

# Client View First: An Exodus From Implementation-Biased Teaching

Timothy Long, Bruce Weide, Paolo Bucci

Computer and Information Science

The Ohio State University

Columbus, OH 43210-1277

+1-614-292-{1408,1517,0066}

{long,weide,bucci}@cis.ohio-state.edu

Murali Sitaraman

Computer Science and Electrical Engineering

West Virginia University

Morgantown, WV 26506

+1-304-293-0405

murali@csee.wvu.edu

## 1. ABSTRACT

**When teaching certain CS topics (e.g., abstract data types, operating systems), the instructor tries to make clear the distinction between the “client” perspective and the “implementer” perspective. But when teaching some programming language features and related programming techniques, this dichotomy often is not respected as strongly as it should be. We illustrate this with a discussion of how to teach recursion, comparing a traditional approach with one that is careful not to blur the distinctions between client view and implementer view. The latter better supports new learners in the creation of a sound and consistent mental model for developing and reasoning about programs that involve recursion.**

### 1.1 Keywords

Programming languages, recursion

## 2. INTRODUCTION

*System thinking* offers a world view that underlies all component-based engineering, including software engineering. By systems 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 [2]. A client is one who uses a system purely via an understanding of a “cover story” that purports to explain what the system *acts like* without re-

---

vealing precisely how it achieves that behavior. An implementer is one who needs to know precisely how the system *actually works* through composition of its subsystems.

Systems thinking is at the heart of many technical computer science topics, such as software development (e.g., abstract data types) and computer architecture and operating systems (e.g., virtual machines). It should be reflected in computer science pedagogy as well. From the perspective of a learner being introduced to a topic, an explanation that blurs the distinction between the client view and the implementer view would seem to promote confusion in the learner’s mind. The student must separate the two views in order to act and think appropriately, depending on whether his or her role is that of client or implementer. This same confusion can serve to inhibit development of clear mental models for the two viewpoints, and negatively impacts the student’s client and implementation skills.

### 2.1 Programming Language Features

A particularly interesting situation where systems thinking is appropriate involves programming languages. On the surface, a programming language seems like a classical virtual machine for which the client-side explanations of most constructs should be much simpler than how they are implemented. But consider how we usually teach such topics as procedures and functions, parameter passing, recursion, and iteration. What cover stories do we offer students who are acting as clients of specific programming language features when we teach the associated techniques for using these features? Here it is not clear that systems thinking has impacted pedagogy to the extent it might or should. The standard cover stories are entirely operational.

Perhaps this is acceptable. Many instructors believe students find operational explanations easier to grasp than purely abstract ones. And because the cover story must be sound — that is, it has to be an *undetectable* lie from the client’s point of view — it is easy for us as instructors to explain a feature by revealing in operational terms how it actually works in a language implementation. The main problem with this approach is that, among the many possible sound operational cover stories, some are easier than others for students to understand. The most comprehensible

cover story seems unlikely to be the “real story” because understandability of implementation is not a prime objective for compiler writers, who generally seek speed.

So, for such programming language features and associated programming techniques as those just listed, we should consider the following fundamental questions:

- What cover story best enables someone to use, and to reason about programs that involve, the feature?
- How is the feature implemented?

The first question ought to have answers independent of answers to the second. Yet, as teachers, how often do we achieve this independence? Can we step back from our own knowledge of how programming language features are implemented and create for students “cleaner” cover stories?

## 2.2 An Example: Recursion

In this paper we use recursion as a working example of a topic whose standard introductory explanation, in our opinion, has been overly influenced by the typical instructor’s knowledge of how the underlying programming language feature is implemented. We assume the perspective of a learner who is being introduced to recursion, and start by considering the first question above: What cover story best enables a student to use, and to reason about programs that involve, recursion? Once this question has been answered, and only then, do we consider teaching students an answer to the second question: How is recursion implemented? We present an approach to teaching recursion in which the client-side explanation is not implementation-biased, yet from which students can formulate and internalize a sound and consistent mental model of recursion that continues to make sense even after they learn how recursion is implemented. Our primary contributions are:

- explaining the advantages of presenting the client view first and making sure students understand it, before presenting the implementer view; and
- illustrating some ways of introducing recursive thinking, and reasoning about why and how recursion works, which reflect the client-implementer dichotomy.

We do not retrace ground already covered well elsewhere: a review of various approaches to teaching recursion [3], an explanation of the reasons why recursion is usually considered to be a difficult topic to teach [1], and a list of other background references [1,3].

Throughout this paper, when we refer to students or learners, we always mean students or learners at the introductory level, such as CS1/CS2 students. Also, we assume a spiral approach to teaching and learning. In a spiral approach, topics are revisited several times, usually with separation in time between successive visits. Subsequent visits develop a topic in greater depth.

## 3. RECURSIVE THINKING

A recursive implementation of an operation typically separates the processing of base-case from non-base-case values. Here we consider only processing of non-base-case values, which for simple situations has this familiar structure:

1. From incoming parameter values, extract one or more “smaller” values of the “same” kind.
2. Apply the operation (recursively) to each of the smaller values to obtain solutions for the smaller values.
3. Use the solutions for these smaller values to obtain a solution for the original incoming parameter values.

Inductively defined mathematical functions, such as factorial, Fibonacci numbers, and exponentiation, are popular first examples of recursion because their inductive definitions are easily translated into recursive operation bodies. However, these inductive definitions essentially are already-completed steps 1–3, so exclusive use of such examples might lead students to believe that successful use of recursion amounts to translation of inductive definitions into code. Instead, what students really need to understand is that successful use of recursion involves *discovering* appropriate inductive structure for themselves, and that such discovery can benefit from using specific strategies to examine the data to be processed.

Thus, we need to equip students with methods and strategies for discovering the answers to, and developing code that accomplishes, steps 1–3. Step 1 is the crucial step; once it is done properly, steps 2 and 3 are often quite routine. How, then, should a student go about identifying appropriate smaller values “within” incoming parameter values; that is, how should he or she go about identifying the recursive or inductive structure of incoming values? For an initial exposure to recursion, we recommend the use of pictures so there is a strong visual suggestion of recursive structure, as opposed to a merely symbolic connotation.

As a working example, we use the problem of reversing a text string. Here is a specification of the operation:

```
operation Reverse_A_Text_String (t: Text)
ensures t = REVERSE (#t)1
```

To avoid the pitfalls associated with factorial, etc., we specifically do *not* want to tell the student the following fact (theorem) about the mathematical *REVERSE* function:

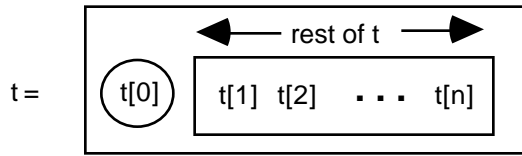
$$\text{REVERSE} (s * \langle x \rangle) = \langle x \rangle * \text{REVERSE} (s)$$

which is the basis for one of several possible recursive implementations.

---

<sup>1</sup> In this specification, the ensures clause  $t = \text{REVERSE}(\#t)$  is the postcondition.  $\#t$  denotes the incoming value of  $t$  while  $t$  denotes the outgoing value of  $t$ . *REVERSE* is a mathematical function on mathematical strings, i.e., the mathematical model of the programming type *Text*. *REVERSE* has the obvious meaning, e.g.,  $\text{REVERSE}(\text{"abc"}) = \text{"cba"}$ .

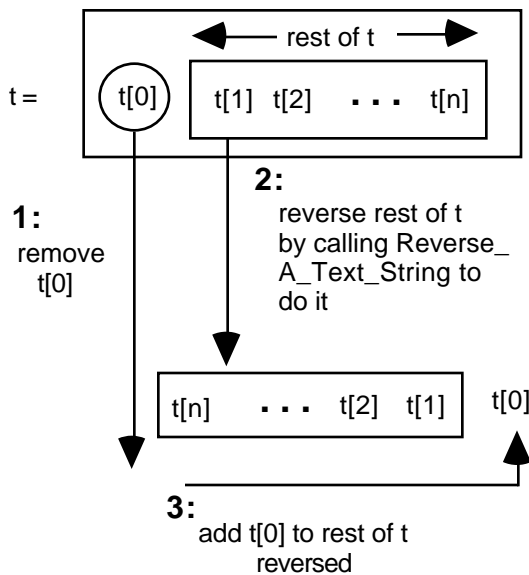
So, what do we say? One view of the recursive structure of the incoming value for parameter  $t$  is illustrated in Figure 1, where the larger, outer box represents the entire text string  $t$  to be reversed. Within this larger, outer box, the circle denotes the first character of  $t$ , and the smaller, inner box represents the rest of  $t$ . This use of boxes and circles is a powerful visualization of one way of identifying recursive structure in strings: a non-empty string consists of a first character followed by the rest of the string, which is itself a string that is merely shorter than the original.



Visualizing Recursive/Inductive Structure

Figure 1

To decide whether such a view of recursive structure is potentially useful, the student should next answer the *usefulness question*: If I had access to an operation that provided solutions for each of the smaller values, would this help me obtain a solution for the original incoming value? For the view in Figure 1, a “yes” answer should be fairly obvious.



Visualizing a Recursive Process

Figure 2

To answer the usefulness question positively, the student should have conceived a strategy for using solutions to smaller values to arrive at a solution for the original incoming value. We suggest that the student illustrate the strategy by annotating the picture of recursive structure. For the sample problem, one obvious annotation appears in Figure 2, where we have told the student that the name of the operation provided to reverse the smaller, inner box is *Reverse\_A\_Text\_String*, the very operation to be imple-

mented. It is a relatively small step from the annotated strategy in Figure 2 to this recursive operation body for *Reverse\_A\_Text\_String*:

```

procedure Reverse_A_Text_String (t: Text)
begin
  if (Length (t) /= 0)
    variable ch: Character
    Remove (t, 0, ch)
    Reverse_A_Text_String (t)
    Add (t, Length (t), ch)
  end if
end Reverse_A_Text_String2

```

Notice that this presentation respects the client-implementer dichotomy and treats the client-view first. There is no mention of how recursion is implemented.

#### 4. WHY RECURSION WORKS

For a typical student given the above introduction to recursive thinking, understanding *why* a recursive operation body is correct is at least as difficult as developing a recursive operation body. He or she usually is unsure of the resulting solution and lacks confidence that it is correct. The student, it seems, is searching for a simple, sound, probably operational cover story about what happens when an operation calls itself. But we recommend temporarily resisting the urge to sate the student’s appetite for an operational explanation of *how* recursion works, concentrating instead on an operational explanation of *why* it works.

For this approach to succeed, the instructor has to understand the subtle difference between the two issues: why vs. how. This is non-trivial because standard explanations of why and how a recursive operation body works often are intertwined with the single idea of “unrolling” recursive calls. Figure 3 is a typical textbook treatment, showing the sequence of calls and returns resulting from a call to *Reverse\_A\_Text\_String*. Depictions of calls and returns are but slightly simplified descriptions of the call-stack mechanism that languages use to implement recursion.

Technically, by learning how recursion works a student can use the mechanism to try to understand why a particular recursive operation body is correct. But this approach is troublesome in several ways. First, as a matter of principle, if a student’s first exposure to recursive thinking is from the client perspective, then reasoning about the effects of executing recursive operation bodies should not need to be based on how recursion is actually implemented. Second, the traditional explanation in Figure 3 is complicated, involving the idea of suspended calls to *Reverse\_A\_Text\_String*; of those calls resuming execution in a specific way and in a specific order; and of variables coming into and out of scope; there should be simpler cover story about

<sup>2</sup> Informally, operation *Remove* ( $t$ , 0,  $ch$ ) removes the leftmost character from  $t$ , and that becomes the value of  $ch$ . *Add* ( $t$ ,  $Length$  ( $t$ ),  $ch$ ) appends  $ch$  onto the right end of  $t$ .

how recursion works. Third, and most importantly, the pattern of recursive calls and returns presents a model of recursion that, on the surface, is inconsistent with inductive thinking. This could have a negative impact on the student's ability to formulate a consistent mental model of recursion and therefore on the ability to discover recursive solutions to new problems.

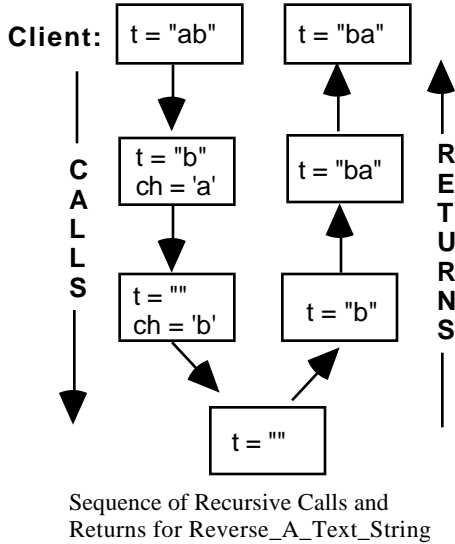


Figure 3

The last concern just raised may not seem compelling yet, but it gets to the heart of the difference between why recursion works and how it works, so we explore it further. The mental processes that a student needs to master in developing recursive operations are, of course, closely related to the mental processes involved in proofs by mathematical induction. In the usefulness question, the assumption that an operation is available to find solutions for the smaller values is but a thinly disguised form of an inductive hypothesis, and the processing of non-base-case values in the form of steps 1–3 is but a thinly disguised proof of the inductive step, where step 2 involves application of the inductive hypothesis. Thus, when we think of a student searching for an explanation of why a recursive solution is correct, we can reinterpret this as the student searching for an explanation of why a proof by mathematical induction is valid.

The inductive step of proof by induction is both extremely powerful and highly abstract. Yet, it merely captures, in a clever way, a simple pattern of reasoning. For the natural numbers, this pattern of reasoning goes as follows: Establish truth at 0; then establish that truth at 0 implies truth at 1 and thereby conclude truth at 1; then establish that truth at 1 (or, at 0 and at 1) implies truth at 2 and thereby conclude truth at 2; and so on. In the case of a recursive operation body, the pattern of reasoning might be similar: Establish the code's correctness at 0; then establish that correctness at 0 implies correctness at 1 and thereby conclude correctness at 1; then establish that correctness at 1 (or, at 0 and at 1) implies correctness at 2 and thereby conclude cor-

rectness at 2; and so on.<sup>3</sup> This is a pattern of reasoning that students can grasp, and it points the way to an appropriate operational cover story about recursion that stops well short of the complications of Figure 3.

To explain why the implementation of *Reverse\_A\_Text\_String* is correct, this pattern of reasoning would be applied as follows: *Reverse\_A\_Text\_String* is first traced on the base-case value of *t* being the empty string. Then *Reverse\_A\_Text\_String* is traced on a text string of length 1. In this trace, the effect of the recursive call is known by the previous trace where the value of *t* was the empty string. *Reverse\_A\_Text\_String* is then traced on a text string of length 2. Again, the effect of the recursive call in this trace is known by the previous trace. Tracing can continue like this (Figure 4) until the student is completely comfortable with the pattern of reasoning and is confident that *Reverse\_A\_Text\_String* will work correctly for all text strings. Notice how this tracing-smallest-values-first explanation captures the essence of inductive reasoning, and that the student can do this form of reasoning without knowing how recursion actually works.

Statement	Object Values
	t = "abc"
<b>if</b> (Length (t) /= 0) <b>variable</b> ch: Character	
	t = "abc" ch = ' '
Remove (t, 0, ch)	
	t = "bc" ch = 'a'
Reverse_A_Text_String (t)	
	t = "cb" ch = 'a'
Add (t, Length (t), ch)	
	t = "cba" ch = 'a'
<b>end if</b>	
	t = "cba"

Figure 4

We can now compare the explanation in Figure 3 with the tracing-smallest-values-first explanation. The latter is not couched in implementation terms; there is no hint of a call stack of suspended executions and no need for an explanation of execution resuming in a particular way and in a particular order. Also notice that the sequence of calls in Figure 3 goes in exactly the *opposite* direction of inductive reasoning; it is the sequence of returns that go in the same direction as inductive reasoning. Having an explanation of recursion that explicitly reveals dynamic behavior in two opposing directions must surely be more difficult to inter-

<sup>3</sup> Here 0, 1, 2, etc., represent the "sizes of the incoming values" and depend on the kind of problem, e.g., for *Reverse\_A\_Text\_String* size would be the length of *t*.

nalize than a non-dynamic explanation that is completely consistent with induction and thus completely consistent with the method described in Section 3 for discovering recursive structure and developing recursive operation bodies.

A student with enough experience with recursion to understand the inductive model can next try to make a convincing case for the correctness of a recursive implementation by explicitly using proof by induction. In particular, this can involve appeal to the inductive hypothesis, as specified in the postcondition of an operation, for the result of a recursive call. This approach removes the need to trace recursion, either by unrolling recursive calls as in Figure 3 or by tracing in the direction of smaller values to larger values.

## 5. HOW RECURSION WORKS

Even after using the above approach to make a convincing case that a recursive operation body is correct, the typical student still wants an operational picture of *how* recursion works. Nearly all explanations of this involve the call-stack mechanism. Details in textbooks vary widely, from informal treatments as in Figure 3 to thorough discussions of activation records and the run-time stack.

We recommend a particularly understandable yet sound explanation that leverages the idea of tracing tables (e.g., Figure 4). Consider a client call to *Reverse\_A\_Text\_String*, say with the incoming value of  $t = \text{"abc"}$ . The student can begin filling in a tracing table for *Reverse\_A\_Text\_String* up to the recursive call. At this point, the current tracing table is temporarily set aside on a pile of partially-completed tracing tables and a new tracing table is started with the incoming value of  $t = \text{"bc"}$ . As this continues, the student creates a “stack” of temporarily-set-aside tracing tables with incoming values  $t = \text{"abc"}$ ,  $t = \text{"bc"}$ ,  $t = \text{"c"}$ , until the current tracing table has  $t = \text{""}$ . This last tracing table can be completed and the result copied back to the point of the recursive call in the top tracing table for  $t = \text{"c"}$ , and so on. This explanation of how recursion works is simple and clean, with each recursive call resulting in a temporary suspension of the current tracing table and the beginning of a new tracing table. The explanation addresses a new aspect of recursion — how it works — but it does so in a way that is consistent both with the previously-presented client views of recursion, and with how recursion is actually implemented in most programming languages.

The tasks of discovering recursive structure and using it to develop a recursive operation body, of understanding why a given use of recursion is correct, and of understanding how recursion works, are three separate intellectual activities. It seems reasonable that a student should be exposed to the first two activities either simultaneously or to the second activity just after the first. On the other hand, from the perspective of the learner, a separation in time would seem desirable between the first two activities and the last. A first exposure to recursion should culminate in the student

using recursion from the client’s perspective and becoming comfortable with the inductive reasoning that underlies recursion. Internalizing these ideas alone will take time, and should not be unnecessarily complicated by the immediate introduction of additional aspects of recursion. More advanced applications of recursion and new ideas (such as how recursion works) can be introduced later in the spiral of visits to the topic of recursion.

## 6. CONCLUSIONS

When the history of CS instruction is written, one theme is sure to emerge, and that is the steady liberation of our thinking, our understanding, and our teaching from the stifling confines of underlying programming languages. At the same time, as we typically teach certain important programming language features and related programming techniques that use them, the client perspective appears to remain compromised. It is overly influenced by our knowledge of how language constructs are realized by compilers and by the historical baggage of our past teaching practices. We need to take seriously the two fundamental questions from the introduction and begin anew searching for client perspectives. As our reconsideration of recursion has shown, this search can lead us to new ways of thinking about and teaching the traditional topics and to additional ways of comparing alternative pedagogical approaches.

## 7. ACKNOWLEDGMENTS

We gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants DUE-9555062 and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, from the Defense Advanced Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office, and from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, the U.S. Department of Education, the U.S. Department of Defense, or Microsoft. We also appreciate the assistance of the many other people who are involved in this project, especially Steve Fridella, Joe Hollingsworth, David Mathias, Bill Ogden, Elley Quinlan, and Scott Pike.

## 8. REFERENCES

- [1] Gal-Ezer, J., and Harel, D. What (Else) Should CS Educators Know? *Comm. ACM* 41, 9 (Sept. 1998), 77-84.
- [2] Long, T. J., et al. Providing Intellectual Focus to CS1/CS2. *In Proc. 1998 ACM SIGCSE Symp.*, ACM, February 1998, pp. 252-256.
- [3] Wu, C., Dale, N.B., and Bethel, L.J. Conceptual Styles and Cognitive Learning Styles in Teaching Recursion. *In Proc. 1998 ACM SIGCSE Symp.*, ACM, February 1998, pp. 292-296.

