Object Oriented Programmingversusabstract Data Type

ShubhamGoyal & Mohit Pahwa
Department of Informational Technology, Dronacharya College of Engineering, Gurgaon, India

Shubham.16936@ggnindia.dronacharya.info

Mohit.16921@ggnindia.dronacharya.info

ABSTRACT

This abstract collects and elaborates arguments for distinguishing between Object oriented programming and abstract data types. The basic distinction is that object oriented programming achieves data abstraction by the use of procedural abstraction, while abstract data types depends upon type abstraction. In this paper we will study about the application of object oriented programming and abstract data type.

Keywords-

Object oriented programming (OOP); Abstract data type (PDP); programming; Operation

INTRODUCTION

The development of abstract data types and object-oriented programming, from their roots in Simula 67 to their current diverse forms, has been prominent in programming language research for the last two decades. This tutorial is aimed at organizing and collecting arguments that distinguish between the two paradigms. The focus of the arguments is on the basic mechanisms for data abstraction, illustrating the differences with examples. Although more advanced topics, like inheritance, overloading, and mutable state, is important features of one or the other paradigm, they are not considered in this presentation. The interpretations of "abstract data type" and "object-oriented programming" compared in this paper are based upon major lines of development recorded in the literature and in general use.

Abstract data types are often called user-defined data types, because they allow programmers to define new types that resemble primitive data types. Just like a primitive type INTEGER with operations +, −, _, etc., an abstract data type has a type domain, whose representation is unknown to clients, and a set of operations defined on the domain. They are also closely related to algebraic specification. In this context the phrase "abstract type" can be taken to mean that there is a type that is "conceived apart from concrete realities" [1].

Object-oriented programming involves the construction of objects which have a collection of methods, or procedures, that share access to private local state. Objects resemble machines or other things in the real world more than any well-known mathematical concept. In this tutorial, Smalltalk is taken as the paradigmatic object-oriented language. The term

"object" is not very descriptive of the use of collections of procedures to implement a data abstraction. Thus we adopt the term procedural data abstraction as a more precise name for a technique that uses procedures as abstract data. In the remainder of this paper, procedural data abstraction (PDA) will be used instead of "object-oriented programming". By extension, the term "object" is synonymous with procedural data value.

HISTORICAL OVERVIEW

In 1972, David Parnas published his seminal work on modularization [2]. He showed the value of decomposition of a system into a collection of modules supporting a procedural interface to hidden local state. He pointed out the usefulness of modules for facilitating modification or evolution of a system. His specification technique [36] for describing modules as abstract machines has not been generally adopted, but the module concept has had a great impact, especially on the development of languages like Modula-2 [3]. Although Parnas recognized that modules with compatible interfaces can be used interchangeably, he did not develop this possibility. As a result, modules are not first-class values, so they cannot be passed as arguments or returned as values.

In 1973, Stephen Zilles published a paper on "Procedural abstraction: a linguistic protection technique" which showed "how procedures can be used to represent another class of system components, data objects, which are not normally expressed as programs" (emphasis added). His notion of procedural abstraction is very similar to Parnas's modules; however, he views them as data and discusses passing them as arguments to other procedures, and returning them as values. He also noted the similarity to objects in Simula. He illustrated them by discussing streams represented as a vector of procedures with local state. Calling an operation was defined as an indirect procedure call through the vector. He shows that different classes of stream objects can be defined by building an appropriate vector of procedures. He also presents two of the main methodological advantages of objects: encapsulation and independence of implementations.

The following year, in 1974, Zilles published an influential paper with Barbara Liskov on ADTs and CLU. Gone was any mention of OOP; type abstraction had taken its place. The formalism of ADTs was still presented as closely related to Simula; the main difference was claimed to be that Simula allowed full inspection of object representations.

In 1975, John Reynolds published a paper called "User-defined data types and procedural data structures as complementary approaches to data abstraction" in which he compares procedural data abstraction to user-defined data types. He argued that they are complementary, in that they each have strengths and weaknesses, and the strengths of one are generally the weaknesses of the other. In particular, he found that PDAs offer extensibility and interoperability but obstruct some optimizations. ADTs, on the other hand, facilitate certain kinds of optimizations, but are difficult to extend or get to interoperate. He also discussed the typing of the two approaches, and identified recursion in values and types as characteristic of PDA. One limitation of

his presentation is that the objects in his examples only have a single method. The introduction of a second method was described as an intellectual "tour de force", implying that multiple methods are too complicated for use in practical designs. After 1975 little was written that related to the theory of object-oriented programming, while investigation of ADTs continued. Yet development of object-oriented languages, like Smalltalk and Flavors , continued, especially in the context of extensible, interactive, open systems which encouraged user programming. Theoretical interest in object-oriented programming was sparked in 1984 by Cardelli's paper on "The semantics of multiple inheritances" [4]. This paper identified the notion of subtyping as central to an understanding of object-oriented programming. Subtyping and parametric polymorphism were combined to form bounded quantification, which could describe aspects of update operations on records A good explanation for the complementarity noted by Reynolds was presented by Abelson and Sussman [1] in 1985, although they do not cite his work. They discuss "data-oriented programming" as a technique for writing flexible and extensible programs in Lisp. They note that abstractions are characterized by observations and representations, where the operation needed to perform an observation depends upon the representation. Data-oriented programming works by grouping all the observations on a particular representation together as components, or methods, of a value containing that representation. This is in contrast to operation-oriented programming, or ADT programming, where a function is written for each observation with cases for each representation. By organizing the observations and constructors into a two-dimensional matrix, it becomes clear that ADTs and object-oriented programming arise from a fundamental dichotomy: there are two ways to organize this table: either by observers for ADTs or by constructors for PDAs

DISTINGUISHING ADTs and OOP

Abstract Data Type (ADP)

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Abstract data type (**ADT**) is a mathematical model for a certain class of data structures that have similar behaviour; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.[1]

For example, an abstract stack could be defined by three operations: PUSH, that inserts some data item onto the structure, POP, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and PEEK, that allows data on top of the structure to be examined without removal. When analysing the efficiency of algorithms that use stacks, one may also specify that all operations take the same

time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the typesystems of programming languages. However, an ADT may be implemented by specific datatypes or datastructures, in many ways and in many programming languages; or described in a formalspecificationlanguage. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This informationhiding strategy allows the implementation of the module to be changed without disturbing the client programs.

Object Oriented Programme (OOP)

Object-oriented programming attempts to provide a model for programming based on objects [5]. Object-oriented programming integrates code and data using the concept of an "object". An object is an abstract data type with the addition of polymorphism and inheritance. An object has both state (data) and behaviour (code).

Objects sometimes correspond to things found in the real world. For example, a graphics program may have objects such as "circle," "square," "menu." An online shopping system will have objects such as "shopping cart," "customer," and "product." The shopping system will support behaviour's such as "place order," "make payment," and "offer discount."

Objects are designed in class hierarchies. For example, with the shopping system there might be high level classes such as "electronics product," "kitchen product," and "book." There may be further refinements for example under "electronic products": "CD Player," "DVD player," etc. These classes and subclasses correspond to sets and subsets in mathematical logic. Rather than utilizing database tables and programming subroutines, the developer utilizes objects the user may be more familiar with: objects from their application domain.[4]

Object orientation uses encapsulation and information hiding. Object-orientation essentially merges abstract data types with structured programming and divides systems into modular objects which own their own data and are responsible for their own behaviour. This feature is known as encapsulation. With encapsulation, the data for two objects are divided so that changes to one object cannot affect the other. Note that all this relies on the various languages being used appropriately, which, of course, is never certain. Object-orientation is not a software silver bullet [6].

WHAT IS ADTs in OOP?

An *abstract class* is a generalization concept. It is a class you invent to only use as a base class for inheritance but not to instantiate objects from.
And *abstract data type* is not necessarily an OOP concept. It is an older term to describe the concepts of for example Stack and Queue in terms of their functionality, without describing the implementation.

Since you are probably interested in abstract class, a small example:

Suppose you have to make a program to deal with cars and motorbikes. You can define the classes (entities) of Car and Bike and you will see they have much (but not all) functionality in common. It would be a mistake to derive Car from Bike or the other way around. What you need to do is to define a common **abstract** base-class MotorVehicle and derive both Car and Bike from that class.

```
Abstract class MotorVehicle {...}
/*concrete*/ class Car: MotorVehicle {...}
/*concrete*/ class Bike: MotorVehicle {...}
```

Note that you would never want to create an object of class MotorVehicle, it would not be 'concrete' (complete). MotorVehicle is only used to build a correct object-model.

COMPARING ADTs and OOP

The difference between ADTs and procedural abstraction involve both the client's use of the abstractionand the implementer's definition of the abstraction. The differences are illustrated in the areas of Incremental programming, optimization, typing, and verification. The client has an abstract view of data in both ADTs and OOP. The major difference between them is the technique used to enforce the encapsulation and abstraction. In an ADT the mechanism is type abstraction, while in OOP it is procedural abstraction. Another major difference is that in OOP the objects act as clients among themselves, and so are encapsulated from each other. In an ADT, the &abstract values are all enclosed within a single abstraction, and so they are not encapsulated from each other.

OPTIMIZING OPERATIONS

Optimizing operations is an important consideration in programming. One of the benefits of abstraction is that some optimizations can be performed in isolation within an abstraction. However, abstraction can also prevent optimization because it prevents access to the information on which the optimization would be based. When the interval representation is added to the lists, the equality operation becomes very inefficient because it must create the complete sequence of numbers in the interval. It is easier to optimize the ADT implementation because the list values are not encapsulated from each other as they are in the procedural data abstraction.

```
Length (l: list) = case l ofNIL) 0
CELL(x, l 0)) 1+length (l 0)
```

Figure 1: A length operation for the list ADT.

```
NilWithLength = Inherit Nil
With [Length = 0]
CellWithLength(x, l) = Inherit Cell(x, l)
With [Length = l.length+1]
IntervalWithLength(x: integer, y: integer)
=Inherit Interval(x, y)
With [Length = (y - x + 1)]
```

Figure 2: Adding a length operation to list constructors using inheritance.

OPTIMIZING ADTs

In the ADT it is possible to improve the efficiency, because the representations of both arguments to the equality function may be inspected. The equality operation in the list ADT can be improved by adding cases for the constructors of both arguments to the operation. Previously, all operations performed a case statement on only their first argument. By using case statement on both arguments of the equal operation, as shown in Figure 13, a much more efficient comparison of intervals is possible. This is still not the most efficient implementation possible, but it does illustrate examination of more than one representation.

OPTIMIZING OOP

In OOP it is much more difficult to optimize operations, because the representation of argument to theequality observation cannot be determined. For example, the equality method on an interval object cannot be optimized because there is no way to determine if its argument is also an interval. This is the cost of the flexibility of objects. The optimization of methods is only one form of optimization; addition of specialized representations can also be viewed as a form of optimization, which is supported by objects. Another promising approach involves compilation techniques that create special code for common combinations of arguments to methods [25, 22, 14]. Direct optimization of methods is possible in some cases. By adding additional messages to the object, it is possible for other objects to query these messages and perform more efficiently. These messages can easily degenerate into simply specifying representational details. The trick is to define sufficiently abstract queries that provide quick answers for some implementations, while not prohibiting other implementations. To give a simple example illustrating this, consider the addition of an append

```
Equal (l: list, m: list) =
Case l of
NIL) null? (m)
CELL(x, l 0)) (not null? (m))
and (x = head (m))
        And equal (l 0, tail (m))
INTERVAL(x, y)) case m of
NIL) false
CELL(y, m0)) (x = y) and equal (tail (l), m0)
INTERVAL(x0, y0)) (x = x0) and (y = y0)
```

Figure 3: Efficient comparison of ADT intervals.

IMPLEMENTING ADTs

A wide variety of languages support the implementation of abstract data types. These languages include use of private types in Ada packages, Clu clusters, ML abstype definitions, and opaque types in Modula-2. The overall structures of these facilities are very similar. The key element is of course that the representation of abstract values is hidden from users of the operations. Exactly how the representation type is defined and how the operations are implemented depends upon the data types and control structures of the language.

Figure 4 defines an ADT implementing integer lists. The syntax is based loosely on ML. The ADT has two distinct parts: a representation and a set of operations. The representation is defined as a labelled union type, or variant record, with cases named NIL and CELL. The NIL variant is simply a constant, while the CELL variant contains a pair of an integer and a list.

A client of the ADT is able to declare variables of type list and use the operations to create and manipulate list values.

```
adtIntList
representation
list = NIL | CELL of integer *list operations
nil = NIL
adjoin(x:integer,l: list) =CELL(x, l)
Null? (l: list) = case l ofNIL) true
CELL(x, l)) false
Head (l: list) = case l of NIL) error
CELL(x, l 0)) x
tail (l: list) = case l of NIL ) error
CELL(x, l 0)) l 0
equal (l: list, m : list) = case l of
NIL) null?(m)
CELL(x, l 0)) not null?(m)
and x = head (m)
and equal (l 0, tail (m))
```

Figure 4: Implementation of an ADT for lists.

IMPLEMENTING OOP

As combinations of procedural observations with shared local state, OOP are naturally implemented a closures containing records of procedures [7]. The procedures are derived from the specification of the data abstraction. The record is formed by using the observations as field names and the procedures as values. The closure is used to encapsulate the constructor's arguments, which act as local state for the procedures. The class constructs in most OOP languages can be viewed as special mechanism for creating closures of records. This form of closure is different from the kind commonly found in functional languages like Lisp for two reasons. First, in functional languages it is often only possible to form closures over functions, not records. Thus records must be simulated as an explicit case over a set of field names [8]. Second, the closures are not a general construct of the language, but are provided only within the class construct.

```
Nil = recursive self = record
        Null? = true
head = error;
tail = error;
cons = fun(y) Cell(y, self);
equal = fun(m) m.null?
end
Cell(x, l) = recursive self = record
null? = false
head = x;
tail = l;
cons = fun(y) Cell(y, self);
equal = fun(m) (not m.null?)
and (x = m.head)
andl.equal(m.tail) end
```

Figure 5: Implementation of lists as OOP.

The two constructors for list objects are defined in Figure 5. The constructor functions, Nil andCell, return record values. The constructor for cells takes two arguments, x and l, which play the role of instance variables of the object. In this example they are not changed by assignment, though there is no essential reason why they could not be modified (if, for example, a set-head method were introduced).

LANGUAGE USED IN ADTs and OOP

Simula 67 was the first object-oriented language. It was defined as an extension of Algol 60 by allowingblocks to be detached from the normal nested activation scheme and have an independent lifetime. The declarations in a detached block were made accessible to other parts of the program through a reference. The definition of such blocks was called classes, which also acted as types or qualifications on references. Classes could also be defined by extension of previous classes, resulting in an inheritance hierarchy. Early versions of the language did not provide sufficient encapsulation of the attributes of classes, but later versions corrected this problem.

Simula was the inspiration for both the pure ADT languages, like CLU, and the pure OOP language Smalltalk. This is not surprising, because Simula embodies aspects of both techniques. This composite approach has been preserved in most of its statically-typed descendants, including C++, Beta, and Eiffel. A class definition is both constructor of objects and a type. If the hidden part is empty then the class resembles an object-oriented interface. Such classes are sometimes called abstract classes. If a class with private components is used as a type, then it is acting more like an ADT.

Simula and C++ also support a distinction between virtual and non-virtual operations. When a virtual operation is invoked, the method to be called is determined from the object on which the operation is being performed. This is the behaviour that has been assumed as normal in the general discussion of OOP. All operations are virtual in Smalltalk, Eiffel, and Trellis. The method for a non-virtual operation is determined from the class of the variable

used to refer to the object, not from the object itself. Nonvirtualoperations model the operations in an ADT, because they are taken from the implementation of the type, not from the abstract values themselves. Classes in which all of the operations are virtual are called virtual classes.

CONCLUSION

The essence of object-oriented programming is procedural data abstraction, in which procedures are usedto represent data and procedural interfaces provide information hiding and abstraction. This technique is complementary to ADTs, in which concrete algebras are used to represent data, and type abstraction provides information hiding. The two paradigms can be derived from a fundamental dichotomy in decomposing a matrix of observers and constructors that specify abstract data.

As would be expected, given the organization biases of the two paradigms, they are complementary in the sense that each has advantages and disadvantages. Using OOP it is easy to add new constructors, and the absence of a shared abstracted type reduces code interdependence. With inheritance and subtyping it is also possible to add new observations (methods). However, the use of strong functional abstraction prevents optimizations that might be performed if more than one representation could be inspected at a time. Binary observations in particular cause difficulties for OOP

Simula was the inspiration for the development of both abstract data types (exemplified by CLU) and OOP (exemplified by Smalltalk). This is not surprising, because Simula embodies a combination of both techniques, a characteristic preserved by its descendants C++, Eiffel, and Beta. The combination is more of an interweaving than unification, because the trade-offs outlined above is still operative.

REFERENCES

[1] J. Stein, editor. Random House Dictionary of the English Language. Random House, 1967.

[2] D. Parnas. On the criteria to be used in decomposing systems into modules.

[3] N. Wirth. Programming in Modula- 2. Springer-Verlag, 1983.

[4] L.Cardelli. A semantics of multiple inheritances. In Semantics Data Types, volume 173 Notes CS.

[5]**Jump up^**Booch, Grady (1986). *Software Engineering with Ada*

[6]**Jump up^** Brooks, Fred P. (April 1987). "No Silver Bullet — Essence and Accidents of Engi.

[7] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In Proc. of the

ACM Conf. on Lisp and Functional Programming, pages 289–297, 1988.