Object-Oriented Programming

Versus

Abstract Data Types

William R. Cook
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA, 94303-0969, USA

Abstract: This tutorial 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 depend upon type abstraction. Object-oriented programming and abstract data types can also be viewed as complimentary implementation techniques: objects are centered around the constructors of a data abstraction, while abstract data types are organized around the operations. These differences have consequences relating to extensibility, efficiency, typing, and verification; in many cases the strengths of one paradigm are the weaknesses of the other. Most object-oriented programming languages support aspects of both techniques, not a unification of them, so an understanding of their relative merits is useful in designing programs.

Keywords: abstract data type, object-oriented programming, higher-order procedures, incremental programming, extensibility, typing.

1 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, are 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. Abstract data types were first formulated in their pure form in CLU [31, 30]. The theory of abstract data types is given by existential types [33, 13]. They are also closely related to algebraic specification [20, 23]. In this context the phrase “abstract type” can be taken to mean that there is a type that is “conceived apart from concrete realities” [41].

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. A useful theory of objects associates them with some form of closure [2, 37, 8], although other models are possible [26]. 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.

It is argued that abstract data types and procedural data abstraction are two distinct techniques for implementing abstract data. The basic difference is in the mechanism used to achieve the abstraction barrier between a client and the data. In abstract data types, the primary mechanism is type abstraction, while in procedural data abstraction it is procedural abstraction. This means, roughly, that in an ADT the data is abstract by virtue of an opaque type: one that can be used by a client to declare variables but whose representation cannot be inspected directly. In PDA, the data is abstract because it is accessed through a procedural interface – although all of the types involved may be known to the user. This characterization is not completely strict, in that the type of a procedural data value can be viewed as being partially abstract, because not all of the interface may be known; in addition, abstract data types rely upon procedural abstraction for the definition of their operations.

Despite the very different approaches taken by ADT and PDA, they can be understood as orthogonal ways to implement a specification of a data abstraction. A data abstraction can be characterized in a very general way by defining abstract constructors together with abstract observations of the constructed values. Using these notions, a data abstraction may be defined by listing the value of each observation on each constructor. The difference between PDA and ADT concerns how they organize and protect the implementation of a data abstraction. The choice of organization has a tremendous effect on the flexibility and extensibility of the implementation.

Abstract data types are organized around the observations. Each observation is implemented as an operation upon a concrete representation derived from the constructors. The constructors are also implemented as operations that create values in the representation type. The representation is shared among the operations, but hidden from clients of the ADT.

Procedural data abstraction is organized around the constructors of the data abstraction. The observations become the attributes, or methods, of the procedural data values. Thus a procedural data value is simply defined by the combination of all possible observations upon it.