

# Omega: A Uniform Object Model Easy to Gain Ada's Ends

Xianzhong Liang, Zhenyu Wang

E-mail: liangxz@public.wh.hb.cn zywang@public.wh.hb.cn

Wuhan Digital Engineering Institute

P. O. Box 74223, Wuhan, P. R. CHINA

**[Abstract]** *Ada provides full capacities of supporting object-orientation, but multiplex object patterns introduced by Ada are so complex that it is difficult for novices to master them. Investigating object patterns implemented by package, protected and task units, this paper presents a uniform object model (Omega), so as to simplify object-orientation with Ada. And exploiting Ada95's capacities, an approach of an Omega pre-processor is also made in confirmation of the model, which rationally hides from the complexity.*

## A. Introduction

From the viewpoint of the historical development [1], Ada provides three kinds of object constructs: package, task and protected unit, all of which manage attributes of its own and present routines for manipulation of attributes. A package is the module structure of rich function, which can be used to implement ADT (abstract data type) and ASM (abstract state machine). A task is a concurrent unit, whose entries are used for the communication between tasks. A protected (unit) is a critical field, whose routines can be used to make an exclusive visit to the protected data.

### Feature-enclosed classes

Each of the encapsulation languages offers a modular construct for grouping logically related program elements. Ada calls it a package; corresponding notions are known as modules in Modula-2 and Mesa, and clusters in CLU.

Besides packages, Ada offers other interesting modular constructs: the task and the protected units. They also deserve a mention purely for their modular concepts, since they actually come closer than packages to supporting object-oriented concepts. Syntactically, task and protected units share many aspects of packages. So besides making up a syntactical unit, a task or a protected unit also describes a semantic component - unlike a package, and like a class [2].

A class is characterized by features, including

attributes (representing fields of the objects of the class) and routines (representing computations on these objects). So in Ada modular constructs, a routine may be a function, which returns a result, or a procedure, which does not, or an entry, which synchronizes communication between tasks, or the mutual exclusive subprogram for coordinated sharing of data between tasks. Architecturally it is routines in modular constructs that bring interaction among components, for an object-oriented system is composed of objects (components).

### Interactive effect on objects

From the viewpoint of software architecture, a composite of attributes and routines represents a system component (instance of a class), with which other components interact through these routines [3]. In this way, there are many kinds of object patterns in Ada, such as a variable of ADT, a package of ASM, a task or, a protected unit. But routines' effect on objects is one of two ways by which one object interacts with the others: explicit - or implicit effect. The routine with explicit effect performs effect through the argument (the object passed as a formal parameter to the routine), while the routine with implicit effect performs effect through side-effect (implicit way). The call to a routine through explicit effect belongs to traditional procedural call. In contrast, the call to a routine through implicit effect embodies message-sending (the called routine seen as a message to the object).

In the paper [4], we have classified the object molds in Ada into two kinds: V-object and U-object. A V-object denotes an ADT (which provides routines with explicit effect) and is used as a common variable, while a U-object denotes an ASM (which provides routines with implicit effect) and is used as a unit cluster of data and routines (similarly task and protected units can be taken into account).

### Diversified objects in Ada95

A class is defined as both a structural system component -- a module (unit) -- and a type, but one hardly finds such a perfect (even closer) notion of class in Ada95 (task and protected units are

exceptions).

In Ada the V-object denoting ADT is object-oriented mainstream, with support for single inheritance, polymorphism and dynamic binding. It's a pity that a V-object is a pure variable – only concerning data structure. U-objects are more quite object-orientated than V-objects, because the unit denoting a U-object may have a type (task and protected type, except for a package). But U-objects support no inheritance, and some of which have very special semantics (e.g. a task can only fit into concurrent process).

As well known, to Ada83, a sophisticated construction, Ada95 has added a whole new set of constructs with many potential interactions both between themselves and with the old constructs. If one comes from the O-O side and are used to the pristine simplicity of the notion of class, he would feel befuddling at first and will find that he has to face the intricacies of objects, each kind having the far-fetched and so-called *class* of its own:

---

➤ V-object

This kind of object denotes an ADT in a package, which concerns a specific type (viewed as *class*) and interactive routines with explicit effect. The object declared from the type is a typical variable (V-object). V-objects are object-oriented mainstream in Ada95, which may involve added-newly concepts characterized by following special types:

- ◆ **tagged type** permits to derive a subtype with extension. For the derived type, besides inheritance of features (attributes and routines), a programmer may add some new features or redefine some of routines.
- ◆ **controlled type** is a descendant of type *Controlled* and gives the user additional control over special operations. In particular, an *Initialize* procedure can be treated as the constructor and a *Finalize* procedure as the destructor for the controlled object.
- ◆ **abstract type** is a tagged type intended for use as a parent type for type extensions, but which is not allowed to have objects of its own. An abstract subprogram has no body, but is intended to be overridden at some point when inherited.

---

➤ U-object

This kind of object embodies different units, which concerns interactive routines with implicit effect and manages attributes of its own. But what is the exact class will depend on which unit is used (a package, a task, or a protected unit). One can freely choose the unit for a U-object pattern according to which unit he uses:

- ◆ **package unit** (denoting ASM) treats automata status as attributes and provides interactive routines with implicit effect. The class can be gotten by attaching the keyword *generic* to the specification unit
- ◆ **protected unit** (for the coordinated sharing of data between tasks) treats critical data as attributes and provides interactive routines with mutual exclusion. The class can be gotten by attaching the keyword *type* to the specification unit
- ◆ **task unit** (defining concurrent computations) treats communicating messages as attributes and provides synchronized routines for communication between tasks. The class can be gotten by attaching the keyword *type* to the specification unit

---

**Convention:** With U-object being constructed by different units, we will generally use “U-object” to refer to a package unit, “task U-object” to a task unit, and “protected U-object” to a protected unit. Sometimes, we use “specific U-object” to refer to both task and protected unit.

Task and protected units in Ada95 are ideal class constructs, which manage attributes and provide interactive routines with implicit effect. But these specific U-objects are not popular because of their specific semantics -- e.g. their routines are only of synchronized or exclusive operation.

An Ada program is composed of one or more program units. Similarly an object-oriented system in Ada95 will be composed of multiplex objects. The objects may be V-objects (which perform an ADT) and U-objects. The later include packages (which perform an ASM), tasks (which perform concurrency) and protected units (which perform exclusive protection).

A little later, different object design patterns in Ada95 will be discussed, and whose characteristics are thoroughly investigated. And last, a uniform object model (known as Omega: Object Model Easy to Gain Ada's Ends) is reasonably recommended. The Omega introduces a pristine notion of class to unify rationally the multiplex objects in Ada95. With prefix modifiers, the essential class construct can be adapted to meet the needs of different object design patterns. The accompanied Omega pre-processor fully exploits Ada95's capabilities, but effectively hides from its complexity.

## B. Object Patterns

### V-Object: denoting ADT

ADT design pattern is the mainstream of object-orientation in Ada, so the V-object (denoting ADT) with new set of concepts such as *tagged*, *class-wide* and *controlled* properties has the capabilities of supporting single inheritance, polymorphism and dynamic binding.

Below is the example of the typical STACK in V-object:

```

-----
V-object and its class
-----
Package Stack_P is
  type Stack is private;
  procedure Put (x: in Integer; s: in out Stack);
  procedure Get(x: out Integer; s: in out Stack);
private
  type tList is array (1..50) of Integer;
  type Stack is record --> class of V-object
    List : tList;
    Top  : Natural := 0;
  End record;
end Stack_P;
-----
with Stack_P; use Stack_P;
procedure Main is
  S : Stack; --> V-object declared
  E : Integer := 10;
begin
  Put(E,S); Put(20,S);
  Get(E, S); ... ..
end Main;
-----

```

In the above example, type *Stack* (denotes *class*) is concretely represented as a record and interactive routines with explicit effect are accompanied.

For the call to a routine, the object *S* has to be passed to the routine, e.g. **Put (E, S)**. This form may bring confusions: where (module) the routine (*Put*) comes from, and who (object) accepts the routine's effect, for *S* (in the call to *Put*) is similar to *E* (as the routine's argument). But the equivalent in the object-oriented approach, *S.Put(E)*, unambiguously indicates the object *S* through the type of *Stack* and the *Put* from the class *Stack*.

### U-Object: denoting ASM

Compared with the V-object, A U-object is well accords with the object-oriented style. The unit (viewed as an object) provides the interactive routines with implicit effect to manipulate the

attributes of its own, which actually embodies the pristine notion, that is, an object is the encapsulation of the features, including attributes and routines.

Below is the example of the typical stack in U-object:

```

-----
U-object and its class
-----
generic --> class of U-object
package GStack_P is
  procedure Put (x: in Integer);
  function Get return Integer;
private
  type tList is array(1..50) of Integer;
  type Stack is record
    List : tList;
    Top  : Natural := 0;
  end record;
  Imp : Stack; -->Implicit variable
end GStack_P;
-----
package body GStack_P is
  procedure Put (x: in Integer) is
  begin
    Imp.Top := Imp.Top + 1;
    Imp.List(Imp.Top) := x;
  end Put;
  ... ..
end GStack_P;
-----
with GStack_P; --> import
procedure Main is
  package S is new GStack_P;
  --> U-object declared
  E : Integer := 10;
begin
  S.Put(E);
  E := S.Get;
  ... ..
end Main;
-----

```

Similar to the above V-object, a type *Stack* is defined in the private part, but which is never exported out of the unit. This type is only used to declare an implicit variable *Imp*, on which the routines make implicit effect. The generic unit denotes the *class*, which is somewhat rational (U-objects are instantiated from the template).

The call to a routine *S.Put(E)* unambiguously indicates that *S* is the object and the routine *Put* comes from the related module. It's a pity that a generic unit is not an exact type although several instances (packages) come from it. And this kind of design pattern cannot support inheritance, which would be the fatal fault.

In Ada task and protected units can be taken into account as U-objects, i.e., the unit denotes an object. Compared with protected unit, a task unit treats

attributes in special way so that its attributes can only appear in the body. Below are two specific U-object classes:

---

**Task U-object**

---

```

package TStack_P is
  type tList is array (1..50) of Integer;
  task type TStack is
    entry Put (x: in Integer);
    entry Get (x: out Integer);
  -- private          -- Appear in the body
  -- List  : tList;
  -- Top   : Natural := 0;
  end TStack;
end TStack_P;

```

---

**Protected U-object**

---

```

package PStack_P is
  type tList is array (1..50) of Integer;
  protected type PStack is
    procedure Put (x: Integer);
    function  Get return Integer;
  private
    List  : tList;
    Top   : Natural := 0;
  end PStack;
end PStack_P;

```

---

Task and protected units are more ideal class constructs than packages and easier to be made typing, but has the same situation without supporting inheritance.

### A-Object: enclosing features

From the syntactic construction, a class is characterized by attributes and routines. The record in Ada is an essential type construct, which can be used to enclose attributes. Furthermore the tagged record allows not only to build parameterized type with discriminants, but also to derive new subtype with extension. It is possible for us to adapt the record type into an ideal class construct, if the following assumptions can be satisfied:

- Assumption-1: routines are allowed embedding as parts of record, i.e. a record can enclose both attributes and routines.
- Assumption-2: the embedded routines can directly access its fellow attributes. In other words, attributes and routines enclosed by the record are fellowship of each other.

Temporarily, we give the record (which satisfies above-mentioned assumptions) a name: *recording class*. The former Stack is described in recording class as following:

---

**Recording class**

---

```

Package Stack_P is
  type tElmt is new Integer;
  type tList is array (1..50) of tElmt;
  type Stack is record          -- Assumption-1
    procedure Put (x: in tElmt);
    function  Get return tElmt;
    List  : tList;
    Top   : Natural := 0;
  end record;
end Stack_P;
----- Implementation -----
package body Stack_P is
  procedure Put (x: in tElmt) is
  begin          -- Assumption-2
    Top := Top + 1;
    List(Top) := x;
  end Put;
  ...
end Stack_P;
---- Application of a recording class ----
with Stack_P; use Stack_P;
procedure Main is
  S : Stack;      -- > object declared
  E : Integer := 10;
begin
  S.Put(E);      S.Put(20);
  E := S.Get;    ... ..
end Main;

```

---

From the example, a unified recognition can be reached that a recording class encloses features containing attributes and interactive routines. In this way, when the class is made tagged a sub-class with extensions can be derived from. It seems that a pristine notion of class would be found. In fact, there is, right now, a long road for us to walk in order to reach the ideal goal.

For an approach toward the ideal class construct, the mentioned assumptions can be divided into following three problems comparatively easy to solve:

- Access to routines -- it is possible for a record to enclose the access to a routine instead of routine. As a result, any object of the class holds *indirect* routines of its own.
- Access binding -- a routine can be called indirectly through its access, as long as the access (enclosed by objects of the class) is bound with the routine.
- Shared routines -- an implicit access to objects can help all objects of the class share routines if only the access appoints the object and let the access accepts routine's manipulation.

Below is the example concerning the three problems:

---

**Exact recording class**

---

```

package Stack_P is
  type tList is array (1..50) of tElmt;
  procedure Put(x : in tElmt);
  type am_Put is access procedure(x : in tElmt);
  function Get return tElmt;
  type am_Get is access function return tElmt;
  type Stack is record
    Put : am_Put; -- 1) access to routine
    Get : am_Get;
    List : tList;
    Top : Natural := 0;
  end record;
  type aStack is access all Stack;
  function this(aObj : aStack) return aStack;
private
  Current : aStack; --> implicit access variable
end Stack_P;

```

---

**Implementation**

---

```

package body Stack_P is
  function this(aObj : aStack) return aStack is
  begin
    aObj.Put := Put'access; -- 2) access binding
    aObj.Get := Get'access;
    Current := aObj; return Current;
  end this;
  procedure Put(x : in Integer) is
  begin -- 3) body shared through Current
    Current.Top := Current.Top + 1;
    Current.List(Current.Top) := x;
  end Put;
  function Get return tElmt is ... end Get;
end Stack_P;

```

---

In above example, ignoring auxiliary trivial declarations in the specification, one will find that the adapted recording class is not only rational, but also concise in Ada95. Since exploiting many access properties of Ada95, such as access type to subprograms, general access type to record types, we call this kind of objects as A-object. Another reason of so-called A-object is that one *prefers* an access object (declared from *aStack*) to an instance object (declared from *Stack*) in application, which is well stated by the immediate example.

The example is sketched rather than complete, but it shows the essential ideas well:

---

**Application of A-object**

---

```

with Stack_P; use Stack_P;
procedure Main is
  S1 : aliased Stack; --> instance object
  P1, P2 : aStack; --> access object
begin
  P1 := This(S1'unchecked_access);
  P1.Put(10); P1.Put(20); ...
  P2 := This(new Stack);
  P2.Put(10); P2.Put(20);
end Main;

```

---

The adopted class brings forth an embryonic notion of class, which well realizes such important concepts as encapsulation of features (attributes and indirect routines), dynamic binding (access bound with routine), implicit *Current* (referring to the current instance within routines), and son on.

For A-object pattern, the function *This* is such an important facility that Ada95 is made closer to pristine notion of class. During the application of A-objects, *This* performs dynamic binding (e.g., the accesses are bound with the corresponding routines), and appoints the object as the *current* which accepts routine's effect.

Now we have outlined the characteristics about several object patterns in Ada. Without being an original object pattern in Ada, an A-object adequately exploits capabilities in Ada95. Although needing many tedious declarations, A-object pattern has many excellent properties. Compared with both V-object and U-object pattern, the A-object takes advantage of all useful aspects and avoiding shortages between them.

From V-object's side, the class of A-objects is the record type which encloses both attributes and indirect routines. So an object can be used as a common variable for assignment.

From U-object's side, the routines can alternate effect on different objects through the implicit *Current*. So any object can accept routine's effect without hesitating how to pass it as an argument because of implicit effect.

Much benefit will be gained from the unification of both V-object and U-object into the A-object. Especially the A-object pattern is easier to support inheritance, polymorphism and dynamic binding, which will thoroughly be discussed in next section.

## OMEGA: modified classes

The Omega (a uniform Object Model Easy to Gain Ada's ends) is rationally recommended to unify all kinds of object-oriented design patterns in Ada95. The evolving unification is shown as following:

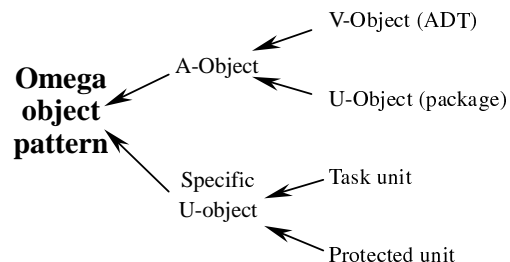


Fig. 1 Evolved unification in Ada95

Keeping the package structure, The Omega introduces a notion of class, with support for inheritance. From the syntactic structure, the class is an applicable cluster, including attributes and routines. Modifiers can precede the class to represent different object design patterns, such as a *general* class (for A-object), a *task* (for concurrency) or a *protected* class (for mutual exclusion)

In Omega the class construct (residing in a package structure) includes an auxiliary part, a class part and a private part, while the corresponding package body is used to implement the routines defined by the class:

---

**Modified class package**

---

```

with ... -- import entities
package CName_P is
-- Auxiliary part
  constant...
  type ...
-- class construct
  [modifier] class CName is [new ...]
    ...routines
    ...attributes
  end CName;

private
...
end CName_P;
```

---

1) **Auxiliary part** is used to declare some auxiliary entities, such as type definitions, constants, which are necessary to serve the class construct.

2) **Class construct** is used to construct the class with specific modifiers: *general* (optional), *protected* and *task* to determine objects of different performance:

- ◆ **general class** is characterized as a record type and must be a descendant of the parent class which hides from such concepts in Ada95, as tagged, controlled and so on.

- ◆ **protected class** is characterized as a protected type unit, with mutual exclusive routines for coordinated sharing of data between tasks.

- ◆ **task class** is characterized as a task type unit, with rendezvous routines (entries) for the synchronized communication between tasks.

3) **Private part** is used to declare import

entities, which are only visible to the package body. Only *general* classes need this part, task and protected classes need not.

In Omega, *general* classes are the mainstream in support for inheritance, polymorphism and dynamic binding. In contrast, *task* and *protected* classes only suit for the objects of special performance. As a result, *task* and *protected* classes are the modified classes, and *general* classes will draw our attention for pristine object-orientation.

### Assessment: different patterns

Ada was a good trailblazer that embodied bold new design ideas in many areas such as exceptions, genericity and concurrency, and on the other hand, Ada was a collector that took in useful advanced ideas such as ADT, ASM and critical monitor (protected object). Nutrient food will benefit one's health, how about eating too much?

Subjectively, regardless of the complexity of diversified objects, Ada allows programmers to choose freely different object design patterns, which would be an effective solution towards concrete problems. Objectively, because long time has passed after Ada83 delivered, Ada95 would be confronted with the conflict between continuity and advancement -- with accommodating itself to continual Ada83 (significant use in the military domain) and advanced technology (spread-wide object-orientation).

Undoubtedly, Ada95 has been provided with the capabilities to support full object-orientation, but the accommodation to Ada83 would demolish the integral simplicity of Ada83 because of many far-fetched complex concepts in Ada95, such as *tagged*, *class-wide*, *controlled* type etc. The fatal problem lies in that Ada lacks a uniform object model that exploits pristine notion of class. And uniform OMEGA is making a significant approach.

The following table shows six aspects of different design patterns. The leftmost column shows the object patterns, where the former four (V-object and U-object) are sophisticated in Ada95. The fifth: A-object, is newly discovered by exploiting Ada95's capabilities. The final column shows the Omega, which would be the desired design pattern.

Object patterns and their properties is illustrated as the following table:

Properties Object pattern	Encapsulation	Attribute's appearance	Routine's effect	Object vs Variable	Inheritance	Notion of class
V-object (ADT)	<b>poor</b>	<b>explicit</b>	<b>explicitly</b>	<b>yes</b>	<b>yes</b>	<b>closer</b>
U-object(ASM)	<b>poor</b>	<b>implicit</b>	<b>implicitly</b>	<b>no</b>	<b>no</b>	<b>far-fetched</b>
Task U-object	<b>poor</b>	<b>implicit</b>	<b>implicitly</b>	<b>yes</b>	<b>no</b>	<b>closer</b>
Protected U-object	<b>good</b>	<b>explicit</b>	<b>implicitly</b>	<b>yes</b>	<b>no</b>	<b>closer</b>
<b>A-object</b>	<b>good</b>	<b>explicit</b>	<b>implicitly</b>	<b>yes</b>	<b>yes</b>	<b>closer</b>
<b>Omega</b>	<b>good</b>	<b>explicit</b>	<b>implicitly</b>	<b>yes</b>	<b>yes</b>	<b>pristine</b>

Caution is wanted, since object patterns in Ada are diversified, some values are somewhat ambiguous, for instance, "poor" does not necessarily imply that a particular object pattern lacks support for the properties. For instance, the *class* of V-object is, in nature, the (tagged) record type which only encloses the attributes, but routines are accompanied outside, so the **encapsulation** of the *class* is "poor". The values "yes" or "no" are unambiguous, for instance, the U-object denoting an ASM is a pure package unit instead of a variable, so its value about "object vs variable" property is "no". With respect to the property "Attribute's appearance", there are two values "implicit" or "explicit"; the former value indicates that the *class* really has attributes, for instance, an ASM of stack, in the specification, only export routines with nothing about attributes, but which are certainly hidden in the body. The same situation occurs in a task unit too.

Up to now, we have examined object-oriented design patterns in Ada95 by discussion of both attributes and interactive routines. OMEGA makes them unified, which will be discussed later.

## C. Uniform Omega

Keeping the package structure, the Omega introduces a notion of class, which not only unifies different design patterns mentioned before, but also achieves very significant simplification for object-orientation in Ada95.

### 1. General classes in Omega

In order to support single inheritance, polymorphism and dynamic binding, Ada95 has to involve many complex concepts, such as *tagged*, *class-wide*, *controlled* type and so on. On the basis of the uniform Omega, general classes successfully hide from complexity in Ada95.

#### Generalized ancestor

In Omega, an upmost root class is needed to provide generalized properties for all descendant classes, such as tagged, controlled properties. Every class (the direct or indirect descendants of the root class) will share those properties, which effectively simplifies redundant and complicated design.

The class of controlled property allows providing a constructor (*Initialize*) and destructor

(*Finalize*), which gives the user additional control over those operations. The constructor will be invoked immediately after a normal default initialization of an object, while the destructor will be invoked immediately before finalization of any of the components of an object.

As to inheritance, only the class of tagged property can be taken as the parent class from which descendant classes are derived with extension, so every class should, in nature, have the ability as the parent class except for task and protected class.

The upmost *Root* class (the simplest form) is described as following:

```



---


Root class in Omega


---


with Ada.Finalization; use Ada.Finalization;
package Root_P is
  class Root is new Controlled
    procedure nil;
  end Root;
private
  procedure Initialize(Obj :in out Root);
end Root_P;


---



```

The upmost *Root* is the class (controlled type) with a nil routine, whose *tagged* and *controlled* properties are a must for inheritance. That **procedure** *nil* does nothing means the null message to be sent to objects, which is useful for the derived class. But here *Root* mainly concerns three important aspects: **class** type (including instance type and access *Root* and its accompanied general **access** type *aRoot*), routine *nil* and object initialization (later

known as object constructor). The same idea will run through all *general* classes.

For simplicity, we will define rules to state important concepts involved by the Omega. And later, it will be found that these rules are key points upon which the Omega pre-processor is based.

A class concerns two types as following rule:

**Rule-1 (class and type)**

A class always responds two types – an instance type – and a general access type (similar to Java’s reference). The class is named by regulation: *Root* for instance type and then *aRoot* for access type.

**Rule-1** allows the user to alternatively declare objects – instance or access to object (*obj-access* for short). The *obj-access* is more convenient than an instance in application, because the access\ can refer to both static objects and dynamic objects.

**Access to routines**

A class is the data structure that can be statically built, so the class can enclose the access to a routine (*rout-access* for short) and the related routine will be implemented out of the class. In this way, every object of the class will hold *rout-accesses* while all objects of the class will share the corresponding routines’ body.

**Rule-2 (enclosed routine)**

That a class encloses a routine means the class holds the *rout-access* and provides the corresponding routine’s body for sharing of objects of the class.

**Object constructor**

Constructing the upmost class *Root* is the significant step to start, and all of general classes must be *Root*’s descendants. Why any general class needs deriving directly or indirectly from the *Root* lies in that an explicit object constructor is eagerly wanted to give the user additional control over *Initialize* operation.

**Rule-3 (object constructor)**

Creating an object needs a constructor, while destroying an object needs a destructor. Procedures *Initialize* and *Finalize* respond to them, respectively

If a general class does not explicitly appoint the parent class, it is meant that the class is the direct descendant of *Root*.

According to **Rule-2** the routine enclosed in a general class refers to an *rout-access* and a routine’s body (implemented in the package body). The object constructor is a controlled procedure that initializes current object, concretely involving the initialization of attributes and *rout-accesses* (bound with the corresponding routines’ body).

An simple example of Stack is described as following, which provides explicit constructor:

```



---




---


Class with an explicit constructor


---


package Stack_P is
  type tList is array (1..50) of Integer;
  class Stack is new Root
    procedure Put(e : in Integer);
    function Get return Integer;
    function Empty return Boolean;
    List : tList;
    Top : Natural;
  end Stack;
private
  procedure Initialize(Obj : in out Stack);
  ...
  procedure Finalize(Obj : in out Stack);
end Stack_P;


---


Class Implementation


---


package body Stack_P is
  ... ..
  procedure Initialize(Obj : in out Stack) is
    sObj : Root; --> Initialize(sObj);
  begin
    Obj := (sObj with Put =>Put’access,
           Get =>Get’access,
           Empty=>Empty’access,
           List =>(1..50=>0),
           Top => 0);
  end Initialize;
  ... ..
end Stack_P;


---



```

As an heir to the parent class (here is *Root*), the derived class inherits all features of the parent class, with extension. The object constructor specifies a value for each of features (*rout-accesses* and attributes) by means of *extension\_aggregate*. Moreover, the constructor in the derived class is inevitably to involve the constructor of the parent class, which indicates that the process of constructor is of transitivity. In some significance, the constructor performs dynamic binding between the object of *rout-accesses* and routines’ body, which will be well understood in discussion about polymorphism.

## Object designator

The call to a routine in pure object-orientation means “sending message to the object”, where the routine is viewed as the message and the object is the target that accepts the message. As a result, when using an object to call a routine (or message-sending), an object designator is needed to definitely specify “which object is accepting the message”.

### Rule-4 (object designator)

Using the designator to select the *rout*-access to call a routine, like  $S \leftarrow \text{Set}(10)$ , the statement at first specifies who is the *current* and then performs call to the routine.

For convenience, we will assume that Ada allows to define a suffix operator on *obj*-access, say “ $\leftarrow$ ”, known as object designator. The designator has two specific functions: the current *obj*-access is assigned to the *Current* through which the called routine can manipulate the attributes of current

object; the value of *obj*-access is returned as the selector to call a routine. Consequently, we can use the designator with object *S*, like  $S \leftarrow \text{Set}(10)$ , to select the *rout*-access for call to the corresponding routine. The Omega pre-processor can make the assumption legal.

All objects of a class will share routines through *Current*, which is achieved by object designator. *Current* must precede the attributes appearing within routine body for manipulation. Preceding *Current* is so definite that it can be ignored, for the Omega pre-processor can concretely determine this situation.

### Rule-5 (preceding current)

For an attribute of the class, the preceding *Current* within routine body can be ignored:

**List(Top) := e;**

The exact equivalence is:

**Current.List(Current.Top) := e;**

Here *List* and *Top* are attributes, but *e* is not an attribute

The following is the adapting version of above example, which provides an explicit designator:

Class with an explicit designator	Class Implementation
<pre> package Stack_P is   type tList is array (1..10) of Integer;   class Stack is new Root     procedure Put(e : in Integer);     function Get return Integer;     function Empty return Boolean;     List : tList;     Top : Natural;   end Stack;   function "&lt;-"(aObj : aStack) return aStack; private   procedure Initialize(Obj : in out Stack);   Current : aStack;   procedure Finalize(Obj : in out Stack); end Stack_P; </pre>	<pre> package body Stack_P is   ... ..   function "&lt;-"(aObj : aStack) return aStack is   begin     (aRoot(aObj))&lt;-nil;     Current := aObj;     return Current;   end "&lt;-";   procedure Put(e : in Integer) is   begin     Current.Top := Current.Top + 1;     List(Top) := e; -- ignore the preceding Current   end Put;   ... .. end Stack_P; </pre>

Similar to the constructor, the designator in the derived class is inevitably to involve the designator of its immediate parent class, which indicates that the process of designator is of transitivity too. The statement  $\text{aRoot}(\text{aObj}) \leftarrow \text{nil}$  only performs parent object designation with sending null message to the *obj*-access (*nil* is defined in class *Root*). The *Current* declared in private part is the mediator between the

current object and routines -- *Current* designates the current object, while routines are shared to manipulate the attributes through *Current*.

In general class of Omega, the object designator is very important for an object to call a routine. **Fig. 2** illustrates the trilogy that the designator works: 1) designation of the object, 2) selection of a *rout*-access, and 3) acceptance of the routine's

effect.

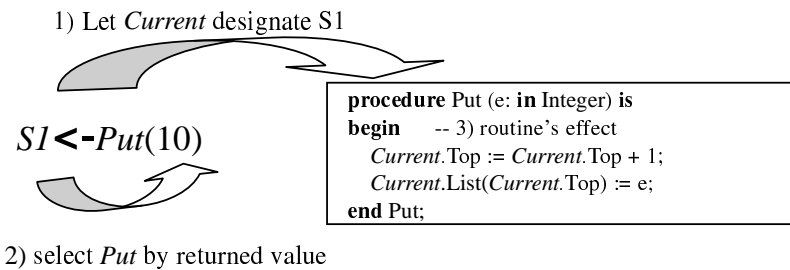


Fig. 2 Functionality of the designator

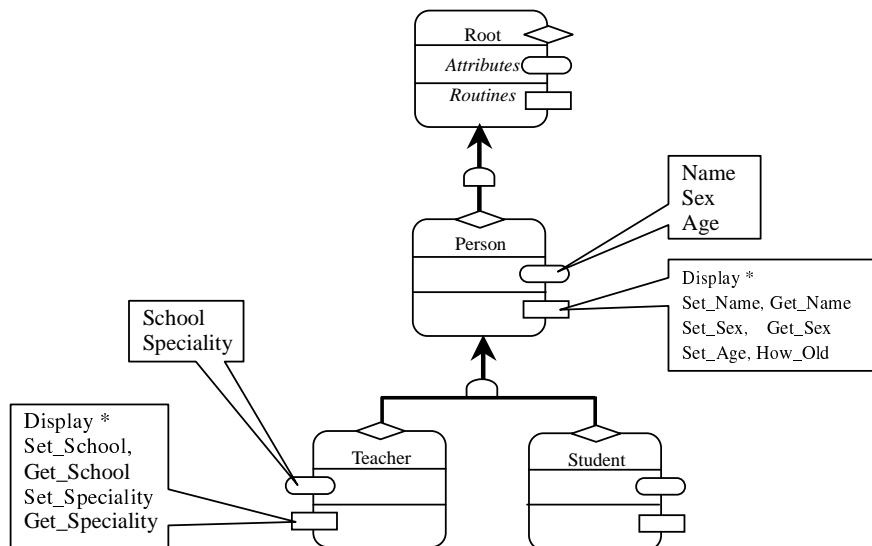


Fig. 3 Inheritance hierarchy tree

With the constructor and designator, a class can be used to declare two kinds of objects: instance and *obj-access*, but an *obj-access* is more convenient in application than an instance, shown as following:

---

**Application of class Stack**

---

```

with Stack_P; use Stack_P;
procedure Main is
    S : aliased Stack; --> Initialize(S1);
    p : aStack;
    aS : aStack ;
begin
    (S1'unchecked_access)<-Put(10);
    (S1'unchecked_access)<-Put(20);
    p := S1'unchecked_access;
    p<-Put(30);    p<-Put(40);
    aS := new Stack; --> implicit call to constructor
    aS<-Put(100); aS<-Put(200);
    if p<-Get = aS<-Get then
        ... ..
    end if;
  
```

---



---

```

end Main;
  
```

---

By using the designator, we can apply objects in the object-oriented style. As *Stack* has tagged and controlled properties (inherited from class *Root*), the binding of the object with its routines automatically occurs at declaration (for an instance) and allocation (for *obj-access*), which is guaranteed by object constructor.

## 2. Derived classes in Omega

In Omega, all general classes have tagged property, which is hidden by class *Root*. That is to say, a new class (sub-class) can conveniently be derived from a parent class. The derived class can only have one direct parent, which means the support for single inheritance. Through derivation an inheritance hierarchy tree can be built. The higher

the class on the “tree” is, the more generalized features the class has, contrarily, the lower the class

## Inheritance

Inheritance allows the derived class to inherit all features held by the parent (up to all of grand parents on the hierarchy tree), to redefine some features and to extend some new features. **Fig. 3** shows a quite simple example of inheritance tree.

To master the basic concepts we will use a simple example. The example is sketched rather than complete, but it shows the essential ideas well.

is, the more specialized features the class has.

With Saying we have a generalized class *Persons* and taking it as the parent class, more specialized classes such as *Teachers* and *Students* can be derived, so we are to describe human abstractions: *Persons*, *Teachers*, *Students* and so on.

For simplicity, the classes only include basic reading and writing operations of attributes. The class inheritance hierarchy may look like this:

Parent class on inheritance tree	Derived class on inheritance tree
<pre> <b>With</b> Root_P; <b>use</b> Root_P; <b>package</b> Persons_P <b>is</b>   <b>type</b> tName   <b>is</b> String(1..10);   <b>type</b> tSex    <b>is</b> (<i>Male</i>, <i>Female</i>);   <b>type</b> tAge    <b>is</b> Integer <b>range</b> 1..130;   <b>class</b> Persons <b>is new</b> Root     <b>procedure</b> Display;     <b>procedure</b> Set_Name(n : <b>in</b> tName);     ...reading and writing of other attributes     Name   : tName;     Sex    : tSex;     Age    : tAge;   <b>end</b> Persons;   ... <b>end</b> Persons_P; </pre>	<pre> <b>with</b> Persons_P; <b>use</b> Persons_P; <b>package</b> Teachers_P <b>is</b>   <b>type</b> tSchool  <b>is</b> String(1..10);   <b>type</b> tSpeciality <b>is</b> (<i>Math</i>, <i>Physics</i>, ...);   <b>class</b> Teachers <b>is new</b> Persons     <b>overridden procedure</b> Display;     <b>procedure</b> Set_School(s : <b>in</b> tSchool);     <b>function</b>  Get_School <b>return</b> tSchool;     <b>procedure</b> Set_Speciality(s : <b>in</b> tSpeciality);     <b>function</b>  Get_Speciality <b>return</b> tSpeciality;     School   : tSchool;     Speciality : tSpeciality;   <b>end</b> Teachers;   ... <b>end</b> Teachers_P; </pre>
<pre> <b>with</b> Text_IO; <b>use</b> Text_IO; <b>package body</b> Persons_P <b>is</b>   <b>function</b> "&lt;-"(aObj : aPersons) <b>return</b> aPersons   <b>is</b>     <b>begin</b>       aRoot(aObj) &lt;- nil;       Current := aObj;       <b>return</b> Current;     <b>end</b> "&lt;-";     <b>procedure</b> Display <b>is</b>     <b>begin</b>       ...display attributes     <b>end</b> Display;     ...   <b>end</b> Persons_P; </pre>	<pre> <b>package body</b> Teachers_P <b>is</b>   <b>package Super</b> <b>renames</b> Persons_P;   <b>function</b> "&lt;-"(aObj : aTeachers) <b>return</b> aTeachers <b>is</b>   <b>begin</b>     (aPersons(aObj)) &lt;- nil;     Current := aObj;     <b>return</b> Current;   <b>end</b> "&lt;-";   <b>procedure</b> Display <b>is</b>   <b>begin</b>     Super.Display; -- &gt; call to parent's Display     ... Display new attributes   <b>end</b> Display;   ... .. <b>end</b> Teachers_P; </pre>

Based on the class *Persons*, a more specialized class *Teachers* can be constructed by inheriting *Persons*. The derived class *Teachers* has all of *Persons*' features, redefines the routine *Display* and adds some new features.

There are two *Display* routines in *Persons* and in *Teachers*, respectively. The “overridden” *Display* in *Teachers* means this class redefines this routine by inheriting the *root*-access of *Display* and providing new version of implementation. The redefinition of a routine involves important object-oriented concept: polymorphism and dynamic binding, which will be discussed below.

## Polymorphism

Through polymorphism, inheritance hierarchies will give us considerable flexibility for the manipulation of objects, while retaining the safety of static typing.

“Polymorphism” means the ability to take several forms. In object-oriented philosophy what may take several forms is an *obj*-access, which will have the ability, at run time, to become attached to objects of different types, all controlled by the static declaration.

For example the routine *Display*: can be sent to any object of *Persons* or *Teachers*. Conceptually, *Display* implies a particular action to be taken. The concept is identical for any object; only the implementation details may be different. So *Display* has different versions for *Persons* and for *Teachers*; let us call them *Display*<sub>PER</sub> and *Display*<sub>TEA</sub>. What happens when a routine with more than one version is applied to a polymorphic entity?

All entities appearing in the preceding cases of polymorphic attachment are of access types: the possible values for *aPoly*, *pObj* and *tObj* are not instances but *obj*-accesses. So the effect of an assignment such as *aPoly := aPersons(tObj)* is

simply to reattach an *obj*-access.

Below is the polymorphic application:

---

**Polymorphic Application**

---

```

with Person_P, Teacher_P; use Person_P, Teacher_P;
procedure Main is
  pObj  : aPersons := new Persons; --> Persons' version;
  tObj  : aTeachers:= new Teachers;-->Teachers' version;
  aPoly : aPersons;
begin
  ...
  aPoly := aPersons(pObj);
  aPoly <-Display; --> taking DisplayPER
-- Polymorphic call
  aPoly := aPersons(tObj);
  aPoly <-Display; --> taking DisplayTEA
end Main;

```

---

When reattached by *aPersons(tObj)*, *aPoly* can only select the features inherited from *Persons*, say *rou*-access *Display*. But *pObj* is bound with the version in *Person* and *tObj* bound with the version in *Teachers*, so *aPoly* can take different forms of routine *Display* according to its current attachment. **Fig. 4** suggests this by showing the *Teachers* object bigger than the *Persons* object. Such differences in object size do not cause any problem if all we are reattaching is an *obj*-access.

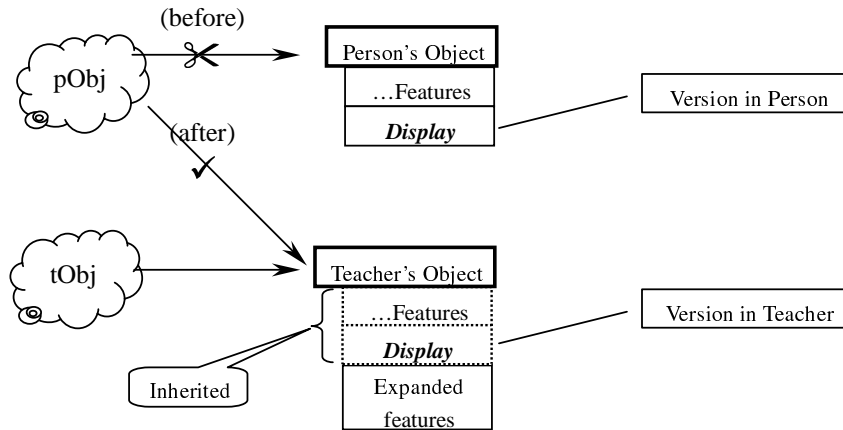


Fig. 4 The overridden routine and polymorphic reattachment

So in spite of the name you should not imagine, when thinking of polymorphism, some run-time transmutation of objects. Once constructed, an object never changes its type. Only an *obj*-access does so by getting reattached to objects of different classes. This also means that polymorphism does not

carry any efficiency penalty; an access reattachment — a very fast operation — costs the same regardless of the objects involved.

## Dynamic binding

Dynamic binding will complement redefinition, polymorphism and static typing to make up the basic tetralogy of inheritance. The rule known as dynamic binding implies that the dynamic form of the object determines which version of the operation to apply.

The “overridden” *Display* means class *Teachers* inherits the *rout-access* of *Display* but provides a new version. Dynamic binding of the overridden routines is realized in the constructor:

### Dynamic binding of overridden routines

```

package body Teachers_P is
  package Super renames Persons_P;
  ... ..
  procedure Initialize(Obj : in out Teachers) is
    sObj : Persons; --> Initialize(sObj);
    -- here bound with the parents' version
  begin
    Obj := (sObj with
      Set_School => Set_School'access,
      Get_School => Get_School'access,
      Set_Speciality => Set_Speciality'access,
      ... .. );
    -- Display is overridden!
    Obj.Display := Display'access;
  end Initialize;
  ... ..
end Teachers_P;

```

Obviously any object of *Persons* and *Teachers* holds the *rout-access* *Display*. The object of *Persons* will be bound with the version implemented in package *Persons\_P* and the object of *Teachers* rebound with the version in *Teachers\_P*.

### Rule-6 (overridden routine)

An overridden routine means the class inherits the *rout-access* but provides a new version of the routine's body. The *rout-access* will be rebound with the new version in the constructor in order to override the parent's version.

As dynamic binding is achieved by the object constructor, the overridden *rout-access* is rebound with the corresponding routine, which is the key step to implement polymorphism.

## 3. Omega pre-processor

The uniform Omega provides a pristine notion of class and is accompanied by a pre-processor. The rules from **Rule-1** to **Rule-6** suit for general classes, which is based on record type with tagged and controlled properties. As the object-oriented mainstream in Ada95, general classes in Omega presents the support for single inheritance, polymorphism and dynamic binding by exploiting many new concepts in Ada95, but effectively hides from the complexity of different object design patterns.

**Rule-7** and **Rule-8** serves task and protected unit pattern, respectively.

**Fig.5** illustrates the sketch of the Omega pre-processor. And Source code in Omega is the input and the mapped code in Ada95 is the output; Ada95 compiler is rear-end of Omega.

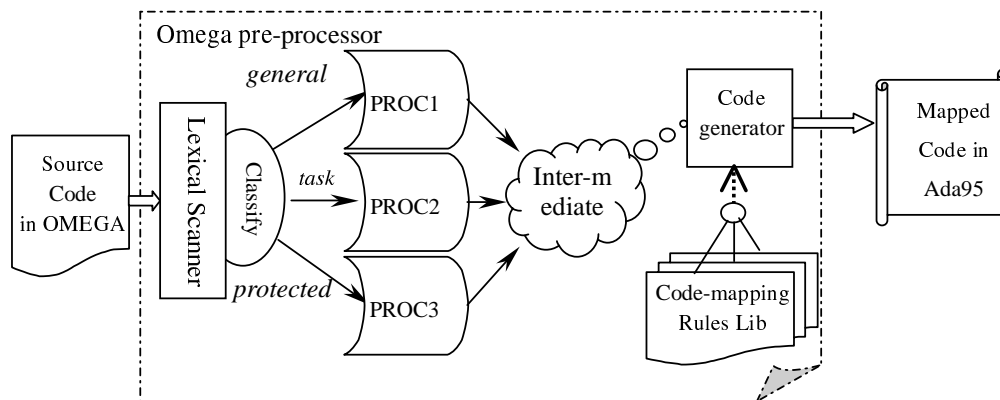


Fig. 5 Sketch of Omega pre-processor

Below table outlines the main rule set and their transformation:

Rules	Description	Transformation
Rule-1 (class & type)	<b>class</b> <i>CName</i> <b>is new</b> Parent ... .. <b>end</b> <i>CName</i> ;	<b>type</b> <i>CName</i> ; --> instance <b>type</b> <i>aCName</i> <b>is access all</b> <i>CName</i> ;--> access <b>type</b> <i>CName</i> <b>is new</b> Parent <b>with record</b> ... enclosed rout-accesses ... enclosed attributes <b>end record</b> ;
Rule-2 (enclosed Routine)	<b>class</b> <i>CName</i> <b>is new</b> Parent <b>procedure</b> Put( <i>x</i> : in tElmt); <b>function</b> Get <b>return</b> tElmt; ... .. <b>end</b> <i>CName</i> ;	<b>procedure</b> Put( <i>x</i> : in tElmt); <b>type</b> <i>am_Put</i> <b>is access procedure</b> ( <i>x</i> : in tElmt); <b>function</b> Get <b>return</b> tElmt; <b>type</b> <i>am_Put</i> <b>is access function return</b> tElmt; <b>type</b> <i>CName</i> <b>is new</b> Parent <b>with record</b> Put : <i>am_Put</i> ; Get : <i>am_Get</i> ; ... .. <b>end record</b> ;
Rule-3 (object constructor)	<b>class</b> <i>CName</i> <b>is new</b> Parent ... .. <b>end</b> <i>CName</i> ;	<b>package</b> <i>CName_P</i> <b>is</b> ... .. <b>private</b> <b>procedure</b> Initialize( <i>Obj</i> : in out <i>CName</i> ); <i>Current</i> : <i>aCName</i> ; <b>procedure</b> Finalize ( <i>Obj</i> : in out <i>CName</i> ); <b>end</b> <i>CName_P</i> ;
Rule-4 (object designator)	<b>function</b> "<-"( <i>aObj</i> : in <i>aCName</i> ) <b>return</b> <i>aCName</i> ;	<b>package</b> <i>CName_P</i> <b>is</b> ... .. <b>function</b> <i>This</i> ( <i>aObj</i> : in <i>aCName</i> ) <b>return</b> <i>aCName</i> ; <b>private</b> ... .. <b>end</b> <i>CName_P</i> ;
Rule-4 (message-sending)	<i>aObj</i> <-Message1 (...); <i>aObj</i> <-Message2; <i>aObj</i> <-Nil;	<i>This</i> ( <i>aObj</i> ).Message1 (...); --> with parameters <i>This</i> ( <i>aObj</i> ).Message2.all; --> without parameter <i>This</i> ( <i>aObj</i> ).Nil.all --> null message-sending
Rule-5 (preceding <i>Current</i> )	--> appear in routine's body Top := Top + 1; List(Top) := e;	--> only if they are attributes <i>Current</i> .Top := <i>Current</i> .Top + 1; <i>Current</i> .List( <i>Current</i> .Top) := e;
Rule-6 (Overridden routine)	<b>overridden procedure</b> Display;	-- rebound in object constructor <i>Obj</i> .Display := Display'access;
Rule-7 (task class)	<b>task class</b> <i>TName</i> <b>is</b> <b>entry</b> Put( <i>x</i> : in tElmt); <b>entry</b> Get( <i>x</i> : out tElmt); <b>private</b> ... attributes... <b>end</b> <i>TName</i> ;	<b>task type</b> <i>TName</i> <b>is</b> <b>entry</b> Put( <i>x</i> : in tElmt); <b>entry</b> Get( <i>x</i> : out tElmt); --...attributes are moved into task body -- as the local declaration <b>end</b> <i>TName</i> ;
Rule-8 (protected class)	<b>protected class</b> <i>PName</i> <b>is</b> <b>procedure</b> Put( <i>x</i> : in tElmt); <b>function</b> Get <b>return</b> tElmt; <b>private</b> ... attributes <b>end</b> <i>PName</i> ;	<b>protected type</b> <i>PName</i> <b>is</b> <b>procedure</b> Put( <i>x</i> : in tElmt); <b>function</b> Get <b>return</b> tElmt; <b>private</b> ... attributes <b>end</b> <i>PName</i> ;

## E. Conclusion

The Omega pattern recommended in this paper successfully makes diversified object patterns in Ada95 unified because a pristine notion of class is introduced. In consequent, there are many excellent properties in the unified object pattern, which are concise and intuitive in representing object-orientation in Ada:

- **Class construct:** the class is characterized by features, so a general class can enclose attributes indirect routines of its own.
- **Concise inheritance:** any general class of extendibility can inherit, redefine and extend features, which is guaranteed by the upmost parent class *Root*
- **Dynamic binding:** any object of a class hold features of its own and is bound with the routines by object constructor with support for dynamic binding..
- **Polymorphism:** the overridden routine of a class makes dynamic form of the object (*obj*-access) dynamically determines which version of the operation to apply.
- **Communication between tasks:** *task* and *protected* classes differ from the *general* classes in semantics and denote specific objects to coordinate message-passing or mutual exclusion between concurrent entities.

Uniform Omega pattern and the accompanied pre-processor not only provide a rational object design pattern, but also make a significant approach, which fully exploits Ada95's capabilities, but

## G. Appendix

### ■ Root Class

Class description in Omega	Mapped code in Ada95
<pre>with Ada.Finalization; use Ada.Finalization; package Root_P is   class Root is new Controlled     procedure Nil;   end Root; end Root_P;</pre>	<pre>with Ada.Finalization; use ada.Finalization; package Root_P is   procedure Nil;   type am_Nil is access procedure;   type Root is new Controlled with record     Nill : am_Nil;   end record;   type aRoot is access all Root;   function This(aObj: in aRoot) return aRoot; private   procedure Initialize(Obj : in out Root);   Current : aRoot;   procedure Finalize(Obj : in out Root); end Root_P;</pre>
<pre>-- **** User can make use of: **** -- -- 1) Instance type: <b>Root</b> -- 2) <i>Obj</i>-access type: <b>aRoot</b> -- 3) Implicit call to <b>object constructor</b> -- 4) Implicit call to <b>object destructor</b> -- 5) Object designator: <b>aObj&lt;-Msg(...)</b></pre>	

effectively hides from its complexity.

Because Omega introduces several good entities that reflect straightforward object-oriented concepts, such as **constructor**, **designator** and **obj-access**, but naming system of pure Ada is somewhat limited.

For a class, say CName, many identifiers are used as accompaniment by default:

- class name: **CName**
- *obj*-access: **aCName**
- package name: **CName\_P**
- designator: **this**
- constructor: **Initialize**
- implicit access: **Current**
- parent class: **Root, aRoot**

A good solution is that we can diagnose the named identifiers in the pre-processor.

In Omega another problem is without support for static overloading of subprogram, for the duplicate name in the same class is forbidden for the *root*-access.

## F. References

- [1] International Standard ISO/IEC 8652:1995(E), **Ada Reference Manual**, Intermetrics, Inc
- [2] Bertrand Meyer, **Object-Oriented Software Construction**, Prentice-Hall International, Inc1997.
- [3] Mary Shaw and David Garlan, **Software Architecture: Perspectives on an emerging discipline**, Prentice-Hall International, Inc1996.
- [4] Xianzhong Liang, Zhenyu Wang **Ada-based Support for Abstraction, Encapsulation and Unit Hierarchy**, Proceedings of Tri-Ada'91 International Conference, San Jose, USA, 1991.10. P116-125

## ■ Derived Classes

Description in Omega	Mapped Code in Ada95	Aiding package
<pre> package Persons_P is   type tName is String(1..10);   type tSex is (Male, Female);   type tAge is Integer range 1..130;   class Persons is     procedure Display;     procedure Set_Name(n : tName);     function Get_Name return tName;     procedure Set_Sex(s : tSex);     function Get_Sex return tSex;     procedure Set_Age(n : tAge);     function How_Old return tAge;     Name : tName;     Sex : tSex;     Age : tAge;   end Persons; end Persons_P; </pre>	<pre> with Root_P,Persons_Aid; use Root_P, Persons_Aid; package Persons_P is   type tName is String(1..10);   type tSex is (Male, Female);   type tAge is Integer range 1..130;   procedure Display;   procedure Set_Name(n : tName);   function Get_Name return tName;   procedure Set_Sex(s : tSex);   function Get_Sex return tSex;   procedure Set_Age(a : tAge);   function How_Old return tAge;   type Persons is new Root with record     Display : am_Display;     Set_Name : am_Set_Name;     Get_Name : am_Get_Name;     Set_Sex : am_Set_Sex;     Get_Sex : am_Get_Sex;     Set_Age : am_Set_Age;     Howt_Old : am_How_Old;     Name : tName;     Sex : tSex;     Age : tAge;   end record;   type aPersons is access all Persons;   function This(aObj : in aPersons)     return aPersons; private   procedure Initialize(Obj : in out Persons);   Current : aPersons;   procedure Finalize(Obj : in out Persons); end Persons_P; </pre>	<pre> -- &gt; Only used for package Persons-P package Persons-Aid is   type am_Display is access     procedure;   type am_Set_Name is access     procedure(n : tName);   type am_Get_Name is access     function return tName;   type am_Set_Sex is access     procedure(s : tSex);   type am_Get_Sex is access     function return tSex;   type am_Set_Age is access     procedure(a : tAge);   type am_How_Old is access     function return tAge; end Persons-Aid; </pre>
<pre> package Teachers_P is   type tSchool is String(1..10);   type tSpeciality is (Math, Physics, ...);   class Teachers is new Persons     overridden procedure Display;     procedure Set_School(s : tSchool);     function Get_School return tSchool;     procedure Set_Speciality(s : tSpeciality);     function Get_Speciality return tSpeciality;     School : tSchool;     Speciality : tSpeciality;   end Teachers; end Teachers_P; </pre>	<pre> with Teachers-Aid; use Teachers-Aid; with Persons_P; use Persons_P; package Teachers_P is   type tSchool is String(1..10);   type tSpeciality is (Math, Physics, ...);   procedure Display; -- overridden   procedure Set_School(s : tSchool);   function Get_School return tSchool;   procedure Set_Speciality(s : tSpeciality);   function Get_Speciality return tSpeciality;   type Teachers is new Persons with record     -- overridden Display;     Set_School : am_Set_School;     Get_School : am_Get_School;     Set_Speciality : am_Set_Speciality;     Get_Speciality : am_Get_Speciality;     School : tSchool;     Speciality : tSpeciality;   end record;   type aTeachers is access all Teachers;   function This(aObj : in aTeachers)     return aTeachers; private   procedure Initialize(Obj : in out Teachers);   Current : aTeachers;   procedure Finalize(Obj : in out Teachers); end Teachers_P; </pre>	<pre> -- &gt; Only used for package Teachers-P package Teachers-Aid is   type am_Set_School is access     procedure(s : tSchool);   type am_Get_School is access     function return tSchool;   type am_Set_Speciality is access     procedure(s : tSpeciality);   type am_Get_Speciality is access     function return tSpeciality; end Teachers-Aid; </pre>
<pre> with Teacher-P; use Teacher-P; procedure Main is   Sheldon : aTeachers := new Teachers; begin   Sheldon &lt;- Set_Name("Sheldon L."); -- inherited   Sheldon &lt;- Set_Age(40); -- inherited   Sheldon &lt;- Set_Speciality(Math);   Sheldon &lt;- Display; -- polymorphism end Main; </pre>	<pre> with Teacher-P; use Teacher-P; procedure Main is   Sheldon : aTeachers := new Teachers; begin   this(Sheldon).Set_Name("Sheldon L."); -- inherited   this(Sheldon).Set_Age(40); -- inherited   this(Sheldon).Set_Speciality(Math);   this(Sheldon).Display.all; -- polymorphism end Main; </pre>	