Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 1988

Objects and Types: A Tutorial

James Kiper
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1988-007

Objects and Types: A Tutorial

James D. Kiper

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

Objects and Types: A Tutorial

by

James D. Kiper
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #88-007 08/88

Objects and Types: A Tutorial

James D. Kiper, Ph.D.
Miami University
August, 1988

1. Introduction.

This paper is a tutorial explaining the concepts that surround abstract data types and object-oriented programming, and the relationships between these groups of concepts. These concepts include types (language-defined, user-defined, abstract), instantiations, differences between operations and functions, overloading, objects, state, inheritance and, messages. Some of the these trems, e.g. "type", have been well defined. Many others are used in several contexts with multiple meanings. This paper is an attempt to identify consistent and meaningful definitions which are the most widely accepted.

2. Background Definitions.

The type of an identifier has become a standard concept in most modern programming languages since typing of identifiers makes the task of identifying potentially inappropriate uses of an identifier easier for both the compiler and the human reader of the program. The type of an identifier defines the kind of entity which that identifier represents. The type may indicate that an identifier represents a procedure, a variable, a constant, a package, or some other syntactic component of the language. In the case of variables, the type, generally called the data type, specifies the set (or range) of values which that variable can have, and the set of operations which are valid upon that variable. A type forms a template or pattern to be used in the creation of an entity. A type is not that entity, but is rather a pattern to be used in its creation. The process of using a type template to create an entity is called instantiation. An instance of that type is created. For example, in the C programming language "int" is the name of the integer type. It is not an integer, but is a template for creating integers. The declaration "int i;" is an instantiation of "int". The variable "i" is an instance of "int".

Another view is that a type is a way of specifying the semantics for a syntactic component of a program. A type of a variable explains how to interpret the binary digits which are stored in the memory location associated with that variable; and reveals the operations which are valid upon those bits. The type declaration of a procedure

specifies what code is to be associated with that procedure, and what its parameters are. Meaning has been added to the procedure name.

An operation is a procedural component of a programming language which transforms data from one form to another. The operation consists of two portions: a specification of the algorithm to be performed, and a specification of the valid data types of its operands. (Section 4 of this paper details the distinction between operations and functions or procedures.)

One useful categorization of various components of a particular programming language is language-defined or user-defined. A syntactic element of a programming language is language-defined if there is a symbol or identifier, i.e. a lexical element, of the language whose semantics are embedded in that language. For example, the term "integer" has a well-defined meaning to a Pascal compiler.

A user-defined entity is a syntactic element which requires a definition in a program to give the semantics to the language processor. In Pascal, for example, the enumerated type "color" has no meaning until its values are specified in the type section of a Pascal program.

The notion of visibility is one which has arisen recently in an attempt to manage the complexity of software. An entity or a portion of an entity in a system has "public" visibility if all other entities or objects in the system can use that entity or portion. "Private" visibility means that the entity or a portion of the entity is hidden from all other entities.

This visibility concept when applied to variables is closely related to the idea of scope. The scope of a variable is the section of the program in which a reference to that variable is syntactically valid. A variable with global scope has public visibility; a variable which is local to a procedure has private visibility. The distinction between these concepts appears in more complex components, e.g. procedures. Some programming languages, e.g. Ada, permit the specification of a public and a private portion of a procedure. The public part, its "specification" in Ada, can be used any where in the scope of the procedure's name. The private portion contains those portions of the procedure which are not revealed outside the procedure. Since these details are not revealed, they can be changed. As long as the public portion (interface) remains constant, such changes effect no other component of the program.

For example, a queue could be implemented in Ada as a circular array with the procedures "enqueue" and "dequeue". (A queue is a first-in-first-out list; enqueue is the operation of added a queue element to the back of the queue; dequeue is the operation of deleting, or servicing, an element at the front of the queue.) The interfaces to these procedures, that is, the procedures' names and parameter lists, and the data type "queue" can be specified with public visibility. The procedure bodies for "enqueue" and "dequeue" and the data structure representation of "queue" can be specified with private visibility. If the queue's implementation is subsequently changed to a linked list, nothing with public visibility will have to be changed. The implementations of the queue data structure and the procedure bodies which have private visibility will change. No other portion of the system which has used queues will have to be changed since the rest of the system can only use the portions of the queue which are public. In particular, the name of the queue data type and the name and parameter lists of the procedures may have been used. Since these have not changed, the modifications have been isolated to the private portion of the queue.

The use of public and private visibility when appropriate is an ideal method of applying information hiding. Information hiding is the principle that the only information which should be revealed about an object is that which is necessary to use the object. The software engineering benefits of this principle in handling complexity and controlling the effect of changes are apparent.

The principle of abstraction is one which has been vital controlling complexity in many situations. To understand any complex system, less important details are ignored to accentuate the salient features. An abstraction is a view of a system or problem which extracts the essential information relevant to the immediate task while ignoring the remainder of the information. (The determination of which features to ignore and which to consider depends upon the purpose of the abstraction.) A hierarchy of successively more detailed abstract views provides a sequence of pictures of a complex system which can be easily understood.

Binding time is an important characteristic of each attribute or property of an identifier in a particular implementation of a programming language. Each attribute of an identifier is bound to that identifier at a specific time. The meaning of a reserved word is bound to the language when the language is defined; the value of a variable is bound to that variable during the execution of a program. As a general rule, the earlier that a property is bound to an identifier, the better the diagnostics which can be automatically provided if that property of the identifier is misused. Conversely, the later the binding occurs, the more flexible that property is. Since the value of

a variable is generally bound at run-time, that value can be changed easily during program execution. The value of a constant identifier is generally bound during compilation, so that no run-time attempt to modify that value is valid. For the purposes of software engineering, e.g. understandability, modifiability, the general rule is that a property should be bound as early as is possible since this allows the compiler to provide more complete diagnostics and warnings of incorrect use.

When applied to the types of identifiers in a programming language, binding time is especially important. Languages which bind types to identifiers during compilation can perform static type checking. Dynamic typing occurs when types are bound to identifiers during program execution. It is generally believed that the use of a language with static type checking is vital to assure quality in the development of large, complex software systems. A language which is strongly type checked is one in which each identifier has a type associated during compilation, and in which all conversions between identifiers of various type are accomplished explicitly. (For example, in Pascal, a conversion from a real value to an integer value can only occur through the explicit use of a function like trunc or round. Since implicit conversions from integer to real are permitted in Pascal, but not in Ada, Ada is more strongly typed checked than Pascal.)

The following sections of this tutorial examine these and related concepts in more detail.

3. **Types and Instantiations.**

As discussed previously, a type is a template which describes some property of a syntactic entity. The type is not that entity, but serves as a template for the creation of the entity, and, after creation, serves as a description of the entity. The term data type is most often used to refer to those types which are language defined, e.g. integer and real in Pascal, int and float in C. A data object, i.e. variable, is the result of the instantiation of such a data type.

Most programming languages provide facilities for the creation of user-defined data types. Examples are enumerated types in Pascal, structures in C, and records in Ada. We will call an instance of such a type a user-defined data object. User-defined data types differ from language-defined types in that the programmer determines the set of values which that type represents. (These user-defined types are somewhat restricted by the language since the language designer determines the set of all possible values from which the user chooses.) The

prescription of the set of operations which can be applied to this user-defined set of values still remains with the language. In Pascal for example, a user can define the set of values for an enumerated type, but the set of operations, e.g. pred, succ, ord, and relational operators, is determined by the language.

More recent programming languages have extended the concept of user-defined type to permit the user to specify the set of values and the set of operations of that type. This capability has the appellation abstract data type; the result of the instantiation of an abstract data type is an encapsulated data object or data abstraction. To describe an abstract data type, it is necessary to specify the values and the operations permitted. The specification of the set of values is often accomplished by building data structure specifications from combinations of language defined data types and/or previously defined user-defined data types. The operations are defined in the form of functions or procedures which operate upon data of these types. (Section 4 describes in more detail the relationships between operations and procedures.) A common example of an encapsulated data object is a LIFO stack with the operations Push, Pop, and IsEmpty. The set of data values permitted may be the set of all integer arrays of one hundred elements (or the set of all linked lists of integers.) Figure 1 presents an Ada implementation of such an abstract data type. When this type is instantiated in a declaration like "operator_stack : STACKS", the identifier "operator_stack" is an encapsulated data object. The operations "push", "pop' and "IsEmpty" are the only valid operations on this object.

```
package STACKS is

    type stack_type is private;

    procedure push( element: in integer; stack : in out stack_type);
    function pop( stack: in out stack_type) returns integer;
    function IsEmpty( stack: in out stack_type) returns boolean;

private
    max_elements : constant integer := 100;
    type list is array (1..max_elements) of integer;
    type stack_type is
        record
            node : list;
            top : integer range 1..max_elements := 1;
        end record;
end stacks;

package body STACKS is
    procedure push( element: in integer; stack : in out stack_type) is

        begin
            stack.top := stack.top + 1;
            stack.node(stack.top) := element;
        end push;
```

```
function pop( stack: in out stack_type) returns integer is
    element : integer;
    begin
        element := stack.node(stack.top);
        stack.top := stack.top - 1;
        return( element );
    end pop;
function IsEmpty( stack: in out stack_type) returns boolean is
    empty : boolean;
    begin
        if stack.top = max_element then empty := true;
        else empty := false;
        return( empty );
    end push;
```

Figure 1 An Encapsualted Data Type for a Stack in Ada.

One advantage of the use of language-defined data types is that the implementation details of a variable of that type are generally not assessable to the programmer. In many languages, the programmer cannot construct a program which manipulates the bit string representation of an integer, real number, or character, but is restricted to manipulating the data object by means of the language-defined operations. This has the advantage that others reading the program can more easily understand this standard use of the data object, and that the program can be more easily ported to another hardware base.

In an analogous manner, it is advantageous for the implementation details of an encapsulated data object to be hidden and not accessible to users of that object. Languages which support abstract data types and data abstraction have a syntax which hides the implementation details of the data and the operations, while revealing a public interface. The "public interface" is a revelation of the names of the operations and the names of the types of the data necessary to use the data abstraction. Figure 1 illustrates this public interface and the hidde, i.e. private, implementation.

4. The Differences between Operations and Procedures.

The terms operator and procedure both refer to syntactic components of a programming language which represent a specification of some action or set of actions to perform. (The term procedure is used here in a generic sense. Some programming languages use other related terms like "function" or "subroutine". There is no semantic difference among these. Hence, we will use the term procedure to incorporate all these.) The primary difference between operators and procedures has been a syntactic one. The symbol "+" represents the addition operator in Pascal; the name "write" represents a standard printing procedure. Both are names which refer to a group of computer instructions. Binary operators are generally represented by an "infix" notation in which the

operators symbol is placed between the operands. Procedures generally use a parenthesized list for parameters or operands in a prefix notation.

In the past, operators were always language-defined. In addition to the language define procedures, facilities were provided to permit the user to define procedures. This distinction between operators and procedures is disappearing in more recent programming languages which support abstract data types. This illusion that procedures and operators are identical becomes more complete with the overloading of operator symbols. Overloading is the use of a symbol for multiple purposes. The context of the use determines the meaning. For example, the "+" symbol in Basic can represent integer addition, real addition, or string concatenation. Some languages, e.g. Ada, permit the programmer to specify an overloading. The user can give the symbol which is most appropriate for an operation (or procedure).

```
package complex is
    type number is private
    procedure set( x : in out number;
                   real_part : in float;
                   imaginary_part : in float);
    function "+" (x,y : in number) return number;
    function "-" (x,y : in number) return number;
    function real_part ( x : in number ) return float;
    function imaginary_part ( x : in number ) return float;
private
    type number is
        record
            real_part : float;
            imaginary_part : float;
        end record;
end complex;
```

Figure 2. Overloading of operators.

Figure 2 gives the Ada specification for an abstract data type called "complex" which represents complex numbers which have a real and an imaginary part. Addition and subtraction are defined on these comlex numbers. The "+" and "-" symbols are overloaded to refer to these operations. The Ada compiler is able to differentiate between this use of these symbols and their ordinary use by determining the type of the oerands. This example is came from the book entitled "Software Engineering with Ada" by Grady Booch [2] in which it is completed.

5. Languages and Abstract Data Types.

The ability to program with abstract data types is not a function of the programming language being used. This method of programming is a disciplined method of using data structures and related operations

(procedures). It is possible to use abstract data types and data abstractions in any language, from assembly language to Ada to Lisp. However, some languages permit and enable this technique more effectively than in others. In particular, there are two adjectives which describe the aid which a programming language gives to abstract data types, support and enforcement. By support, we mean that the language has some syntactic elements which enable the creation and use of abstract data types. Enforcement means that, besides allowing a correct use of abstract data types, the language does not permit an incorrect, or inappropriate, use of the data abstractions and abstract data types.

Languages in the category which provides neither support or enforcement, e.g. assembly language and Basic, allow data abstractions, but only through the discipline of the programmer. Data of various types can be specified, and operations on that data can be created. However, the language provides no syntactic units which support types, or the allow operations and data to be packaged together. The data can be accessed via the requisite operations, but there is nothing to restrict the programmer from accessing the data in alternative ways that do not involve legitimate uses of the operations of that data. The data types are not strictly bound to the data. That is, the data can be interpreted in many ways other than that intended by the type.

Consider the task of creating a binary search tree in Basic. Each node of the tree is to store a person's name and telephone number and pointer to the left subtree and right subtree. The left subtree is to contain nodes of persons whose names are alphabetically less than the root node's node; the right contains those which are alphabetically greater. This definition continues recursively since each subtree is also a tree. This can be implemented in Basic by means of several arrays. One array contains the names; a second array contains the cooresponding telephone numbers; the third array contains integers which serve as pointers. An "operation" called print tree could be created through the mechanism of the "gosub". Although possible, such a program would be cryptic at best. The translator would not check for any misuses of the data or the operations.

Languages which provide support, but no enforcement, for abstract data type and data abstraction are typified by Pascal and C. Programmers can define their own data types and corresponding data, and operations upon those data. The language supports the creation of the data types, and the specification of operations (procedures or functions) in its syntax. Enforcement of data abstractions and data types is absent. The details of implementation of the operations or data structures are not not hidden from the programmer. For example, if an

integer stack data structure with operations of Push and Pop is implemented in Pascal or C as an array of integer, a programmer can very easily access the array which represents the stack directly, rather than through the Pop and Push. This may seem to be inconsequential to program development. However, the maintenance problem introduced may be enormous. In order to change the implementation of the stack data structure from an array to a linked list, the entire program may potentially be affected. Every reference to the array which was previously used will have to be modified. Conversely, if the programmer had used the stack in a disciplined manner in which the only accesses to the stack were via the Pop and Push operations, then the array could be changed to a linked list and the Pop and Push operations modified to access the list instead of the array. If the interface to these operations remained consistent, i.e. the names and parameter list was unchanged, no other portion of the program would have to be modified.

The more modern languages which support and enforce data abstractions and abstract data types have eliminated the possibility of the scenario discussed above. These languages, e.g. Ada, Modula 2, Objective C, Clu, give the programmer a set of syntactic components for the specification of data types and associated operations, and mechanisms for the packaging of these data objects and operations together. The language enforces the constraint that these data objects can be accessed and modified only through these associated operations. This has the benefit of explicit specification of operations for data structures, and the decoupling of data objects from one another. Understandability, maintainability, and modifiability of the code is enhanced.

6. Object-Oriented Programming.

To successfully engineer complex software, development methods are necessary. Two general classes of methods which have been promoted and have proven at least moderately successful are the process-oriented and the data-oriented methods. The first class consists of those which first concentrate upon the design of the processes required by the software; the second consists of those which approach the design of the data first, i.e. the input and output, and from the data structures required develop the requisite processes. A third general approach or class of methods has recently found success in many application domains: object-oriented design and object-oriented programming. This method has been described as an approach which blends the design of data and processes. The design of software in this method consists of breaking the desired system into the logical

objects which compose it. (These objects are often determined by modelling the components of the physical system.) This method is illustrated and described more completely in Booch's paper [2].

Objects, as used in object-oriented programming, have the following properties. (1) Objects are data abstractions and are derived as instances of an abstract data type. Specifically, this means that objects have a private data store, a private set of operations (often called methods), and a public interface which reveal the names and parameters required to use this object. (2) Objects have state. This means that the internal memory has its own on-going existence. Pascal procedures cannot be used to implement objects since local variables in Pascal do not maintain their value between calls to the procedure. In C, static, local variables do maintain their values and could therefore be used to give state to a function in C. (3) Objects request service from another object and communicate with other objects by the means of messages. (The differences between messages and procedure calls are discussed in the following section.) (4) The abstract data types which describe objects are organized into a hierarchy of classes and subclasses. A subclass inherits all the data structures and operations (methods) of its parent class. In addition, a subclass includes some other more specific methods and/or data structures. An instance of such a class or subclass is an object. (5) Methods (operations) are bound to an implementation dynamically (at run-time). Most modern programming languages, including Ada and C, do not provide this capability since the implementation details for an operation must be known during compilation. Some exceptions are Lisp and Smalltalk. A parenthesized list can be input at run time, and subsequently executed. Lisp's delayed bindings are the source of this flexibility. Smalltalk is the quintessential object-oriented programming language, and, as such, allows dynamic binding. This means that additional methods (operations) can be added to the system without having to recompile the entire system. The papers by Kaehler [5] and Seidewitz [10] prrovide an introduction to Smalltalk.

7. Messages versus Procedure Calls.

Objects communicate by means of messages. A message is often implemented by means of a procedure-like call. The difference lie in the fact that messages operate in an environment in which objects have state. The results of a message are dependent not only upon the parameters (as in standard procedure calls), but also on the

state of the object which receives the message. Another difference is that the implementation of the methods (operations) which result from the message are determined at run-time.

8. Summary

The software engineering terminology which has arisen over the past decade has led to many mis-interpretations and misunderstandings, particularly to those who are not involved in the literature of the field on a regular basis. An understanding of these concepts is vital for anyone who is interested in improving software design and implementation skills. However, the terminology is still in flux. This tutorial presents and defines some of the important terminology in this field in a consistent manner.

The terms introduced are listed below in the order in which they are introduced in the paper. This list can be used by the reader to find a particular term more quickly.

1. type
2. data type
3. instantiation
4. instance
5. operator
6. language-defined
7. user-defined
8. visibility
9. scope
10. information hiding
11. abstraction
12. binding time
13. static type checking
14. dynamic type checking
15. strongly typed
16. data object
17. user-defined type
18. user-defined object
19. abstract data type
20. encapsulated data object
21. data abstraction
22. operator
23. procedure
24. support
25. enforcement
26. process-oriented method
27. data-oriented method
28. object-oriented design
29. object-oriented programming
30. objects
31. methods
32. state
33. message
34. class
35. subclass

9. Annotated Bibliography.

1. Booch, Grady, "Object-Oriented Development", IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, Feb. 1986, pp. 211-221. This article uis a good introduction to Object-Oriented Design by one of the early supporters of the method. The examples given illustrate the concepts well.

2. Booch, Grady, Software Engineering with Ada, Second Edition, Benjamin-Cummings Publishing Co., Menlo Park, Ca., 1987. This text book is a coverage of Object-Oriented Design integrated with Ada. It provides a detailed description of the OOD method, and illustrates how Ada supports OOD. It is not a good Ada reference.

3. Cox, Brad, Object Oriented Programming, Addison-Wesley, Reading, Ma., 1987. This book presents another view of OOD.. The terminology is slightly differnt from that of Booch, but the same concepts are presented.

4. Guttag, John V., Ellis Horowitx, and David R. Musser, "Abstract Data Types and Software Validation", Communication of the ACM, vol. 21, no. 12, Dec. 1978. This paper is a classic which illstrates the use of abstract data types in a formal manner. Undergraduate students will need some guidence in applying these techniques.

5. Kaehler, Ted and Dave Patterson, "A Small Taste of Smalltalk", BYTE, August 1986, pp. 145-159. This article provides a laypersons view of Smalltalk.

6. Love, Tom, "Hope with Objects", Productivity Products International, Inc., Sandy Hook, Ct. This paper provides a good description and illustration of object. It is available from the author.

7. Pascoe, Geoffrey A., "Elemnets of Object-Oriented Programming", BYTE, August 1986, pp. 139-144. This is a ldescription of OOP in a layperson's terminology.

8. Pomberger, Gustav, Software Engineering and Modula-2, Prentice-Hall International, Englewood Cliffs, NJ, 1984. This text book presents the important concepts of Software Engineering in the context of the language Modula 2.

9. Pratt, Terrence W., Programming Languages: Design and Implementation, Second Edition, Pretice Hall, Englewood Cliffs, NJ, 1984. This classic text book present the traditional programming language concepts, e.g. typs, binding, in a very understandable way. (It does not cover OOP.)

10. Seidewitz, "Object-Oriented Programming in Smalltalk and Ada", OOPSLA '87 Proceedings, October 4-8, 1987, pp. 202-213. This article compares and illustrates the use of Ada with that of Smalltalk in performing OOP.

11. Verity, John W., "The OOP Revolution", Datamation, May 1, 1987, pp.73-78. This is another view of Object-Oriented Programming in terminology that the layperson can understand.