

TOYS ARE US: PRESENTING MATHEMATICAL CONCEPTS IN CS1/CS2

Paolo Bucci, Timothy J. Long, Bruce W. Weide¹
Joe Hollingsworth²

Abstract - *Presentation and use of formally-specified software components in CS1/CS2 presents interesting pedagogical challenges. Specifications may involve unfamiliar mathematical concepts and notation. We have found that the use of toys, such as stacking plastic cups and Lego® blocks, to be amazingly effective in helping students develop mental models for mathematical concepts. With the aid of these mental models, students are able to understand the behavior of software components through cover stories — their specifications — without knowing the implementations of the components.*

1. A PEDAGOGICAL CHALLENGE

System thinking offers a worldview that underlies all component-based engineering, including software engineering. By system thinking, we mean viewing or understanding things as units that can be viewed from the outside — the *client view* — as indivisible, or from the inside — the *implementer view* — as compositions of other systems, a.k.a. subsystems [4].

Our CS1/CS2 sequence is based first and foremost on system thinking. Throughout CS1, students act as clients of many “off-the-shelf” software components without seeing the implementations of any of the components. Instead, students are presented with a client description of each component, including a formal mathematical model of the values that objects can assume as well as formal pre- and post-conditions for all operations. Thus, from the very beginning of CS1, students are expected to read and understand formal mathematical descriptions of software components.

Not surprisingly, this commitment to formal specifications in CS1 and CS2 presents interesting pedagogical challenges. For example:

- Students may be asked to understand mathematical structures not yet introduced in any of their math courses, such as strings, binary trees, and weighted graphs.
- Even for mathematical structures that are familiar, students may be asked to understand new or different notation. Mathematical concepts are often presented in a highly symbolic form. Because of

this, something as simple as different syntax can give the appearance of a new mathematical concept.

- The whole idea of taking mathematics outside its usual context — the mathematics classroom — and using it to model other things presents an intellectual hurdle.

Other challenges may come to mind as well, not to mention the usual issue of students’ attitudes towards mathematics.

When using mathematics to formally describe software components, we are primarily interested in students being able to *read* mathematics as opposed to being able to write mathematics. This observation is crucial because of its consequences; namely, *the central pedagogical challenge is to help students formulate appropriate mental models of mathematical concepts so they can attach “meanings” to the symbols they are reading.* Through appropriate mental models, students can translate written symbols into meaningful representations of those symbols in their minds. In turn, they can formulate meaningful models for the behavior of software components.

2. A “LOW-TECH” APPROACH

The challenge just described will not surprise computer-science educators. After all, computer-science textbooks are full of pictures visualizing symbolic concepts, as are the chalkboards at the ends of our lectures [2]. Similarly, there are many clever computer animations of data structures and algorithms aimed at helping students visualize concepts otherwise presented symbolically (for example, [1, 3, 5]). In addition to this excellent work, we would like to suggest a “low-tech” approach to cultivating mental models of mathematical concepts: toys.

The use of physical manipulatives has been part of mathematics instruction for several decades. It has been shown that the use of manipulative materials can increase mathematics achievement and improve students’ attitudes towards mathematics [6]. It is interesting to note that the same study did not find significant differences between instruction with pictures and diagrams and instruction with symbols. There seems to be something special about the clear kinesthetic nature of physical manipulatives that gives them this edge with many students. We, too, have found the use of physical manipulatives, such as children’s stacking

¹ Department of Computer & Information Science, The Ohio State University, Columbus, Ohio 43210

² Computer Science Department, Indiana University Southeast, New Albany, Indiana 47150

cups and Lego[®] blocks, to be amazingly effective. When mathematical models for software-component behavior are demonstrated and practiced using manipulatives, heretofore uninterpreted symbols acquire actual intuitive meaning!

In this paper we will illustrate our ideas through three examples. Each example includes:

- a client description of a software component using formal mathematical models to describe component behavior, and
- a discussion of the physical manipulatives we use as metaphors to help students formulate appropriate mental models.

3. QUEUE MODELED BY MATHEMATICAL STRING

As a first example, we'll use a queue component. `Queue_Template` (see the Appendix) is parameterized by a type T , where T will be the client's choice for the type of item to go into a queue. The formal mathematical model for the values that queue objects can assume is **string of T** , often denoted T^* , consisting of all finite sequences with entries from T . The pre- and post-conditions for operations appear in requires and ensures clauses, respectively, and are statements from mathematical string theory. For example, the equation for the post-condition of `Add` is $q = \#q * \langle \#x \rangle$. In this equation, $\#q$ denotes the incoming value of parameter q , q denotes the outgoing value of parameter q , $*$ denotes the concatenation operator for strings, and $\langle \rangle$ is a string constructor operation from items in T to strings of length one. So, $\#x$, a value in T , is the incoming value of parameter x , while $\langle \#x \rangle$ is the string of length one whose single entry is $\#x$. As a specific example, if $\#q = \text{"abc"}$ and $\#x = \text{'d'}$, then after the `Add` operation, $q = \text{"abcd"}$. (The operation header for `Add` specifies that parameter x will be **consumed**. This means that the outgoing value of x will be an initial value for its type.) Specifications of the other operations are similar.

String theory is extremely useful for specifying software components. We use it for stacks, text strings, one-way and two-way lists, and tokenizing machines, for example. How do we help students in CS1 read and understand specifications involving string theory? We use stacking plastic cups, the kind sold in the toddler section of toy stores. Through in-class demonstrations using "strings of plastic cups", we can easily and quickly give meaning to the idea of a string of T and to equations such as $q = \#q * \langle \#x \rangle$. Figures 3.1 and 3.2 show the behavior of the `Add` operation.

By the way, plastic stacking cups also work great for demonstrating sorting algorithms (see Figures 3.3-3.7 for pictures of a queue being sorted with quicksort). Our experience is that students grasp the algorithms much more quickly, requiring less class time, and that their implementations of the algorithms are less likely to be defective.

4. PARTIAL MAP MODELED BY FINITE MATHEMATICAL FUNCTIONS

The partial map component is like a dictionary, symbol table, or look-up table. `Partial_Map_Template` (see the Appendix) is parameterized by two types, D and R , so that clients can customize the domain and range types for the map. The mathematical model for the programming language type `Partial_Map` is just a finite function, described here as a finite set of (d,r) -pairs with the function property (no d value is paired with more than one r value; see the **constraint**). Operation pre- and post-conditions are mathematical set-theory assertions. For example, consider the `Undefine` operation used to "undefine" the finite function m at d . The precondition for `Undefine` is `IS_DEFINED` (m,d) where `IS_DEFINED` is a mathematical definition stating that the value of d is in the domain of m . The post-condition for `Undefine` has three parts:

- (d,r) is in $\#m$ (the incoming value of m maps d to the outgoing value of r ; parameter d is **preserved**, so its incoming and outgoing values are the same)
- $m = \#m - \{(d,r)\}$ (the outgoing value of m is the incoming value of m without the pair (d,r))
- $d = d_copy$ (the outgoing value of d_copy will just be the "copy of d " that was in the incoming value of m and that m no longer needs).

The specifications of the other operations are similar.

To help students understand the mathematical model for the `Partial_Map` component, we use a clear plastic bag, like a sandwich bag, and Lego blocks. The plastic bag represents the set. Small Lego blocks are the domain items, and larger Lego blocks are the range items. When a "domain" Lego (D-Lego) is mapped to a "range" Lego (R-Lego), the D-Lego is snapped onto R-Lego and the pair is put into the plastic bag. It is easy to distinguish between a D-Lego and an R-Lego because they are different sizes, and it is easy to distinguish among D-Legos or R-Legos because they are different colors. Math operations such as `IS_DEFINED` (m,d) are easy to explain — just look into the plastic bag and see if there is a D-Lego equal to (i.e., the same color as) d . The function property is equally easy to explain — each color of D-Lego can appear in the bag at most once. Figures 4.1 and 4.2 show the behavior of the `Undefine` operation.

5. BINARY TREE MODELED BY MATHEMATICAL BINARY TREES

Our last example is a binary tree component. `Binary_Tree_Template` (see the Appendix) is parameterized by a type T so that clients can customize the type of items labeling the nodes of a binary tree. The mathematical model for the values that binary tree objects can assume is a mathematical **binary tree of T** , that consists of all mathematical binary trees labeled by values from T . The

pre- and post-conditions of operations are statements from binary-tree theory. Consider the Decompose operation, for example. The pre-condition is that $t \leq \text{empty_tree}$; that is, we cannot decompose an empty tree. The post-condition is $\#t = \text{COMPOSE}(x, l_tr, r_tr)$, where COMPOSE is a mathematical function in binary-tree theory. The post-condition just says that if we take the outgoing value of x (a value of type T) and the outgoing values of l_tr and r_tr (both of type binary tree of T) and compose them in the obvious way, the result is the incoming value of t . In other words, Decompose splits the incoming value of t into its three constituent parts. Also, since t is **consumed**, its outgoing value is the `empty_tree`.

To help students understand the mathematical concept of a binary tree, we use PVC pipe (white plastic pipe used for plumbing projects). The items in the tree are PVC joints with bright colors, such as red and blue. If an item, say a , is the left child of an item, say b , then there is a section of white PVC pipe connecting the two PVC joints for a and b and this represents the parent/child relationship in a binary tree. To illustrate the effect of COMPOSE, it is a simple matter to take two PVC models of binary trees and a new PVC-joint item, and “hook them up” with two new sections of pipe. Explaining the inverse operation and the recursive structure of binary trees is just as easy. Figures 5.1 and 5.2 show the behavior of the Decompose operation.

PVC toys also work great for illustrating binary tree operations such as inserting into a binary search tree or deleting from a heap. Item values can be written on post-it notes and stuck on the PVC joints. Then, the algorithm is “animated” by moving the post-it notes according to the movement of values by the algorithms. (PVC pipes and different kinds of joints are also useful when illustrating template instantiation, which is explained as analogous to plumbing. The details of this metaphor are a bit too complicated to include in this short paper.)

6. WHAT DO STUDENTS THINK?

Having been convinced in advance by the literature that physical manipulatives would be effective for us, we did not attempt to conduct controlled studies of student outcomes with and without them. Instead we played with various toys in class and, with student as well as instructor input, invented more and more ways to leverage them. Student reaction has been overwhelmingly positive. Our student surveys include the statement “Physical metaphors, such as plastic cups and Lego blocks, can help in understanding software.” The six possible responses range from strongly disagree to strongly agree. Cumulative data covering over 400 students and several different instructors, all of whom used this technique, show over 90% of students responding with moderately agree, agree, or strongly agree.

7. FINAL COMMENTS

Our use of toys and other physical manipulatives in CS1 and CS2 certainly reflects a spirit of light-heartedness. They are just plain fun to use in the classroom. At the same time, it is important to keep in mind that the component specifications we are asking students to understand are, in no sense of the word, light-hearted. The component designs presented here are of “professional” strength and reflect, without compromise, what we consider to be the right designs and right specifications, presented in full mathematical formality. We are continually impressed by our students’ ability to read and understand this type of formalism, as indicated by their ability to use software components (even on exams) given only their formal specifications, and especially by their lack of fear at the sight of such things. It is our sincere belief that the mental models formed by students through exposure to physical metaphors is what makes this understanding and attitude possible.

8. ACKNOWLEDGMENTS

We gratefully acknowledge financial support from our own institutions, 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 number P116B60717.

9. REFERENCES

- [1] Baker, R., M. Boilen, M. Goodrich, R. Tamassia, and B. Stibel, Testers and Visualizers for Teaching Data Structures, In Proc. 1999 ACM SIGCSE Symp., ACM, March 1999, pp. 261-265.
- [2] Goodrich, M. and Tamassia, R., Teaching the Analysis of Algorithms with Visual Proofs, In Proc. 1998 ACM SIGCSE Symp., ACM, February 1998, pp. 207-211.
- [3] Goodrich, M. and Tamassia, R., Data Structures and Algorithms in Java, John Wiley and Sons, New York, 1998.
- [4] Long, T. J., et al., Providing Intellectual Focus to CS1/CS2, In Proc. 1998 ACM SIGCSE Symp., ACM, February 1998, pp. 252-256.
- [5] Sangwan, R., J. Korsh, and P. LaFollette, Jr. A System for Program Visualization in the Classroom, In Proc. 1998 ACM SIGCSE Symp., ACM, February 1998, pp. 272-276.

[6] Sowell, E. J., Effects of Manipulative Materials in Mathematics Instruction, Journal for Research in Mathematics Education, 20(5), 1989, pp. 498-505.

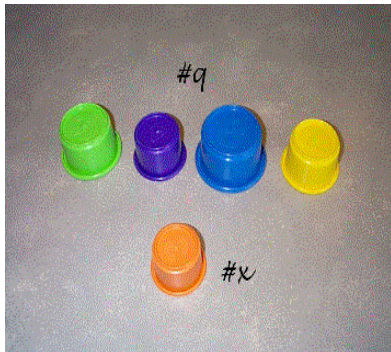


Figure 3.1. Add - before

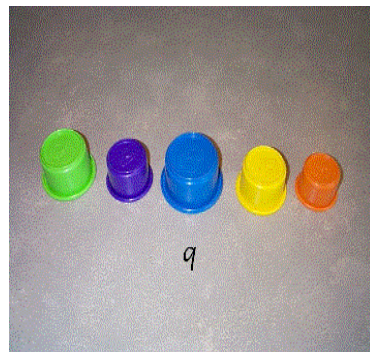


Figure 3.2. Add - after



Figure 3.3. Queue before Quicksort



Figure 3.4. Selecting partition element



Figure 3.5. Two queues after partition



Figure 3.6. Two queues sorted



Figure 3.7. Original queue sorted

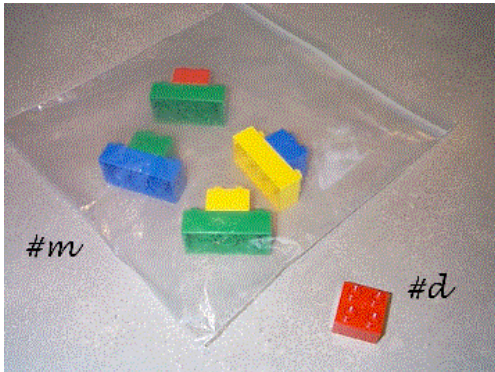


Figure 4.1. Undefine – before

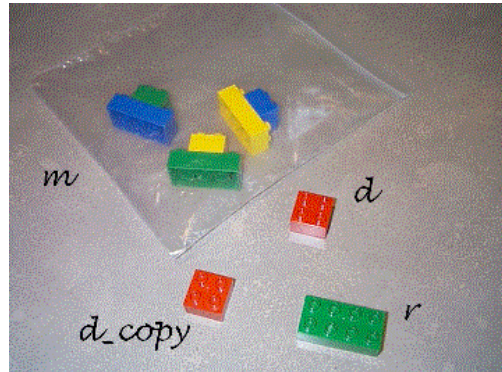


Figure 4.2. Undefine - after

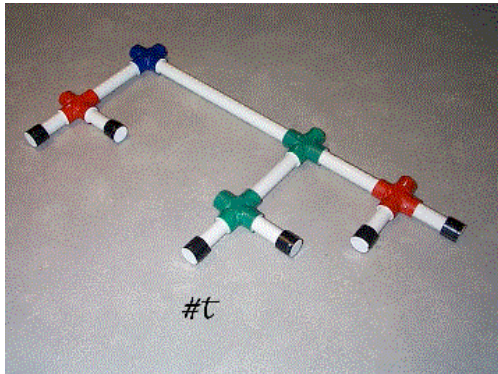


Figure 5.1. Decompose – before

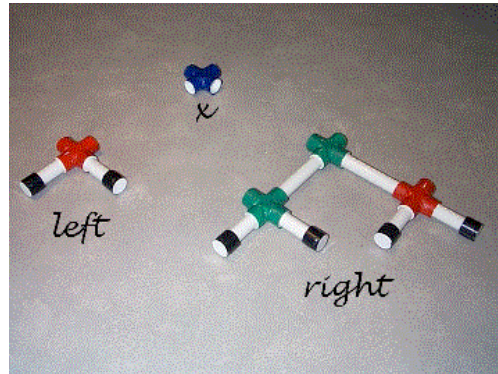


Figure 5.2. Decompose - after

Appendix:

Queue_Template:

parameters

type T

interface specification

type Queue = string of T;

Initialize as empty_string

operation Add (

alters q: Queue,
consumes x: T

);

ensures q = #q * <#x>

operation Remove (

alters q: Queue,
produces x:T

);

requires q <= empty_string

ensures #q = <x> * q

operation Size (

preserves q: Queue

):INTEGER;

ensures Size = |q|

Partial_Map_Template:

parameters

type D

type R

interface specification

type Partial_Map = finite set

of (d_value: D, r_value: R);

Initialize as empty_set

constraint d:D r1,r2:R

((d,r1) is in m and (d,r2)

is in m implies r1 = r2)

operation Define (

alters m: Partial_Map,

consumes d: D,

consumes r: R

);

requires not IS_DEFINED (m,d)

ensures m = #m union {(#d,#r)}

operation Undefine (

alters m: Partial_Map,

preserves d: D,

produces d_copy: D,

produces r: R

);

requires IS_DEFINED (m,d)

ensures (d,r) is in #m and
m = #m - {(d,r)} and

d = d_copy

operation UndefineAny (

alters m: Partial_Map,

produces d: D,

produces r: R

);

requires m <= empty_set

ensures (d,r) is in #m and

m = #m - {(d,r)}

operation Is_Defined (

preserves m: Partial_Map,

preserves d: D

): BOOLEAN;

ensures

Is_Defined = IS_DEFINED(m,d)

operation Size (

preserves m: Partial_Map

): INTEGER;

ensures Size = |m|

Binary_Tree_Template:

parameters

type T

interface specification

type Binary_Tree =

binary tree of T;

Initialize as empty_tree

operation Compose (

produces t: Binary_Tree,

consumes x: T;

consumes l_tr: Binary_Tree,

consumes r_tr: Binary_Tree

);

ensures

t = COMPOSE (#x,#l_tr,#r_tr)

operation Decompose (

consumes t: Binary_Tree,

produces x: T;

produces l_tr: Binary_Tree,

produces r_tr: Binary_Tree

);

requires t <= empty_tree

ensures

#t = COMPOSE(x,l_tr,r_tr)

operation Size (

preserves t: Binary_Tree

): INTEGER;

ensures Size = |t|