Data Abstraction and Hierarchy

** This research was supported by the NEC Professorship of Software Science and Engineering.*

Barbara Liskov

Affiliation: MIT Laboratory for Computer Science
Cambridge, MA , 02139

ABSTRACT

Data abstraction is a valuable method for organizing programs to make them easier to modify and maintain. Inheritance allows one implementation of a data abstraction to be related to another hierarchically. This paper investigates the usefulness of hierarchy in program development, and concludes that although data abstraction is the more important idea, hierarchy does extend its usefulness in some situations.

1. Introduction

An important goal in design is to identify a program structure that simplifies both program maintenance and program modifications made to support changing requirements. Data abstractions are a good way of achieving this goal. They allow us to abstract from the way data structures are implemented to the behavior they provide that other programs can rely on. They permit the representation of data to be changed locally without affecting programs that use the data. They are particularly important because they hide complicated things (data structures) that are likely to change in the future. They also simplify the structure of programs that use them because they present a higher level interface. For example, they reduce the number of arguments to procedures because abstract objects are communicated instead of their representations.

Object-oriented programming is primarily a data abstraction technique, and much of its power derives from this. However, it elaborates this technique with the notion of "inheritance." Inheritance can be used in a number of ways, some of which enhance the power of data abstraction. In these cases, inheritance provides a useful addition to data abstraction.

This paper discusses the relationship between data abstraction and object-oriented programming. We begin in Section 2 by defining data abstraction and its role in the program development process. Then in Section 3 we discuss inheritance and identify two ways that it is used, for implementation hierarchy and for type hierarchy. Of the two methods, type hierarchy really adds something to data abstraction, so in Section 4 we discuss uses of type hierarchy in program design and development. Next we discuss some issues that arise in implementing type hierarchy. We conclude with a summary of our results.

2. Data Abstraction

The purpose of abstraction in programming is to separate behavior from implementation. The first programming abstraction mechanism was the procedure. A procedure performs some task or function; other parts of the program call the procedure to accomplish the task. To use the procedure, a programmer cares only about what it does and not how it is implemented. Any implementation that provides the needed function will do, provided it implements the function correctly and is efficient enough.

Procedures are a useful abstraction mechanism, but in the early seventies some researchers realized that they were not enough [15], [16], [7] and proposed a new way of organizing programs around the "connections" between modules. The concept of *data abstraction* or *abstract data type* arose from these ideas [5], [12].

Data abstractions provide the same benefits as procedures, but for data. Recall that the main idea is to separate what an abstraction is from how it is implemented so that implementations of the same abstraction can be substituted freely. The implementation of a data object is concerned with how that object is represented in the memory of a computer; this information is called the *representation*, or *rep* for short. To allow changing implementations without affecting users, we need a way of changing the representation without having to change all using programs. This is achieved by encapsulating the rep with a set of operations that manipulate it and by restricting using programs so that they cannot manipulate the rep directly, but instead must call the operations. Then, to implement or reimplement the data abstraction, it is necessary to define the rep and implement the operations in terms of it, but using code is not affected by a change.

Thus a data abstraction is a set of objects that can be manipulated directly only by a set of operations. An example of a data abstraction is the integers: the objects are 1, 2, 3, and so on and there are operations to add two integers, to test them for equality, and so on. Programs using integers manipulate them by their operations, and are shielded from implementation details such as whether the representation is 2's complement. Another example is character strings, with objects such as "a" and "xyz", and operations to select characters from strings and to concatenate strings. A final example is sets of integers, with objects such as { } (the empty set) and {3, 7}, and operations to insert an element in a set, and to test whether an integer is in a set. Note that integers and strings are built-in data types in most programming languages, while sets and other application-oriented data abstractions such as stacks and symbol tables are not. Linguistic mechanisms that permit user-defined abstract data types to be implemented are discussed in Section 2.2.

A data or procedure abstraction is defined by a *specification* and implemented by a program *module* coded in some programming language. The specification describes what the abstraction does, but omits any information about how it is implemented. By omitting such detail, we permit many different implementations An implementation is correct if it provides the behavior defined by the specification. Correctness can be proved mathematically if the specification is written in a language with precise semantics; otherwise we establish correctness by informal reasoning or by the somewhat unsatisfactory technique of testing. Correct implementations differ from one another in how they work, i.e., what algorithms they use, and therefore they may have different performance. Any correct implementation is acceptable to the caller provided it meets the caller's performance requirements. Note that correct implementations need not be identical to one another; the whole point is to allow implementations to differ, while ensuring that they remain the

same where this is important. The specification describes what is important.

For abstraction to work, implementations must be *encapsulated*. If an implementation is encapsulated, then no other module can depend on its implementation details. Encapsulation guarantees that modules can be implemented and reimplemented independently; it is related to the principle of "information hiding" advocated by Parnas [15].

2.1.
Locality

Abstraction when supported by specifications and encapsulation provides *locality* within a program. Locality allows a program to be implemented, understood, or modified one module at a time:

1. The implementer of an abstraction knows what is needed because this is described in the specification. Therefore, he or she need not interact with programmers of other modules (or at least the interactions can be very limited).

2. Similarly, the implementer of a using module knows what to expect from an abstraction, namely the behavior described by the specification.

3. Only local reasoning is needed to determine what a program does and whether it does the right thing. The program is studied one module at a time. In each case we are concerned with whether the module does what it is supposed to do, that is, does it meet its specification. However, we can limit our attention to just that module and ignore both modules that use it, and modules that it uses. Using modules can be ignored because they depend only on the specification of this module, not on its code. Used modules are ignored by reasoning about what they do using their specifications instead of their code. There is a tremendous saving of effort in this way because specifications are much smaller than implementations. For example, if we had to look at the code of a called abstraction, we would be concerned not only with its code, but also with the code of any modules it uses, and so on.

4. Finally, program modification can be done module by module. If a particular abstraction needs to be reimplemented to provide better performance or correct an error or provide extended facilities, the old implementing module can be replaced by a new one without affecting other modules.

Locality provides a firm basis for fast prototyping. Typically there is a tradeoff between the performance of an algorithm and the speed with which it is designed and implemented. The initial implementation can be a simple one that performs poorly. Later it can be replaced by another implementation with better performance. Provided both implementations are correct, the calling program's correctness will be unaffected by the change.

Locality also supports program evolution. Abstractions can be used to encapsulate potential modifications. For example, suppose we want a program to run on different machines. We can accomplish this by inventing abstractions that hide the differences between machines so that to move the program to a different machine only those abstractions need be reimplemented. A good

move the program to a different machine only those abstractions need be reimplemented. A good design principle is to think about expected modifications and organize the design by using abstractions that encapsulate the changes.

The benefits of locality are particularly important for data abstractions. Data structures are often complicated and therefore the simpler abstract view provided by the specification allows the rest of the program to be simpler. Also, changes to storage structures are likely as programs evolve; the effects of such changes can be minimized by encapsulating them inside data abstractions.

2.2.

Linguistic Support for Data Abstraction

Data abstractions are supported by linguistic mechanisms in several languages. The earliest such language was Simula 67 [3]. Two major variations, those in CLU and Smalltalk, are discussed below.

CLU [8][11] provides a mechanism called a *cluster* for implementing an abstract type. A template for a cluster is shown in Figure 2-1. The header identifies the data type being implemented and also lists the operations of the type; it serves to identify what procedure definitions inside the cluster can be called from the outside. The "**rep** = " line defines how objects of the type are represented; in the example, we are implementing sets as linked lists. The rest of the cluster consists of procedures; there must be a procedure for each operation, and in addition, there may be some procedures that can be used only inside the cluster.

```
int_set = cluster is create, insert, is_in, size,...

     rep = int_list

     create = proc ... end create

     insert = proc ... end insert

     ...

end int_set
```

Figure 2-1: Template of a CLU Cluster.

In Smalltalk [4], data abstractions are implemented by *classes*. Classes can be arranged hierarchically, but we ignore this for now. A class implements a data abstraction similarly to a cluster. Instead of the "**rep** =" line, the rep is described by a sequence of variable declarations; these are the *instance variables*. * *We ignore the class variables here since they are not important for the distinctions we are trying to make. CLU has an analogous mechanism; a cluster can have some "own" variables [9].* The remainder of the class consists of *methods*, which are procedure definitions. There is a method for each operation of the data type implemented by the class. (There cannot be any internal methods in Smalltalk classes because it is not possible to preclude outside use of a method.) Methods are

called by "sending messages," which has the same effect as calling operations in CLU.

Both CLU and Smalltalk enforce encapsulation but CLU uses compile-time type checking, while Smalltalk uses runtime checking. Compile-time checking is better because it allows a class of errors to be caught before the program runs, and it permits more efficient code to be generated by a compiler. (Compile-time checking can limit expressive power unless the programming language has a powerful type system; this issue is discussed further in Section 5.) Other object- oriented languages, e.g., [1], [13], do not enforce encapsulation at all. It is true that in the absence of language support encapsulation can be guaranteed by manual procedures such as code reading, but these techniques are error-prone, and although the situation may be somewhat manageable for a newly-implemented program, it will degrade rapidly as modifications are made. Automatic checking, at either runtime or compile-time, can be relied on with confidence and without the need to read any code at all.

Another difference between CLU and Smalltalk is found in the semantics of data objects. In Smalltalk, the operations are part of the object and can access the instance variables that make up the object's rep since these variables are part of the object too. In CLU, operations do not belong to the object but instead belong to a type. This gives them special privileges with respect to their type's objects that no other parts of the program have, namely, they can see the reps of these objects. (This view was first described by Morris [14].) The CLU view works better for operations that manipulate several objects of the type simultaneously because an operation can see the reps of several objects at once. Examples of such operations are adding two integers or forming the union of two sets. The Smalltalk view does not support such operations as well, since an operation can be inside of only one object. On the other hand, the Smalltalk view works better when we want to have several implementations of the same type running within the same program. In CLU an operation can see the rep of any object of its type, and therefore must be coded to cope with multiple representations explicitly. Smalltalk avoids this problem since an operation can see the rep of only one object.

3. Inheritance and Hierarchy

This section discusses inheritance and how it supports hierarchy. We begin by talking about what it means to construct a program using inheritance. Then we discuss two major uses of inheritance, implementation hierarchy and type hierarchy; see [18] for a similar discussion. Only one of these, type hierarchy, adds something new to data abstraction.

3.1. Inheritance

In a language with inheritance, a data abstraction can be implemented in several pieces that are related to one another. Although various languages provide different mechanisms for putting the pieces together, they are all similar. Thus we can illustrate them by examining a single mechanism, the subclass mechanism in Smalltalk.

In Smalltalk a class can be declared to be a *subclass* of another class * *We ignore multiple inheritance to simplify the discussion.*

mechanism is what code results from such a definition. This question is important for understanding what a subclass does. For example, if we were to reason about its correctness, we would need to look at this code.

From the point of view of the resulting code, saying that one class is a subclass of another is simply a shorthand notation for building programs. The exact program that is constructed depends on the rules of the language, e.g., such things as when methods of the subclass override methods of the superclass. The exact details of these rules are not important for our discussion (although they clearly are important if the language is to be sensible and useful). The point is that the result is equivalent to directly implementing a class containing the instance variables and methods that result from applying the rules.

For example, suppose class T has operations o_1 and o_2 and instance variable v_1 and class S, which is declared to be a subclass of T, has operations o_1 and o_3 and instance variable v_2. Then the result in Smalltalk is effectively a class with two instance variables, v_1 and v_2, and three operations, c_1, o_2, c_3, where the code of o_2 is supplied by T, and the code of the other two operations is supplied by S. It is this combined code that must be understood, or modified if S is reimplemented, unless S is restricted as discussed further below.

One problem with almost all inheritance mechanisms is that they compromise data abstraction to an extent. In languages with inheritance, a data abstraction implementation (i.e., a class) has two kinds of users. There are the "outsiders" who simply use the objects by calling the operations. But in addition there are the "insiders." These are the subclasses, which are typically permitted to violate encapsulation. There are three ways that encapsulation can be violated [18]: the subclass might access an instance variable of its superclass, call a private operation of its superclass, or refer directly to superclasses of its superclass. (This last violation is not possible in Smalltalk.)

When encapsulation is not violated, we can reason about operations of the superclass using their specifications and we can ignore the rep of the superclass. When encapsulation is violated, we lose the benefits of locality. We must consider the combined code of the sub- and superclass in reasoning about the subclass, and if the superclass needs to be reimplemented, we may need to reimplement its subclasses too. For example, this would be necessary if an instance variable of the superclass changed, or if a subclass refers directly to a superclass of its superclass T and then T is reimplemented to no longer have this superclass.

Violating encapsulation can be useful in bringing up a prototype quickly since it allows code to be produced by extension and modification of existing code. It is unrealistic, however, to expect that modifications to the implementation of the superclass can be propagated automatically to the subclass. Propagation is useful only if the resulting code works, which means that all expectations of the subclass about the superclass must be satisfied by the new implementation. These expectations can be captured by providing another specification for the superclass; this is a different specification from that for outsiders, since it contains additional constraints. Using this additional specification, a programmer can determine whether a proposed change to the superclass can usefully propagate to the subclass. Note that the more specific the additional specification is about details of the previous superclass implementation, the less likely that a new superclass implementation will meet it. Also, the situation will be unmanageable if each subclass relies on a different specification of the superclass. One possible approach is to define a single specification

for use by all subclasses that contains more detail than the specification for outsiders but still abstracts from many implementation details. Some work in this direction is described in [17].

3.2.

Implementation Hierarchy

The first way that inheritance is used is simply as a technique for implementing data types that are similar to other existing types. For example, suppose we want to implement integer sets, with operations (among others) to tell whether an element is a member of the set and to determine the current size of the set. Suppose further that a list data type has already been implemented, and that it provides a *member* operation and a *size* operation, as well as a convenient way of representing the set. Then we could implement set as a subclass of list; we might have the list hold the set elements without duplication, i.e., if an element were added to the set twice, it would be appear in the list only once. Then we would not need to provide implementations for *member* and *size*, but we would need to implement other operations such as one that inserts a new element into the set. Also, we should suppress certain other operations, such as *car*, to make them unavailable since they are not meaningful for sets. (This can be done in Smalltalk by providing implementations in the subclass for the suppressed operations; such an implementation would signal an exception if called.)

Another way of doing the same thing is to use one (abstract) type as the rep of another. For example, we might implement sets by using list as the rep. In this case, we would need to implement the *size* and *member* operations; each of these would simply call the corresponding operation on lists. Writing down implementations for these two operations, even though the code is very simple, is more work than not writing anything for them. On the other hand, we need not do anything to take away undesirable operations such as *car*.

Since implementation hierarchy does not allow us to do anything that we could not already do with data abstraction, we will not consider it further. It does permit us to violate encapsulation, with both the benefits and problems that ensue. However, this ability could also exist in the rep approach if desired.

3.3.

Type Hierarchy

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a *subtype* is one whose objects provide all the behavior of objects of another type (*the supertype*) plus something extra. What is wanted here is something like the following substitution property [6]: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T. (See also [2], [17] for other work in this area.)

We are using the words "subtype" and "supertype" here to emphasize that now we are talking about a semantic distinction. By contrast, "subclass" and "superclass" are simply linguistic concepts in programming languages that allow programs to be built in a particular way. They can be used to implement subtypes, but also, as mentioned above, in other ways.

We begin with some examples of types that are not subtypes of one another. First, a set is not a

subtype of a list nor is the reverse true. If the same element is added to a set twice, the result is the same as if it had been added only once and the element is counted only once in computing the size of the set. However, if the same element is added twice to a list, it occurs in the list twice. Thus a program expecting a list might not work if passed a set; similarly a program expecting a set might not work if passed a list. Another example of non-subtypes are stacks and queues. Stacks are LIFO; when an element is removed from a stack, the last item added (pushed) is removed. By contrast, queues are FIFO. A using program is likely to notice the difference between these two types.

The above examples ignored a simple difference between the pairs of types, namely related operations. A subtype must have all the operations ** It needs the instance methods but not the class methods.* of its supertype since otherwise the using program could not use an operation it depends on. However, simply having operations of the right names and signatures is not enough. (A operation's *signature* defines the numbers and types of its input and output arguments.) The operations must also do the same things. For example, stacks and queues might have operations of the same names, e.g., *add_ el* to push or enqueue and *rem_el* to pop or dequeue, but they still are not subtypes of one another because the meanings of the operations are different for them.

Now we give some examples of subtype hierarchies. The first is indexed collections, which have operations to access elements by index; e.g., there would be a *fetch* operation to fetch the i^{th} element of the collection. All subtypes have these operations too, but, in addition, each would provide extra operations. Examples of subtypes are arrays, sequences, and indexed sets; e.g., sequences can be concatenated, and arrays can be modified by storing new objects in the elements.

The second example is abstract devices, which unify a number of different kinds of input and output devices. Particular devices might provide extra operations. In this case, abstract device operations would be those that all devices support, e.g., the end-of-file test, while subtype operations would be device specific. For example, a printer would have modification operations such as *put_char* but not reading operations such as *get_char*. Another possibility is that abstract devices would have all possible operations, e.g., both *put_char* and *get_char*, and thus all subtypes would have the same set of operations. In this case, operations that not all real devices can do must be specified in a general way that allows exceptions to be signalled. For example, *get_char* would signal an exception when called on a printer.

An inheritance mechanism can be used to implement a subtype hierarchy. There would be a class to implement the supertype and another class to implement each subtype. The class implementing a subtype would declare the supertype's class as its superclass.

4. Benefits of Type Hierarchy

Data abstraction is a powerful tool in its own right. Type hierarchy is a useful adjunct to data abstraction. This section discusses how subtypes can be used in the development of the design for a program. (A detailed discussion of design based on data abstraction can be found in [11].) It also discusses their use in organizing a program library.

4.1.

Incremental Design

Data abstractions are usually developed incrementally as a design progresses. In early stages of a design, we only know some of a data abstraction's operations and a part of its behavior. Such a stage of design is depicted in Figure 4-1a. The design is depicted by a graph that illustrates how a program is subdivided into modules. There are two kinds of nodes; a node with a single bar on top represents a procedure abstraction, and a node with a double bar on top represents a data abstraction. An arrow pointing from one node to another means that the abstraction of the first node will be implemented using the abstraction of the second node. Thus the figure shows two procedures, P and Q, and one data abstraction, T. P will be implemented using Q (i.e., its code calls Q) and T (i.e., its code uses objects of type T). (Recursion is indicated by cycles in the graph. Thus if we expected the implementation of P to call P, there would be an arrow from P to P.)

This figure represents an early stage of design, in which the designer has thought about how to implement P and has invented Q and T. At this point, some operations of T have been identified, and the designer has decided that an object of type T will be used to communicate between P and Q.

The next stage of design is to investigate how to implement Q. (It would not make sense to look at T's implementation at this point, because we do not know all its operations yet.) In studying Q we are likely to define additional operations for T. This can be viewed as refining T to a subtype S as is shown in Figure 4-2. Here a double arrow points from a supertype to a subtype; double arrows can only connect data abstractions (and there can be no cycles involving only double arrows).

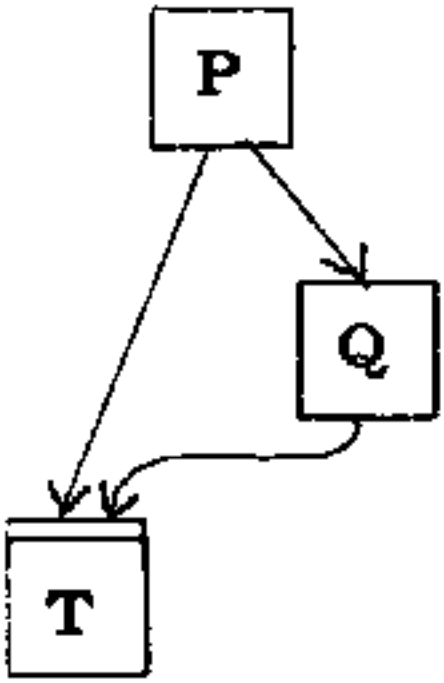

Figure 4-1: The Start of a Design.

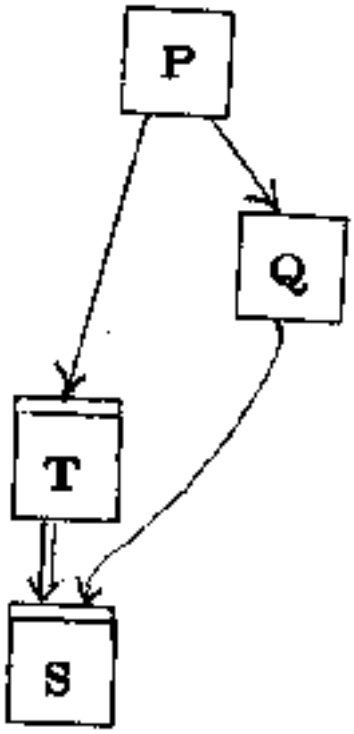

Figure 4-2: Later in the Design.

The kind of refinement illustrated in the figures may happen several times; e.g., S in turn may have a subtype R and so on. Also, a single type may have several subtypes, representing the needs of different subparts of the program.

Keeping track of these distinctions as subtypes is better than treating the group of types as a single type, for several reasons. First it can limit the effect of design errors. For example, suppose further investigation indicates a problem with S's interface. When a problem of this sort occurs, it is necessary to look at every abstraction that uses the changed abstraction. For the figure, this means we must look at Q. However, provided T's interface is unaffected, we need not look at P. If S and T had been treated as one type, then P would have had to be examined too.

Another advantage of distinguishing the types is that it may help in organizing the design rationale. The design rationale describes the decisions made at particular points in the design, and discusses why they were made and what alternatives exist. By maintaining the hierarchy to represent the decisions as they are made over time, we can avoid confusion and be more precise. If an error is discovered later, we can identify precisely at what point in the design it occurred.

Finally, the distinction may help during implementation, for example, if S, but not T, needs to be reimplemented. However, it may be that the hierarchy is not maintained in the implementation.

Frequently, the end of the design is just a single type, the last subtype invented, because implementing a single module is more convenient than having separate modules for the supertype and subtypes. Even so, however, the distinction remains useful after implementation, because the effects of specification changes can still be localized, even if implementation changes cannot. For example, a change to the specification of S but not T means that we need to reimplement Q but not P. However, if S and T are implemented as a single module, we must reimplement both of them, instead of just reimplementing S.

4.2.

Related Types

The second use of subtypes is for related types. The designer may recognize that a program will use several data abstractions that are similar but different. The differences represent variants of the same general idea, where the subtypes may all have the same set of operations, or some of them may extend the supertype. An example is the generalized abstract device mentioned earlier. To accommodate related types in design, the designer introduces the supertype at the time the whole set of types is conceived, and then introduces the subtypes as they are needed later in design.

Related types arise in two different ways. Sometimes the relationship is defined in advance, before any types are invented; this is the situation discussed above. Alternatively, the relationship may not be recognized until several related types already exist. This happens because of the desire to define a module that works on each of the related types but depends on only some small common part of them. For example, the module might be a sort routine that relies on its argument "collection" to allow it to fetch elements, and relies on the element type itself to provide a "<" operation.

When the relationship is defined in advance, hierarchy is a good way to describe it, and we probably do want to use inheritance as an implementation mechanism. This permits us to implement just once (in the supertype) whatever can be done in a subtype-independent way. The module for a subtype is concerned only with the specific behavior of that subtype, and is independent of modules implementing other subtypes. Having separate modules for the super- and subtypes gives better modularity than using a single module to implement them all. Also, if a new subtype is added later, none of the existing code need be changed.

When the relationship is recognized after the types have been defined, hierarchy may not be the right way to organize the program. This issue is discussed in Section 5.1.

4.3.

Organizing a Type Library

There is one other way in which hierarchy is useful, and this is to aid in the organization of a type library. It has long been recognized that programming is more effective if it can be done in a context that encourages the reuse of program modules implemented by others. However, for such a context to be usable, it must be possible to navigate it easily to determine whether the desired modules exists. Hierarchy is useful as a way of organizing a program library to make searching easier, especially when combined with the kind of browsing tools present, e.g., in the Smalltalk

environment.

Hierarchy allows similar types to be grouped together. Thus, if a user want a particular kind of "collection" abstraction, there is a good chance that the desired one, if it exists at all, can be found with the other collections. The hierarchy in use is a either a subtype hierarchy, or almost a subtype hierarchy a subtype differs from an extension of the supertype in a fairly minor way). The point is that types are grouped based on their behavior rather than how they are used to implement one another.

The search for collection types, or numeric types, or whatever, is aided by two things: The first is considering the entire library as growing from a single root or roots, and providing a browsing tool that allows the user to move around in the hierarchy. The second is a wise choice of names for the major categories, so that a user can recognize that "collection" is the part of the hierarchy of interest.

Using hierarchy as a way of organizing a library is a good idea, but need not be coupled with a subclass mechanism in a programming language. Instead, an interactive system that supports construction and browsing of the library could organize the library in this way.

5. Type Hierarchy and Inheritance

Not all uses of type hierarchy require language support. No support is needed for the program library; instead all that is needed is to use the notion of type hierarchy as an organizing principle. Support is also usually not needed for hierarchy introduced as a refinement technique. As mentioned earlier, the most likely outcome of this design technique is a single type at the end (the last subtype introduced), which is most conveniently implemented as a unit. Therefore any language that supports data abstraction is adequate here, although inheritance can be useful for introducing additional operations discovered later.

Special language support may be needed for related types, however. This support is discussed in Section 5.1. Section 5.2 discusses the relationship of inheritance to multiple implementations of a type.

5.1.

Polymorphism

A *polymorphic* procedure or data abstraction is one that works for many different types. For example, consider a procedure that does sorting. In many languages, such a procedure would be implemented to work on an array of integers; later, if we needed to sort an array of strings, another procedure would be needed. This is unfortunate. The idea of sorting is independent of the particular type of element in the array, provided that it is possible to compare the elements to determine which ones are smaller than which other ones. We ought to be able to implement one sort procedure that works for all such types. Such a procedure would be polymorphic.

Whenever there are related types in a program there is likely to be polymorphism. This is certainly the case when the relationship is indicated by the need for a polymorphic module. Even when the

relationship is identified in advance, however, polymorphism is likely. In such a case the supertype is often *virtual*: it has no objects of its own, but is simply a placeholder in the hierarchy for the family of related types. In this case, any module that uses the supertype is polymorphic. On the other hand, if the supertype has objects of its own, some modules might use just it and none of its subtypes.

Using hierarchy to support polymorphism means that a polymorphic module is conceived of as using a supertype, and every type that is intended to be used by that module is made a subtype of the supertype. When supertypes are introduced before subtypes, hierarchy is a good way to capture the relationships. The supertype is added to the type universe when it is invented, and subtypes are added below it later.

If the types exist before the relationship, hierarchy does not work as well. In this case, introducing the supertype complicates the type universe: a new type (the supertype) must be added, all types used by the polymorphic module must be made its subordinates, and classes implementing the subtypes must be changed to reflect the hierarchy (and recompiled in a system that does compilation). For example, we would have to add a new type, "sortable," to the universe and make every element type be a subtype of it. Note that each such supertype must be considered whenever a new type is invented: each new type must be made a subtype of the supertype if there is any chance that we may want to use its objects in the polymorphic module. Furthermore, the supertype may be useless as far as code sharing is concerned, since there may be nothing that can be implemented in it.

An alternative approach is to simply allow the polymorphic module to use any type that supplies the needed operations. In this case no attempt is made to relate the types. Instead, an object belonging to any of the related types can be passed as an argument to the polymorphic module. Thus we get the same effect, but without the need to complicate the type universe. We will refer to this approach as the *grouping* approach.

The two approaches differ in what is required to reason about program correctness. In either case we require the argument objects to have operations with the right signature and behavior. Signature requirements can be checked by type-checking, which can happen at runtime or compile time. Runtime checking requires no special mechanism; objects are simply passed to the polymorphic module, and type errors will be found if a needed operation is not present. Compile-time checking requires an adequate type system. If the hierarchy approach is used, then the language must combine compile-time type checking with inheritance; an example of such a language is discussed in [17]. If the grouping approach is used, then we need a way to express constraints on operations at compile-time. For example, both CLU and Ada [19] can do this. In CLU, the header of a *sort* procedure might be

```
sort = proc [T: type] (a: array[T])
        where T has lt: proctype (T, T) returns (bool)
```

This header constrains parameter T to be a type with an operation named *lt* with the indicated signature; the specification of *sort* would explain that *lt* must be a "less than" test. An example of a call of *sort* is

```
sort [int] (x)
```

In compiling such a call, the compiler checks that the type of *x* is array[int] and furthermore that int has an operation named *lt* with the required signature. We could even define a more polymorphic sorting routine in CLU that would work for all collections that are "array-like," i.e., whose elements can be fetched and stored by index.

The behavior requirements must be checked by some form of program verification. The required behavior must be part of the specification, and the specification goes in different places in the two methods. With hierarchy, the specification belongs to the supertype; with related types, it is part of the specification of the polymorphic module. Behavior checking also happens at different times. With hierarchy, checking happens whenever a programmer makes a type a subtype of the supertype; with grouping, it happens whenever a programmer writes code that uses the polymorphic module.

Both grouping and hierarchy have limitations. More flexibility in the use of the polymorphic module is desirable. For example, if *sort* is called with an operation that does a "greater than" test, this will lead to a different sorting of the array, but that different sorting may be just what is wanted. In addition, there may be conflicts between the types intended for use in the polymorphic module:

1. Not all types provide the required operation.

2. Types use different names for the operation.

3. Some type uses the name of the required operation for some other operation; e.g., the name *lt* is used in type T to identify the "length" operation.

One way of achieving more generality is to simply pass the needed operations as procedure arguments, e.g., *sort* actually takes two arguments, the array, and the routine used to determine ordering. * *Of course, this solution would not work well in Smalltalk because procedures cannot conveniently be defined as individual entities nor treated as objects.* This method is general, but can be inconvenient. Methods that avoid the inconvenience in conjunction with the grouping approach exist in Argus One way of achieving more generality is to simply pass the needed operations as procedure arguments, e.g., sort actually takes two arguments, the array, and the routine used to determine ordering. This method is general, but can be inconvenient. Methods that avoid the inconvenience in conjunction with the grouping approach exist in Argus [10] and Ada.

In summary, when related types are discovered early in design, hierarchy is a good way to express the relationship. Otherwise, either the grouping approach (with suitable language support) or procedures as arguments may be better.

5.2.

Multiple Implementations

It is often useful to have multiple implementations of the same type. For example, for some matrices we use a sparse representation and for others a nonsparse representation. Furthermore, it is

sometimes desirable to use objects of the same type but different representations within the same program.

Object-oriented languages appear to allow users to simulate multiple implementations with inheritance. Each implementation would be a subclass of another class that implements the type. This latter class would probably be virtual; for example, there would be a virtual class implementing matrices, and subclasses implementing sparse and nonsparse matrices.

Using inheritance in this way allows us to have several implementations of the same type in use within the same program, but it interferes with type hierarchy. For example, suppose we invent a subtype of matrices called *extended-matrices*. We would like to implement extended-matrices with a class that inherits from matrices rather than from a particular implementation of matrices, since this would allow us to combine it with either matrix implementation. This is not possible, however. Instead, the extended-matrix class must explicitly state in its program text that it is a subclass of sparse or nonsparse matrices.

The problem arises because inheritance is being used for two different things: to implement a type and to indicate that one type is a subtype of another. These uses should be kept separate. Then we could have what we really want: two types (matrix and extended-matrix), one a subtype of the other, each having several implementations, and the ability to combine the implementations of the subtype with those of the supertype in various ways.

6. Conclusions

Abstraction, and especially data abstraction, is an important technique for developing programs that are reasonably easy to maintain and to modify as requirements change. Data abstractions are particularly important because they hide complicated things (data structures) that are likely to change in the future. They permit the representation of data to be changed locally without affecting programs that use the data.

Inheritance is an implementation mechanism that allows one type to be related to another hierarchically. It is used in two ways: to implement a type by derivation from the implementation of another type, and to define subtypes. We argued that the first use is uninteresting because we can achieve the same result by using one type as the rep of the other. Subtypes, on the other hand, do add a new ability. Three uses for subtypes were identified. During incremental design they provide a way to limit the impact of design changes and to organize the design documentation. They also provide a way to group related types, especially in the case where the supertype is invented before any subtypes. When the relationship is discovered after several types have already been defined, other methods, such as grouping or procedure arguments, are probably better than hierarchy. Finally, hierarchy is a convenient and sensible way of organizing a library of types. The hierarchy is either a subtype hierarchy, or almost one; the subtypes may not match our strict definition, but are similar to the supertype in some intuitive sense.

Inheritance can be used to implement a subtype hierarchy. It is needed primarily in the case of related types when the supertype is invented first, because here it is convenient to implement common features just once in the superclass, and then implement the extensions separately for each subtype.

We conclude that although data abstraction is more important, type hierarchy does extend its usefulness. Furthermore, inheritance is sometimes needed to express type hierarchy and is therefore a useful mechanism to provide in a programming language.

Acknowledgments

Many people made comments and suggestions that improved the content of this paper. The author gratefully acknowledges this help, and especially the efforts of Toby Bloom and Gary Leavens.

REFERENCES

1. Bobrow, D., et al. "CommonLoops: Merging Lisp and Object-Oriented Programming". *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, SIGPLAN Notices 21,* 11 (November 1986).
2. Bruce, K., and Wegner, P. "An Algebraic Model of Subtypes in Object-Oriented Languages (Draft)". *SIGPLAN Notices* 21, 10 (October 1986).
3. Dahl, O.-J., and Hoare, C.A.R. Hierarchical Program Structures. In *Structured Programming*, Academic Press, 1972.
4. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Ma., 1983.
5. Hoare, C. A. R. "Proof of correctness of data representations". *Acta Informatica 4* (1972), 271-281.
6. Leavens, G. *Subtyping and Generic Invocation: Semantic and Language Design.* Ph.D. Th., Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, forthcoming.
7. Liskov, B. A Design Methodology for Reliable Software Systems. In *Tutorial on Software Design Techniques*, P. Freeman and A. Wasserman, Eds., IEEE, 1977. Also published in the Proc. of the Fall Joint Computer Conference, 1972.
8. Liskov, B., Snyder, A., Atkinson, R. R., and Schaffert, J. C. "Abstraction mechanisms in CLU". *Comm. of the ACM 20*, 8 (August 1977), 564-576.
9. Liskov, B., et al.. *CLU Reference Manual.* Springer-Verlag, 1984.
10. Liskov, B., et al. *Argus Reference Manual.* Technical Report MIT/LCS/TR-400, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1987.
11. Liskov, B., and Guttag, J.. *Abstraction and Specification in Program Development.* MIT Press and McGraw Hill, 1986.
12. Liskov, B., and Zilles, S. "Programming with abstract data types". *Proc. of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9* (1974).
13. Moon, D. "Object-Oriented Programming with Flavors". *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices 21,* 11

(November 1986).

14. Morris, J. H. "Protection in Programming Languages". *Comm. of the ACM 16*, 1 (January 1973).

15. Parnas, D. Information Distribution Aspects of Design Methodology. In *Proceedings of IFIP Congress*, North Holland Publishing Co., 1971.

16. Parnas, D. "On the Criteria to be Used in Decomposing Systems into Modules". *Comm. of the ACM 15*, 12 (December 1972).

17. Schaffert, C., et al. "An Introduction to Trellis/Owl". *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices 21*, 11 (November 1986).

18. Snyder, A. "Encapsulation and Inheritance in Object-Oriented Programming Languages". *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices 21*, 11 (November 1986).

19. 19. U. S. Department of Defense. *Reference manual for the Ada programming language*. 1983. ANSI standard Ada.