Object Oriented PL/I

Alpha Release 0.001

written by Patrick Senti, Dec/94
Email patrick.senti@gmx.net

System Requirements: OS/2 2.1 or OS/2 WARP, IBM PL/I compiler, REXX/2
(back in 1994) (will compile and run under PL/I for MVS but I have not yet converted the OOPLI procedure)

Requirements in 2006: any REXX and PL/I compiler combination

1) Introduction

At the time of writing (1994), my job was in a traditional IBM mainframe shop where most of the (internal) legacy code was written in PL/I. However, I had strongly committed myself to object oriented technology because I felt that OO was the only way that could help developers increase their productivity and lead out of the 'software crisis'.

The problem with OO is that most programmers are very suspicous about it because at first glance it seems very confusing and unnatural, while - if you give it a more serious look - the opposite is true. Another concern is that learning to program in an object oriented way involves learning a new language such as Smalltalk or C++. While this might be interesting enough to do, it also requires a great deal of effort and you might end up not being as productive as with the 'old' language because you simply don't have enough expierence or maybe you just don't feel comfortable about the syntax, or...

C++ is ok for C programmers, but for PL/I programmers it's just too different to be learnt in some weeks' time. This is where my object oriented PL/I comes in. It is a precompiler that introduces a set of new keywords for object oriented code. It produces pure PL/I code that can be compiled with a usual PL/I compiler. While the resulting PL/I code is not really object oriented (as no other language is, by the way - after all the processor requires serialized input!)
it transparently simulatates OO concepts such as data encapsulation, inheritance,
object hierarchy and even a little bit of the 'single type' concept.

Of course, in order for OO PL/I to be an alternative, class libraries would have to be written... anyone...?

Note: The code that OOPLI produces looks ugly and is not meant to be the best possible solution. My goal was to produce something that works. Any comments, suggestions etc are welcome! Planned improvements include to generate more readable PL/I code as well as a reference listing for 'input line' vs 'out put line(s)/statements' to make such programs easier to debug.

2) What is this OO about, anyway?

Testers: what do you think about this section? Suggestions? Better idea do describe
what OO is about? (btw I don't like the animal samples because I hardly ever have to write a zoo animation... but it's simple to write and that's why the sample
included in the package is of this kind)

Most of us are used to writing code in a functional approach and we usually tend to look at problems as if we were the computer to solve it, rather than figure

out
what the problem would look like in a real world. So writing OO code is
just about the opposite of what we're used to.

Is it? Well, yes and no. OO is about writing code that correlates more closely
to the real world. With OO, you don't write routines that belong together, but
you write code that belongs to data. For example, all data related to a
particular customer, for example name, address, telephone number, contact person
etc. will be handled by code that is very tightly coupled with that data. This
means that the data can not be modified directly but rather some routine (called
method in OO terms) must be called in order to access and change the data. In
OO this is called data encapsulation.

Like in the real world (remember, the world you live in, not the computer
bit and bytes!), problems that are to be solved with computers are composed
of different entities or objects. These objects have certain attributes and
some of them have a certain behaviour depending of what you do to them. Take
our customer again, for example. The attributes are his name, address etc.
and his behaviour can be 'order', 'change address', 'call up', 'pay', 'refuse
to pay' etc. In a functional approach, we would have written a program for
each of these actions (behaviours), and each of these programs would have
accessed the customer's attributes (such as name, address, customer number
etc.). So far, there is nothing wrong with this. But consider yourself in
about a year's time when say, the business process for orders changed and
the orders had to be filled with additional data.

Imagine that you have about five to ten different types of customers all of
which
are handled in a seperate module. Imagine you're not the only person who wrote
those order handling systems. Imagine this is not the only thing that changed
in the process. Well, there you go with a bunch of work and a lot of time to
be spent... Does this sound familiar to you? Try object oriented programming!

With OO you would have an object class 'Customer' that can handle messages
(messages are - conceptually speaking - the only thing an object understands;
basically methods are invoked by 'sending' an object a message) such as 'order',
'change address', 'call up', 'pay', 'refuse to pay'. All you would have to do is
just change the 'order' message (and related messages if any) in this object
class
which - in a true OO environment - is defined and coded in one and only one
place.
The latter is not just one of those "would be really nice to have"s but is made
possible by the way OO works!

Yes, you're right. I said that you'd have more than just one type of customer.
But basically they're all customers and so they are all based upon the
'Customer'
object class and that's why all changes are in the Customer object class. Now,
what
does it mean, they are 'based' upon another object class?

In OO this is called inheritance and it means that any object class that is
based on/inherited from another object class can handle the same and maybe even
more messages than its base class (called the parent class). Consider that when
we designed the Customer class we put all the code for the 'order' message that
is common to all types of customers into the order method. Because we did that,
we don't have to change all other customer classes but only the Customer class
itself! Writing a different customer type is easy: just include the Customer's
class definitions, supply additional methods and attributes if you want to -
that's it! You do not have to write forwarding procedures for every function
you want to use in the base class (as you would have to do for a conventional
approach using external modules). It is important to note that you do only
provide *additional* information for your class, everything that's been defined
in the base class is automatically there through inheritance.

While a scenario similiar to this could be achieved by writing an external
module that handles all the common code for customers, writing code for a
different
customer type would not be very likely to use this module and it would be rather

inconvenient to write the calling mechanism for this module (several parameters that tell the module what exactly you want to do: order, pay etc.).

This is a rather pratical look at OO, so I'd strongly recommend you attend a course and/or (better and!) read a book, such as "Concepts of Object-oriented Programming" by David N. Smith (ISBN 0-07-059177-6) or "An Introduction to Object-Oriented Programming" by Timothy Budd (ISBN 0-201-54709-0). The first one is one of the best for OO newcomers, the second is more detailed.

--> Conclusion <--

Basically, OO technology provides the *mechanism* for code reuse (beside other things such as data encapsulation and polymorphism) and it encourages its use because it's simple once you understand the concept. This is true if you look at it from the programmer's point of view but there are other advantages both in the analysis, design and maintenance of software.

3) OOPLI Syntax

3.1 Object class declaration

Declaring an object class is very much like declaring a structure in PLI, but it includes the declaration of methods as well:

```
DCL 1 <className>[:module] OBJECT[(<baseClass>)],
     2 <memberName> <attributes>[,..];
```

where <memberName> is an attribute or a method for object class. If it's an attribute you can use any of the PL/I attributes, if it's a method, you have to use the PROC attribute. You may or may not provide the argument list for a PROC attribute. OOPLI does not check or match the arguments within the class declaration and the actual procedure statement.

If the class is external to the module where it is defined in, you must use the :module extension where <module> is the name of the external module name.

I suggest you place the class declarations in an include file, so others can use your class. If you do, make sure you have the :module extension set correctly. Otherwise the class cannot be resolved correctly.

Note: the module name and the class name must not be the same.

3.2 Object method coding

After you have declared the object class along with its attributes and methods you must code the methods. You do this by writing a procedure just as you would in a conventional PL/I program except that the name of the procedure is of the form className.methodName.

The complete syntax is:

```
className.methodName: PROC[(parameters)] [RETURNS(PTR)];

 (your code)
END;
```

Within a method you have access to the special variable THIS. This is the same as the self variable in Smalltalk or the this variable in C++. Use THIS in the form THIS.membername to access an attribute or to call a method. You MUST use THIS to access a membername.

Use the special variable PARENT to call a method of the parent class from within a method. Syntax is CALL parent.method(). OOPLI will generate an error if the current class does not have a parent class.

Note that currently you can only write functions that return a pointer. But, of course, the pointer can point to anything you like! Polimorphysm as it is implemented in C++ is not possible, but I consider to implement it.

3.3 Object variable declaration

Before you can use an object, you must declare a variable that is an instance of the object class. You do this by declaring a variable as you would in a conventional PL/I program except that the attribute is the name of the object class rather than any other PL/I attribute. So the syntax is:

```
DCL varName className;
```

3.4 Calling a method

After you have declared an object variable (see 3.3), you can use the methods and attributes defined for that class. Do this by writing

```
CALL varName.methodName[(prmList)];
```

or

```
ptr = varName.methodName([prmList]);
```

The method call will be dynamically resolved during run time. If the method is not found the program will report an error and abort.

Note that the attribute of any parameter in <prmList> must excatly match the attributes of the method's procedure declaration. The compiler will not report any missmatches since this information is lost in the 'pure' PL/I code. If you pass a CHAR array (a string), declare the arguments in the method procedure as CHAR(*); the same goes for VAR CHAR strings. I have not tested arrays, so I don't know whether that will work...

Note that declaring a variable will always allocate memory for it. You cannot defer allocation nor can you dynamically remove an instance from memory. This limitation will be removed in the future.

It is possible to assign one instance variable to another, so that the target variable will become the same instance as the source variable (they need not be of the same class). The instance of the target variable will be lost.

3.5 Using an attribute

Although direct access to an object's attribute is not allowed in a true OO environment you can do this in OO PL/I. Currently there is no way to restrict the access to attributes, but support for this will be provided in a future release (PUBLIC and PRIVATE sections).

```
value = varName.attributeName;
```

To use an attribute from within a method, you must use the special variable THIS. So the syntax is value = THIS.attributeName. OOPLI will not discover any attributes not accessed by THIS.

3.6 Using external modules

If you pass the OOPLI compiler a file that does not contain an OPTIONS(MAIN) attribute, OOPLI assumes that this is an external object module an generates the necessary bindings. Make sure that the name of the ENTRY for the module matches the name of the :module extension in the object class declaration. If they don't match, you will get unresolved external entries.

4) Running the OO PLI (pre)compiler

To run OOPLI to produce the PL/I code out of your OO PL/I modules, at an OS/2 command prompt type

```
OOPLI infile outfile
```

where infile is your OO PL/I source file
outfile is the 'pure PL/I' file for the PL/I compiler

I suggest naming OO PL/I source files *.OPL and the <outfiles> *.PLI. The samples are compiled using the OOPLI.MAK file which is based on suffixes.

5) The sample

The sample is a simple 'animal-type' demonstration of OO PL/I. The following are the source files:

CANIMAL.OPL	OO source code for the Animal class
ANIMAL.INC	the Animal class declaration
CDOG.OPL	OO source code for the Dog class
DOG.INC	the Dog class declaration
TESTOO.OPL	the main program
TESTOO.MAK	the make file for the OS/2 2.1 Toolkit NMAKE utility
TESTOO.EXE	the compiled version of the above. Requires IBM PL/I v1.1 installed (for the runtime libraries)

6) Known restrictions, bugs etc

These are restrictins in the current version. These might or might not be removed depending on their importance. See section 7 for planned improvements.

- you cannot declare methods to return anything other than a pointer
- objects references within classes are not supported
- the name of each method plus its classname must not exceed the PL/I compiler's limit for internal symbols.
- syntax check is rather poor and will not discover typing errors, but the PL/I compiler will. However the PL/I code is difficult to relate to the OO PL/I code...
- the OS/2 PL/I for Professionals compiler (v1.10) generates a warning message / rc 4 for classes that do not have any data attributes. The program runs fine, though.
- for strings, the INITAL attribute in a class declaration is translated to uppercase

7) Planned improvements

- remove the limitation in length for class and member names by restructuring the output PL/I code (nested procedures for each class)
- allow functions to return anything PL/I allows
- implement poloymorphism in the way it's implemented in C++
- provide operator overriding
- implement the concept of public/private interfaces
- generate more readable code as well as a reference listing for OO PLI code vs actual PL/I code/statements
- provide condition handling
- allow to defer instance allocation/deallocation
- allow the programmer to override constructors/destructors (constructors are there already but you must not override them)
- provide support for accessing SOM/DSOM objects (is it possible?)

There's much more and I'll extend the list as ideas come to my mind... Anything
YOU would like to have implemented?

8) Support

I have written the OOPLI procedure in my spare time at home and it is not related
to any particular project. Therefore support and enhancements will be provided as
time is available.

9) Questionaire

Please return to PSEN@CHVM1

1. What do you think about OO PL/I? Do you think it should be extended?

2. Anything basic missing in this doc? (I'm sure there is...) Mistakes?

3. Comments? Suggestions? Flames?