APPELBE84

Appelbe, William F. and A.P. Ravn. "Encapsulation Constructs in Systems Programming Languages". ACM Transactions on Programming Languages and Systems, Volume 6, Number 2, (April 1984). In this paper the authors discuss programming language constructs for encapsulation. They identify two main uses for these constructs: encapsulation of environments (for which they use packages) and constructing abstract data types (for which they use classes). They argue that these two uses are distinct and should be supported by distinct constructs in the language. They describe the properties these two constructs should have. They illustrate these points by considering the design and implementation of a model file system, patterned loosely on the UNIX* file system. They describe extensions to Ada+ to support their ideas, and then show how the model file system would be programmed in their extended Ada. Finally, they examine the encapsulation constructs in a few other programming languages intended for systems programming. The authors describe two kinds of encapsulation constructs. The "package" is used to enclose an environment: a static collection of declarations. Any kind of object can be exported from a package: constants, types, objects and subprograms. These exported declarations constitute the package's interface. The package also contains a collection of private declarations, called the body, that implement the interface. Examples of packages include an implementation of a "virtual machine", and a Logical I/O system. The other kind of encapsulation construct is the "class". The class is used to implement abstract data types. A class is a collection of data and operations. Taken together, these define a new type, and the exported operations define all the legal operations upon values of that type. The class declaration is then used to declare variables of the new type. A very important point is that *all* operations on variables of this new type must be defined by the class. This includes initialization, termination, assignment, copying and comparison. For example, consider an object of class type local to a procedure. When this procedure is called, the object is instantiated, and must be initialized. When the procedure exits, the object is destroyed, and resources may have to be reclaimed. Both of these operations must be under the control of the programmer of the class. If they are not, then the client must explicitly initialize the object, and explicitly destroy it, which is insecure. In summary, a class can only export operations, the declaration of a class introduces a new type, and there can be arbitrarily many instances of this type. A package can export any declaration, its declaration introduces an environment, and there can be only one instance of a particular package. Appelbe and Ravn argue that these two uses are different enough to warrant different language constructs and that a systems programming language should include both. They point out that it is possible to simulate a class using a package. However, they describe several problems with this simulation, primarily the lack of control over all operations. In [2], Wirth discusses the lack of a class construct in Modula, the predecessor to Modula-2. He asserts that typical packages in systems programming tend to be relatively large, with only one instance. He concludes that the class construct is not required for systems programming languages. Applebe and Ravn would argue that the class is needed in systems programming and that the language should guarantee the integrity of class objects. The authors describe a way of simulating packages with classes, but only if the classes can export declarations other than routines. Even in this case, there are problems with the security of the classes. They also describe other desired attributes of encapsulation constructs. The interface specification should be strictly separated from the implementation of that specification. This makes it possible to specify several different implementations for the same interface, which they use to advantage in their implementation of the physical I/O package. The authors proceed to illustrate these ideas via a model file system. They describe the file system's user interface, which consists of the new type *File*, and the operations upon it, embedded in a Logical I/O package. In order to implement the file system, they must choose a programming language. A key point of the paper is that no existing language contains the necessary encapsulation constructs. They chose to extend Ada with classes to satisfy the requirements set out at the beginning of the paper. Their motivation for choosing Ada is discussed briefly, and the reasons do not seem compelling. They chose Ada because it is "state-of-the-art", and has a "comprehensive, though informal, reference manual". The stated goal of the paper is "not to design a 'better' SPL ... Instead, ... the difficulties of implementing [the model] file system directly with existing SPLs are analyzed." However, the bulk of the paper is concerned with the implementation of the file system in extended Ada. Perhaps a smaller language, e.g., Modula-2, would have displayed the issues more sharply, allowing a more concise exposition. The syntactic extensions are indeed quite small, and are shown as modifications to the Ada grammar. The semantics of the constructs are described as the construction of the model file system proceeds. Various semantic and implementation issues are discussed as they arise during the exposition of the

implementation. "By-referene" parameter passing and assignment are discussed at length. The authors present a good argument for this, pointing out that the alternatives present serious problems for shared resources. Classes, as described here, do not require run-time support and the compilation overhead is minimal. They also discuss dynamic binding of class bodies to instances of the class. They motivate this very effectively as a natural way to provide the same user interface to files implemented on different kinds of devices. They discuss various ways of implementing dynamic binding. After presenting the model file system, the authors consider several existing programming languages which they consider representative of the encapsulation mechanisms currently available. The languages evaluated are Ada, Pascal Plus, Concurrent Pascal, Modula-2, Mesa and CLU. The evaluations are fairly short, and in each case the language is found to be inadequate. They argue convincingly that their proposals are more secure and more flexible than *private* types in Ada. Their criticisms of the other languages are valid considering the authors' stated goals. However, they incorrectly state that in Modula-2 the represntation and specification of modules are not separable. The authors conclude that no existing language provides the encapsulation constructs needed to directly and securely implement the model file system and other similar systems. They argue that better constructs, such as those they have designed are needed.

* UNIX is a trademark of Bell Laboratories
+ Ada is a trademark of the U.S. Department of Defense (AJPO)

References

[1] Wirth, N. The Module: A System Structuring Facility in High-Level Programming Languages. Institute fur Informatik, ETH-Zentrum, Zurich, Switzerland.
[2] Wirth, N. "Design and Implementation of Modula". Software - Practice and Experience, Volume 7, pp. 67-84 (1977).