The Design of ObjectClass, A Seamless Object-Oriented Extension of Pop11

Stephen F. Knight
Intelligent Networked Computing Laboratory
HP Laboratories Bristol
HPL-93-98
November, 1993

object-oriented,
multi-methods,
Pop11

The ObjectClass library adds object-oriented programming into Pop11. In contrast to similar previous work, its main goal is to integrate the procedural and object-oriented paradigms in the most natural and fluent way. The article describes the problems encountered and the solutions adopted in the ObjectClass work with those of other hybridization efforts such as C++ and the Common Lisp Object System CLOS.

To be published in the proceedings of *Expert Systems 93*, December, 1993.

1 Seamless Integration

1.1 Object Oriented Programming

The consensus notion of object-oriented programming is not specifically addressed in this article. It is apparent that the term "object-oriented" is both informal and open to many interpretations by different authors. A useful discussion of this topic may be found in the appendix of [Booch91].

Here, the term object-oriented is taken to mean two things. Firstly, that datatypes are hierarchical; possibly involving complex, tangled hierarchies. Secondly, that some procedures, called generic procedures, are written in separate units, called methods. The method units are distinguished by the types of formal parameters they handle. The generic procedure is a fusion of these methods to form a single entry point.

The terminology adopted is that of the Common Lisp Object System (CLOS) because that system has a sufficiently rich vocabulary to discuss the other approaches. In this terminology, classes are datatypes which can be combined through inheritance; *methods* are individual code units; *generic procedures* are the entry point of a collection of identically named methods; and *slots* are what are commonly called instance variables or fields.

1.2 The Criteria of Seamless Integration

There have been many attempts to extend existing procedural languages to provide object-oriented programming facilities. C has been extended to produce Objective-C and C++ [Ellis90]. Pascal was extended by Apple to produce Object Pascal. Modula-2 has been extended to produce Modula-3 and has been partially redesigned to produce Oberon. Lisp has been extended on many occasions [Bobrow88, Cannon82, Stefik86], notable efforts being Common Loops, Flavors and CLOS. These solutions are of widely differing quality and there appears to be no real consensus concerning how to evaluate the differing approaches.

The ObjectClass work [Knight93] is an attempt to closely integrate object-oriented programming facilities into POP11[Barrett85]. In order to pursue this work it was necessary to develop criteria by which the quality of integration should be judged. These criteria were formed by inspecting other language extensions and formalising our intuitive judgement.

For example, in the FLAVOURS extension to POP11, there is a distinct syntactic difference between applying a procedure and applying a generic procedure (called a method in the jargon of FLAVOURS). This is intuitively an uncomfortable distinction since they are conceptually very similar. To instantiate this similarity, consider the procedure `applist` that iterates a procedure over a list. It cannot not iterate a FLAVOURS method over a

list. Instead a programmer would be obliged to write a new variant of `applist` that works on methods, but not procedures.[1] One might then go on to formalise this intuitive understanding as methods and procedures are unjustifiably distinct.

This introspective process of formalisation led the author to the following criteria or principles for successful integration.

Generalisation When a proposed language construct is a generalisation of another existing construct, the two must be semantically unified. The existing construct should be a straightforward specialisation of the new construct. For example, classes can be seen as a generalisation of record. In the same vein, generic procedures are a generalisation of ordinary procedures. This criterion demands that the existing concepts are specialisations of classes and generic procedures.

Extension When a proposed language construct performs the same job as an existing language construct but in the new context, the existing language construct must be extended to cope with the new task. For example, the selective hiding of instance variables is a issue of name-visibility control. Since POP11 already provides mechanisms for control the visibility of variables it is necessary to arrange that they can be applied within a class definition.

General invention When a proposed language construct has no existing counterpart, it should, if possible, be extended to cover existing contexts and not just the new object-oriented context. So if there is a mechanism for adding pre- and post- actions to methods (e.g. before and after methods) then it should be extended to ordinary procedures.

Stated in such abstract terms, these criteria are probably uncontroversial – indeed, they are merely variants of Occam's Razor. Yet applying them to existing hybrid languages typically generates a great deal of debate.

For example, in C++ the "private" keyword is used to control the visibility of an identifier. In this case, the identifier becomes restricted to descendants of the class. However, the notion of class-privacy cannot be applied to ordinary variables or functions, even though there is a sensible interpretation of this. So the new name-space control construct has not been sufficiently generalised.

However, it is doubtful that this criticism would ever be widely accepted since supporters of the status quo would prepare justifications for the exceptional role of class-privacy – and there may well be valid justifications. Yet these justifications do not prevent the nagging suspicion that, had the new language been designed from scratch, this feature would have been treated differently.

Thus, it is argued that it is necessary to develop criteria first in order to avoid the post-rationalisation of uncomfortable compromises as planned design.

[1] An alternative approach would be to write a syntax word that transformed a method into a procedure. But this would be an admission that the two concepts are tightly related.

2 Integrating with Existing Language Constructs

2.1 Classes as Records, Methods as Procedures

From the view of object-orientation taken above, hybridisation must introduce at least two new programming languages constructs to a procedural language, namely methods and classes. In POP11, and most procedural languages, both of these constructs have strong similarities with existing constructs. Classes can be viewed as records definitions with the ability to include earlier definitions. Methods are somewhat more complex. Each individual method definition is a procedure whilst the invocation of a method performs a selection of the definitions based on the types of the actual parameters. Following the terminology of CLOS, we call the individual definitions "methods" and their entry point a "generic procedure".

The generalisation criterion demands that classes that do not employ their ability to inherit should be identical to record definitions. Similarly, methods that do not employ their selective ability should be ordinary procedures. Furthermore, it suggests that, if possible, classes should always be instantiated as records and generic procedures should always be procedures.

2.2 Extending Procedure Definitions

The extension criterion can be applied to the design of methods. The difference between a method definition and an ordinary procedure definition is that a method definition includes a selection criterion. Whatever syntactic policy is employed to denote the selection criterion should be "reverse engineered" into the existing procedure syntax. This principle can be used to choose between the hybridisation strategy of, say, FLAVOURS and CLOS.

In hybrid solutions such as FLAVOURS or C++, the selection criterion is based on the type of a single, privileged argument. To denote this, methods are written within the lexical scope of a class definition in all of these languages. This approach rejects the idea of integrating methods and procedures together. Taking this line would mean that there would always be a conceptual distinction between methods, which are can only be written in the lexical context of a class definition, and procedures which ignore the context.

```
;;; Illustrating the distinction between method and
;;; procedure definitions.

flavour animal;
    defmethod wakeup;        ;;; method.
    enddefmethod;
endflavour;

define sleep( animal );      ;;; ordinary procedure.
enddefine;
```

The approach taken by CLOS, in contrast, is to require a method to specify optional selection criteria against each formal parameter.

```
; method.
(defmethod wakeup ((a animal)) ...)

; procedure.
(defun sleep (a) ...)
```

This approach removes the dependence of method definitions on class definitions. In addition, it eliminates the need to have a single privileged argument on which the method-dispatch is based. Importantly, this approach permits writing the procedure **sleep** as if it was a generic procedure.

```
; method/procedure.
(defmethod sleep (a) ...)
```

Following the principle of extension, the ObjectClass work pursued the CLOS strategy. Ideally, the **define** syntax would be altered to arrange for ordinary procedures and methods to be defined using the same syntax. The distinction between the two would be reduced to the question of whether or not formal parameters were specialised by type restrictions.

```
;;; method, specialised on class animal.
define wakeup( a:animal ); ... enddefine;

;;; procedure or a catchall method.
define sleep( a ); ... enddefine;
```

For performance reasons, it was appropriate to compromise and use an extension of the **define** syntax. Thus the actual syntax in use is shown below. It is planned to eliminate this compromise at a later date through an enhancement to the garbage collection algorithm that will allow transparent redirection of store. This technique will permit the implementation of procedures to be changed, and therefore the performance profile varied, silently and at the small cost of a single garbage collection.

```
define :method wakeup( a:animal );
    ...
enddefine;
```

2.3 Extending Record Definitions

Class definitions differ from record definitions in two ways. Firstly, they allow the inclusion of other class definitions, a mechanism that is usually referred to as class inheritance. Secondly, the organisation within a class definition is expected to be opaque whereas the organisation within a record is expected to be transparent. This is also presented as the distinction between abstract and concrete data types.

The basic syntax chosen for ObjectClass departs from the **recordclass**, **defclass** syntax. This compromise arises because the existing syntax cannot easily be revised to include the considerable number of extra constructs.

```
define :class person;
    ;;; There are two slots, name and age.
    ;;; Age defaults to 0.
    slot name, age = 0;
enddefine;
```

A good design would unify record syntax and class syntax by making the latter an extension of the former. C++ provides a good model for such a design. In C++, a class definition is simply a record (struct) definition in which fine control is exercised over the visibility the slots (members) and other classes can be included.

This approach can be summarised as treating data type encapsulation as an issue of selective name hiding. By providing convenient name hiding operators it would appear to be possible to unify the two concepts. This is discussed in more detail in the following section.

However, there are several objections to this summary of data encapsulation. A basic objection is that if slots have any distinctive properties, such as their syntactic role, then they violate the encapsulation. For example in FLAVOURS accessing the slot of a class and applying a method can be syntactically distinct. This approach owes much to the influence of SMALLTALK[Goldberg83].

```
flavour person;
    defmethod birthday;
        1 + age -> age;          ;;; age is not a method.
        self <- celebrate;       ;;; celebrate may be.
    enddefmethod;
endflavour;
```

This problem can be solved if slots and methods are unified. The simplest resolution is to make no real distinction between slots and methods. This raises a further issue, however. Slots are always updatable (i.e. a slot access may appear as the target of an

assignment) but methods frequently are not. The consequence of this is that methods should be generalised to be updatable as well as slots.

Thus in ObjectClass, there is no difference between a slot and a method beyond the way in which they are created. Furthermore, both slots and methods may appear in the same syntactic role as the target of an assignment. For example, in the following code fragment, **name** might be a slot, method, or general procedure. Furthermore, it is not easy to write code that would mistakenly reveal any distinction.[2]

```
name( p ) -> x;
y -> name( p );
```

A more difficult problem is that if there are general operators on records, then these operators can expose the data encapsulation. This is exactly the case in POP11. There is no possible immediate solution to this problem – except by abandoning the attempt to integrate classes and records altogether. Procedures such as **appdata**, **length**, **explode** are currently designed to break encapsulation barriers that the programmer would like to erect. The best that can be claimed, in the case of ObjectClass, is that it suffers no worse from these "encapsulation breakers" than do existing records.

In the future, many of these encapsulation breakers will be redesigned into a high-level generic procedure and a low-level encapsulation breaker. For example, **length** is normally thought of as a generic procedure and its companions **datalength** and **datasize** as lower-level versions. The lower-level versions would then be isolated from the casual user by moving them into an appropriate name space such as **$-sys**.

2.4 Encapsulated Data Types

The main technique for encapsulating a data type [3] is the selective hiding of identifiers. In order for it to be a practical technique, it must also be relatively convenient to hide or expose related groups of identifiers and cope with exceptions to the groups.

In C, for example, the ability to hide or expose identifiers is poorly developed – variables are either local to a procedure, local to a file, or global to the application. It is the lack of control over the global namespace which is problematic, forcing programmers to compensate with sophisticated naming conventions – recently illustrated by the X-window system and its associated toolkits. Thus the presence of novel declarations in C++ that control name visibility is expected. This is the role of the **public**, **private**, and **protected** keywords. It is rather unsatisfactory that these declaration modifiers are not available outside of the context of a class. This dissatisfaction can be explained by reference to the principle of extension.

[2] It is possible for the programmer to distinguish between slots, methods and procedure by applying the appropriate recognition procedures. Only in this way can dependencies be introduced.

[3] In this article, encapsulating a data type is to implement a data types so the implementation may be safely revised with no impact on client code. This is sometimes referred to as abstract data typing.

By contrast, POP11 has a highly developed system of name-space control that includes full lexical binding, a fully hierarchical module system (sections), compile-time lexical scoping (lconstants) and file-local variables (top-level lexicals). So it would be unacceptable to provide yet another name-visibility control mechanism. ObjectClass was designed to redeploy the existing mechanisms to achieve data encapsulation.

The design concept was to ensure that the declaration facilities available to ordinary variables applied to class declarations. In line with this, each slot declaration can be individually declared as **vars**, **constant**, **lvars** or **lconstant**. [4]

```
define :class computer;
    slot constant benchmark;     ;;; public
    slot lvars real_speed;       ;;; private
enddefine;
```

This concept needed expanding to cover, for example, the case when all the identifiers needed hiding. The solution adopted was to permit a default declaration, much in the style of the existing record declaration syntax, which could be overridden by specific declarations. So the previous example could have been written as shown below.

```
define :class lvars computer;    ;;; default to private
    slot constant benchmark;     ;;; declared as public
    slot real_speed;             ;;; defaults to private
enddefine;
```

This provides a good example of extending existing concepts fill a new requirement.

A second, less obvious consequence of data encapsulation is that instances of the class cannot be created in the same way as records. [5] The construction of a record exposes the precise number of slots in the record. For example, the constructor **conspair** takes two arguments because it has precisely two slots. This exposure of implementation is unsatisfactory because it obliges a client to rely on information that might change.

It is therefore necessary to provide a means for instance construction that is independent of the implementation of the class. This is accomplished through the use of a nullary constructor that allocates a new default instance. This new instance is modified using the public updaters. In the example below, a **person** can have any number of slots (including none, see following text).

[4] Note that a constant slot is still updatable. This potentially confusing terminology arises from the terminology of the parent language. The attribute of constancy refers to the updatability of the variable bound to the method implementing the slot and not the slot itself.

[5] It should also be noted that there are many occasions on which data encapsulation is not required and is not desirable. In these situations, it is quite natural to construct classes in the same way as records i.e. by using constructor functions.

```
instance person          ;;; create a new person
    name = 'eliza';      ;;; assign 'eliza' to its name
    age  = 42            ;;; and 42 to its age
endinstance -> x;        ;;; put the result in x
```

The most significant decision in ObjectClass was to avoid making the fields of the **instance** syntax required to be slots or even methods. They have the minimal requirement of having values that can be run in update mode.[6] This decision makes construction code independent of the choice of implementation of the updaters, too. Not all object-oriented solutions have adopted this approach. It contrasts with **make-instance** in CLOS where the names of slots are inevitably published. As a consequence, programmers using CLOS are advised to write constructors for the classes that they define to overcome this loss of encapsulation.

Finally, it remains to be observed that there is nothing that distinguishes classes from ordinary records as far as instance construction is concerned. Thus **instance** syntax should also be applicable to ordinary records as well as classes, according to the generalisation criterion discussed earlier. This is not the case in the present implementation of ObjectClass. This compromise is planned for elimination in the near future through continuing to extend the class system to all system and user-defined record types.

3 Correctly Generalising New Constructs

3.1 New Constructs

With the introduction of a new programming style, many new idioms emerge. Programmers legitimately expect these idioms to be supported when, as in the case of object-oriented programming, there is a considerable legacy of programming idioms. Each of the new constructs should be reviewed for generalising over the existing language concepts.

In the case of ObjectClass there are many novel constructs: mixins, singletons, class slots, if-needed methods, class-wrappers, before/after methods, and class adoption. The most difficult cases are reviewed below. In the case of class slots, if-needed methods, class-wrappers, their proper integration is a simple issue dependent partly on the revision of existing syntax and partly on extension of the datatype system. Class adoption is a novel operator, specific to the ObjectClass package. However, it has no meaning for ordinary records since it is an inheritance process and so is not discussed here.

3.2 Mixins and Singletons

Mixins are class definitions that cannot be instantiated and singletons are definitions that can only be instantiated once. The motivation behind both constructs is the same – they document a common intention of the programmer whilst providing superior performance.

[6] In POP11 this means that they need not even be procedures even though this would not be usual.

A first view is that mixins cannot be usefully generalised to record classes. However, there is a language construct called **p_typespec** that acts rather like a record type. Normally this only introduces simple types but, in the context of declaring foreign functions, it can indeed declare record types that cannot be instantiated. Because of the very different origins of these concepts there is no agreement on the proper resolution at present.

Singletons are a relatively simple issue. They can be applied to record types as easily as classes. This is not the case in ObjectClass at present since the best solution depends on enhancing the type-allocator **conskey**.

3.3 Before and After Methods

Before and after methods in ObjectClass broadly follow the CLOS model. However two underlying concepts have been distinguished to facilitate integration, before and after actions and method combination.

Before and after actions are procedural hooks that can be placed on a generic procedure that are invoked before or after the main body of the procedure. This idea should, following the criterion for new constructs, be applied to ordinary procedure definitions. This generalisation is potentially very useful. For example, there are many subtle problems concerned with the tracing of procedures in POP11. Using before and and after actions these problems, which are rooted in the loss of store identity, are solved elegantly.

Method combination is the term used for the process by which the methods associated with a generic procedure are fused to synthesise the underlying procedure that implements it.[7] ObjectClass supports three different method combinations – only the most specific (primary), most specific to least specific (before), least specific to most specific. This concept does not generalise to ordinary procedures except in a trivial sense.

4 Summary

This paper presents the goals behind the ObjectClass package, an object-oriented extension to POP11 which is designed to be well integrated with the existing language. By defining the integration criterion explicitly it is possible to evaluate how effective or ineffective the integration has been. In particular, areas of deviation from the ideal of integration are identified and this lays the foundation for future evolution of the ObjectClass work.

The ObjectClass work is not quite a seamless extension of POP11. The unification of classes with record types and methods with procedures is effective at the syntactic level. However, at the programmatic level there are gaps owing to technical issues involved in the redefinition of system variables.[8] Furthermore, there remain several corner cases where the

[7] The underlying procedure is sometimes called the effective procedure.

[8] Although it is possible to redefine a system variable by cancelling it from the dictionary and redeclaring it, this change does not affect previously compiled code. This means that this process must be repeated for all clients of the modified system variable. In general this is an impractical solution.

unification is imperfect or novel extensions have not been generalised outside the confines of the object-oriented work. Having made the integration agenda explicit, these corner cases are accepted as defects in the current implementation rather than interpreted as reasoned compromises between today and tomorrow's technology.

References

[Barrett85] Barrett, Ramsay, and Sloman. *POP-11: a Practical Language for Artificial Intelligence.* Ellis Horwood, Chicester, 1985.

[Bobrow88] Bobrow, DeMichiel, Gabriel, Keene, Kiczales, and Moon. *Common Lisp Object System Specification.* X3J13 Document 88-02R, June 1988.

[Booch91] G.Booch. *Object-Oriented Design with Applications.* Redwood City, Calif., Benjamin/Cummings, 1991. ISBN 0-8053-0091-0.

[Cannon82] H.I.Cannon. *Flavours:A Non-Hierarchical Approach to Object-Oriented Programming.* Symbolic Inc., 1982.

[Ellis90] Ellis and Stroustrup. *The Annotated C++ Reference Manual.* Reading, Mass., Addison-Wesley 1990. ISBN 0-201-51459-1.

[Goldberg83] Goldberg and Robson. *Smalltalk-80: The Language and its Implementation.* Reading, MA., Addison Wesley, 1983.

[Knight93] S.F.Knight. *REF OBJECTCLASS* On-line documentation of the ObjectClass package, planned for inclusion in POPLOG version 14.5, 1993.

[Stefik86] Mark Stefik and Daniel Bobrow. *Object-Oriented Programming: Themes and Variations.* The AI Magazine, Vol 6 No 4. Winter 1986.