Do We Really Teach Abstraction?

Paolo Bucci, Timothy J. Long, and Bruce W. Weide
Computer and Information Science
The Ohio State University
Columbus, OH 43210
{bucci,long,weide}@cis.ohio-state.edu

Abstract

Abstraction is one of the cornerstones of software development and is recognized as a fundamental and essential principle to be taught as early as CS1/CS2. Abstraction supposedly can enhance students' ability to reason and think. Yet we often hear complaints about the inability of CS undergraduates to do that. Do we supply students with the tools they need to reach their potential to think carefully and to reason rigorously about software behavior? Typically we do not, but as educators there are techniques we can use to help our students develop such skills starting in CS1/CS2.

1 Introduction

Abstraction and information hiding have long been recognized as fundamental and essential principles in software development, and are usually given positions of prominence in computer science curricula as early as CS1/CS2 [1, 2, 5]. Information hiding and abstraction can be thought of as complementary aspects of one general idea: hide the details of a complex system while providing a simpler "cover story" to explain those aspects of the system that are of interest. In this context then, we view abstraction as presenting a cover story for hidden detail.

Why are these ideas so important? As Don Norman [4] points out: "A good representation captures the essential elements of the event, deliberately leaving out the rest. ... The critical trick is to get the abstractions right, to represent the important aspects and not the unimportant. This allows everyone to concentrate upon the essentials without distraction from irrelevancies. Herein lie both the power and the weakness of representations: Get the relevant aspects right, and the representation provides substantive power to enhance people's ability to reason and think; get them wrong, and the representation is misleading, causing people to ignore critical aspects of the event or perhaps form misguided conclusions."

According to Norman, when done properly, abstraction introduces the possibility for completely new ways of thinking about the system being abstracted. The new representation enhances our ability to think about the external behavior of the original system through the use of terms that are fundamentally different from those in which its internal details would be explained. It is critical to note that the beneficiaries of both information hiding and abstraction are human beings; computers do not care at all about either.

It is the central claim of this paper that, despite broad agreement about their fundamental importance, both information hiding and abstraction are severely shortchanged by current CS1/CS2 pedagogy. More specifically, we claim that standard pedagogical practices tend to compromise the human dimension of information hiding and fall well short of helping student learners to realize an increased ability to reason and think *carefully* and *rigorously* as a benefit of using abstraction.

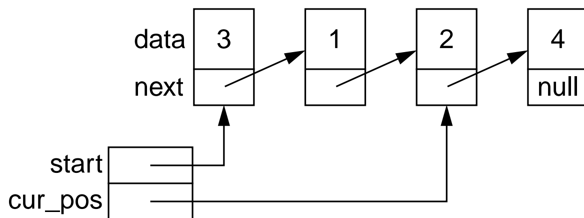
Our claim is perhaps surprising and certainly requires justification. After all, when we teach objects using, for example, abstract classes in C++ or interfaces in Java together with concrete classes, are we not teaching information hiding and abstraction? Clearly these programming-language mechanisms provide information hiding in a technical sense — access to concrete-class internals is restricted by the compiler. But what about the human dimension of information hiding? If our presentations to students of concrete classes immediately follow our presentations of abstract classes, inevitably information hiding will be compromised in the minds of the students, though not technically compromised in programs. Further, the programming-language mechanisms we teach do very little to encourage the creation of appropriate cover stories, other than to require operation prototypes. Operation prototypes alone do little to increase students' ability to reason and think about the behavior of operations being prototyped, even when combined with specification-by-wishful-naming. Of course, by standard practice we often include natural-language descriptions of the operations in addition to the prototypes. However, it is our contention that such descriptions still do not help students reach their potential to reason and think *carefully* and *rigorously* about the behavior of operations.

In the remainder of this paper, we present two examples in support of our contentions. The first example is a traditional container class that is typically presented in CS2. The second example is a simple class that might be presented quite early in CS1.

2 Traditional Presentation of Lists

For our first example, we consider a typical cover story for a list abstraction. Usually the data type is described first; that is, we start with a description of the values that list objects might assume. For example, we might say "a list is a linear sequence of items with a current position indicator". This is followed by the operation prototypes and descriptions of their behavior. Let's assume there are operations to insert and remove items from a list and to change the position indicator. Then the description of an insert operation might be "*Insert (l, x)* inserts item *x* in list *l* at the current position". (At times, requires and ensures clauses might be used to describe the effect of the Insert operation. Even so, the degree of formality is essentially what we have just stated.) Other operations would be described similarly.

The cover story above might be followed by an example use for lists. Whether an example is present or not, it is customary for CS2 books to present one or more implementations of lists in close textual proximity to the cover story. A frequently used implementation involves a singly linked list of nodes and pointers. Here is a possible representation of a list containing the integers 3, 1, 2, and 4, in that order, and with current position indicator on 2:



To begin to understand the pedagogical shortcomings of this presentation of lists, suppose that immediately following the presentation, we asked students to complete the following tracing table.

Statement	Object Values
object loi: List_Of_Integer **object** i: Integer i = 5	
	loi = ??? i = 5 **1**
Insert (loi, i)	
	loi = ??? i = ??? **2**

In the table, a list of integers, *loi*, and an integer, *i*, are declared, the value of *i* is set to 5, and the list operation Insert is invoked to insert the integer *i* into the list *loi*. If students were asked to fill in the values of the objects in box 1 after the declaration of *loi* and before the invocation of the Insert operation, and in box 2 after the call, what values would we expect our students to record?

It should not be surprising if some students drew pictures of a singly linked list representation of *loi* in both boxes. Why? Immediate presentation of one or more implementations of lists has compromised information hiding in students' minds and thus any abstraction has been lost [3]. Other students might attempt to record values for *loi* based on the description "linear sequence of items with a current position indicator". Even in this case, it is not clear exactly what students might write down. Why? Unless notation has been established for recording values of "linear sequences with a current position indicator", we should not be surprised to see a considerable variety in how students record values in the two boxes, even among students who are attempting to think in terms of the cover story.

At this point, it is tempting to dismiss concerns about how students might fill in the tracing table. Are uniformity and rigor in such tasks really that important? After all, students somehow "get" lists, even if each in their own way. However, suppose we asked students to trace through a much more substantial example. Would their informal intuitions and individualized notations serve them sufficiently well to achieve clear individual understanding? Would differences between individuals facilitate or impede clear communication between them? Does this presentation of lists employ abstraction in such a way as to provide "substantive power to enhance people's ability to reason and think"? We are convinced it does not.

3 Improved Presentation of Lists

In order to realize the full potential of abstraction as described by Norman, we need more explicit models of type values and of operation descriptions. One way to accomplish this is to use the time-tested and time-proven idea of mathematical modeling. Specifically, we can use mathematical types to describe (i.e., to model) program types. This powerful idea allows for exact descriptions of the values of complex objects (beyond built-in types) and of the behavior of operations on those objects. In CS1/CS2, almost all of the models we need to use in this approach are familiar to students, such as sets, functions, tuples, integers, etc.

We should be careful to introduce students to specific notation to be used with these models. The models then provide an excellent mechanism for creating cover stories. In a precise notation, all relevant information can be described while leaving out unimportant details. As Norman points out, extraneous detail can create confusion and can cause errors and misguided conclusions.

Here is the abstract description of a List component in a dialect of the RESOLVE language [6, 7]. The abstraction

employs mathematical modeling. Note that this is a generic component (i.e., a template) parameterized by the type of the items that can be inserted in a list (**type** Item).

```
concept List_Template (type Item)

   type List is modeled by
         (left: string of Item,
          right: string of Item)
      exemplar s
      initialization ensures
         |s.left| = 0 and |s.right| = 0

   operation Insert (updates s: List,
                     clears x: Item)
      ensures  s.left = #s.left and
               s.right = <#x> * #s.right

   operation Remove (updates s: List,
                     replaces x: Item)
      requires |s.right| > 0
      ensures  s.left = #s.left and
               #s.right = <x> * s.right

   operation Advance (updates s: List)
      requires |s.right| > 0
      ensures  |s.left| = |#s.left| + 1 and
               s.left * s.right =
                              #s.left * #s.right

   operation Reset (updates s: List)
      ensures  |s.left| = 0 and
               s.right = #s.left * #s.right

   operation Advance_To_End (updates s: List)
      ensures  |s.right| = 0 and
               s.left = #s.left * #s.right

   operation Left_Length (restores s: List)
         returns length: Integer
      ensures  length = |s.left|

   operation Right_Length (restores s: List)
         returns length: Integer
      ensures  length = |s.right|

end List_Template
```

The mathematical model of the type List is (left: **string of** Item, right: **string of** Item). This says that the value of a List object is an ordered pair of **string of** Item. **string** is a mathematical type describing ordered finite sequences of items. To denote strings, we use angle-bracket notation. For example, the notation <10, 20, 30> denotes the string containing the three integers 10, 20, and 30, in that order. The left and right strings in the model for List represent the items before and the items after the insertion/removal point of a List, respectively. The initial value of an object of type List is (< >, < >), that is, the pair of two empty strings.

The operations provided allow the client to insert and remove items from the list (specifically at the left end or front of the right string), to advance through the list one position at a time, to move directly to the front or the end of the list, and to obtain the lengths of the left and right strings. The pre- and post-conditions for the operations appear in requires and ensures clauses, respectively, and are statements from mathematical string theory. For example, the post-condition of Insert is $s.left = \#s.left$ **and** $s.right = \langle\#x\rangle * \#s.right$. In this equation, *#s* denotes the incoming value of parameter *s*, and *#s.left* and *#s.right* denote the left and right strings of the mathematical value of *#s*, respectively. Similarly, *s* denotes the outgoing value of parameter *s*. * denotes the concatenation operator for strings, and < > is a string constructor operation from items of type Item to strings of length one. So, *#x*, a value of type Item, is the incoming value of parameter *x*, while <*#x*> is the string of length one whose single entry is *#x*. As a specific example, if $\#s = (\langle 3, 1, 2\rangle, \langle 4\rangle)$, and $\#x = 5$, then after the Insert operation, $s = (\langle 3, 1, 2\rangle, \langle 5, 4\rangle)$. (The operation header for Insert specifies that parameter *x* will be cleared. This means that the outgoing value of *x* will be an initial value for its type.) Specifications of the other operations are similar.

The presence of a mathematical model for the program type List gives us a precise way of describing values of List objects and allows us to write precise specifications for the behavior of the operations, completely independent of any implementation details. The real power and leverage of abstraction can then be exploited to their full potential. The mathematical model and the operation specifications allow students to form an appropriate mental model of a List: what a List looks like and how the value of a list can be written down, and how each operation manipulates its parameters, in particular, objects of the List type. Armed with these tools, students can now easily trace through and understand statements involving List objects. For example, here is the earlier tracing table completed in terms of the mathematical model for List.

Statement	Object Values
object loi: List_Of_Integer **object** i: Integer i = 5	
	loi = (< >, < >) i = 5
Insert (loi, i)	
	loi = (< >, < 5 >) i = 0

Note that because the specification for Insert states that the second argument is cleared, the value of *i*, after the execution of Insert, is the initial value for an object of type Integer, which is 0.

We have used a simple example to illustrate how we can help students experience the added power of abstraction resulting from mathematical modeling and formal specifications. Our students use such tracing tables routinely to trace through much more complex code. An extended example appears in [7].

The list example may be suspect because it presents a traditional abstract data type for which abstract specifications are known. However, this does not alter the fact that most text-

books do not include precise models of lists. Our presentation of an exact model demonstrates that, as instructors, we can capture the power of abstraction for students in a very understandable way, at least for those software components traditionally taught as abstract data types.

4 A Flying_Robot Component

Our next example is intended to show that we can use mathematical modeling from the beginning to enhance students' ability to reason and think; that is, we do not need to wait until the chapter on abstract data types. We consider a fun component that might be presented quite early in CS1.

A Flying_Robot is an object designed to simulate a gadget that can get off the ground, fly around while in the air, and then land safely (e.g., by using a jet pack). It has operations to ascend to a fixed height (Ascend), to return to the ground (Descend), and to apply some acceleration in the direction it wants to go (Accelerate). Here is a typical informal description of the component.

```
component to simulate a flying robot

    A Flying_Robot can either be on the ground
    or in the air at a fixed height. When in
    the air, it can move in any direction on a
    horizontal plane.

operations

    Ascend (r)

        lifts the robot r from the ground
        to a fixed height in the air -
        the robot must be on the ground

    Descend (r)

        returns the robot r to the ground -
        the robot must be in the air and its
        speed must be less than 1

    Accelerate (r, ax, ay, dt)

        robot r accelerates by ax units in
        the x direction and by ay units in
        the y direction for dt time units -
        the robot must be in the air
```

Suppose we gave students this description of Flying_Robot and asked them to complete the following tracing table.

Statement	Object Values
object r: Flying_Robot	
	r = ???
Ascend (r)	
	r = ???
Accelerate (r, 5.0, 10.0, 1.0)	
	r = ???

As with the list example, it is not at all clear what students might record in the table. In fact, things might even be less certain in this case as compared with lists. Also, just as before, as instructors we might tend to dismiss the importance of students being able to complete such a table.

However, suppose that we tell students that the Flying_Robot *r* in the above table is flying over the surface of Mars and that there is a mountain range located close by at a specific location. Further, we ask students to determine if, after execution of the two operations in the table, the Flying_Robot will crash into the mountains. With the above presentation of Flying_Robot, have we, as instructors, provided students with a cover story that can be used to answer the question? Again, we have not.

Here is a precise description of the component, including a mathematical model and formal specifications for a few operations (the others have been omitted for brevity).

```
concept Flying_Robot_Template

   type Flying_Robot is modeled by
         (up: boolean,
          x, y: real,
          vx, vy: real)
      exemplar r
      initialization ensures
         r = (false, 0.0, 0.0, 0.0, 0.0)

   operation Ascend (updates r: Flying_Robot)
      requires r.up = false
      ensures  r = (true, #r.x, #r.y, 0.0, 0.0)

   operation Descend (updates r: Flying_Robot)
      requires r.up = true and
               r.vx * r.vx + r.vy * r.vy < 1.0
      ensures  r = (false, #r.x, #r.y, 0.0, 0.0)

   operation Accelerate (
                     updates r: Flying_Robot,
                     preserves ax, ay, dt: Real)
      requires r.up = true
      ensures  r.up = true and
               r.x = #r.x + #r.vx * dt +
                     ax * dt * dt / 2.0 and
               r.y = #r.y + #r.vy * dt +
                     ay * dt * dt / 2.0 and
               r.vx = #r.vx + ax * dt and
               r.vy = #r.vy + ay * dt

   ... other operations omitted for brevity ...

end Flying_Robot_Template
```

The mathematical model of a Flying_Robot has been chosen as a 5-tuple that makes explicit the information that is necessary to describe the state of a Flying_Robot: Is it in the air or not? What are its x and y coordinates? What is its velocity in the x and y directions? Given this model, it is possible to write down precise specifications for what Ascend, or Accelerate, or any other operation does, and it is then easy to fill out the tracing table (see next page).

After comparing these two presentations of Flying_Robot, it might be thought that we "stacked" the example with a deliberately inadequate first presentation. In fact, we did not, at least not intentionally. We suspect that many text-

Statement	Object Values
object r: Flying_Robot	
	r = (false, 0.0, 0.0, 0.0, 0.0)
Ascend (r)	
	r = (true, 0.0, 0.0, 0.0, 0.0)
Accelerate (r, 5.0, 10.0, 1.0)	
	r = (true, 2.5, 5.0, 5.0, 10.0)

book examples would be quite similar to our first presentation. We also suspect that any effort to make the first presentation more exact would ultimately lead to a mathematical model, or else how could we manage to say how the position or velocity of the robot are changed by the operations. As CS1/CS2 educators, should we not be doing this modeling in the first place?

A possible objection to the validity of this example is that, since this example is rooted in physics, mathematical modeling and equations of motion are, of course, the right approach to describing the component. But it is exactly because of how effective mathematical modeling is in providing abstraction (as witnessed by this example) that we should strive to use it to model components from other domains as well.

5 Conclusions

It is not unusual to hear faculty complaining about the inability of CS undergraduates to think and reason carefully and rigorously. Faculty often are dismayed that this is the state of affairs even among upper-level students. As CS1/CS2 educators, we might be part of the problem. When it comes to programming, student habits are often well established by the completion of CS2, at the latest. Do we, as CS1/CS2 educators, provide CS1/CS2 students with appropriate "mental" tools? Do we, for example, teach information hiding and abstraction so as to enhance students' ability to think and reason carefully and rigorously about programs, starting in CS1/CS2?

It may seem that an approach based on mathematical modeling and formal specifications is premature in CS1/CS2. However, in our experience after four years of teaching this approach, it is not. On exams, homeworks, and in-class activities, students routinely trace through code that they write or that we provide, proving that they have the appropriate conceptual model of the components used and that they have a clear understanding of the behavior of the operations involved. It is not surprising that even beginning students can do this for programs involving integers and other scalar types for which the mathematical models are so well-known that no one realizes they *are* models. The key thing is that CS1/CS2 students can do this for much more sophisticated program types; they are not intimidated by "mathematical modeling" in CS any more than in calculus or physics.

A simple, informal experiment at the beginning of the third quarter of our CS1/CS2 sequence gave us interesting and positive results. We asked about 40 students to "draw a picture of the value" of a queue, a set, a partial map (like a symbol table), and a list, given some code fragments that constructed values of these types using the operations available for each type. The overwhelming majority of the students wrote down the correct mathematical model values of the objects, even though we did not tell them which models to use. It is important to note that the students had been exposed to the client view of these components for the first 5 weeks of the previous course in the sequence, and had seen or written at least one implementation for each of these abstractions during the last 5 weeks of that course. Yet, when asked to draw the values of the objects, no student drew any implementation-related pictures or values.

Based on our experience [8], we are confident that real abstraction can be taught as early as CS1/CS2. Students at this level may not yet be conscious of the power of the ideas to which they are being exposed, but they can use them successfully and comfortably.

6 Acknowledgments

We gratefully acknowledge financial support from the National Science Foundation under grants DUE-9555062 and CDA-9634425, and from the Fund for the Improvement of Post-Secondary Education under project P116B60717.

References

[1] Astrachan, O.L., *A Computer Science Tapestry*. McGraw-Hill, 1997.

[2] Liskov, B. with Guttag, J. *Program Development in Java*. Addison-Wesley, 2001.

[3] Long, T.J., et al., Client View First: An Exodus From Implementation-Biased Teaching. In *Proc. 1999 ACM SIGCSE Symp.*, ACM, March 1999, pp. 136-140.

[4] Norman, D.R., *Things That Make Us Smart*. Perseus Books, 1993, p. 49.

[5] Shackelford, R.L., *Introduction to Computing and Algorithms*. Addison-Wesley, 1998.

[6] Sitaraman, M. and Weide, B.W., eds., Component-Based Software Using RESOLVE. *ACM Software Eng. Notes*, Vol. 19, No. 4, 1994, pp. 21-67.

[7] Sitaraman, M., et al., Reasoning About Software-Component Behavior. In *Proc. 6th Intl. Conf. on Software Reuse*, LNCS 1844, Springer-Verlag, 2000, pp. 266-283.

[8] Sitaraman, M., et al., *An Approach to Component-Based Software Engineering Education and Its Evaluation*. Tech. Report, Computer Science Dept., Clemson University, September 2000, 10 pages.