Strong typing

? Support for self-reference

One widely available product (Delphi) evidences the general style, i.e. non-standard object construction via a "create" method - a combination of the "Model" language's (see next section) type constructors and an *initial* method. Typically, method interfaces are defined in type definitions and method bodies are defined in the implementation part of a module.

Appendix 7 contains an example written in Delphi.

"Widely Used" Object Oriented Languages

As the reader might well suspect, a large number of object oriented programming languages is (are?) currently available ! The list below contains languages which are (arguably) "widely used":-

Statically-Typed:

Cobol with Objects.

C++

Classic-Ada

Dragoon

Emerald/Jade

Java

Object Pascal

Trellis/Owl

Dynamically-Typed:

Actors Languages

C+@

Flavors

Python

Self

Smalltalk

Both:

Actor

Ada95

BETA

C++ (With RTTI)

Cecil

CLOS

Eiffel

Modula-3

Objective-C

Sather

The "Model" Object Oriented Programming Language

A basic design principle of a "model" programming language is that the language *is strongly-typed*. The compiler will then ensure an operation may only be applied to entities which support that operation, thus ensuring that a major class of programming errors can be detected "automatically". In addition, to avoid "run-time" disaster, the following steps can be taken:-

? All values in the language are initialised to well-defined defaults.

? All accesses to structured values are checked to ensure that they are indexed within valid bounds - this sometimes involves a run-time check.

These two decisions mean that a fatally flawed program will always fail in a controlled manner.

The syntax of the "model" language has been designed for perspicuity and conciseness[21]. Consistency is at a premium, and special restrictions remain only where the removal of them would cause theoretical or implementation difficulties. No distinction between persistent and non-persistent data exists in the language, thus there are no special mechanisms required to preserve complex structures on storage devices[22].

In the "model" language, a program comprises of a set of one or more types, e.g.

```
TYPE first_type;
END;

TYPE second_type;
END;
.
.
.
TYPE n_th_type;
END.
```

Each type may be further elaborated in terms of its supertype(s), instance variables, type and value parameters and the methods (procedure and functions) that it supports, e.g.

```
TYPE first_type;

  SUPERTYPES window;

END;

TYPE second_type;

  SUPERTYPES first_type;

  VAR i: integer;
      s: string;

END;
.
.
.
TYPE n_th_type(i: integer);

  PROCEDURE initial;
  BEGIN

  END;

  FUNCTION mapping: integer = i + 1;

END.
```

An example of a simple "counter" type written in the "model" language is shown below:-

```
TYPE counter;
  SUPERTYPES window;
  VAR n: integer;
  PROCEDURE initial;
  BEGIN
    n:=0; self.request_ticks(60).set_font(self.times_font).set_font_size(72)
  END;
  PROCEDURE display; self.show_integer(n, 90, 40);
  PROCEDURE tick;
  BEGIN
    n:=n + 1; self.clear.display
  END;
END.
```

In this example, the predefined type "window" provides a supertype for the type counter which encapsulates a single instance variable `n` of type `integer`. The `initial` procedure (which is guaranteed to be invoked "first") sets `n` to the value `0`, and "cascades" applications of the predefined methods `request_ticks`, `set_font`, and `set_font_size` with appropriate arguments to `self` (an *object* of type counter). The procedure `display` applies the predefined operation `show_integer` (inherited from the supertype `window`) to `self`, and the procedure `tick` increments `n` by `1` and then applies the predefined methods `clear` and `display` to `self`. When compiled and executed, the program produces a window containing a integer value 0 which is then refreshed every second with its successor value.

Appendix 8 contains further examples of programs written in the "model" language.

A comprehensive manual for the "Model" language, which is actually called the "Feynman Language" after the world famous physicist[23], is provided at the end of the Appendices for students who wish to further their understanding of its syntax and semantics.

<u>Summary and Conclusions</u>

This lecture has examined a variety of different programming language "styles" each of which supports the object oriented paradigm. Central to any object oriented programming language's design is its support for defining objects, support for collecting objects together to form classes, support for creating objects, support for a subtyping relation, support for genericity, and also support for other notions which enable software systems to be "engineered", e.g. type checking/type inferencing, assertion checking, exception handling, etc.

Chris Harrison, January 1977.

Appendix 1

Algol and Algol-like Languages

In 1959, John Backus presented a paper [Bac59] on "the proposed international algebraic language" (which soon after evolved into the language that became known as Algol 60). Backus begins by giving informal descriptions of the syntax and semantics of the language, and outlines reasons why more precise descriptions are needed. But he then makes the following admission.

"The author had hoped to complete a formal description of the set of legal programs and of their meanings in time to present it here. Only the description of legal programs has been completed however. Therefore the formal treatment of the semantics of legal programs will be included in a subsequent paper."

Backus introduced the meta-notation we now know as Backus-Naur formalism and used it to specify the syntax of the language; however, the "subsequent paper" on the semantics never materialized, and the semantics was again described informally in the Revised Report on Algol 60.

Many computer scientists today are unfamiliar with this important document, subsequent corrections and discussion of various "ambiguities" may be found in [Knu67 , DHW76].

The importance of having precise descriptions of syntax and semantics is today generally appreciated; however, more than 35 years after Backus's talk, researchers are still trying to produce satisfactory descriptions of Algol-like languages. This situation is not unprecedented: mathematicians needed hundreds of years to sort out the semantic issues underlying the differential and integral calculi, and some thirty years passed between Alonzo Church's formulation of the (untyped) lambda calculus [Chu41] and the first mathematical models by Dana Scott [Sco72b].

Scott's work on the lambda calculus was partly motivated by a collaboration with Christopher Strachey [SS71] in which they outlined a "mathematical" approach to the semantics of procedural programming languages, using the powerful domain-theoretic tools Scott had developed. This approach, now termed denotational semantics, has become well known.

In [SS71], Scott and Strachey had suggested that if "you put your domains on the table first" then this would help to reveal language-design issues and possibilities.

One of the most significant contributions to semantic analysis of programming languages by Strachey and his collaborators, such as Scott [Sco72a], Rod Burstall [Bur70], and Peter Landin [Lan65], is the distinction between the environment and the store (or state). A domain analysis of Algol 60 reveals the disjointness of the "storable" and "denotable" values in this language; this was later described by John Reynolds as the distinction between "data types" and "phrase types."

In Strachey's paper, labels and jumps are not treated in any detail, but the important concept of continuations, which became the "standard" approach to these features, had already been discovered; see [SW74 , Rey93].

More information on traditional denotational semantics may be found in [MS76 , Sto77].

Basic Principles

A significant change in attitudes to Algol 60 was initiated by the paper "The Essence of Algol" by John Reynolds. At the time (1981), this language was generally regarded as having been superseded by new and improved "Algol-like" languages, such as Algol 68 [vW#69] and Pascal [Wir71]. But Reynolds argues that such languages are not truly Algol 60-like, and, in many important respects, are less satisfactory than the original. This is reminiscent of Tony Hoare's opinion [Hoa74] that Algol 60 is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors.

Furthermore, Reynolds points out that most formal models of programming languages then in use were failing to do justice to typed languages and to important concepts such as representation-independent storage variables and stack-oriented storage management. As a result, languages designed with such models in mind would inevitably be influenced by concepts that are not truly Algol-like.

In more detail, Reynolds characterized "Algol-like" languages as follows.

1.The procedure mechanism is based on the fully typed, call-by-name lambda calculus, and equational principles such as the fi and j laws are valid even in the presence of imperative features such as assignment commands and jumps.

2.The language has assignments, but procedures, variables, and other denotable meanings are not assignable.

3.Apart from overflow, roundoff error, and error stops, expressions are purely "mathematical," without non-local jumps, side effects, or ambiguous coercions.

4.Allocation and de-allocation are based on a stack discipline, but the treatment of storage variables is otherwise representation independent.

5.Generic features such as conditionals, recursion, and procedures are uniformly applicable to all types of phrases.

According to these criteria, languages such as Algol 68 and Pascal and their descendants are not Algol-like. (In some minor respects, such as the incomplete typing for parameters, even Algol 60 is not Algol-like! but these are regarded as design mistakes.)

Despite being imperative, Algol-like languages preserve all of the reasoning principles used in "pure" functional programming. However, this does not mean that Algol-like languages are necessarily simple to reason about, since this only involves the easy part: reasoning about parts of programs that don't involve change. The underlying problem is that interactions between assignments and procedures in conventional higher-order procedural languages can produce undesirable phenomena such as aliasing and covert interference via non-local variables. For this reason, many researchers [Bac78, WA85 , BW88, Hug89, Hud89] abandoned procedural languages entirely and promoted "purely" functional languages in which there is no interference at all. Other language designers have attempted to avoid these problems by significantly circumscribing the procedures in their languages; see, for example, the discussion of Euclid in [Mor82].

References and Associated Reading

[Abr93] S. Abramsky. Computational interpretations of linear logic. Theoretical Computer Science, 111(1-2):3-57, April 12 1993.

[Abr94] S. Abramsky. Interaction categories and communicating sequential processes. In Roscoe [Ros94], chapter 1, pages 1-16.

[AW85] S. K. Abdali and D. S. Wise. Standard, storeless semantics for Algol-style block structure and call-by-name. In A. Melton, editor, Mathematical Foundations of Programming Semantics, volume 239 of Lecture Notes in Computer Science, pages 1-19, Manhattan, Kansas, April 1985. Springer-Verlag, Berlin (1986).

[Bac59] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM -GAMM Conference. In Information Processing, Proceedings of the International Conference on Information Processing, pages 125-131, Paris, June 1959.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. Comm. ACM, 21(8):613-641, August 1978.

[BC82] G. Berry and P-L. Curien. Sequential algorithms on concrete data structures. Theoretical Computer Science, 20:265-321, 1982.

[Bro93] S. Brookes. Full abstraction for a shared variable parallel language. In Proceedings, 8th Annual IEEE Symposium on Logic in Computer Science, pages 98-109, Montreal, Canada, 1993. IEEE Computer Society Press, Los Alamitos, California.

[Bur70] R. M. Burstall. Formal description of program structure and semantics in first-order logic. In B. Meltzer and D. Michie, editors, Machine Intelligence 5, pages 79-98. Edinburgh University Press, Edinburgh, 1970.

[BW88] R. Bird and P. Wadler. Introduction to Functional Programming. Prentice-Hall International, London, 1988.

[CD78] M. Coppo and M. Dezani. A new type assignment for lambda-terms. Archiv. Math. Logik, 19:139-

156, 1978.

[Chu41] A. Church. The Calculi of Lambda Conversion. Princeton University Press, Princeton, 1941.

[Cla79] E. M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare-like axiom systems. J. ACM, 26(1):129-147, 1979.

[DHW76] R. M. De Morgan, I. D. Hill, and B. A. Wichmann. A supplement to the Algol 60 Revised Report. The Computer Journal, 19(3):276-288, 1976.

[FMS96] M. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the pi-calculus. In [LIC96], pages 43-54.

[HJ82] W. Henhapl and C. B. Jones. Algol 60. In D. Bjorner and C. B. Jones, editors, Formal Specification and Software Development, pages 141-173. Prentice-Hall International, London, 1982.

[HJ89] C. A. R. Hoare and C. B. Jones, editors. Essays in Computing Science. Prentice Hall International, 1989.

[HO94] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. Submitted for publication, 1994.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. Comm. ACM, 12(10):576-580 and 583, 1969.

[Hoa74] C. A. R. Hoare. Hints on programming-language design. In C. Bunyan, editor, Pages 193-216 of [HJ89].

[Hud89] P. Hudak. Conception, evolution, and application of functional programming languages. Computing Surveys, 31:359-411, 1989.

[Hug89] J. Hughes. Why functional programming matters. The Computer Journal, 32:98-107, 1989.

[Knu67] D. E. Knuth. The remaining troublespots in Algol 60. Comm. ACM, 10(10):611-617, 1967.

[Lan64] P. J. Landin. A formal description of Algol 60. In Steel [Ste64], pages 266-294.

[Lan65] P. J. Landin. A correspondence between Algol 60 and Church's lambda notation. Comm. ACM, 8(2,3):89-101 and 158-165, 1965.

[LIC96] Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Jersey, USA, 1996. IEEE Computer Society Press, Los Alamitos, California.

[LP95] J. Launchbury and S. Peyton Jones. State in Haskell. Lisp and Symbolic Computation, 8(4):293-341, December 1995.

[Mil89] R. Milner. Communication and Concurrency. Prentice-Hall International, 1989.

[Mor82] J. H. Morris. Real programming in functional languages. In J. Darlington, P. Henderson, and D. A. Turner, editors, Functional Programming and its Applications, pages 129-176. Cambridge University Press, Cambridge, England, 1982.

[Mos74] P. Mosses. The mathematical semantics of Algol 60. Technical monograph PRG-12, Oxford University Computing Laboratory, Programming Research Group, Oxford, January 1974.

[MS76] R. E. Milne and C. Strachey. A Theory of Programming Language Semantics. Chapman and Hall, London, and Wiley, New York, 1976.

[NB#60] P. Naur (ed.), J. W. Backus, et al. Report on the algorithmic language Algol 60. Comm. ACM, 3(5):299-314, 1960. Also Numerische Mathematik 2:106-136.

[NB#63] P. Naur, J. W. Backus, et al. Revised report on the algorithmic language Algol 60. Comm. ACM, 6(1):1-17, 1963. Also The Computer Journal 5:349-67, and Numerische Mathematik 4:420-53.

[Ole85] F. J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J. C. Reynolds, editors, Algebraic Methods in Semantics, pages 543-573. Cambridge University Press, Cambridge, England, 1985.

[OR95] P. W. O'Hearn and U. S. Reddy. Objects, interference, and the Yoneda embedding. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference, volume 1 of Electronic Notes in Theoretical Computer Science, Tulane University, New Orleans, Louisiana, March 29-April 1 1995. Elsevier Science (http://www.elsevier.nl).

[OR96] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus.

Unpublished draft, 1996.

[OT92] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, Applications of Categories in Computer Science, volume 177 of London Mathematical Society Lecture Note Series, pages 217-238. Cambridge

University Press, Cambridge, England, 1992.

[Plo73] G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI RM-4, School of Artificial Intelligence, University of Edinburgh, October 1973.

[PS93] A. Pitts and I. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In A. M. Borzyszkowski and S. Sokolowski, editors, Mathematical Foundations of Computer Science, volume 711 of Lecture Notes in Computer Science, pages 122-140, Gdansk, Poland, 1993. Springer-Verlag, Berlin.

[PW93] S. Peyton-Jones and P. Wadler. Imperative functional programming. In Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 71-84, Charleston, South Carolina, 1993. ACM, New York.

[Red94] U. S. Reddy. Passivity and independence. In Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 342-352, Paris, France, 1994. IEEE Computer Society Press, Los Alamitos, California.

[Rey81] J. C. Reynolds. The Craft of Programming. Prentice-Hall International, London, 1981.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, Information Processing 83, pages 513-523, Paris, France, 1983. North-Holland, Amsterdam.

[Rey89] J. C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, Automata, Lan guages and Programming, 16th International Colloquium, volume 372 of Lecture Notes in Computer Science, pages 704-722, Stresa, Italy, July 1989. Springer-Verlag, Berlin.

[Rey93] J. C. Reynolds. The discoveries of continuations. Lisp and Symbolic Compu tation, 6(3/4):233-247, 1993.

[Ros94] A. W. Roscoe, editor. A Classical Mind, Essays in Honour of C. A. R. Hoare. Prentice-Hall International, 1994.

[Sco69] D. S. Scott. A type-theoretical alternative to Cuch, Iswim, Owhy. Privately circulated memo, Oxford University, October 1969. Published in Theoretical Computer Science, 121(1/2):411-440, 1993.

[Sco72a]D. S. Scott. Mathematical concepts in programming language semantics. In Proc. 1972 Spring Joint Computer Conference, pages 225-34. AFIPS Press, Montvale, N.J., 1972.

[Sco72b]D. S. Scott. Models for various type-free calculi. In P. Suppes et al., edi tors, Logic, Methodology, and the Philosophy of Science, IV, pages 157-187, Bucharest, 1972. North-Holland, Amsterdam.

[SS71] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, Proceedings of the Symposium on Computers and Automata, volume 21 of Microwave Research Institute Symposia Series, pages 19-46. Polytechnic Institute of Brooklyn Press, New York, 1971. Also Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, Oxford.

[Sta85] R. Statman. Logical relations and the typed lambda-calculus. Information and Computation, 65:85-97, 1985.

[Sta96a]I. Stark. Categorical models for local names. Lisp and Symbolic Computation, 9(1):77-107, feb 1996.

[Sta96b]I.A. Stark. A fully abstract domain model for the pi-calculus. In [LIC96], pages 36-42.

[Ste64] T. B. Steel, Jr., editor. Formal Language Description Languages for Computer Programming, Proceedings of the IFIP Working Conference, Baden bei Wien, Austria, September 1964. North-Holland, Amsterdam (1966).

[Sto77] J. E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Pro-gramming Language Theory. The MIT Press, Cambridge, Massachusetts, and London, England, 1977.

[Str67] C. Strachey. Fundamental Concepts in Programming Languages. Unpublished lecture notes, International Summer School in Computer. Programming, Copenhagen, August 1967.

[SW74] C. Strachey and C. P. Wadsworth. Continuations: a mathematical semantics for handling full jumps. Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, Oxford, 1974.

[Ten94] R. D. Tennent. Correctness of data representations in Algol-like languages. In Roscoe [Ros94], chapter 23, pages 405-417.

[THM83] B. A. Trakhtenbrot, J. Y. Halpern, and A. R. Meyer. From denotational to operational and axiomatic semantics for Algol-like mlanguages: an overview. In E. M. Clarke, Jr. and D. Kozen, editors, Logics of Programs 1983, volume 164 of Lecture Notes in Computer Science, pages 474-500, Pittsburgh, PA, 1983. Springer-Verlag, Berlin, 1984.

[vW63] A. van Wijngaarden. Generalized Algol. In R. Goodman, editor, Annual Review in Automatic Programming, volume 3, pages 17-26. Pergamon Press, Oxford, 1963.

[vW64] A. van Wijngaarden. Recursive definition of syntax and semantics. In Steel [Ste64], pages 13-24.

[vW#69] A. van Wijngaarden (ed.) et al. Report on the algorithmic language Algol 68. Numerische Mathematik, 14:79-218, 1969.

[WA85] W. W. Wadge and E. A. Ashcroft. Lucid, the Dataflow Programming Language, volume 22 of APIC Studies in Data Processing. Academic Press, Lon- don, 1985.

[Wex81] R. L. Wexelblat, editor. History of Programming Languages. Academic Pres, New York, 1981.

[WH66] N. Wirth and C. A. R. Hoare. A contribution to the development of Algol. Comm. ACM, 9(6):413-432, June 1966.

[Wic73] B. A. Wichmann. Algol 60 Compilation and Assessment. Academic Press, London, 1973.

[Wir71] N. Wirth. The programming language Pascal. Acta Informatica, 1:35-63, 1971.

Appendix 2: Introduction to Simula

Here are some exmaples of simple programs written in Simula which should be understandable to anyone with a knowledge of a block-structured high-level imperative programming language.

An initial program:-

```
BEGIN
  WHILE 1=1 DO
  BEGIN
    outtext("Hello World!);
    outimage;
  END;
END;
```

Compute average value and the highest value from a list of whole numbers:-

```
BEGIN
  integer x, n, sum, max;
  IF lastitem THEN outtext("NULL LIST") ELSE
  BEGIN
    sum:=max:=inint;
    n:=1;
    WHILE  lastitem DC
    BEGIN x:=inint;
      n:=n+1;
      IF x > max THEN max:=x;
      sum:=sum+x;
    END;
    outtext("LIST LENGTH = ");
    outint (N, 6);
```

```
    outtext(",            HIGHEST = ");
    outint(MAX, 6);
    outtext(",      AVERAGE = ");
    outfix(SUM/N, 2,, 8);
  END;
  outimage;
END
```

Simula References

Textbooks on SIMULA:

SIMULA begin Graham M. Birtwistle, Ole-Johan Dahl, Bjoern Myhrhaug, and Kristen Nygaard. Petrocelli/Charter, New York (1975). ISBN 0-88405-340-7. Studentlitteratur, Lund, Sweden (1973) ISBN 91-44-06211-7. Bratt Institut fur Neues Lernen, Goch, GDR Chartwell-Bratt Ltd., Bromley, England. Auerbach, U.S.A.

Introduction to SIMULA 67 Guenther Lamprech Vieweg Verlag, Braunschwig, Wiesbaden (1981)

An Introduction to Programming in SIMULA R.J. Pooley. Blackwell Scientific Publications, Oxford, London, Edinburgh, Boston, Palo Alto, Melbourne (1987). ISBN 0-632-01611-6. ISBN 0-632-01422-9 Pbk

Object-Oriented Programming with SIMULA Bjoern Kirkerud. International Computer Science Series. Addison- Wesley Publishing Co. (1989). ISBN 0-201-17574-6.

Appendix 3

Simulation and Object Orientation

Building a simulation model for execution involves mapping some "real world" system into a programming language description. Meeting constraints such as reliability, accuracy and efficiency is particular difficult in simulations. As models become larger and more complex the likelyhood of errors increases causing reliability and accuracy problems. In addition, larger and more complex models are harder to set up and require more computational power to be simulated causing efficiency problems.

One widely accepted means of enhancing reliability, accuracy and efficiency is to maximize reuse of model parts. Using such an approach, complex models can be set up more efficiently and assembled from previously used and tested model

parts.

A large variety of techniques have been developed for maping real world systems into software simulations. Object orientation offers features for managing the inherent complexity in simulations and also for addressing issues such as heterogeneity, distribution and reuse. The most important features offered by object orientation are encapsulation of behavior and data, a common paradigm for system analysis, design and implementation of software, inheritance and polymorphism.

Encapsulation promotes the notion of distribution - a major issue for complex system modeling and simulation. Object orientation enables a systemic view of reality to be mapped (more or less) directly onto a software simulation without the usual semantic gaps and paradigm shifts related with other approaches, e.g. data modeling, structured analysis or functional decomposition.

Inheritance and polymorphism provide techniques for improving reusability and for dealing with dealing with heterogeneity. The notions of abstraction, encapsulation, inheritance and polymorphism combine to form a collective approach to managing heterogeneity, more specifically, abstraction enables generalized features and interfaces of heterogeneous notions to be captured directly. Inheritance provides support for organising heterogeneous notions in terms of a specialization hierarchy. Polymorphism provides support for organising descriptions in terms of the encapsulation of objects well-defined abstract interfaces.

Appendix 4

Smalltalk Texts

A Taste of Smalltalk

Ted Kaehler, Dave Patterson

Norton, 1986, ISBN 0-393-95505-2

An Introduction to Object-Oriented Programming and Smalltalk

Lewis J. Pinson, Richard S. Wiener

Addison-Wesley, 1988, ISBN 0-201-19127-X

Inside Smalltalk, Volume I

Wilf R. LaLonde, John R. Pugh

Prentice Hall, 1990, 0-13-468414-1

Inside Smalltalk, Volume II
Wilf R. LaLonde, John R. Pugh
Prentice Hall, 1991, 0-13-465964-3

Practical Smalltalk
Dan Shafer, Dean A. Ritz
Springer-Verlag, 1991, ISBN 0-387-97394-X

Smalltalk-80
Bits of History, Words of Advice
Glenn Krasner, ed.
Addison-Wesley, 1983, ISBN 0-201-11669-3

Smalltalk-80
The Interactive Programming Environment
Adele Goldberg
Addison-Wesley, 1984, ISBN 0-201-11372-4

Smalltalk-80
The Language and its Implementation
Adele Goldberg, David Robson
Addison-Wesley, 1983, ISBN 0-201-11371-6

Appendix 5

The following example is included for those unfamiliar with C++

In C++ the class keyword is used to define an abstract data type..

Example: A Linked List.

```
class List {
 private:
    Listelement * listHead;
 public:
   List() {listHead = NULL;}
   ~List() { release(); }
   void prepend(char data);
   void print();
   void release();
   // ...
 };
}
```

The above definition is of a C++ class named List . The body of the definition is divided into two parts labelled private and public. The first part contains a data member (a pointer variable), the second part contains the methods needed to manipulate the list. In addition, there are two definitions with the same name as the class - List(), and ~List(). The first is termed a (default) constructor and provides an "empty list"whose "head" pointer points to the NULL address. The second is a destructor calling the release() method in order to dispose of the elements of the list sequentially and thus release the storage space they occupy. All the other methods declared in this class remain to be elaborated. For convenience the class definition consisting of method declarations only is kept in its own file, e.g. List.hh, and the definitions of the methods in a separate file List.cc which includes the header file(s) using the #include compiler directive.

The meaning of the partitioning becomes clear when trying to use or instantiate the definitions: outside of the class body, only the use of methods in the public part is allowed and recognized. Thus, anything in this group is truly private to the user-defined type List. On the contrary, the public methods and variables can be used anywhere -- in other ADT definitions or in client programs. The fragment below is part of a sample client. At this point, it shall only show how the ADT might be used in a program called main():

```
main() {
  List w;
  w.prepend('A');
  w.prepend('B');
  w.print();
  // ...
```

The file which contains main() must include the class definition. The output, facilitated by the print() method, might simply result in printing the letters added to the (initially empty) list named w. You can see that a period is used as an operator to access the member functions.

Note: When there is no need for more than one object of a type, the modular programming style suffices and there is no need for data abstraction.

Appendix 6

NOTE: This article is dated 1989 and hence does not reflect changes to C++ since that date.

TECHNICAL DIFFERENCES Between Eiffel and C++

(Meyer 1989 in response to various questions etc)

Software structure

Eiffel software is organized in autonomous software units (classes), meant to be compiled separately. There is no main program. This is what I believe should be the case in object-oriented

programming. In contrast, I understand that C++ still follows the traditional C model. Quoting from Dr. Bjarne Stroustrup's ''The C++ Programming Language'' (Addison-Wesley, 1986), which seems to be the major reference on C++, page 22, lines 13-14: ''A C++ program typically

consists of many source files, each containing a sequence of declarations of types, functions, variables, and constants''. This is very far from the object-oriented model of software decomposition. Furthermore, reports from actual users of C++ seem to indicate a heavy use of ''include files'', a technique which I don't fully understand in the C++ context, and which

has no equivalent in Eiffel.

Assertions

A fundamental property of Eiffel software is that it may be equipped with assertions. Assertions are elements of formal specification that serve to characterize the semantics of classes and their routines independently of their implementation. Assertions include in particular routine preconditions (which must be satisfied when a routine is called), routine postconditions (ensured by the routine on exit) and class invariants (global consistency conditions applying to every instance of a class).

Assertions are essential for documenting components. As a matter of fact, I do not understand how one can talk about the very idea of reusable software components without assertions. Using a hardware analogy, a software component without assertions is similar to, say, an amplifier without precondition (the acceptable input voltage), postcondition (the gain, expressed as acceptable ratio of output to input) and invariant (including for example the temperature limits expected andmaintained by the amplifier). Yet of widely available programming languages, only Eiffel has these notions. (A system that does have assertions, and in fact ones that are more sophisticated than Eiffel's current ones, is David Luckham's Anna system, developed at Stanford on top of Ada. As far as I know, however, this is not a deliverable product.)

Beyond their documentation uses, assertions, which optionally may be monitored at run-time, provide a remarkable debugging and testing aid. At the recent Eiffel conference in Paris, one user organization (Cognos Inc.) reported that they no longer perform traditional unit testing, having replaced it by assertion monitoring.

Exceptions

Eiffel has exception handling. Its exception mechanism is original and I believe it is one of our major contributions, based on the theory of ‘‘Programming by Contract’’. As far as I know, there is no exception mechanism in C++. I believe that one cannot write serious software without having a way to recover cleanly from unexpected cases.

Global variables

Consistent with the absence of main program is the absence of global variables. Global variables are well known to be detrimental to modularity and more generally to quality. The Eiffel technique of ‘‘once routines’’ is used to ensure disciplined sharing between classes when needed. (See my column in the Journal of Object-Oriented Programming, vol. 1, no. 3, pages 73-77, ‘Bidding Farewell to Globals’’.) In contrast, C++ seems to support global variables in the C style.

Genericity

Eiffel classes may be generic, i.e. parameterized by types, as in LIST [T]. Here actual uses of the class may use any type (class) as actual generic parameter, as in

    my_list: LIST [TEXT_LINE]

The genericity may be constrained, as in MATRIX [T -> NUMERIC], which specifies that actual generic parameters must be descendants, in the sense of inheritance, of class NUMERIC (equipped with the operations "+", "-", "*" etc.). Descendants of NUMERIC include (in version 2.2) predefined types such as INTEGER and REAL. The operations of NUMERIC are available,

within the class, on any variable of type T - so that it can define, for example, routines for adding and multiplying matrices. Note that in this example MATRIX itself may inherit from NUMERIC.

Nothing of the sort exists in C++. This means that generic structures must be simulated by forcing type conversions, or ‘‘casts’’, using low-level C techniques. This defeats any attempt at static typing.

A paper was published not long ago to describe a proposal for class parameterization in C++. (Although the paper was published in a refereed journal, it regrettably did not mention any of the

two object-oriented languages that offer such a facility: Eiffel and Trellis-Owl, the latter designed by Craig Schaffert and others from DEC). Since by all reliable accounts the inclusion of such a facility in any form accessible to C++ users is several years away, it cannot be considered in any serious discussion. On that kind of time scale one can promise anything.

Dynamic binding

Dynamic binding is the default mechanism for routine calls in Eiffel (achieved without any undue effect on performance). The default policy in C++ is static binding; dynamic binding is only applied to routines declared as ‘‘virtual’’. This may look like an acceptable requirement to impose on programmers but I believe it is not. The whole idea of inheritance is that you may reuse a class

later on by writing a descendant and adapting it to new uses by overriding some of the routines of the original - within the original semantic constraints, as defined by assertions. This should be done without impacting the original, which may be used by many other ‘‘client’’ classes. (These concepts are explained in my book ‘‘Object-Oriented Software Construction, Prentice-Hall, 1988, as the ‘‘Open-Closed Principle’’, section 2.3.) In such a case the designer of the original routine may have had no inkling whatsoever that the routine would ever be redefined and subjected to dynamic binding. This is incompatible with the requirement that the original designer should have declared the routine as virtual in the first place.

Instead of forcing the programmer to take care of low-level optimizations, the Eiffel approach makes the compiler responsible for exploiting the performance of static over dynamic binding. The optimizer, working on a set of classes, generates code that applies static binding to any routine which warrants it (because it is never redefined).

Performing tedious and potentially dangerous optimizations in a safe way should be the role of computers, not humans.

In-line expansion

In C++ as in Ada a routine may be declared as ‘‘in-line’’, meaning that calls will be expanded in-line to gain performance. No such mechanism is available in Eiffel. Contrary to what one might think at first sight, I believe this to be a serious advantage for Eiffel. As soon as a routine is

declared as in-line, its usefulness is severely limited because it no longer is a normal routine that can be redefined and subjected to dynamic binding. The discussion of the previous paragraph

applies even more strongly.

In Eiffel, once again, the corresponding optimizations are performed by the compiler, not by the human user. The optimizer will automatically expand certain routines in-line based on systematic criteria beyond programmer control. One of the criteria is of course that the routine not be subject to redefinition and dynamic binding; the number of calls in the code is another.

Again, this seems the safe and efficient approach. Computers can perform this kind of task both more efficiently and more safely than humans.

Operator overloading

The term ‘‘operator overloading’’ is not entirely adequate since the issue is whether functions may be assigned names that will be used in prefix or infix form in calling expressions. This is a syntactic, not a semantic issue; the more important form of overloading, the semantic one, is provided in the object-oriented context by redefinition and dynamic binding.

C++ offers the possibility of using an operator (from a set of predefined ones) as function name; a similar possibility is offered in Eiffel 2.2, although it was not present in earlier releases. So the two languages are now indeed comparable in this respect.

Consistency of the type system

Beginning with version 2.2, Eiffel has a fully consistent type system in which every type, including basic types such as INTEGER, REAL and so on, is defined by a class (using the multiple inheritance mechanism). This was made possible by the introduction of the notion of expanded class, of special BITS M classes (whose instances are bit strings of length M), and of infix/prefix operators as discussed above. This is achieved without any effect on the efficiency of dealing with simple values such as integers, characters and the like. The advantage is mainly a conceptual one - being able to work with a single set of concepts admitting few

special cases.

There doesn’t seem to be anything similar in C++, which uses the C types as basis.

Type checking

Because of the absence of genericity and the presence of the full C type system with its casts and other unsafe mechanisms, C++ cannot be reasonably be called a statically typed language. In contrast, Eiffel was designed as fully typed.

The present Eiffel compiler misses a small number of type violations (arising in particular from cases in which polymorphism enables a client to evade an export or redefinition constraint). These cases seldom arise in practice, which is not an excuse for not handling them properly. Even with the current implementation, however, Eiffel is incomparably more type-safe than C++ because of the presence of genericity, of the strict enforcement of type checks in assignments, and of the absence of any unsafe casts or conversions.

Friend functions

C++ has a notion of friend function which, as I understand it, makes it possible to define routines outside of the object-oriented framework. There is nothing equivalent in Eiffel. This facility is not missed; I would see its introduction as a dangerous violation of the object-oriented principles.

Deferred classes

An extremely important notion in Eiffel is that of deferred class, which describes a non-fully-implemented abstraction. Deferred classes are used to capture commonalities and are central to the object-oriented approach. Two aspects are particularly important: the ability to define a partially deferred class, which contains both implemented and non-implemented routines; and the ability to attach assertions to a deferred class and its deferred routines, and thus to specify the behavior of yet to be implemented software.

C++ as described in published references does not appear to support a similar notion. I have heard, however, that the forthcoming version of C++ has a notion of abstract class, which is meant to play the same role. Perhaps someone will describe this facility in detail so that readers

can judge.

Multiple inheritance

Multiple inheritance is fundamental in the Eiffel approach. We made every effort to handle it in a very clean way; name clashes, in particular, are treated in what I believe is the right way. (More precisely, I do not know of any satisfactory solution in any other language. This is a strong statement, and proponents of other languages are welcome to respond to the challenge.)

...

Renaming

Eiffel offers a powerful technique in connection with inheritance: renaming. A class can rename inherited routines and attributes (i.e. methods and attribute variables for those who prefer such terms). This is used for removing name clashes in multiple inheritance and also, perhaps even more importantly, to provide locally adapted terminology when you inherit the right features but under the wrong names. As discussed in my OOSC book referenced above (section 10.4.7) and in a JOOP column (Vol. 1, no. 4, pages 48-53), this is essential if inheritance is to provide support for reusability in a practical industrial context.

Garbage collection

This item and the next violate Mr. Geary's request to limit the discussion to language features. I have included them anyhow because, even though they are environment rather than language features, they are made possible or next-to-impossible by the language design.

To write serious object-oriented software, which at run time will inevitably generate many objects, some of which may become useless, one needs a good garbage collector. This is the case in Eiffel (which uses an incremental, parallel scheme so as not to impair performance). As far as I know, C++ systems do not support garbage collection, which would be extremely difficult if not impossible to implement because of the presence of C types and mechanisms.

Automatic recompilation

One of the most important practical aspects of Eiffel is the automatic compilation mechanism, based on automatic analysis of inter-class dependencies (multiple inheritance and client). This removes the need for make files and include files. Although I recall some seemingly interminable

notes on the feasibility (or lack thereof) of a similar mechanism in comp.lang.c++, I don't know of any implemented mechanism for C++. Again, this seems due to the very design of the language; and again, the difference seems to result from irreconcilable views of what should be

done by computers and what should be done by humans. The Eiffel view is that error-prone and tedious management tasks should be handled by tools, and that programmers should concentrate on solving programming problems.

Pointer arithmetic etc.

One of my major objections to C++ stems from what that language has rather than what it has not. Because C++ retains almost total compatibility with C, it keeps all its low-level and dangerous features. The design of C dates back to the late sixties and is obsolete by modern software engineering standards. Compatibility with C means that in C++ you still have pointers,

type casts, pointer arithmetic, function pointers, malloc, free, bizarre operator precedence (the famous asterisk/parenthesis bugs), weak type checking and so on.

I strongly disagree with this approach if the goal is to obtain software quality. Take pointer arithmetic, for example. I would contend that you can have quality software, or you can have pointer arithmetic; but you cannot have both at the same time.

In Eiffel, the choice has been made. None of these low-level features

are present ...; needless to say, they are not missed.

....

Simplicity and ease of learning

Much of the plea for C++ is based on the observation that it provides an easy transition from C, which (for better or worse) is the language many programmers know nowadays. Using Dr. Brad Cox's expression (meant for Objective-C), this supports an ''evolutionary'' approach.

I can certainly respect this view and its appeal to software managers in industry. But I believe that by considering it more closely one will find it short-sighted and ill-founded.

Learning a new language such as Eiffel is nothing for a competent programmer. For Eiffel, which is small and simple, the learning process typically lasts a few days at most. Nobody has ever told us that Eiffel was difficult to learn. (If you read this, have tried to learn Eiffel, and found otherwise, please respond!) I believe that the process of going to Eiffel is in fact much smoother, as you don't have to use a confusing mix of old and new concepts. In a language that you master totally, you feel confident and you can concentrate on your job rather than on the language intricacies.

Also, the brief initial shock produced by the realization that you cannot easily write your programs in a traditional way any more is, in the experience reported by Eiffel users, highly salutary.

CONCEPTUAL DIFFERENCES

The considerable differences listed above more than offset, in my mind, any similarity that may seem to exist between Eiffel and C++. Beyond these individual technical differences, the contrast between the two languages is deep and conceptual. Eiffel is a new language and environment designed with a precise charter (enabling the production of very high quality software by professional programmers). C++, as I see it, is an attempt at a more modern version of C. (Dr. Stroustrup's book, in the ''historical note'' on page 5 of his book, writes that ''the difference between C and C++ is primarily in the degree of emphasis on types and structures'') .

There are undoubtedly arguments for both approaches. Obviously, I believe that arguments for the first are much stronger.

Appendix 7

A List ADT in Object Pascal

```
UNIT a_list;

INTERFACE

TYPE list =

CLASS

  PUBLIC

    FUNCTION is_empty: Boolean; VIRTUAL;
```

```
    FUNCTION head_of: integer; VIRTUAL;
    FUNCTION tail_of: list; VIRTUAL;
  PRIVATE
END;
empty =
CLASS(list)
  PUBLIC
    FUNCTION is_empty: Boolean; OVERRIDE;
  PRIVATE
END;
cons =
CLASS(list)
  PUBLIC
    CONSTRUCTOR create(h: integer; t: list); VIRTUAL;
    FUNCTION is_empty: Boolean; OVERRIDE;
    FUNCTION head_of: integer; OVERRIDE;
    FUNCTION tail_of: list; OVERRIDE;
  PRIVATE
    head: integer;
    tail: list;
END;
IMPLEMENTATION
  FUNCTION list.is_empty: Boolean; {VIRTUAL;}
  BEGIN
    is_empty:=false
  END;
  FUNCTION list.head_of: integer; {VIRTUAL;}
  BEGIN
    head_of:=0
  END;
  FUNCTION list.tail_of: list; {VIRTUAL;}
  BEGIN
    tail_of:=nil
  END;
  FUNCTION empty.is_empty: Boolean; {OVERRIDE;}
  BEGIN
    result:=true
  END;
  CONSTRUCTOR cons.create(h: integer; t: list); {OVERRIDE}
  BEGIN
    INHERITED create;
    head:=h;
```

```
    tail:=t
  END;

  FUNCTION cons.head_of: integer; {VIRTUAL;}
  BEGIN
    head_of:=head
  END;

  FUNCTION cons.tail_of: list; {VIRTUAL;}
  BEGIN
    tail_of:=tail
  END;

END.
```

Appendix 8

Consider, first, the interface component of an axiomatic specification[24] of a "list" adt:-

```
ADT list;
  INTERFACE
    TYPE list;
    CONSTRUCTOR empty: list;
    CONSTRUCTOR cons(x: integer; l: list): list;
    FUNCTION head(l: list): integer;
    FUNCTION tail(l: list): list;
    FUNCTION is_empty(l: list): Boolean;
  EQUATIONS
    ...
END.
```

Consider, next, an implementation of a list data type in the "model" object-oriented language. Development of the implementation proceeds top-down, with each step adding an additional level of refinement.

1) Describe the type "list", which repesents the top or "root" of the implementation:

```
TYPE list;
END;
```

2) We know that lists can be either:

Empty, or

Non-empty.

Both Empty and Non-empty lists are different **kinds** of list, which is reflected in the implementation by subtyping[25] the type list to "contain" both empty and non-empty lists:

```
TYPE list;
  TYPE empty;
  END;
  TYPE cons;
  END;
END;
```

The model language provides an explicit SUPERTYPES clause which is used to express a type-supertype relationship in those cases where a type has more than one supertype. The above example could be rewritten in this style, producing a semantically equivalent type hierarchy:

```
TYPE list;
END;
TYPE empty;
  SUPERTYPES list;
END;
TYPE cons;
  SUPERTYPES list;
END;
```

3) We can usefully include comments (in quotes) drawn from a formal language in the source text for list - this indicates how each component of theimplementation relates to the corresponding specification.

```
TYPE list;
  "CONSTRUCTOR" TYPE empty;
  END;
  "CONSTRUCTOR" TYPE cons;
  END;
END;
```

4) In this example, there are no further types defined inside `empty` and `cons`. We can now begin to refine each type with attributes - a finer level of refinement than that associated with the type hierarchy.

```
TYPE list;
  "CONSTRUCTOR" TYPE empty;
  END;
  "CONSTRUCTOR" TYPE cons(x: integer; l: list);
  END;
END;
```

5) Finally, we complete the lowest level of refinement which is to enumerate the operations applicable to each type in turn. Operations in the specification are partitioned with respect to the type hierarchy derived in the implementation.

```
TYPE list;
  "CONSTRUCTOR" TYPE empty;
      "PREDICATE" FUNCTION is_empty: boolean = true;
  END;
  "CONSTRUCTOR" TYPE cons(x: integer; l: list);
      "PREDICATE" FUNCTION is_empty: boolean = false;
      "DESTRUCTOR" FUNCTION head: integer = x;
      "DESTRUCTOR" FUNCTION tail: list = l;
  END;
END;
```

6) The above type definition represents the set of all potential objects of type list - it does not define any particular object of type list. If we want to create a list object, we must define an owner type for the type list, which encapsulates the list object:

```
TYPE list_owner;
  VAR l: list;
  PROCEDURE initialise_list; l:=empty;
END;
```

6.1) An alternative form of "ownership" occurs for types which are aggregations of **objects** of other types, some of which are lists:

```
TYPE list_owner(i: integer; b: boolean; l: list);
END;
```

To create on object of type list_owner, we could use:

```
VAR l: list_owner;
l:=list_owner(42, false, cons(21, empty));
```

6.2) An further form of "ownership" occurs for types which are aggregations of other **types**, some of which are lists:

```
TYPE list_owner{i <= integer, b = boolean, l <= list};
END;
```

In this example, the type list_owner is an aggregation of three *types*, and not three *objects*. An object of this list_owner type is created as follows:

```
VAR l: list_owner;
l:=list_owner{natural_numbers, boolean, cons};
```

7) We can augment an existing type definition in several ways:

Add an additional constructor,

Add an additional instance variable,

Add an additional operation,

Redefine the meaning of an existing operation.

In the example below, an additional "auxilliary" operation, cardinality, augments the existing operations defined over the type list::

```
TYPE list;
  "CONSTRUCTOR" TYPE empty;
      "PREDICATE" FUNCTION is_empty: boolean = true;
      "AUXILLIARY" FUNCTION cardinality: integer = 0;
  END;
  "CONSTRUCTOR" TYPE cons(x: integer; l: list);
      "PREDICATE" FUNCTION is_empty: boolean = false;
      "DESTRUCTOR" FUNCTION head: integer = x;
      "DESTRUCTOR" FUNCTION tail: list = l;
      "AUXILLIARY" FUNCTION cardinality: integer = 1 + tail.cardinality;
  END;
END;
```

In the example below, the meaning of the operation cardinality is redefined for objects of type pair_cons, to include the cardinality of l2:

```
TYPE list;
  "CONSTRUCTOR" TYPE empty;
      ...
  END;
  "CONSTRUCTOR" TYPE cons(x: integer; l: list);
      ...
      "CONSTRUCTOR" TYPE pair_cons(l2: list);
          "AUXILLIARY" FUNCTION cardinality: integer =
            1 + tail.cardinality + l2.cardinality;
```

```
      END;

  END;

END;
```

8) Whilst apparently a conceptually simple notion, multiple inheritance introduces a significant change in the way we choose to reason about things. In the absence of multiple inheritance, we reason in terms of the stepwise refinement of a collection of types, as described above. When we want to aggregate types together, we use:

? The type-subtype relationship,

? Parameterisation by a type,

? Parameterisation by objects of a type.

It is only after an initial refinement (or alternatively, because it has been recognised at the specification or design stage) that we can factor out commonalities and adopt an alternative form of aggregation, i.e. aggregation by multiple inheritance.

In the example below, we have recognised that many types may require a name associated with their objects, and have therefore defined a type whose sole purpose is to capture a name. For any subsequent type which requires a name, we can form an arrgegation by multiple inheritance.

```
TYPE named(s: string);
END;

TYPE car;
END;

TYPE fast_car;

  SUPERTYPES named, car;

END;
```

However, in the example above, we utilise the type-subtype relationship in two contrasting ways:

? A subtype is conceptually a member of the set of objects described by a supertype (a fast_car is always a car),

? A subtype reuses the definitions of a supertype (a fast_car *has* a name).

9) "self"

The language requires that all method application is associated with either:

An attribute of a type,

Self.

For example,

```
TYPE example;

  FUNCTION factorial(x: integer): integer;
  BEGIN
    IF x = 0 THEN result:=1 ELSE result:=1 + self.factorial(x - 1)
  END;

END.
```

Such a scheme enables the language to be kept regular, as "self" cannot be omitted if the method is defined locally.

Appendix 9

The Model Language Type Graph (Lattice)

The diagram below provides a "snapshot" of the static organisation of the type lattice supported by the implementation of the model language. As the language manual makes clear, taking an arbitrary type (known as 'self type'), that type may inherit information from a number of supertypes, and may in turn be an ancestor of a number of subtypes.

Every type is considered to be a supertype of itself, and hence it is also a subtype of itself.

We consider that at the 'top' of the graph is a special type called 'any', which is a supertype of every type.

In general, the developer is interested in which types are compatible with each other, or which types are compatible with the constructs of the language. Type compatibility is concerned with where differing types fit to the type graph.

[pic]

[pic]

A large collection of pre-written types are stored in the "service layer" of the language's implementation. The attached manual for the service layer details these components which are available to users of the system by simple reference.

----

[1] This is never-the-less a "big" lecture ! In order to make its content easier to assimilate key notions are introduced and explained in the various sections whilst detailed examples and notes have been included as appendices. These appendices provide a starting point for further study for those students who wish to develop their understanding.

[2] Self is an experimental object oriented language designed for expressive power and to be "adaptable". It combines a pure, prototype-based object model with uniform access to state and behavior. Unlike other languages, Self allows objects to inherit state and to change their patterns of inheritance dynamically. Self's customizing compiler can generate very efficient code compared to other dynamically-typed object-oriented languages.

[3] A discrete event simulation model describes physical systems in terms of state changes which occur in response to individual events occuring at discrete instants. In a continuous siulation a systems state is modelled in terms of continuous evolution. Both are essentially modelling techniques which may equally well be applied to developing simulations of physical systems regardless of whether a physical system ius inherently continuous or discrete. See appendix 3 for a short discussion of simulation and object orientation.

[4] I wrote my first "high level" programs in Algol 60 nearly twenty years ago, yet, if you care to look at Appendix 1, Algol's design still has ramifications for today's language designers. I don't suggest that anyone reads all of the associated references - they are there to demonstrate how notions which may appear "obsolete" in the 1990's are in fact fundamental to the design and implementation of even so-called "state of the art" object oriented languages !

[5] As Algol 60 is an (almost) strict subset of Simula most Algol programs will compile using a Simula compiler.

[6] It is instructive to consider the term "classical language". Classic and classical are sometimes interchangeable when used as adjectives, as in such phrases as classic/classical design or look. Classical is, in fact, more common in pertaining to an ancient Greek or Roman culture. Classic has a more general range of use including the broadest sense of "highest rank or excellence". In Physics, for example, the phrase "classical theory" is used to denote a theory which is conventional or authoritative rather than new or experimental, e.g. Einstein's general theory of relativity (because it does not take account of the uncertainty principle of quantum mechanics as it should for consistency with other theories). The term classic has in fact undergone considerable semantic development in recent years with its meaning of "typical" "appropriate" and its widespread use in informal speech, e.g. "that's classic". Now wonder natural language processing is

so problematic !

[7] The default static binding ensures that if *n* is a non-virtual feature declared at level *A* then `a1.n` denotes the *A* version of *n* even if there is a different version in *A* . Dynamic binding is forced by the QUA construct, e.g.

```
(a1 QUA B).n
```

The automatic adaptation of every operation to its target (which is lost with the use of the QUA construct) may be achieved by declaring polymorphic routines to be virtual. The INSPECT construct enables one or more operations to be performed as a result of a type inspection, e.g.

```
INSPECT a1
  WHEN A DO ...;
  WHEN B DO ...;
  .
  .
```

[8] As you should be aware, *co-routine*s are modelled after parallel processes as they manifest themselves in operating systems and "real-time" software. Consider, first, how a call to a program unit, e.g. a procedure or subroutine, in a <u>sequential</u> program results in that unit executing <u>completely</u> before control is returned back to the caller. Co-routines are modelled after a parallel execution of program units in which execution is interrupted only when a unit needs to provide information or obtain information from another unit but this execution takes place on a <u>sequential</u> computer, i.e. co-routines emulate parallel execution.

[9] In common with other programming languages which provide support for dynamic data structures, object oriented programming languages must provide some means of managing the persistent representation of objects. One approach is to make such management "invisible" to a user of the language by implementing some form of "automatic" garbage collection. Alternatively, explict commands may be provided in the language on order that "programmer controlled" memory managment may be realised.

When "automated" storage management is made part of the implementation of a programming language we must develop some highly-reliable means of determining when an "object" is no longer referenced by any other object so that its storage can be "reclaimed" and "recycled". One primitive technique is usually termed *reference counting* - each object maintains a "count" of the number of references to it, and , when this count becomes zero, the object (more properly the space it "occupies") may be "recycled". A more general technique is usually termed *mark-sweep* garbage collection after its two phase nature, i.e. first a "garbage collector" starts at the various root(s) of some structure and traverses the "active" part of the structure marking all of the "live" objects that are found, then, in a second phase, the garbage collector "sweeps" through the whole structure putting any objects which are unmarked into a "list" of "recyclable" objects and at the same time unmarks all objects it encounters.

Algorithms for automatic garbage collection have been known for some time yet remain an area of fundamental research.

[10] See, for example:-

http://www.isima.fr/asu/ ASU, The Association of Simula Users

http://home.sn.no/~simula/ Simula Standards Group

[11] We must be careful to differentiate between Smalltalk the <u>language</u> and Smalltalk the <u>environment</u>. The Smalltalk environment addresses many of the issues usually associated with operating systems and underlying hardware and provides a distinctive WISIWYG user-interface. Smalltalk is a particularly "pure" language whose syntax is somewhat ideosyncratic.

[12] Lisp is often described as a "pure" funtional language, however, just as Prolog fails to achieve the "goals" of logic programming by having to incorporate features which improve execution efficiency, Lisp implementations similarly must make allowances for an inappropriate underying (machine) architecure. In its original form Lisp was a pure functional language but many additional features have been incorporated into langage implementations to improve execution efficiency and also to make it more "useful" for program development.

[13] In Smalltalk every object has a "tag" which gives its class. Dynamic type checks at run-time ensure compatibility but a programmer may check the class of an object explicitly if required. One advantage of dynamic type checking is that composite objects can be heterogenous, i.e. an object can be enriched (after its initial creation) with components of different classes. These additional classes need not even have been envisaged when the object was created. Variables always contain references to objects, thus, a given variable can refer to objects of any class.

[14] In general:-

E0 I denotes the evaluation of a subexpression (*E0*) to yield a reference to a receiver *object* which is requested to perform the operation *I*

E0 <operator> E1 requests the receiver object to perform the operation named *<operator>* with the object yielded by *E1* as an argument

E0 I 1 : E1... I n : En requests the receiver object to perform the operation named *I 1 : ...I n* with the objects yielded by *E1 , ...En* as arguments

[15] This section is included for completeness.

[16] Uniformity is a vital property of any formal language since it reduces the reasoning involved in its use. Language designers are guided by general principles, e.g. the correspondence principle, the type-completeness principle etc. Adhering to such principles helps to ensure that a language is regular in the sense that few (hopefully none) of the "structures" in the language is "special", i.e. is treated differently (under certain conditions) that other "structures".

[17] By the term "documentation value" Eliens is presumably referring to the relative "ease" with which it is possible to read, write and reason about strings written in the language. Such a criterion is particularly difficult to quantify sinse it depends not only on the syntax and semantics of the language but the style of use adopted.

[18] The notion of *efficiency* has little meaning (to a computer scientist) except in the context *execution efficiency*. How "quickly" something can be compiled and how "fast" it then executes, or how quickly interpretation "takes place", are entirely separate issues.

[19] The notion of a language's *complexity* is particularly problematic. How would you determine if one particular language is more or less "complex" than another ?

[20] Many "modular Pascal's" are "direct" descendents of the famous UCSD Pascal.

[21] The language provides a comprehensive collection of predefined and "built-in" types. See the language reference provided and also the type hierarchy diagram in Appendix 9.

[22] Persistence is a separate but related notion which has been the subject of research for a number of years, see for example, PSAlgol and other work on *persistent object stores*.

[23] Not only was Richard Feynman a world renowned physicist (and Nobel laureate), he was also a gifted teacher. Those with an interest in Physics, and in science in general, are recommended to read his lecture notes which provide an insight into the mind of a truly great scientist (now sadly deceased).

[24] A detailed consideration of *axiomatic specification* and its use in the description of ADT's and of objects on attributes is provided in a later lecture. The specification component is shown here to demonstrate how "software engineering" involves the use of formal notations to describe potential products and how implementations of such products are developed systematically from such formal descriptions.

[25]At this point, we are introducing the notion of inheritance. In the model language, a type defined defined within another type denotes a subtyping relation, i.e., the enclosed type is a proper subtype of the enclosing type and is not encapsulated by the enclosing type.