

The Persuasive Programming Style

By **JERUD J. MEAD** AND **ANIL M. SHENDE**

For years students in introductory programming classes have been fed a diet of formal syntax and very informal semantics. They have been encouraged to write nicely formatted programs with imprecise comments documenting more the structure of the code than its meaning. Check out the following vignette:

Vignette—“Programming by Approximation”

Sam is anguishing over the current programming project in his CS course. The program is almost working correctly, but . . .

“. . . there's just this one last bug. And it seems to reside in this loop. What can be wrong with this thing? Maybe the loop bound is wrong -- I'll change it. Ah, that's a bit better, but the output still isn't quite right. Maybe it's where I -- I'll move that before this stuff. Rats! Now nothing works. I'd better change that back . . .”

And on he goes into the night.

This is a programming style, familiar to most programming instructors, that we call programming by approximation. In this style the student repeatedly manipulates a program based on very fuzzy semantic understanding in the hope that the next small modification will result in a program that better approximates its specification (the correctness condition). Sam seems to understand that there is a relationship between changing the loop bound and the program's output, but he doesn't analyze exactly what the effect of the change will be.

A new, semantics-based approach to programming and programming education is needed—an approach that redirects the attention of the student programmer to semantics; an approach that integrates semantics from the start and provides a mechanism for representing semantic content within the text of a program.or in a later course, but it need not wait.

Through this repeated process of “survival-of-the-fittest modification,” Sam’s program evolves. In the end, Sam has a program whose execution approximates (or maybe meets) the specification, but he may have no clear idea why the program behaves as it does.

Okay, this picture of student programming is not entirely fair. Most students have a good understanding of the semantics of the assignment and IO statements and even of simple selection statements and abstraction calls. But when it comes to more complex abstraction calls and selection and especially repetition statements, most students’ semantic intuition is fuzzy at best. This intuition is even fuzzier when these basic statement types are combined through nesting or sequencing.

Contrast with Sam’s plight the situations of two other students: Sue, who is working on a programming assignment for her CS1 class, and Tom, who is working on an essay for a history class.

A program written in the persuasive style will carry evidence of its own correctness.

Vignette—“Writing vs. Programming”

For Sue, there is no question as to what is expected of her. She must develop a program that meets the problem specification. She is expected to use certain design principles and to carefully test the program to ensure that it meets the specification. The instructor who will read the completed program will examine the output from sample executions and then study the program to see if it is well designed and written.

Tom is equally certain as to what is expected of him. He knows that a thesis and a simple listing of historical facts will not do. He knows that the facts must be accompanied by supporting evidence set in a logical framework so that the reader of the essay (i.e., the instructor) is persuaded that the thesis is valid. This is a rhetorical (we will use the term persuasive) style of writing.

The contrast is clear. Sue expects to produce a program that produces a specified output (analogous to the facts and thesis of Tom’s essay). But there is no perceived requirement that the program be accompanied by a convincing explanation for the instructor that the program does what it’s supposed to do—just the evidence from test executions of the program.

Let’s check up on Sam again to see if he has made any progress. Perhaps we can gain some new insight on the programming process.

Vignette—“Why Comment?”

Sam has finally gotten the program to work! Now the final task.

“Finally done!! Now all I have to do is comment the code. Let’s see, I can comment each variable declaration. That’s easy. Then I’m supposed to put in each function . . . what are

they called? Oh right, pre- and post-conditions. Then some comments in the code and I'll be done.”

Here is a sample from the code that Sam turned in—a function `insert` that is meant to insert a value into an ordered list.

```
//*****
//***** insert *****
//*****
void insert(List list, int value) {
    // pre: list is a list and value is an integer
    // post: value is inserted in list
    ...
    // this loop inserts value in list
    while(...) {
        ...
        X = X + 1;    // increment X by 1
        ...
    }
}
```

So what kind of comments do we have here? The first three lines may be useful for signaling the beginning of the code for this particular function, but, in fact, they are purely syntactic. The pre- and post-conditions are welcome, but they don't really convey information not already implied by the function's name. Where is the value inserted? Is the list sorted before `insert` is called? After? And the comment before the repetition statement seems to be more a statement of hope than of fact. The comment on the assignment statement is welcome, because it is an attempt to represent the semantics of the commented statement. But the semantics of this particular assignment statement is so simple that the comment is redundant. In our experience such comments can be found in the programs of even experienced programmers. The meaning in such comments is often inversely proportional to the complexity of the statement being commented. Comments appear before a statement, not after; the pre- and post-conditions are associated with a function's definition, but are not restated at the point of a call to the function, where the program state is actually affected.

Persuasive Programming

The problems illustrated by the vignettes above are a consequence not of focusing too much on syntax, but rather of not focusing enough on semantics. A better grounding in semantics

might give Sam a base from which to seek a solution for his loop problem; in fact, a better semantic understanding might have allowed Sam to analyze the problem and avoid the loop problem from the start. What about Sue's situation? Perhaps the reason that she isn't expected to supply some evidence of program correctness is because the instructor knows she hasn't the semantic grounding to do it. And the comments in Sam's program? More focus on semantics might motivate Sam to include more detailed indications of semantic effects.

To address these problems, a new, semantics-based approach to programming and programming education is needed—an approach that redirects the attention of the student programmer to semantics; an approach that integrates semantics from the start and provides a mechanism for representing semantic content within the text of a program. This redirection is the goal of our new approach to programming, which we call persuasive programming.

The persuasive style makes use of assertions (in the form of program comments) to reflect properties of a program's state (i.e., semantic content) at selected locations in a program's text. Including the assertions naturally draws the programmer's attention toward the semantic side. A program written in the persuasive style will carry evidence of its own correctness. In this way, persuasive programming could also be called a rhetorical style for programmers.

Teaching Persuasive Programming

Why don't we teach our students about semantics from the start? Probably because program semantics is one of those areas of computer science perceived to be too difficult for a CS1/CS2 sequence. While this may be true in a formal sense, does that mean that we can't make a start in CS1/CS2? Why not lay down the groundwork and do some simple semantic analysis early on? Our approach to teaching persuasive programming addresses this question directly and makes "semantics early" an attainable goal. Here are some key strategies:

Teach students about program environment and state when the notion of variable (or object) is first introduced. We define state to be the set of variable/value pairs that are active (i.e., in scope) at a particular time during execution. It is this definition that drives the use of assertions in the persuasive style.

We also find it useful to bring the basic notions of input and output streams into the picture early on as components of the program environment. The modification of these streams is a semantic detail often glossed over in CS1 courses.

We introduce assertions immediately after the first discussions of the assignment statement. Assertions help to focus attention on the result of executing an assignment and, consequently, on assignment semantics. The example that follows reflects a basic idea about assertions: an assertion that precedes a statement (a pre-condition) is assumed to be true; an assertion that follows a statement (a post-condition) is determined to be true based on the semantics of the statement.

```
// Assert: X == 5 AND Y == 3
X = X + Y * 3;
// Assert: X == 14 AND Y == 3
Here is another example that is important at this stage:
// Assert: X > 0
X = X - 1;
// Assert: X >= 0
```

The examples above can, of course, also be extended to the other simple statements, i.e., the input, output, and declaration statements. Here's an example involving a C++ input statement. The notation `<v, w, r, ...>` is meant to reflect that the input stream has three values at its head, possibly followed by more data values.

```
// Assert: cin == <v, w, r, ...>
cin >> X >> Y;
// Assert: X == v AND Y == w AND cin == <r, ...>
```

By the time the first structured statement is introduced, students should be used to seeing and using simple assertions. But the structured statements are much more complex semantically, and it is unreasonable to expect students to grasp their semantic details immediately. Instead, we take the intuitive semantics of a statement and encourage a strategy that reflects that intuition in the form of pre- and post-conditions. The strategy is key to the students building their confidence in semantics. Here is an example of a two-step strategy for asserting a C++ while-loop.

```
// Assert: X > 0
Count = 0;
while (Count < X) {
    // Assert: Count < X
    Count = Count + 1;
    // Assert: Count <= X
}

// Assert: X > 0
Count = 0;
while (Count < X) {
    Count = Count + 1;
    // Assert: Count <= X
}
// Assert: Count >= X AND
//          Count <= X
```

In the code on the left we focus on asserting the loop's body, which we understand from the semantics of the assignment statement and the fact that the loop condition must be

true. The post-condition on the right reflects that when the loop terminates, (i) the loop condition must be false and (ii) the body's post-condition must still be true.

One thing to point out here is that the code on the left should be seen as analytic. In order to come up with the loop's post-condition, we need the body's post-condition. The asserted code on the right retains those assertions needed to make the code persuasive. When current CS1 texts are examined, the one place where assertions are routinely used is in function definitions. The pre- and post-conditions for a function are meant to define its semantics. What the persuasive style demands, and what is not generally seen in these CS1 texts, is the use of the pre- and post-conditions at each point where the function or procedure is called.

A function's pre-condition places restrictions on its formal parameters. Assuming that the pre-condition holds for the actual parameters in the call, then the post-condition will also hold for the actual parameters and the return value after the call returns. Using these at the point of the call is critical to understanding the code that contains the call.

The semantics of individual program statements is only part of the persuasive programming process. Just as important is the integration of the statement semantics into a program's text. Applying semantic reasoning to a sequence of statements is critical to the success of the process.

A final point is necessary. While it may seem that this persuasive style might require lots of extra class time, this has not been our experience. What we have found is that once the notions of state and assertion are discussed, then the presentation of statement semantics can be accomplished by using assertions and state to clarify concepts that are already being explained. Assertions and state just provide a context for presenting normal CS1 material.

What this says is that persuasive programming is a process that must be experienced—it shouldn't be taught. Students should see the persuasive style in examples, lab assignments, and project solutions. By always seeing the asserted style, they will come to accept it as the accepted way to program.