From Rigor to Rigor Mortis: Avoiding the Slippery Slope

Allen B. Tucker
Bowdoin College

www.bowdoin.edu/~allen

Undergraduate CS Goals and Outcomes

We prepare graduates for industry and for PhD programs. Among other goals, we prepare them to develop reliable software. How well do they do?

Often well:

Successful software products are many: E.g., TurboTax, Linux.

But often not well:

Companies spent over $250 billion on IT projects in 1995, yet 31% of these were canceled.

Only 9% of all IT projects are delivered on time and on budget [Software; April 1998].

Specific software failures are humbling:

In 1992, AMR cancelled development of "Confirm" at a cost over $125 million, 4 years, and 200 programmers. *Reason*: component software designs were incompatible and incomprehensible.

In 1996, the Ariane 5 launcher crashed on take-off, at a cost of over $500 million. *Reason*: insufficient software specifications that failed to trap an exception [Meyer 1997].

Evidence suggests that something is missing:

Rigorous software design and verification is not always practiced in the *software industry,* and so *Rigorous* software design and verification is not always taught in our *undergraduate programs*

What do we mean by “rigor” in Computer Science?

Dictionary: “rigor” = “rigid accuracy, or precision,” or “great care or thoroughness in making sure that something is correct.”
“Rigor mortis” = “a muscular stiffening at time of death. (the rigor of death)

For computer science:

Rigor$_{cs}$ = a careful, thorough, systematic, and precise process for assuring that solutions to computational problems are correct and robust.

Rigor mortis$_{cs}$ = a careless, partial, unsystematic, and imprecise process for solving computational problems. (the death of rigor)

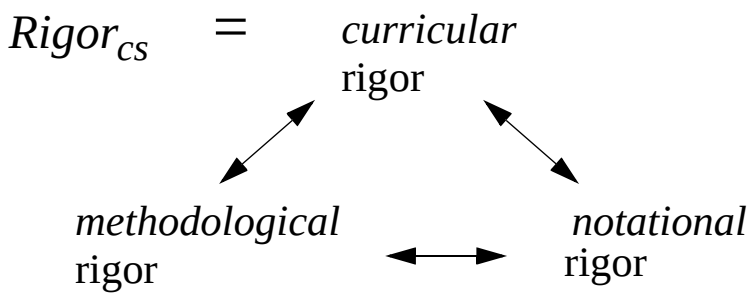
Computer Science *is* a Rigorous Discipline

Liberal arts model curriculum [Gibbs 1986]:
"computer science is the systematic study of algorithms and data structures, specifically 1) their formal properties, 2) their mechanical and linguistic realizations, and 3) their applications."

Computing as a Discipline [Denning 1988]:
"The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application."
The discipline has three paradigms: theory (mathematics), abstraction (science), design (engineering), and these occur in all subject areas.

Recent Curriculum models [ACM/IEEE 1991; Walker 1996]:
Theory (as well as abstraction and design) should be interwoven through all courses, including algorithms, data structures, computer organization, programming languages, and software engineering.

Viewing $Rigor_{cs}$ in Three Dimensions



I *Curricular* rigor = ensuring that the CS courses we teach are rigorous
II *Methodological* rigor = ensuring that the process for doing CS is rigorous
III *Notational* rigor = ensuring that our languages and notations are rigorous

I. Curricular Rigor

To be rigorous, a curriculum should meet three goals:

Goal 1: ensure that students can use precise mathematical ideas and notations in all subject areas.

Goal 2: ensure that students can read and write precise problem specifications and systematically demonstrate correctness in all subject areas.

Goal 3: ensure that students pay equal attention to design and verification, as to coding and debugging.

(How) Can a Curriculum Meet these Goals?

Yes, claims the Revised Liberal Arts Model Curriculum [Walker 1996], and in the following way:

Introduce the tools for rigor early (i.e., make sure that *discrete mathematics has equal status with CS1*, and is thus a prerequisite for data structures)

Use rigorous techniques in CS1 and CS2 (i.e., confirm that *the principles of mathematical logic are integral to good programming*)

Continue using rigorous techniques in each core course (i.e., *integrate the theory* and rigorous precision *with the practice*)

Below are five examples...

Example 1: the CS1 course

Students should learn that deMorgan's law in logic is related to the rigorous implementation of programs that use loops and conditionals.

```
boolean search (Argument x, List L) {
   int i = 1;
   while ( ¬found(x) ∧ ¬exhausted(L) )
     i = i + 1;
   //  found(x) ∨ exhausted(L)
   return found(x);
}
```

Students should learn to write programs from rigorous specifications, not informal invitations to write code and play with the debugger.

```
boolean search (Argument x, List L) {
   require:  L = {e1, e2, …, en} ∧ n ≥ 0
   int i = 1;
   while ( ¬found(x) ∧ ¬exhausted(L) )
     i = i + 1;
   //  found(x) ∨ exhausted(L)
   return found(x);
   ensure:  ∃i ∈ {1, …, n}: x = ei ∧ result ∨ ¬result
}
```

Example 2: the CS2 and Software Design Courses

Students should use preconditions, postconditions, loop invariants, class invariants, and "design by contract" ideas. [Meyer 1997; Liskov 2001]

Design by Contract [Meyer 1997]:

Reliable software must be both *correct* and *robust*

Correctness is enabled using *assertions* (preconditions and postconditions).

Robustness is enabled using *exceptions*

The relationship between a piece of software (a class) and its clients is a *formal agreement*, with each having rights and obligations. To have validity, these rights and obligations must be:

* stated explicitly and formally (using *assertions*), and
* built into the programming language.

Assertions can be well-written only with fluency in logic, so:

1. Discrete mathematics is a natural prerequisite for any software design course,
2. *Assertions* and *exception-handling* should be major topics in CS1 and CS2, and
3. The language(s) and text(s) used in CS1 and CS2 should incorporate assertions and exception-handling as key themes.

Specification of an Integer Set [Liskov 2001]:

```
public class IntSet {
  // OVERVIEW: IntSets are unbounded sets of integers
  // A typical IntSet is {x1, x2, ..., xn}.
  public IntSet ()
    // EFFECTS: Initializes this to be empty.

  public void insert (int x)
    // MODIFIES: this
    // EFFECTS: Adds x to the elements of this,
    //          i.e., this_post = this + {x}.

  public void remove (int x)
    // MODIFIES: this
    // EFFECTS: Removes x from this,
    //          i.e., this_post = this - {x}.

  public boolean IsIn (int x)
    // EFFECTS: If x is in this returns true,
    //          else returns false.

  public int size ()
    // EFFECTS: Returns the cardinality of this.

  public int choose ( ) throws EmptyException
    // EFFECTS: If this is empty, throws EmptyException
    //          else returns an arbitrary element of this.
}
```

Specification of a Stack [Meyer 1997]:

```
indexing
   description: “Stacks, with elements of arbitrary type G”
deferred class STACK[G]
   -- a typical stack is of the form [x1, x2, ..., xcount]
   count: INTEGER is
            -- number of items
          deferred
          end
   empty: BOOLEAN is
            -- are there no items?
          deferred
          end
   item: G is
            -- item at top of stack
          deferred
          end
   put (x:G) is
            -- add x on top
          require
            not full
          deferred
          ensure
            not empty
            item = x
            count = old count + 1
          end
   remove is
```

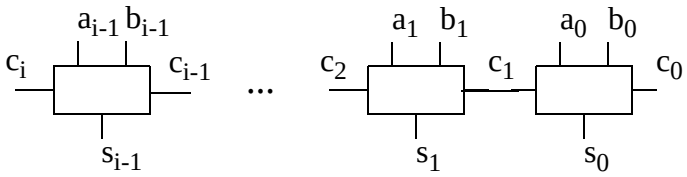
```
            -- remove top element
          require
            not empty
          deferred
          ensure
            not full
            count = old count - 1
          end
invariant:
   count_non_negative: count >= 0
   count_bounded: count <= capacity
   empty_if_no_elements: empty = (count = 0)
   item_at_top: (count > 0) implies
                (representation.item(count) = item)
end -- class STACK
```

Example 3: The Computer Organization Course

Students should learn to:

* prove that a device correctly implements its logical function.
* show equivalence of functions, and how that relates to optimal design.

E.g., generating the carry bit in a serial adder [Tananbaum 2000]:



The carry bit c_i can be expressed as a logical function (recurrence):

$$c_i = a_{i-1} \cdot b_{i-1} + (a_{i-1} + b_{i-1}) \cdot c_{i-1}$$
$$= P_{i-1} + S_{i-1} \cdot c_{i-1}$$

The trick is to derive a function that eliminates the dependency of c_i on c_{i-1} (this helps optimize the adder). This derivation is made by solving the recurrence:

$$
\begin{aligned}
c_i &= P_{i-1} + S_{i-1} \cdot c_{i-1} \\
&= P_{i-1} + S_{i-1} \cdot (P_{i-2} + S_{i-2} \cdot c_{i-2}) \\
&= P_{i-1} + S_{i-1} \cdot P_{i-2} + S_{i-1} S_{i-2} \cdot c_{i-2} \\
&= \ldots \\
&= P_{i-1} + S_{i-1} P_{i-2} + S_{i-1} S_{i-2} P_{i-3} + \ldots + S_{i-1} S_{i-2} \ldots S_1 P_0 + S_{i-1} S_{i-2} \ldots S_1 c_0
\end{aligned}
$$

Student comfort with logic and algebraic manipulation is essential here.

Example 4: The Programming Languages Course

Students should integrate formal semantics with the study of language design.
E.g., here is the meaning function *M* for a *Loop* in a program [Tucker 2001]:

$$M(Loop\ l, State\ \sigma) = M(l, M(l.body, \sigma)) \quad \text{if } M(l.test, \sigma)$$
$$= \sigma \quad \text{otherwise}$$

This is modeled in Java, which is a near copy of the functional definition:

```
State M  (Loop l, State sigma) {
   if (M (l.test, sigma))
        return M(l, M (l.body, sigma));
   else return sigma;
}
```

With formal semantics, students can reason abstractly about loop behavior:

M(while(i>1){fact=fact*i; i=i-1;}, σ)
= M(while(i>1){fact=fact*i; i=i-1;}, M({fact=fact*i; i=i-1;}, {⟨i, 3⟩, ⟨fact, 1⟩}))
= M(while(i>1){fact=fact*i; i=i-1;}, M({fact=fact*i; i=i-1;}, {⟨i, 2⟩, ⟨fact, 3⟩}))
= M(while(i>1){fact=fact*i; i=i-1;}, M({fact=fact*i; i=i-1;}, {⟨i, 1⟩, ⟨fact, 6⟩}))
= {⟨i, 1⟩, ⟨fact, 6⟩}

... and thus master the principles of language design rather than special cases.

Example 5: The Algorithms Course

Students should argue rigorously that an algorithm is correct [Cormen 1990], as well as analyze its complexity bounds.

Various methods of proof should be used: induction, construction, counterexample, equivalence, ...

... many more examples can be added!

II. Methodological Rigor

In the practice of computer science, professionals should use:

Formal tools: discrete mathematics, including propositional and predicate logic, the calculational method of logic [Gries 1993] and other techniques that demonstrate what a proof is and how to develop proofs.

Programming languages and paradigms that promote rigor by properly including formal specifications.

Software designs that result from a rigorous process, and are correct and robust.

... this subject merits a separate talk!

III. Notational Rigor

The *consistency* issue is huge, both in formal methods and in programming languages! E.g., the notation $[N/x]M$ means "substitute N for every free occurrence of x in expression M." But so does each of the following:

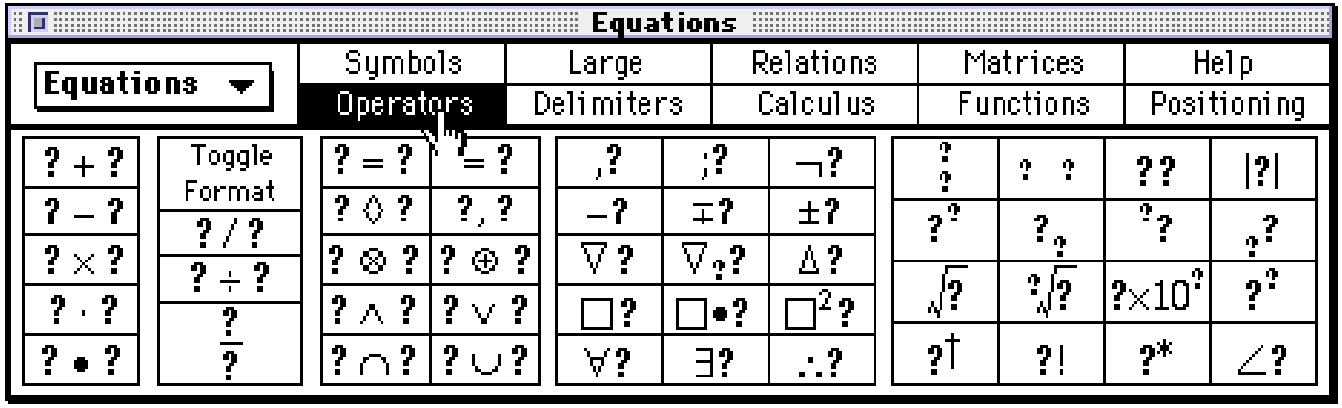
$S?_N^x M$	Quine, 1940
$S_N^x M\|$	Church, 1956
M_N^x or $M[x:=N]$	Gries, 1993
$M[x:=N]$	Stansifer, 1995
$M[x \leftarrow N]$	Meyer, 1990 and Watt, 1991
$M[N/x]$	Winksel, 1993 and Friedman, 1999
$\{N/x\}M$	Sethi, 1996
$M[N\backslash x]$	Scott, 1999

The *completeness* issue is also huge! E.g., There is no direct representation for even the basic logical operators in programming languages. (Instead we are forced to use surrogates like && for $\wedge$, || for $\vee$, and so forth.)

The “ASCII/QWERTY Bottleneck”

Our ability to specify and write programs is *severely hampered* by the limits of the ASCII character set and the QWERTY keyboard.

Other fonts are available: Lics, Zed, Ophir, Symbol, and Framemaker equations:



... but none of these is well-integrated with a programming language.

Programming Language Design: A Wish List

1. A more *complete* character set
2. Better *consistency* and uniformity of expression
   E.g., *one* way to specify a loop (collapse while, for, repeat, ... into one form)
   E.g., *one* way to specify a conditional (collapse case, if, guard into one form)
   E.g., *one* way to write a fundamental operation (E.g., only *one* of `:=`, `<-`, `is`, `let`, `setq`, and `=` should mean "assignment")
3. A *compilable integration of code and commentary.* E.g., the "publication language" of Algol 60 [Naur 1962] has some of these features:

Reference language	Publication language
`A[i]`	A_i
`x^i`	x^i
`10.5e-6`	10.5×10^{-6}
`sigma`	σ
`a<=b and b<=c`	$a \leq b \wedge b \leq c$
italics, **boldface,**	**procedure** *Ising*

and support for rigorous comments...

E.g., an Algol program [*CACM* 1969 p 562]

procedure *Ising(n,x,t,S)*; **integer** *n*, *x*, *t*; **integer array** *S*;

comment *Ising* generates *n*-sequences $(S_1, \ldots, S_n)$ of zeros and ones where $x = \sum_{i=1}^{n} S_i$

and $t = \sum_{i=1}^{n-1} |S_{i+1} - S_i|$ are given.

...

begin

integer *k*; **integer array** *L*, *M*[1 : $t \div 2 + 1$];

...

end *Ising*

... we must begin to view compilers as *document editors* and not just language translators.

Example of “literate programming” in Haskell [Tucker 2001]

Exercise 8-24: Consider the following (correct, but inefficient) Haskell implementation of the familiar Fibonacci function:

```
>   fibSlow n
>      | n == 0  = 1
>      | n == 1  = 1
>      | n > 1   = fibSlow(n-1) + fibSlow(n-2)
```

The correctness of this function is apparent, since it is a direct encoding of the mathematical definition discussed in chapter 3:

$fib_0 = 1$

$fib_1 = 1$

$\forall n > 1: fib_n = fib_{n-1} + fib_{n-2}$

a. But the efficiency of this function is suspect. Try running `fibSlow(25)` and then `fibSlow(50)` on your system and see how long these computa-

tions take. What causes this?

b. An alternative definition of the *fib* function can be made in the following way. Define a function `fibPair` that generates a 2-element tuple that contains the nth Fibonacci number and its successor. Define another function `fibNext` that generates the next such tuple from the current one. Then the Fibonacci function itself, which we optimistically call `fibFast`, is defined by selecting the first member of the nth `fibPair`. In Haskell, this is written as follows (this program is adapted from [Haskell 1999]):

```
>   fibPair n
>      | n == 0   = (1,1)
>      | n > 0    = fibNext(fibPair(n-1))
>   fibNext (m,n) = (n,m+n)
>   fibFast n = fst(fibPair(n))
```

Try running the function `fibFast` to compute the 25th and 50th Fibonacci numbers. It should be considerably more efficient than `fibSlow`. Explain.

c. Prove by induction that $\forall n \geq 0$: $fibFast(n) = fibSlow(n)$.

Summary: How Can We Achieve Rigor$_{CS}$?

A rigorous curriculum

- Create student comfort with mathematical notation and use it.
- Expect students to read and write rigorously, and to make clear correctness arguments.
- Require students to be rigorous in all subject areas.

A rigorous software methodology

- Integrate contemporary features of the software process into the curriculum.
- Develop better formal methods for design and verification.

A rigorous notation and language

- An extended alphabet.
- A more literate programming style that encourages programmers to be more like authors and programs to be more like refereed publications than secret codes!

Achieving Rigor$_{CS}$ will serve our graduates well and will help address the software quality problem.

In closing, I hope you...

Have at least one *rigorous experience* at this conference (and enjoy it too!)

Think about the *level of rigor*:

- in your own courses (curricular rigor)
- in your own software research (methodological rigor)
- in your own programming languages research (notational rigor)

Thank you again, ACM SIGCSE: "... because we teach *rigorously!*"

References

[ACM/IEEE 1991] ACM/IEEE Joint Curriclum Task Force, *Computing Curricula 1991*, ACM Press, 1991.

[Denning 1988] Denning, P (ed), D. Comer, D. Gries, M. Mulder, A. Tucker, A.J. Turner, P. Young, *Computing as a Discipline*, ACM Press, 1988.

[Gibbs 1986] Gibbs, N. and A. Tucker, "Model Curriculum for a Liberal Arts Degree in Computer Science," *Communications of the ACM* 29,6 June 1986.

[Gries 1993] Gries, D. and F. Schneider, *A Logical Approach to Discrete Mathematics*, Springer-Verlag, 1993.

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

[Meyer 1997] Meyer, B., *Object-Oriented Software Construction* 2e, Prentice-Hall, 1997.

[Naur 1962] Naur, P (ed), "Revised Report on the Algorithmic Language Algol 60," IFIP Press 1962.

[Tucker 2001] Tucker, A. and R. Noonan, *Programming Languages: Principles and Paradigms*, McGraw-Hill, 2001 (in preparation).

[Walker 1996] Walker, H. and M. Schneider, "Revised Model Curriculum for a Liberal Arts Degree in Computer Science," *Communications of the ACM* 39,12 December 1996.