Is There a Way of Combining Array-Processing and Object-Oriented Programming?

Georg Reichard
Institute fur Baustatik
Technical University of Graz,
Lessingstrasse 25 A-8010 Graz, Austria
Tel.: 0316/ 873-6187   Fax: 0316 / 873-6185
Reichard@ifb.tu-graz.ac.at

Abstract

Based on the seven levels of principals of object-oriented programming described by B. Meyer [2], it is shown that with today's APL interpreters it is possible to combine array processing and object-oriented programming on a very high level of abstraction.

Introduction

By reading computer magazines, you can find everywhere articles concerning object-oriented software development. It seems as if suddenly everything is object oriented. You find object-oriented programming languages, databases, user interfaces and even object-oriented hardware.

"Object-oriented programming" is "in", but still developers may have heard about it, but do not know exactly what is actually meant by it.

The term *object-oriented programming* comes from the concept of objects, which form the main pillars of this programming style. A task is divided in its elementary components, the so-called objects, which can be manipulated by the user.

From this point of view "object-oriented thinking" has a tradition lasting over 2000 years and starts with no less a person than Aritoteles, who designed in 350 (B.C.) with his "disposition of knowledge in categories" a class scheme for all objects.

Object-oriented programming is not only using an object-oriented programming language, but means also a completely different way of thinking, which is already used in analysis and design of the theoretical, semantic solution.

For making code re-usable, extensible, and compatible there is a demand for a modular system architecture.

Like "object-oriented", "modular" is another popular catchword, but with many interpretations. Originally it meant developing an application in pieces, mostly called subroutines. The degree of reusability can only increase, if the modules are really autonomous.

But today modularity demands more from the programmer. B. Mayer defines the five criteria for rating modularity, the attributes of modular decomposition, combination, understanding, durability and protection.

This will not be discussed in detail here, but it should be pointed out, that modularity is the key for reusability and the flexibility of future extensions.

Object-Oriented Models

Instead of the two entities of data and procedures there is only one entity in object-oriented systems—the object. It offers both procedures and data as a self-contained unit to the rest of the program.

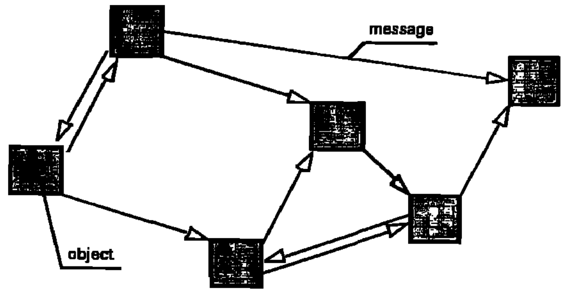

Figure 1: Basic structure of object oriented systems

Furthermore, the data of this object can not be manipulated directly, but only by messages sent to the object. The object itself decides how—and with which one of its own procedures—it will respond to this message.

For evaluating the quality of a system architecture, not only should the simplicity of design be considered, but also the facilities: how changes can be realized. In this respect objects get the drop on functions.

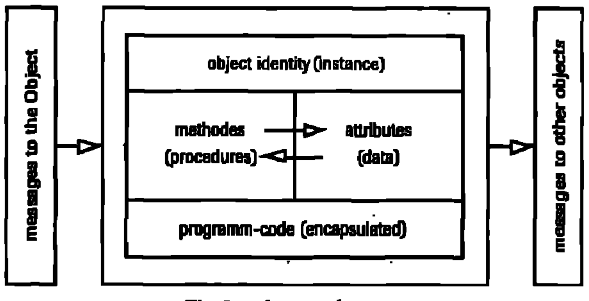
messages to the Object
object identity (instance)
methodes (procedures)
attributes (data)
programm-code (encapsulated)
messages to other objects

Figure 2: Object architecture

Object-Oriented Programming Languages

Looking at classical programming languages, how far they are able to support object-oriented concepts, you find mainly three types:

The first one allows programming with abstract data types. All accesses to data are performed by routines. Without linguistical support this is just a methodical agreement and of course not object-oriented programming yet. Representatives of this type include languages like FORTRAN.

The second category covers languages, which allow definitions of modules though suitable data structures for encapsulating information (for example the language Modula-2), but do not offer heredity.

The last type contains the real object-oriented languages, which include all concepts, like heredity, polymorphism and redefinition.

Object-oriented programming in Fortran?

The oldest "living" programming language, FORTRAN, offers just a primitive support for encapsulation.

A FORTRAN system consists of a main-routine and normally several subroutines. Due to a language facility, which has been part of the language since FORTRAN77, encapsulation can be emulated.

The facility to allow several entries to a procedure was originally likely designed for other purposes, but can also be used in this case.

This extension allows FORTRAN routines to have entry-points other than the one at the head of the routine. These entries can be called from other places as if they were independent routines. All entries of a given routine use the same persistent data, which can be stored between different calls by the SAVE directive.

This technique can be used for developing modules for managing abstract objects. The module hides thereby in a routine with several entries, which represent the methods of the object. Note that the routine is never called by its intrinsic name.

All entries must contain the following form:

```
ENTRY (arguments)
...instructions...
RETURN
```

Note that the FORTRAN routine and all its entries must either be subroutines or functions. So if any operation of the object needs a result, all other entries must be defined as functions, even if the result is irrelevant.

Although this programming technique works, it suffers from strict limitations:

Internal calls of an operation by another one are not allowed, since this would be a recursive call.

There is no support for dynamical creation of objects (instances). Thus it is only possible to implement single objects instead of defining classes with any number of exponents created at run time.

Therefore this programming style should only be used, if there is no other choice than using FORTRAN and applying encapsulation techniques.

Object-oriented programming in APL?

When APL was invented in 1962 by K.E. Iverson nobody thought about object-orientation, but it was a milestone of array processing. The most significant step for APL as an array-processing language was the further development of APL2 with the introduction of mixed arrays and nested arrays combined with suggestive, primitive functions.

Although there are several different APL interpreters controlling the market, just the environment Dyalog APL provided by Dyadic Systems Ltd. seems to permit object-oriented software development.

Dyalog APL/W combines the facilities of array handling with the object-oriented concepts of modern GUIs. The result is a highly powerful and productive tool for developing CUA (Common User Applications) conforming software.

At this time the namespace technology is the most important reason why Dyalog APL/W is so extraordinary and gets the drop on many other programming languages.

I think I will not have to explain namespaces in detail for the circle of the APL community, but I would like to cling to certain features of namespaces, which are:

Namespaces can contain functions, operators, variables, and, of course, also further namespaces. That means that functions and variables for a certain task can be combined in their own namespace.

Within a namespace you can see only the functions and variables of this namespace, so that functions with the same name in other namespaces are invisible.

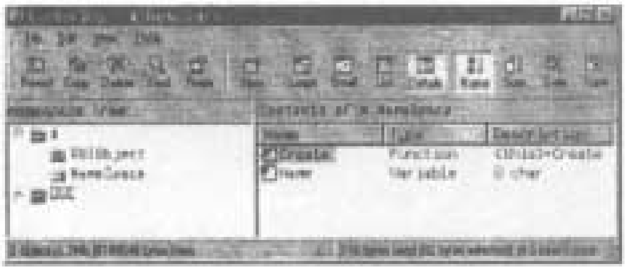

Figure 3: The Dyalog APL/W Workspace Explorer

Originally namespaces might have been implemented to organize and divide the workspace into smaller entities. So a namespace can be copied from one workspace to another without creating naming conflicts.

Furthermore, namespaces permit object-oriented concepts to be introduced to APL.

By using namespaces it is possible to encapsulate procedures on data, which leads to code that is easier to develop, easier to understand, and above all easier to maintain.

Implementation of Object-Oriented Programming Concepts in APL

B. Meyer defines "seven levels of the object-oriented facility", which I have used as criteria for the feasibility of object-oriented APL.

Level 1 – an object based modular structure

"Systems are unitised on basis of their data structure."

The classification of a system into modules can be easily realized by using the namespace technology and hierarchy for structuring the needed data of an application system.

Level 2 – data abstraction

"Objects must be described as implementations of abstract data types."

By structuring classes, especially objects, in separate namespaces abstract data types can easily be defined, due to the fact that namespaces can manage variables and routines in common. Because of the invisibility of these elements from outside of the namespace it is possible to encapsulate the structure. But there is one minor point against the pure object-oriented concept: Values of attributes can be accessed without calling a proper procedure by simply specifying the whole class path, e.g.:

#.Project.MyClass.Variable

It should be pointed out that this facility is also provided in "pure" object-oriented languages like C++, but for reasons of maintainability and reusability it should be avoided.

Level 3 – automated memory allocation

"Unused objects should be deallocated by the subjacent language system without the interference of the programmer"

This level is covered by the APL interpreter, which actually allocates and deallocates memory for all variables used in the system.

Level 4 – classes

"Every non-simple type is a module and each module of higher level is a type."

By interpretation of the definition "non-simple type", which allows the usage of predefined data types (like integer, string, etc.), APL fulfils this level too. All other program elements can build up on these types and can be defined as variables containing mixed and/or nested arrays, implemented as classes in separate namespaces.

Level 5 – heredity

"A class can be defined as an extension or as a restriction of another class."

Heredity techniques can be realized by nesting the different classes in a tree structure of namespaces. Through adding all parent namespaces in the global system variable ⎕*PATH* with

```
⎕PATH←'+'
```

the whole class path is searched for a requested method to run.

By adding functions in a new sub-namespace classes can be extended, by redefining functions as "null-functions" and variables with "null-values" classes can be restricted referring to their parent class.

Level 6 – polymorphism

"Program elements may refer to objects of more than one class, and operations may have different implementations in different classes."

Through the principles of heredity and encapsulation of data in separate namespaces polymorphic software development can easily be realized with Dyalog APL. For the purpose of extensibility of software systems this technique of polymorphism should be implicitly used.

Level 7 – multiple heredity

"It is possible to define classes, which inherit from more than one class, and more than once from one class."

This level can only be realized laboriously in Dyalog APL. By implementing a cover function

```
Result←'Method' Do arguments
```

it is possible to search the path stored in the variable "classpath" for a requested method.

This principle of object-oriented programming is very controversial, since it is not defined how methods which exist in more than one class are inherited. Also the maintenance of these classes often leads to problems. For these reasons this technique is not supported in Java, for example.

Literature:

[1] I. Jacobson, M. Christerson, P. Jonnson, and G. Övergaard: *Object-Oriented Software Engineering – A Case Driven Approach*, Addison Wesley, Wokingham, 1992.

[2] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, Ltd., London, 1988.