Algebraic Specification of Abstract Data Types

John Guttag
Professor and Head of Computer Science Department
Massachusetts Institute of Technology
guttag@mit.edu

Ph.D., University of Toronto

Co-leader,
MIT's Networks and
Mobile Systems Group

Member of the Board of Directors,
Empirix, Inc.

Member of the Board of Trustees,
MGH Institute of Health Professions

Major contributions:
abstract data types and specification techniques, the Larch family of specification languages and tools

Current interests: networking, wireless communication, medical computing

M. Broy, E. Denert (Eds.): Software Pioneers

John V. Guttag

Abstract Data Types, Then and Now

Data abstraction has come to play an important role in software development. This paper presents one view of what data abstraction is, how it was viewed when it was introduced, and its long-term impact on programming.

1. Abstract Data Types in a Nutshell

The notion of an abstract data type is quite simple. It is a set of objects and the operations on those objects. The specification of those operations defines an interface between the abstract data type and the rest of the program. The interface defines the behavior of the operations – what they do, but not how they do it. The specification thus defines an abstraction barrier (Figure 1) that isolates the rest of the program from the data structures, algorithms, and code involved in providing a realization of the type abstraction.

Abstraction is all about forgetting. There are lots of ways to model this. I find it easiest to model it on my teenage children. They conveniently forget

everything I say that they don't consider relevant. That's a pretty good abstraction process unless, as is often the case, their notion of relevance and mine conflict. The key to using abstraction effectively in programming is finding a notion of relevance that is appropriate for both the builder of an abstraction and the user of the abstraction. That is the true art of programming.

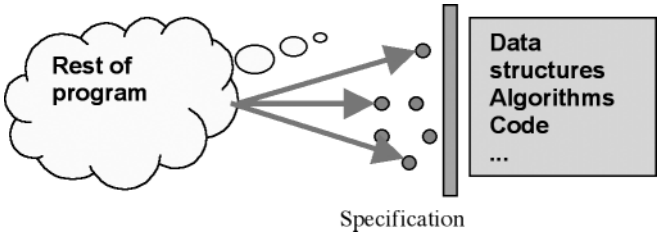

Fig.1. A data abstraction hides information about how a type is implemented behind an interface.

2. The 1970s

Who invented abstract data types? No one, really. Like so many timely ideas it was "in the air." Parnas was preaching the virtues of information hiding [14]. Dahl and Nygaard were using Simula 67 [2] to create rich sets of abstractions. Wirth [15] and Liskov [10] were demonstrating how programming languages could be used to protect programmers from themselves. Hoare was showing how monitors [9] could be used to structure programs and axiomatic systems used to reason about them [7, 8], etc.

It was an exciting time to be thinking about how people should go about building programs. In 1975, under the guidance of Jim Horning, I completed a dissertation [4] that started with:

At most points in the program, one is concerned solely with the behavioral characteristics of the data object. What one can do with them; not with how the various operations on them are implemented. I will use the term "abstract data type" to refer to a class of objects defined by a representation independent specification.

Today this idea is commonplace, and it seems perfectly obvious that abstract data types are among the most useful kinds of abstractions.

So, abstract data types were an indisputably good idea with an impeccable pedigree. One would have assumed that in the mid-1970s everybody would have stood up and saluted. Wrong. There was significant resistance in both academia and industry.

There were other, apparently competing, ideas in the air. There was a popular (and good) book called *Algorithms + Data structures = Programs* [16]. Conferences and journals were full of papers on successive refinement. Thoughtful students of project management argued that code should belong to everyone on the project. Many interpretations of these ideas were in direct conflict with the notion of organizing programs around abstract data types and consequently led people to conclude that abstract data types were just a bad idea.

How did people working on the same problem reach such different conclusions? My guess is that they started from different sets of (often not clearly articulated) assumptions.

3. Assumptions Underlying Abstract Data Types

An important, and arguable, assumption is that software is not fractal. This is in contrast to the fractal view of software, "it's the same thing all the way down." The fractal view leads one to treat the process of creating software as the same at each level, and from there to development methods in which the same process is applied recursively all the way from the early design phases to executable code. An alternative view is that high-level design is qualitatively different from low-level design. In particular, the design of a system architecture is very different from the design of an algorithm or data structure. This non-fractal view motivated much of the work on abstract data types.

Another assumption underlying a lot of the early work on abstract data types is that knowledge is dangerous. "Drink deep, or taste not the Pierian Spring," is not necessarily good advice. Knowing too much is no better, and often worse, than knowing too little. People cannot assimilate very much information. Any programming method or approach that assumes that people will understand a lot is highly risky. More importantly, the more widely information is disseminated, the more likely it is that decisions believed by designers and implementers to be local will have global ramifications. This makes change challenging. If information hiding and data abstraction had been widely understood and practiced in the 1970s, there would have been no Y2K problem in the 1990s. It should have taken but a few minutes to change the number of digits in dates, but it is no mystery why it didn't. Programmers did not use data abstractions, and too many people knew how dates were implemented. A clear and concise discussion of how views on information hiding have evolved since the 1970s can be found in the 1995 edition of *The Mythical Man Month* [1].

An important attribute of programming with data abstraction is that it limits direct access to key data structures. A perceived risk of data abstraction is that preventing client programs from directly accessing critical data structures leads to unacceptable loss of efficiency. In the early days of data abstraction, many were concerned about the cost of introducing extraneous procedure calls. Modern compilation technology makes this

concern moot. A more serious issue is that client programs will be forced to use inefficient algorithms. Consider a set of integers abstraction that supports only the operations *createEmpty*, *insert*, *delete*, and *elementOf*. There is no efficient way to write a client program that sums the elements of the set. One solution is to abandon the abstraction and allow the client program to directly access the data structure used to store the values of the set. A far better solution is to add an iterator operation [11] to the abstraction.

An assumption underlying approaches to programming that emphasize abstract data types is that performance is not primarily determined by local decisions. Building an efficient program rarely requires inventing clever data structures and algorithms. It more often depends upon developing a global understanding of the program that leads to a design that facilitates change. Certainly one needs to perform local optimization, but it is often difficult to know which things need to be optimized until relatively late in the day.

An apparent paradox of software is that it is so easy to replace that it is nearly immortal. It is not unusual for a programmer to be asked to maintain a program that originated before he or she was born. If one discovers that it is impossible to change some part of a program because the ramifications of that change cannot be ascertained, both the utility and the performance of that program will inevitably degrade over time – no matter how good the program was originally. Ultimately performance is determined by ease of repeated modification and optimization.

A final assumption underlying a belief in the usefulness of data abstraction is that inventing application-domain-specific types is a good thing, and a fixed set of type abstractions is not sufficient. That this assumption was not universally shared in the mid-1970s was driven home forcefully to me during my thesis defense. I vividly remember one of the examiners stating rather forcefully that he was quite certain that only a small set of types was needed. On another occasion, I recall someone saying something on the order of "I've looked at hundreds of thousands of lines of code and only eight types were ever used. Clearly that's all you need."

In 1980, I gave a short course at Softlab in Munich at the invitation of Ernst Denert. While preparing this paper, I uncovered the transparencies I used in that course. Apparently, I still found it necessary in 1980 to argue for the utility of inventing types. One of the (handwritten) transparencies contained the text, "Take a leap of faith, the programming process is facilitated by the introduction of new domains of discourse, it requires only imagination on the part of the designer."

Of course, it is possible to build too many highly specialized types. As it happens, many problems are not unique. They are similar to other problems, and one might just as well use the types that exist. Sets and numbers, vectors, and such can take one a long way.

4. From Assumptions to Abstract Data Types

From where did these assumptions arise? I think they came from trying to think about what's truly important in commercial software development. Time, cost, and quality matter. Truth and beauty are important, but they are means not ends. We would like our software to be elegant not because we are striving for elegance for its own sake, but because it helps us to build high-quality software on time and on budget. But, of course, even today software systems cost too much to build, take too long to build, and often do not do what people want.

Why is this? Most fundamentally, it is because programming is just plain hard. I don't think I understood this in the 1970s. At the time, I thought if only we could understand how best to go about building software it would become easy. I no longer believe that. I believe that we can improve our efficiency, but I don't think it will ever be easy to build interesting software systems. They are complex systems that people expect to be simultaneously reliable and infinitely malleable.

Programming is about managing complexity in a way that facilitates change. There are two powerful mechanisms available for accomplishing this, decomposition and abstraction. Decomposition creates structure in a program, and abstraction suppresses detail. The key is to suppress the appropriate details. This is where data abstraction hits the mark. One can create domain-specific types that provide a convenient decomposition. Ideally, these types capture concepts that will be relevant over the lifetime of a program. If, at the beginning, one devises types that will be relevant a decade later, one has a great leg up in maintaining that software.

Another thing that data abstraction achieves is representation independence. Think of the implementation of an abstract type as having several components:

- Implementations of the operations of the type,
- A data structure that holds values of the type, and
- Conventions about how the implementations of the operations are to use the data structure. The conventions are captured by the
    - Representation invariant and the
    - Abstraction function.

The representation invariant states which values of the data structure are valid representations of abstract values. Imagine implementing bounded sets of integers with lists of integers. The representation invariant might well restrict the lists that represent sets to those that contain no duplicates or perhaps to those in which the integers are in ascending order. The implementation of the operations would then be responsible for establishing and maintaining that invariant.

The abstraction function [8] maps values of the data structure implementing an abstract type onto the abstract values that the concrete value

represents. For example, if one uses a sorted list to implement a set, the abstraction function might map NIL to { } and the list containing the elements <1, 2, 2, 3> to the set {1,2,3}.

The commutative diagram in Figure 2 captures the notion of representation independence. The box in the lower left corner stands for a representation of an abstract value. One can go across the bottom and apply the code implementing an operation of the abstract type to that data structure and get another concrete value. If one then applies the abstraction function, one gets an abstract value.

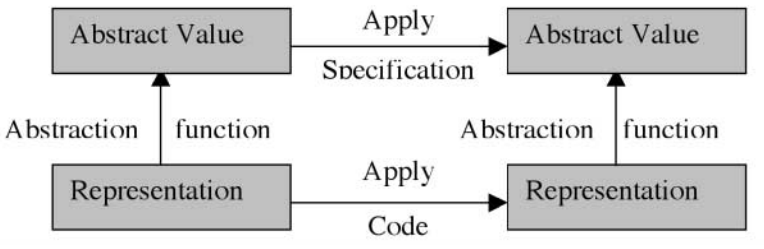

Fig. 2. A simplified commutative diagram. It is simplified in that it assumes that both specification and the code define a mapping (rather than a general relation).

The box in the upper left corner stands for the abstract value yielded by applying the abstraction function to the representation in the lower left corner. Think of applying an operation to that abstract value. The specification of the operation defines the abstract value that results. That each route yields the same abstract value indicates that the behavior of programs that rely on the interface provided by the abstract type is independent of the way the type is implemented. This is so because the interface to a module is characterized by the operations on that module.

Representation independence facilitates the construction of client programs, because those programming the clients need not be concerned with the representation of the abstract type. It also provides those implementing the abstract type with free reign to substitute one implementation for another.

5. Where Data Abstraction Has Led

Today, data abstraction is taken for granted. It is covered (more or less well) in many texts, e.g., [12], and supported (more or less well) by a number of programming languages, e.g., Java. I want to take this opportunity to speculate on some of the more indirect impacts of the work done on data abstraction. Cause and effect is rarely easy to prove, and I expect that some readers will see things quite differently.

The advent of data abstraction was a harbinger of a shift of emphasis in programming from decomposition, which dominated people's thinking for many years, to abstraction. This shift of emphasis appears in the literature on programming methods and in the design of programming languages.

A related development was an increased emphasis on interfaces; in particular on interfaces characterized by operations. The study of data abstraction led to a deeper understanding of the role of specification, including as an aid in designing and communicating interfaces [5].

The study of abstract data types led to a deeper understanding of types in general. In some sense, abstract data types existed from the very beginning of higher-level programming languages. FORTRAN programmers could ignore, at least some of the time, the way numbers were implemented. However, by better understanding such things as how to specify abstract types, how to support them in programming languages, and how to reason about them, we came to a deeper understanding of what types should be about. We came to understand the importance of things like representation invariants and abstraction functions. And, perhaps most importantly, we came to understand how to use induction to reason about types.

The data type induction principle [6] is quite simple, as illustrated in Figure 3.

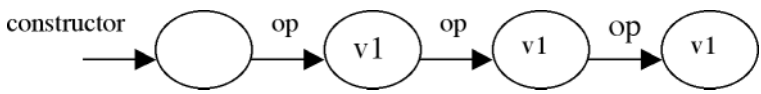

Fig. 3. Data type induction

Data type induction can be used to prove that something is always true about a type at the abstract level, i.e., an abstract invariant. It can also be used to prove representation invariants as part of the process of proving that an implementation of a type is correct. One starts with a constructor that creates some abstract value, *v1*, in this figure. One then applies an operation to get another abstract value, and so forth. The key is that the only abstract values that one can derive are those values that can be reached by applying a finite number of operations to the result of the constructor. This is so because the implementation is hidden, and (assuming the representation is not exposed through aliasing) there is no way to modify the data structures implementing the type except through the operations appearing in the interface. This yields an induction principle that allows one to reason about types. In short, data type induction provides for types what Floyd gave us for control structures. Floyd's inductive assertions method of reasoning [3] allows one to prove something about an infinite set of program executions in a finite number of proof steps. Data type induction allows one to do the same thing for data types.

Data abstraction encouraged program designers to focus on the centrality of data objects rather than procedures. That one should think about a program more as a collection of types than as a collection of procedures leads to a profoundly different organizing principle. It also encourages one to think about programming as a process of combining relatively large chunks, since data abstractions typically encompass much more functionality than do individual procedures. This, in turn, leads one to think of the essence of programming as a process not of writing individual lines of code, but of composing modules.

The reuse of relatively large modules is facilitated by flat program structures. The utility of flat structures was not obvious when abstract data types first came on the scene. As mentioned above, successive refinement was the rage. In some hands, it led to excessive levels of refinement. The problem with deep hierarchies is that, as one gets deeper in the hierarchy, there is a tendency to build increasingly specialized components. Flatter program structures typically lead one to build modules that assume relatively little about the context in which they are to be used. Such modules are more likely to be reusable.

The availability of reusable abstractions not only reduces development time, but also usually leads to more reliable programs because mature software is usually more reliable than new software. For many years, the only program libraries in common use were statistical or scientific. Today, however, there is a great range of useful program libraries, often based on a rich set of data abstractions.

6. Concluding Remarks

For the first 30 years of programming almost all of the real improvements in productivity were attributable to better tools, mostly better machines, and especially better programming languages. The movement from machine languages, to symbolic assembly languages, to higher-level programming languages produced huge improvements in programmer productivity.

In 1975, I thought that times were changing and that almost all future progress in software development would stem from innovations in the ways people thought about programs and programming and not from innovations in programming language technology.

I was not completely wrong. I was correctly pessimistic. There has not been much progress in programming languages since the late 1970s. If I look at the languages of today, I don't think that they are significantly better than some of the languages, e.g., CLU [11], that were available then. Certainly the compilers have gotten faster, they generate better code, and they do a bit more checking. But clearly nothing that has happened in the language domain has had anything like the impact of say FORTRAN, LISP, or Algol60. I was also correctly optimistic; there has indeed been considerable conceptual progress, e.g., the acceptance of abstract data types.

What I didn't anticipate was the enormous impact of improved hardware. This is a difficult thing for me to say as a computer scientist, but nothing in the last quarter century has had a comparable impact on programmer productivity and software utility. The vast memory and the fast processors in today's machines have done several things. Perhaps most importantly they have made it easier for us to use abstractions. The fact that we can so often ignore details of memory management has made it far easier to produce programs. In a similar vein, fast processors often allow one to use simple straightforward algorithms rather than the more subtle and efficient algorithms that were previously necessary.

What next? For one, we need to think about better ways to build embedded software. It seems likely that the majority of software will be part of embedded systems or at least connected to sensors or actuators. Such blue-collar software presents a number of challenges not usually present in white-collar software. Some of the difficulties are the difficulties of the past; embedded processors are often slow (by modern standards) and have little memory (again, by modern standards). The most fundamental aspect of this kind of software is that it will control systems that, with minimal or no human intervention, will take actions of critical importance. Such software will provide realtime control for automobiles, surgical tools, etc. The software will have to be immensely predictable and resilient. Static and dynamic analysis and principled testing of both designs and code will be essential.

Another challenge will be making everyone into a programmer. One of the reasons that programming is hard is that the programmer is often confused about what the user wants. The advent of spreadsheets solved this problem for one class of user. Tools like MatLab have helped to solve the problem for a different class of users. Many similar opportunities surely exist.

So where should you look for these things? Not in this volume. The software pioneers represented in this book did important work. For the most part, however, our day has passed. Look to the new pioneers.

Acknowledgements

Several people provided me with feedback both on earlier drafts of this paper and on a talk that preceded the paper. The overall shape and content of the paper owes much to the advice of Michael Ernst (one of the young pioneers to keep an eye on), Daniel Jackson (another young pioneer worth watching), Steve Garland, and John Ankcorn.

My thinking about data abstraction has been shaped by extensive interactions with some extraordinary colleagues. In particular, I owe much to Jim Horning, Dave Musser, and Steve Garland.

References

[1] F. Brooks, The Mythical-Man Month, Anniversary Edition: Essays on Software Engineering, Addison-Wesley (1995).

[2] O-J., Dahl, K. Nygaard, and B. Myhrhaug, "The SIMULA 67 Common Base Language," Norwegian Computing Centre, Forskningsveien 1B, Oslo (1968).

[3] R.W. Floyd, "Assigning Meaning to Programs," Proceedings of Symposium in Applied Mathematics, vol. IX, American Mathematical Society (1967).

[4] J.V. Guttag, The Specification and Application to Programming of Abstract Data Types, Ph.D. Thesis, Dept. of Computer Science, University of Toronto (1975).

[5] J.V. Guttag and J.J. Horning, "Formal Specification as a Design Tool," Seventh ACM Symposium on Principles of Programming Languages, Las Vegas (1980).

[6] J.V. Guttag, "Notes on Type Abstraction, Version 2," IEEE Transactions on Software Engineering vol. SE-6, no. 1 (1980).

[7] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," Communications of the ACM, vol. 12, no. 10 (1969).

[8] C.A.R. Hoare, "Proofs of Correctness of Data Representations," Acta Informatica, vol. 1, no. 4 (1972).

[9] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," Communications of the ACM, vol. 17, no. 10 (1974).

[10] B. Liskov and S.N. Zilles, "Programming with Abstract Data Types," Proceedings of ACM SIGPLAN Symposium on Very High Level Programming Languages, SIGPLAN Notices, vol. 9, no. 4 (1974).

[11] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. "Abstraction Mechanisms in CLU," Communications of the ACM, vol. 20, no. 8 (1977).

[12] B Liskov and J. Guttag, Program Development in Java; Abstraction, Specification, and Object-Oriented Design, Addison-Wesley (2000).

[13] J.H. Morris, "Types Are Not Sets," ACM Symposium on the Principles of Programming Languages (1973).

[14] D.L. Parnas, "Information Distribution Aspects of Design Methodology," Proceedings of IFIP Congress 1971 (1971).

[15] N. Wirth, "The Programming Language Pascal," Acta Informatica, vol. 1, no. 1 (1972).

[16] N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliff (1976).