2009

The Hidden Injuries Of Overloading 'ADT

Duane Buck
Otterbein University, DBuck@otterbein.edu

David J. Stucki
Otterbein University, DStucki@otterbein.edu

The Hidden Injuries of Overloading "ADT"

Duane Buck
Otterbein College
Mathematical Sciences Dept.
Westerville, OH 43081
614-823-1793
DBuck@Otterbein.edu

David J. Stucki
Otterbein College
Mathematical Sciences Dept.
Westerville, OH 43081
614-823-1722
DStucki@Otterbein.edu

ABSTRACT
The most commonly stated definition of *abstract data type* (ADT) is that it is a domain of values and the operations over that domain. So, for example, a language's built-in types, like int are seen to be ADTs. It is our opinion that a pure interpretation of this definition yields a semantics in which using an ADT is the same as using built-in types: the operations are side effect free and there is no concern over alias, shallow copy or synchronization problems. Unfortunately, the term abstract data type has over time been associated with at least three distinct meanings, and those incompatible definitions have often been conflated, causing confusion to students and textbook authors alike. We believe that this has resulted in a loss of appreciation for the value-based semantics of ADTs.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, classes and objects, data types and structures, modules.* E.1 [**Data**]: Data Structures – *lists, stacks, queues, and trees.* K.3.2 [**Computers and Education**]: Computer science education

General Terms
Performance, Design, Standardization, Languages.

Keywords
Abstract data types, containers, interfaces, modeling, data representation, value semantics.

1. INTRODUCTION
One of the inevitable consequences of growth and progress within any discipline, particularly one as young as computer science, is the corruption of terminology. As a concept evolves and is fleshed out (or as a technology changes) it will often retain its earliest designation, even when it develops well beyond the literal semantic connotation of this label. In many cases, this is immaterial. For example, modern machine execution models don't follow the simplistic description *fetch-execute cycle*; yet the term continues to prevail without ill effect. Occasionally however, it becomes necessary to recognize a particular use of language as abusive and bring to the community's attention its harmful effects.

SIGCSE'09, March 3–7, 2009, Chattanooga, Tennessee, USA.

The term *Abstract Data Type* has a long history, intertwined with the parallel developments of data representation, programming languages, data algebras, and object-oriented concepts. We do not intend to recapitulate that history here. However, we do feel that the current broad uses to which this term is applied are untenable, inconsistent, and ultimately damaging to students' understanding of best software practices.

In order to simplify our discussion, we would like to identify three broad genres that have been described as Abstract Data Types. First, in its purest usage, an ADT is defined to be a set of values along with the operations on those values. For the time being, let us call these *Platonic Types*. Secondly, it has become standard practice to denote classical data structures (stacks, list, trees, etc.) as ADTs. We will use the more recent label, *Containers*. From a theoretical perspective Containers can be subsumed under the first genre. However, for pragmatic reasons implementations usually utilize mutable objects so we will treat them separately. Finally, many authors have begun to use the term ADT more generally to describe classes that present a public interface and private implementation. This case represents such a diversity of application (e.g., interfaces in Java, packages in Ada, classes or user-defined types in language neutral texts, etc.) that there is not a uniform term we can identify, so we will use the term *interface* in its broad sense.

2. ABSTRACTION & REPRESENTATION
Before we examine these genres in detail, it is important to lay out the larger context of representation. When we model within a computer system, we are creating an abstraction. The model is abstract in that we only codify those aspects of the thing being modeled that are necessary to carry out the responsibilities of the system. Consider the following two general categories that are modeled within computer systems.

We owe our operational conceptualization of numerical values to an ancient philosophical tradition. It was the Greek thinker Plato who first described an ideal realm of intangible, immutable archetypes, of which numbers were only a special case. The majority of practicing mathematicians and scientists today continue to think of mathematical objects as existing in this *platonic universe*, having an abstract nature that is different in kind from physical entities. These abstract types are pure concepts and their constituent values are immutable, unchanging. The

integer 5 does not suddenly become the integer 7. Any domain of mathematically defined values thus exists in the platonic realm. For example, strings of symbols can be formalized as a mathematical domain (the language design choice that made String instances immutable in Java is consistent with this view).

On the other extreme of the spectrum are those entities that exist in the physical world. They are distinct from mathematical objects in that they are mutable through their interactions in the physical world. A student could change her name, and she would still be the same student. An automobile could be painted a different color and still be the same automobile. Software reuse and other engineering principles lead us to organize abstract taxonomies of these real-world entities (e.g., inheritance hierarchies of classes: Student, etc.). This significant distinction between platonic and physical entities should impact the way we model them in order to create more understandable and maintainable software.

We will argue that only models of the Platonic Type genre, whose constituent instances are immutable, should properly be called ADTs. We should note here that we recognize that the social momentum behind the current practices may be commensurate with the QWERTY keyboard, or the inclusion of GOTO statements in programming languages. This doesn't negate our argument in principle—it merely means that it may be too late to recall the overloaded term ADT and have the community use it as we argue it should properly be used. We will, however, argue that the conflation of different usages of ADT has caused confusion in the community and produces suboptimal results in software modeling. At the very least, elucidating the different kinds of models to which the term ADT is applied, correctly or in some sense incorrectly, will strengthen the modeling acumen of us and our students.

3. ADT: AN OVERLOADED MONIKER

First, let's get past a point on which there is little controversy. The most common definition for ADT is that it is a domain (a set of values) and the operations defined on that domain (See, for example [3, 6, 9, 12, 16]). Common examples found in textbooks are integers and strings, which are modeled in Java for instance as int (or Integer) and String. An operation for an int would be the dyadic function addition, while an operation for String would be the dyadic function concatenation. There are also monadic functions like negation and toUpperCase(). We agree that these are clearly ADTs.

This first definition is largely derived from the theory of types, and carries the pure mathematical quality of abstraction, or intangibility. When we build computer models, we must necessarily choose representations for these abstract types in terms of the primitives of the computational environment. We can observe two things about this situation. First, it really is a choice of representation, as there is always more than one feasible mapping from the abstract domain to the computational primitives (e.g., sign and magnitude vs. two's complement). Second, this choice is transparent to the resulting model. That is, there is an abstraction barrier separating the conceptual entities from their representation. The presence of both of these elements justifies use of the term *abstract data type*.

In light of this, it makes little sense to us to refer to models of physical entities as abstract data types. Physical entities simply don't fit well into the platonic universe of values. A Student class would not have an operator that takes in two students and produces a third student that is somehow a function of those two students (although we concede that there is a biological basis for that!). Likewise, there would not be a monadic operator that produces a new student that is somehow functionally related to that single student. Also, students change, and thus their representations are modified. Because the objects that represent a student are mutable, instances of the Student class are not representations of pure values, so how can they form the elements of a domain called for in the definition of an abstract data type? Therefore, models of physical entities are clearly not ADTs, although this use of the term is not uncommon.

Many textbooks (see [1, 2, 4, 16]) will skip over built-in ADTs, such as int or String, and immediately present Containers as the canonical examples of ADTs. Containers raise interesting issues because they are ambiguous relative to the platonic/physical distinction. It is possible to conceptualize Containers in a purely mathematical manner. This can be seen in the way that lists are implemented as immutable structures in pure functional languages like LISP. On the other hand, there are entities in the world, like box-office queues and cafeteria tray stacks, that bear a structural relationship to the mathematical abstraction, but which are, by their very natures, mutable. It is also interesting to note that the fact that Containers are defined as ADTs independent of the underlying type of their elements (à la generics) affords them a greater degree of abstraction than is implied by the term ADT. We will return to a discussion of implementation issues with containers below.

So, where does the controversy arise? There are several contexts in which the use of the term ADT can be confusing, misleading, or inconsistent.

Like so many issues in computer science, the first source of confusion arises from concerns of computational efficiency—specifically, Platonic Types for which representing values is "expensive." First, let's look at a Platonic Type for which Java has two implementations, one an ADT and the other not an ADT. Java implements the concept of string as an ADT using the String class. Operations defined on objects of that class treat the objects as values, never altering them and instead producing new objects functionally related to the inputs. However, it also supplies the StringBuffer class, whose instances are mutable and can denote different platonic strings at different times. This class can increase efficiency because in a long series of operations multiple String objects do not have to be created. Care must be taken not to treat the StringBuffer object as a platonic value, or confusion could result when it is modified.

In a pure ADT, an object is a surrogate for a particular value drawn from the domain, not a variable that can take on any value within the domain. Therefore a reference variable of the type has a bit pattern whose interpretation is unchanging, just as the bit pattern for an integer variable that holds the representation for 5 does not suddenly represent some other value, say 7. (This reminds one of the authors about a FORTRAN program that inadvertently passed the integer constant 5 to a subroutine that also used that parameter for output [all parameters in that version of FORTRAN were passed by reference]. The storage location in the constants table that should have always held 5 came to actually hold 7. Now, where the integer value (constant) 5

appeared in the code the value 7 was actually used. The debugging session was horrific.)

For types derived from pure mathematical domains, this choice to represent values with immutable objects ought to be natural, yet we have found it to be rare. One text that observes this discipline [6] has two well-crafted examples: 3D_Vector and Rational.

As a Container, the list is a common example of a Platonic Type in the sense that every possible list of elements can be viewed as a member of an ideal realm of list values that is not subject to the vagaries of the physical world. However, it is costly to model it in terms of explicit values and so it is commonly modeled in terms of mutable objects. For example, in most implementations that we have seen there is an `add()` method that rather than returning a new value (i.e., a new list containing the additional element), the method inserts the new element into the existing list. So at the implementation level, it becomes a hybrid that has some qualities of a platonic value but is also open to modification. Another way of looking at it is that its attachment to specific platonic values changes over time. That is, an object of the type points to one value in the platonic realm at any one time, but the value that it denotes changes as the result of operations. We know that platonic values do not change, but within this hybrid model we do not have a permanent way of denoting them. So, when we apply a function to a list, its "value" gets updated. Modeling in this way is certainly justified in many cases, and in fact can often be considered a best practice. There is nothing wrong with it as long as it is understood.

So we've seen both with strings and lists, that there is a design tension between value-based modeling and modeling with mutable objects. We feel that it is problematic to call both of these methods of modeling Platonic Types *abstract data types*. They are very different kinds of models whose usage requires a significantly altered approach. In particular, the latter approach does not have the same degree of transparency of representation as the former. Implementation details must be known and accounted for by the user of the type.

A second source of confusion is related to what we referred to above as the third genre of uses of the term ADT, in which any class that presents a public interface, in order to engage in data encapsulation and information hiding, can be called an ADT, even when it is representing physical entities. One objection to this is that although such a class possesses the same sort of abstraction barrier with implementation independence and user transparency that we discussed above, it does not map to a domain of values and so is not a data type in this narrow sense.

Further problems arise when these different definitions are conflated. One popular data structures text [3, p. 12] has the following introduction to ADTs:

> "An ***abstract data type*** consists of a collection of values, together with a collection of operations on those values. In object-oriented languages such as Java, abstract data types correspond to interfaces in the sense that, for any class that implements the interface, a user of that class can: (1) Create an instance of that class ('instance' corresponds to 'value') (2) Invoke the public methods of that class ('public method' corresponds to 'operation')."

Now if the class is modeling a Platonic Type with instances that have immutable state, then the above conflation of definitions is fine. But the very first example in the text following this definition(s) is an Employee class. But people are not values! Our students struggle to make sense of this inconsistency.

This is not an isolated incident. A random, small sampling of textbooks shows that it is not uncommon to equate ADTs and interfaces [7, 8, 13], ADTs and user-defined types [2, 5, 14], ADTs and classes [1, 11], and ADTs and Containers [12, 16]. In fact, many of these do not even restrict ADTs to classes that engage in encapsulation, but have public data elements. At this point the term ADT becomes so broad that it acquires that unfortunate status of meaningless buzz word.

4. "HIDDEN INJURIES"

It might be reasonable to pause at this point and say something like, "Well sure, the history and use of the term ADT is a bit confused and overloaded, but so what? What's the harm?" We feel that there is a real potential for harm in the following ways.

First, there is a temptation to allow both approaches to the modeling of Platonic Types, as values and as mutable objects, to fall under the same label. If we yield to this temptation we are masking the distinctions of mental discipline and acuity that are required in each case. For example, when modeling with values there is no danger in performing a shallow copy, whereas cloning would be necessary in the case of mutable objects. This represents a lack of transparency, requiring knowledge of implementation details to ensure correctness. Johannes J. Martin ([10, p. 97]) observed this danger more than two decades ago:

> "Viewing stacks (or other data forms) as mutable objects instead of values makes it necessary to introduce the implicit set of stores. The elements of this set are mappings (functions) that associate objects (containers) with their contents. The resulting axioms are more complicated than those used previously for the value interpretation. Allowing the dynamic creation of mutable objects leads to even more complexity. This must be considered a disadvantage of mutable objects, for simple semantic rules promote—and complex rules obstruct—both the correct implementation and use of a data type. Also, the fact that the set of stores does not explicitly appear in programs may be a possible source of error."

His margin notes are even more telling: "Mutable objects complicate specifications and call for discretion;…Deciding between values and objects deserves care." If one of the functions of language is to make distinctions apparent, then we do a disservice to students by not providing unambiguous terminology.

A second potential harm, related to references and aliasing, is observed by Louden [9, p. 393] in his popular programming languages text (`x` and `y` are instances of an Ada package `ComplexNumbers`).

> "Part of the problem comes from the use of assignment. Indeed, when `x` and `y` are pointer types, `x := y` performs assignment by sharing the object pointed to by `y`, with resulting potential unwanted side effects."

If ADTs were always models of Platonic Types, then there would be no risk of unwanted side effects.

Third, we have found that the lack of concern for separating these subtle distinctions in many textbooks has resulted in students who don't apprehend or appreciate the benefit of modeling with ADTs. They don't understand clearly the *relief of burden* that the value approach to modeling Platonic Types yields. This impairs their modeling faculties. For example, it is typical that once a student learns that objects passed to methods may be modified, they inevitably pass an Integer object and then try to set its value inside the method. Of course, Integer is an ADT and is therefore immutable, so the student finds that they cannot set its value. The student, having no appreciation of the great burden added when objects no longer represent values, assumes that whoever designed Integer must be either stupid or an egghead.

We find in [15, 16] a "Fraction" ADT that is a valuable example except that it inexplicably includes mutator methods in a class otherwise written in the style of a true Java ADT, like String. The standard Fraction operations, such as add, subtract, multiply, etc., return a result that is a new Fraction object; however each object is also mutable (e.g., there are setNumerator and setDenominator methods). Choosing to model a platonic type in this way, especially a simple numeric domain that doesn't introduce efficiency concerns, demonstrates a lack of contemplation in design. That a respected textbook author can have such a poor example in a popular textbook is a sign of the injury to the community caused by neglect to these issues.

5. CONCLUSION

There are three popular uses of the term ADT. The most often quoted, and we think the correct one, is that an ADT is a set of values and the operations on those values. The second usage identifies Containers as ADTs. That is somewhat understandable because indeed, there are such structured platonic values and operations. However, using the term ADT for Containers is problematic because for efficiency reasons Containers are most often mutable and therefore the representation is no longer value based; the "operations" have side effects that complicate the semantics. Finally, the third usage calls any interface with a private implementation an ADT. In this guise, ADTs are claimed to model real world entities in addition to platonic ones.

In his excellent programming languages text [9] Kenneth Louden recognizes the historical/developmental problems that have arisen from the varied uses of the term ADT. Although his motivation is in the context of language design, and his discussion is aimed at a somewhat different audience, some of the observations we have made here were anticipated even as far back as his 1st edition more than a decade ago. Louden stops short, however, of identifying the injuries caused when the overloaded definitions are conflated. During the intervening 14 years, the misuse of ADT has continued unabated.

In many ways the object-oriented approach to software construction followed a parallel evolutionary path to that of the structured programming movement, out of which the phrase abstract data type was coined. It is our belief that the vocabulary that has emerged from the object-oriented sector (e.g., *interface*) more appropriately characterizes the kind of modeling that occurs for real-world, physical entities. Further, the adoption of the label *container* is also an improvement for those data structures (e.g., list, stack, queue, map, set, etc.) that have historically been called ADTs. We therefore recommend that the term ADT be reserved for those Platonic Types that legitimately can be defined as a domain of values with a set of operations over the domain, *and* which are represented as values (i.e., immutable objects).

6. REFERENCES

[1] Aho, Alfred V., and Ullman, Jeffrey D. *Foundations of Computer Science*. W. H. Freeman, 1992.

[2] Brookshear, J. Glenn. *Computer Science: An Overview*, 6th ed. Addison-Wesley, 2000.

[3] Collins, William J. *Data Structures and the Java Collections Framework*, 2nd ed. McGraw-Hill, 2005.

[4] Dale, Nell, and Walker, Henry M. *Abstract Data Types: Specifications, Implementations, and Applications*. D. C. Heath and Company, 1996,

[5] Deitel, H. M., and Deitel, P. J. *Java: How To Program*, 7th ed. Prentice-Hall, 2007.

[6] Delillo, Nicholas J. *A First Course in Computer Science, with ADA*. Irwin, 1993.

[7] Goodrich, Michael T, and Tamassia, Roberto. *Data Structures and Algorithms in Java, 3rd ed*. John Wiley & Sons, 2003.

[8] Jia, Xiaoping. *Object-Oriented Software Development Using Java: Principles, Patterns, and Frameworks*, 1st ed. Addison-Wesley, 1999.

[9] Louden, Kenneth C. *Programming Languages: Principles and Practice*. Brooks/Cole, 2003; and PWS Publishing, 1993.

[10] Martin, Johannes J. *Data Types and Data Structures*. Prentice-Hall, 1986.

[11] Meyer, Bertrand. *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 2000.

[12] Musser, David R., Derge, Gillmer J., and Saini, Atul. *STL Tutorial and Reference Guide*, 2nd ed. Addison-Wesley, 2001.

[13] Sahni, Sartaj. *Data Structures, Algorithms, and Applications in C++*. McGraw-Hill, 1998.

[14] Schach, Stephen R. *Object-Oriented and Classical Software Engineering*, 7th ed. McGraw-Hill, 2006.

[15] Wu, C. Thomas. *An Introduction to Object-oriented Programming with Java*, 4th ed. McGraw-Hill, 2005.

[16] Wu, C. Thomas. *A Comprehensive Introduction to Object-Oriented Programming with Java*. McGraw-Hill, 2007.