A Web-capable Persistent Programming Environment

C. J. Harrison and O. M. Sallabi

Department of Computation
UMIST
Manchester, M60 1QD, U.K.

Chris.Harrison@co.umist.ac.uk , Omar.sallabi@stud.umist.ac.uk

Abstract

This paper describes a persistent programming environment designed specifically for use in a teaching role. The environment supports an interpreted idealised object-oriented programming language and includes pre-built classes that directly support the teaching of fundamental and general concepts that underpin the object-oriented paradigm. Users interact with the environment via a structure-editor in order to ensure that only syntactically correct programs are written, and semantic checking is provided incrementally. In order that the environment is web-capable, it provides users with the ability to invoke external applications, e.g. a web-browser, and it also includes an in-built FTP class. The basis for the environment's implementation lies in its manipulation of the underlying abstract syntax representation of an application. Such representations may be split and merged and are held in a persistent store. Applications within the environment can then be unparsed into the idealised programming language supported by the environment, and also into the commonly used languages C++ and Java.

Keywords: *Persistent programming environment, object-oriented, web-capable.*

1. Introduction

Recent studies[1], [2] indicate that a wide variety of programming languages and language paradigms have been used to teach initial programming skills. Where "off-the-self" programming languages have proved unsuitable for use in a teaching role, it is evident that other languages and language implementations have been developed specifically for use in such a role. Early purpose-built "teaching languages" are exemplified by the Pascal language[3], and especially its UCSD implementation[4], by developments of the Pascal language, e.g. the Turing Language[5], the original (1972) SmallTalk language[6], and also by more recent languages, e.g. the F-language[7], and the object-oriented language Blue[8] and its implementation [9].

However, fundamental limitations of such systems can be identified, for example, earlier systems like UCSD Pascal were essentially "stand-alone" language implementations, i.e. a compiler and run-time support, and were developed before the advent of the Web. More recent systems like F, and Blue are essentially vehicles that support a single "teaching" language rather than enabling users to compare and contrast implementations generated automatically from the "teaching" language representation. Finally, even the most sophisticated systems for teaching programming that are currently available exploit conventional techniques for manipulation and storage which preclude the kind of support we believe it is vital to provide, i.e. immediate feedback to users on semantic errors.

This paper describes an idealised object-oriented programming language called IOPL (Initial Object-oriented Programming Language) which has a combination of Pascal-[4] and SmallTalk-[14] like syntax. In addition, we describe key features of the IOPL environment which build on experience gained in the development of earlier persistent systems, in particular, a persistent programming system[10] used for several years to support undergraduate teaching.

The IOPL environment manipulates the abstract syntax representation of an application, and all objects computed by an application are also similarly stored in a form that enables them to be reloaded after the execution of the application which created those objects has ceased. The IOPL environment provides pre-built classes that implement the simple types *Integer*, *Boolean* and *String*, and the type constructors *Array* and *Record*, together with other types for developing interactive applications, e.g. a "window" type and a "button" type.

In order that the IOPL environment is "web-capable" it provides users with the ability to invoke external applications, e.g. a web-browser, and most importantly, it also includes an in-built FTP class that implements an FTP client. Stored representations of user-defined classes and objects may be split and merged and their persistent representation may be FTP'd between users

via shared servers. Such applications can then be unparsed into the idealised programming language supported by the environment, and also into two other commonly used languages, i.e. C++[15] and Java[16]. This support for un-parsing into alternative languages enables users to compare and contrast applications developed in the idealised language with automatically generated implementations in C++ and Java.

2. THE INITIAL OBJECT-ORIENTED PROGRAMMING LANGUAGE (IOPL)

IOPL adheres to several fundamental principles that ensure it is suitable for use in a teaching role. These principles are enumerated below.

- Δ The language should exhibit "pure" features drawn from a given language paradigm.
- Δ The language should have an easy to read and consistent syntax.
- Δ The language should only include constructs that have semantic value.
- Δ The language should have a well-defined and minimally complex execution model.
- Δ The language should be compact and avoid redundant constructs.
- Δ The language should have an associated development environment designed for users whose level of skill changes significantly over time.

The following subsections give a brief overview of IOPL. Section 3 describes how the IOPL environment provides support for developing applications, in particular how such representations are stored and how they can be transferred between users. Section 4 describes how users interact with the environment.

2.1 SYNTAX AND INFORMAL SEMANTICS

IOPL is a "pure" object-oriented language, i.e. all the object-oriented concepts are presented in the language in a clean and consistent way. Most importantly, it supports strong typing, single inheritance, generic classes, persistence, and it is made available via a powerful interactive development environment.

Classes and methods

All applications in IOPL is implemented as classes. An IOPL class describes the implementation of a set of objects that represent the same kind of component. There are four kinds of class in IOPL, i.e. system classes, enumerations, user-defined and generic classes.

A class structure in IOPL contains two parts:-

- A data definition component that defines the class instance variables.
- A data manipulation component that defines the methods within a class. A method describes how an object will perform one of its operations.

The following simple example illustrates a user-defined class *person* which has two instance variables, *person-name* and *person-age*, and two methods, *initialize* and *print()* .

```
CLASS person  SUBCLASS OF Object

Instance variables:
      person_Name   : string
      person_age    : Integer

METHOD initialize (n: String , g: Integer) RETURN Void
BEGIN
        person_name = n;
        person_age  =  g;
END.

METHOD print()  RETURN Void
BEGIN
       PRINT  " Person Name : " , person_name;
       PRINT  " Person Age  : " ,   person_age;
END.
```

An EBNF definition of the general structure of a class, a method, a declaration, a type and an expression in IOPL is shown below in Figure (1):

```
(Class)         C::= Class ClassId Subclass Of ClassId
                       Instance variables  D1..Dk
                       M1.....Mn

(Method)        M::= Method  MethodId (D1..Dk) Return T
                        Begin
                           E1.. En
                        End.

(Declaration)  D::=  Id : T

(Type)          T::= Self | Void | ClassId

(Expression)   E::= Id := E | E.MethodId(E1..En)
                     | if E then E else E  | New (classId)
                     | case E of E : E | for E to E do E|
                     | Self | Id | nil | print E |
```

Figure 1: General Structure of IOPL Syntax

2.2 INHERITANCE

Inheritance enables an instance of a child class (or subclass) to access both data and behavior (or methods) associated with a parent class (or super class) [18].

IOPL defines a simple, straightforward inheritance mechanism, i.e. single inheritance is supported. Syntactically, naming the parent class in the class header specifies inheritance. Consider, for example, the classes *employee* and *person* defined below:-

```
Class Employee Subclass Of  Person
Instance Variables :
                Salary : Integer;
Method initialize(s:Integer,n:string,a:Integer) Return void
Begin
           Super. initialize(n,a);
           Salary := s;
End;
Method cal_salary(s:integer) Return Integer
Var
         Net_pay : Integer;
         Tax        : Integer;
Begin
         Tax  := salary *15/100;
         Net_pay :=Salary – Tax;
         Return Net_pay;
End;
Method print() Return Void
Begin
         Super.print();
         Print " Salary : ", Salary
End;
```

In this example, the class *Employee* inherits from *Person*. The effect of this relationship is that all instance variables and methods are inherited by the subclass. Note the use of the keyword *Super* to indicate the parent class.

2.3 GENERICS

Generics provide a way of parameterizing a class or a method. In IOPL, generics are denoted by the type <*T*>, i.e. in IOPL, generics comprise a variable defined as a type parameter. This parameter can then be used within the class definition just as if it were a type. Consider, for example, a stack declared in the following fashion:-

```
Class Stack<T> subclass of Object
Instance variables:
            Stack : Array<T>;
            Top   : Integer;
Method initialize (size:integer) Return Void;
Begin
         Stack.create(size);
          Top := 0;
End;

Method push(val:T) Return Void
Begin
         Stack.at_put(top,val);
          Top:=Top+1;
End;
Method pop() Return T
Var
    Result : T;
Begin
    Result := stack.at(top);
    Top:= Top-1;
    Return  Result ;
End;
```

In this example, *T* is being used as a type parameter. To create an instance of the class *stack*, the user must provide a type value for the parameter *T*. For example:-

```
st :=  new STACK <Integer>
```

will construct a new stack object *st* of type *integer*. The stack size can be defined by sending the message *st.create(size)* to the new stack object *st.*

2.4 THE IOPL TYPE SYSTEM

Type systems ensure readability, reliability and efficiency of software. For object-oriented languages, typing is an especially challenging problem because of inheritance, assignment and late binding.

In IOPL environment, the structure editor provides incremental type checking during the construction of a class definition. This form of type checking enables code generation and optimization to be undertaken during class construction.

In this section we will focus in the usage of the IOPL type system, in particular, how type checking is performed.

2.4.1 TYPE ANNOTATIONS

A class is "typed" when it contains type annotations. In IOPL, types are either system classes, e.g. Integer, String, Boolean and Window, or user-defined classes e.g. point, circle, etc, or enumeration classes, e.g. colors, days, etc.

Types can be attached in three places in classes:

- At instance variables.
- At formal arguments.
- At method results.

When developing a class, the structure editor provides a menu of permissible types that can be assigned to each one of the above annotations. This technique, i.e. a menu of permissible types and incremental type checking, ensures that no syntax errors occur during construction or subsequent execution. More generally, this technique ensures that users interact in a semantically meaningful manner at all times.

2.4.2 ASSIGNMENT

The assignment of an expression is similarly performed via the structure editor to ensure type correctness. For example the expression:

Variable := *Expression*

requires the user to chose a variable such that the expression is of a valid type, i.e. the expression must be either:-

- A variable - the structure editor provides a list of variables of the same type.
- A constant - the structured editor provides a mask to write the constant (there are three masks for primitive types Integer, String, Boolean).
- An arithmetic expression - only a combination of integer variables, integer constants and arithmetic operands are allowed).
- Nil - assign universal type nil if the variable doesn't reference any other object.
- New - assign to the variable to a newly created object.
- Message - assign to the variable the returned object from a message.

2.4.3 MESSAGES

Objects receive messages to invoke one of their class methods or to invoke an inherited method from their super-class. To avoid the common error that in a typical system usually results in a "**Message Not Understood**" response, the IOPL environment provides a list of all valid methods an object can receive when executing. Similarly, the system deals with arguments in the same way, i.e. in this case the message should be error free in terms of the message selector and arguments. In the example below, a new object of type employee is assigned to the variable *emp* of type employee.

emp := new (*Employee*)

The object *emp* can then accept the following messages (given the definition in *section **2.2***) as shown in Figure(2).

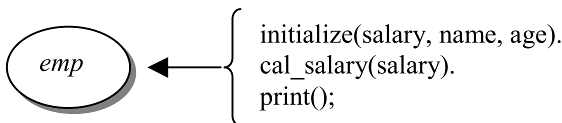

Figure 2 - Ensuring Valid Messages.

Each message appears with its arguments, each argument represented by "masked" box and the system accepts only data of the same type as shown in the example below.

emp.initialize(1500,"John",33)
emp.cal_salary(emp.salary);
emp.print();

Another benefit gained from the enforcement of a rigorous typing system is code optimization, i.e. the IOPL interpreter executes the intermediate code directly without any need to perform type checking during execution.

3. SUPPORT FOR PERSISTENCE AND INTERCOMMUNICATION

Persistence has been defined as supporting data values for their full life-times however brief or long those lifetimes may be. Persistence requires that data values are treated uniformally for all aspects of system services independently of their longevity, size or type[17].

The IOPL persistent store builds on experience gained from the design of persistent stores for a number of different applications, in particular the Modular Persistent Store[11], the Abstract Data Store[13] and the POOL persistent store [12].

3.1 PERSISTENT STORE ARCHITECTURE

The store structure is composed of three parts as shown in Figure (3). Firstly, the store header contains the store description (i.e. store identifier, current size of store, block size, and starting block numbers of other store sections). Secondly, the Persistent Object Table maintains a list of all object names and their object identifiers (OID's) in the store. Finally, the Data section holds the values of an object. Each object occupies a number of contiguous bytes, which hold the object's information and data. There are two kinds of objects held in the store, i.e. simple objects such as Integer, Boolean, and string, and structured objects such records and arrays.

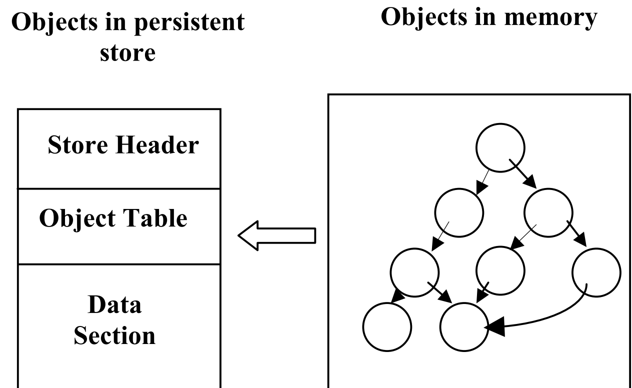

Figure 3 - Representation of objects in the memory and persistent store.

3.2 STORE OPERATIONS

There is four store operations: -

- **MOVE TO STORE -** All created objects are stored in a heap, therefore, to keep the objects persistent we need to move the heap contents to the persistent store by creating a new store or overwriting an existing store.
- **RESTORE FROM STORE -** This operation opens a store with a given name and loads all entries from the store to the system heap.
- **DIVIDE STORE -** This operation divides the persistent store into two stores. This requires choosing the required objects in the first store by marking them, and the rest of the objects and then transferred to the second store. The system then compacts all objects in the first store. Figures (4a) & (4b) below show the effect of the divide operation.

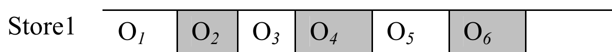

Figure 4(a) – The store before division

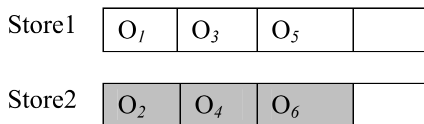

Figure 4(b) - The resulting stores 1,2 after division.

- **MERGE STORE -** This operation merges two stores in one single store and involves choosing the required stores for concatenation. All objects in the second store are appended to the end of first store. Referencing will subsequently be made to all objects in the new store.

3.3 THE FTP CLASS

The FTP class is a pre-built class that enables the transfer of values between users via an FTP client. Users send a message to the method *connect* with the parameters *host name*, *user name* and *password* to make a connection, and this connection enables the export & import of other classes or persistent stores.

The user can send the following messages to the FTP client:-

- Create new directory
- Change directory
- Delete directory
- Select files
- Delete files
- Export & Import selected files

3.4 THE UN-PARSER CLASS

In the IOPL environment, classes are stored in the persistent store as *objects* of type *class*. An internal representation of a class object contains the class name and its super class, instance variables are stored in an instance variable vector, methods are stored in a methods vector (which contains method name, method type, method temporary variables), and code is stored in a code vector. The code vector holds all the commands and expressions as a tree structure in which each node contains the operation code and the parameters needed to perform execution. Figure(5) below shows the general structure of a class in the IOPL implementation.

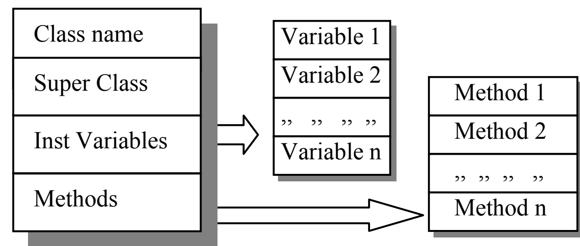

Figure 5 - General structure of an IOPL Class

The un-parser class has three main methods, which generate the class's code in IOPL, C++ and Java respectively. For example, suppose we have a class with a method called print () and this method is used to print the string "Hello world". This class can be "switched" between the three un-parsing methods to show its realisation in three different languages. The resulting implementations are shown in Figure (6) below:

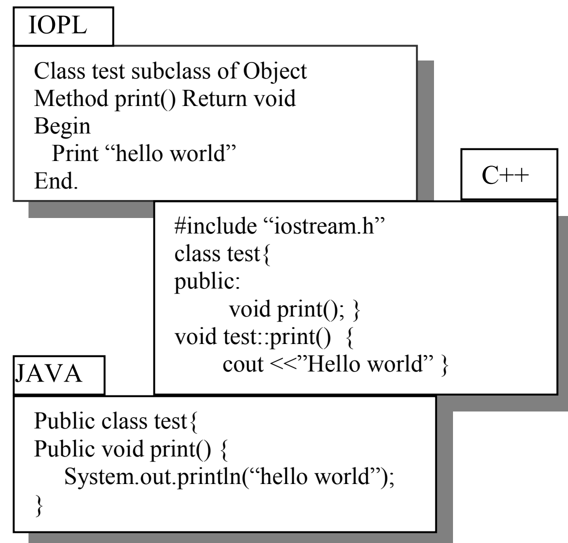

Figure 6 - Three implementations of a single class.

The non-IOPL implementations generated by the un-parser class can be executed by compiling the unparsed representation using an external compiler, i.e. by sending the message *compile-to ()* to the un-parsed code.

4 USER INTERACTION

The main aim of the IOPL environment is to encourage users to think exclusively in terms of objects and classes and to provide an environment that embodies this important separation of related concerns.

Figure (7) illustrates the desktop window presented to users when they log on to the system. Before entry to the system, users enter their user name and password (which are allocated by the tutor or system administrator), and each user has a workspace that maintains their classes via the persistent store.

The system's main window has three components: a pop-up menu and toolbar at the top, class information (at the left) shows the class tree and the active class instance variables and methods, and the active object and the heap contents (at the right). These main desktop components as they appear to a user are shown in Figure(7).

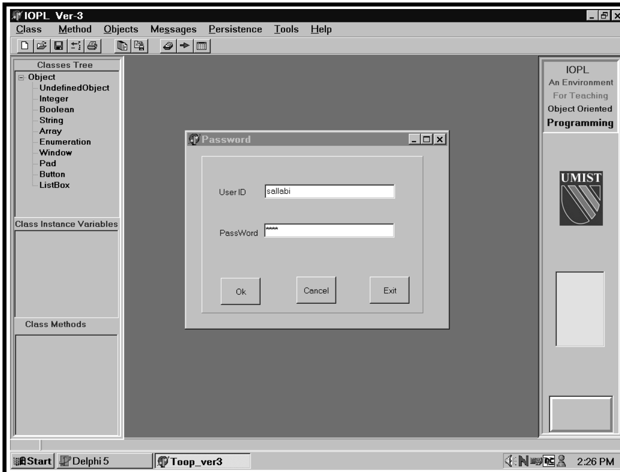

Figure 7 - The Desktop

4.1 CREATING A NEW CLASS

Classes are constructed via an interactive structure editor that ensures syntactic correctness, and the editor also ensures that users choose a variable or a method of a valid type when defining a class. The structure editor provides templates in order that users can "fill-in" incomplete entries during software construction. The system maintains the abstract syntax tree representation of a users program in the persistent store and executes it directly without the need to recompiling each time a change is made.

To create a new class, users select the New Class operation from the menu. A dialog then appears which enables users to enter a class name, a super-class, a class type, and the instance variables for the new class. Figure(8) shows the create class dialog.

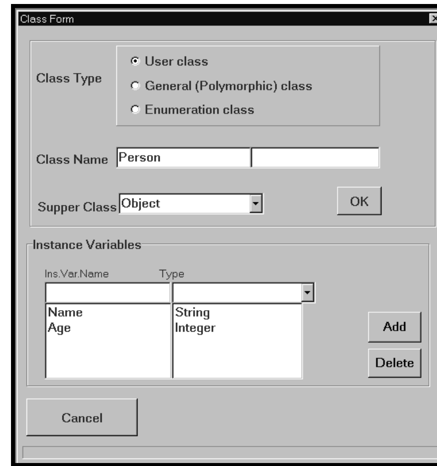

Figure 8 - The Create Class Dialog

4.2 CREATING A NEW METHOD

A method is made up of a message pattern and a sequence of expressions. The method structure contains:

- **METHOD HEADER** -The method header contains the method name, arguments and the return type.
- **METHOD VARIABLES -** A methods local variables (or a method's temporary variables) comprise a set of variables names and types used during execution of the method.
- **METHOD BODY -**The body of a method is a sequence of instructions. Users can add an instruction by selecting the proper expression template, completing the blanks and then selecting the Add operation. The added instruction will be appended to the instruction part.

Figure (9) illustrates a method's header, variables and the addition of a method body.

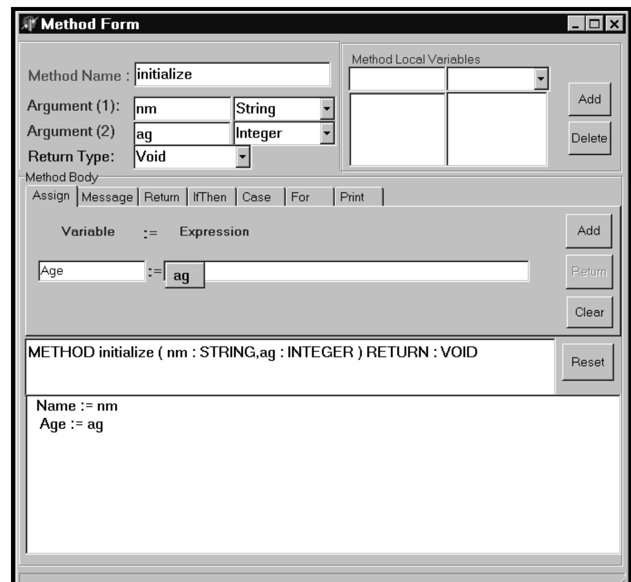

Figure 9 – The Create Method Dialog

The language provides a set of statements that can be used to implement a class. These statements are also made available via templates. For example, if the user chooses the If-Then statement, the appropriate template is displayed to the user, as shown in Figure (10) below.

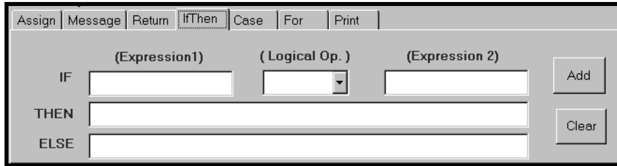

Figure 10 - If-then template

4.3 OBJECT CREATION

Once a class is defined, objects of that class may be created. This operation is similar to interactively sending a "new" message to a class in a Smalltalk environment, i.e. an instance is interactively created and made available to users. Users must provide an object name, and the resulting named object will appear in the active object box in the bottom right of the user interface as shown in Figure (11) below:

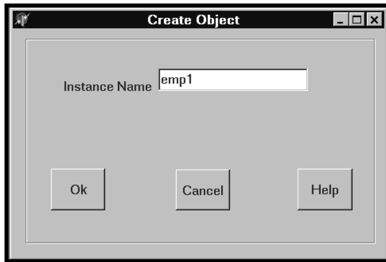

Figure 11 - Object creation dialogue

4.4 SENDING A MESSAGE

Once an object in the object box, selecting it will invoke a pop-up menu that lists all messages which can respond to the active object as shown in Figure(12) below:

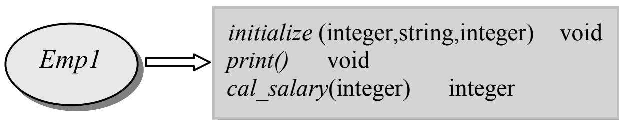

Figure 12 - Sending a message

4.5 ENTERING PARAMETERS

Selecting one of the object's methods makes the system request any parameters for that message, otherwise the method is executed directly. For example, to send the message *initialize* to the object *emp1* the dialogue in Figure(13) appears and the user must enter the *Salary*, *Name* and *Age* in the proper boxes.

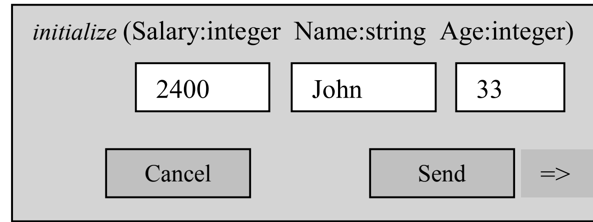

Figure 13 - Entering parameters

If one of the parameters requests a structured object, then a double click on the parameter box will display a menu of objects of that type from which the user may select their choice.

4.6 THE RUN-TIME VIEWER

The runtime viewer shows all messages received by the object during run-time as shown in Figure (14). In this example, the message *initialize* invoked the method *initialize* in the employee class, then the method *initialize* sends another message *initialize* to the super-class person.

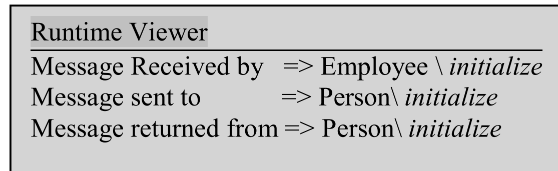

Figure 14 - The Run-time viewer.

4.7 INSPECTING AN OBJECT

Object inspection allows users to examine the structure of an object and determine the current values of its instance variables as shown in Figure (15).
Users can inspect an object by selecting the Inspect option from the object menu or by dragging the mouse over the object in the object corner in the bottom right of the screen.

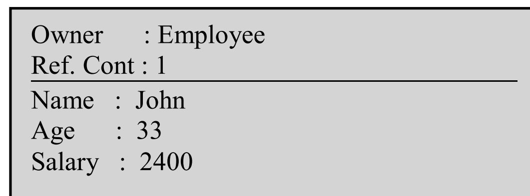

Figure 15 - Inspecting an object's contents

The object dialog shows the object owner, a reference count and the instance variables names and values.
If the value of a variable is itself a structured object only an <object reference> is shown. If this reference is itself then inspected, the inspection operation is applied recursively.

5. SUMMARY AND CONCLUSIONS

This paper has described an Initial Object-oriented Programming Language (IOPL) and its environment. The IOPL language and environment differ from other teaching languages and language implementations, e.g. the Turing Language[5], SmallTalk[6], F[7], and Blue[8], in several important ways. First, the IOPL language was specifically designed to be suitable for incremental type checking, and the IOPL environment supports this form of type checking via an interactive structure editor that manipulates the underling abstract syntax representation of an application. Secondly, the same underlying representation enables the IOPL environment to efficiently and correctly generate code, and to provide support for automatically un-parsing the stored representation into other languages, e.g. C++ and Java. Finally, the environment is "web-capable", i.e. it enables users to invoke external applications such as a web-browser, and it has an in-built FTP class that enables users to communicate stored representations via shared servers.

In order to support a range of undergraduate courses including initial courses in programming skills and later courses in OO techniques, the environment has been populated with teaching materials. These materials include web-based tutorials, sample implementations of model solutions to exercises, and also model applications, e.g. a calculator, a software implementation of a processor.

6. REFERENCES

[1] "A Perspective on Language Wars", Bowman, H., Papers for CTI Annual Conference 1994. http://www.ulst.ac.uk/cticomp/papers/bowman.html

[2] "The Selection of a First Programming Language", Essendal, H. T. Papers for CTI Annual Conference 1994. http://www.ulst.ac.uk/cticomp/papers/esendal.html

[3] Jensen, K. and Wirth, N. "Pascal User Manual and Report. 2nd Edition", Springer-Verlag, ISBN 0-387-90144-2, (1975)

[4] UCSD P-System Version IV, SofTech Microsystems, Inc., San Diego, California, (1982)

[5] "The Turing programming language", Holt, R. C., and Cordy, J. R., CACM 31, 12 (Dec. 1988), Pages 1410 - 1423

[6] "The Early History of Smalltalk", Kay, A. C., ACM SIGPLAN Notices Volume 28, No. 3, March 1993 Pages 69-95

[7] ``The F programming language", Reid, J., and Metcalf, M., Oxford University Press, 1996, ISBN 0-19-850026-2.

[8] "Blue - A Language for Teaching Object-Oriented Programming", Michael Kölling and John Rosenberg Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, March 1996, pp. 190-194.

[9] "An Object-Oriented Program Development Environment for the First Programming Course", Michael Kölling and John Rosenberg, Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, March 1996, pp. 83-87

[10] Harrison, C.J. and Powell, M.S. "An Environment for Initial Software Engineering Teaching", Software Engineering Journal, November 1992

[11] Harrison C.J and Powell, A Modular Persistent store POS 1990,pp171-184, (1990).

[12] Harrison C.J and N. Majid, POOL: A Persistent Object-Oriented Language. ACM symposium on Applied computing, 2000.

[13] M. S. Powell. A Program Development Environment based on Persistence and Abstract Data Types. Proceedings of 3rd International Conference on Persistent Object Systems, Newcastle, Australia (January 1989).

[14] Adele Goldberg, David Robson. Smalltalk-80 - the Language and its Implementation. Addison-Wesley, 1983.

[15] R. McGregor. Practical C++. QUE Corporation 1999.

[16] H. Schildt. Java 2: The complete reference. McGraw-Hill 2001.

[17] M.P. Atkinson, R. Morrison. Orthogonally Persistent Object Systems. VLDB Journal 4,3. 1995, pp319-401.

[18] T. Budd. An Introduction to Object-Oriented Programming, Addison Wesley, 1997.