Programs, not Code

Steven J. Beaty
Metropolitan State College of Denver
Campus Box 38, P.O. Box 173362,
Denver, CO, 80217-3362
+1 303 556 5321

beatys@mscd.edu

ABSTRACT
Students too often write code instead of programs. There are fundamental differences between the two. This paper suggests programming should be encouraged and gives an approach to doing so.

Categories and Subject Descriptors
D.2.9 [**SOFTWARE ENGINEERING**]: Management – *life cycle, productivity*.

General Terms
Documentation, Design.

Keywords
Programming, Coding.

1. INTRODUCTION
Consider one of the definitions for the word *code* from http://www.m-w.com:

3 a : a system of signals or symbols for communication b : a system of symbols (as letters or numbers) used to represent assigned and often secret meanings

While it is a system for communication, it is often used to *hide* the information being communicated. Consider examples from everyday life: Morse code, Universal Product Code (UPC) bar codes, even ZIP codes (what percentage of those does anyone know?); all are if not in purpose, in practice, unreadable. In contrast, consider one of the definitions of *program*:

2 a : a brief, usually printed, outline of the order to be followed, of the features to be presented, and the persons participating (as in a public performance).

One receives programs when attending a ballet, musical, or other public event. They are used to explain what occurs, in what order, and to and by whom. Indeed, programs are meant to de-code some event for those who are not intimately familiar with it.

It is the central thesis of this paper that the production of code needs to stop, and that what we need to foster is the production of programs. The issues from a (computer) generation ago (executable size and performance) are generally not our issues. What was difficult no longer is, and we are now faced with problems unsolvable by individual coders.

2. WHAT IS DIFFICULT
The breadth of problems that computer programs are asked to solve, even at the undergraduate level, is staggering. Indeed, we (correctly) ask undergraduates to become proficient at a number of programming styles and a large set of data structures. I submit that the difficulty in creating programs for anyone, freshman or thirty-year veteran, is not syntax. It is the job of a compiler or interpreter to know what the allowable syntax is and communicate it to the user. No doubt, some compilers or interpreters have less than ideal error detection, recovery, and reporting schemes. Nonetheless, with the current state of Integrated Development Environments (IDEs), program editors, etc., few would argue that syntax is the major issue facing programmers or that syntax errors lead to the majority of ill-behaved programs. Semantics, the meanings, understanding the meanings, and controlling the behavior of programs, are what is difficult, from a person's very first encounter with programming. For the sake of argument, not to produce a taxonomy, I'll break the semantics issue into three major parts: understanding the problem, dealing with complexity, and communication between humans.

2.1 Understanding
Syntax is not the issue. Consider writing a paper such as this one. Is the most difficult task spelling or creating passable grammar? Independent of this particular paper's proper grammar or lack of it, there are many tools that assist in producing a polished paper, at least in its most basic form. I contend that what is difficult about writing papers is knowing what to say, and how to say it such that the major points are communicated. I also contend that writing programs is more substantially akin to writing good papers or music than other creative processes such as doing math, but that is a subject for another tome.

Given we agree that the basic issue facing software engineers these days is not syntax, what are the issues? In my mind, first and foremost is understanding the problem and knowing what a good solution should look like.

2.1.1 Requirements
One of the most crucial preconditions at every level, from the system down to the unit level, is understanding what needs to be solved. Important corollaries include knowing what *does not* need to be solved, and what should happen when exceptional circumstances occur. The process of capturing the set of requirements and producing a workable specification from them is one that needs to be taught from the first day.

Professors certainly provide a subset of specifications when we give assignments. Most (correctly) do not provide what a professional software engineer would consider a complete specification. Our implicit assumption is that the students will "fill in the blanks" while creating their programs. I believe this assumption should be made explicit; i.e.: we should tell our

students that we have not thought about all possible input and their mapping to all possible output. This process needs to be at least partially owned by the students so that they must think about the problem at hand. I believe thinking about the problem at hand is at the heart of computer science. When a student finds an incomplete specification, and there is not a single correct approach to choosing a solution, I suggest they "decide and document." That is, decide how to solve the problem and document their solution.

Thinking about what a software component should do needs to be captured (documented) at every level of a solution. Of course for us, the documentation is most often in the form of comments. I believe it is crucial not to have external documentation for a program, although I recognize this opinion may be a controversial one. As with all of this paper's ideas, I am not the first, nor the most eloquent to express these ideas. Knuth's "Literate Programming" [1] predates many people's work in the area of documenting programs. In keeping with my no external documentation policy, I require user documentation to be extractable from a program's source – Javadoc [2] is a method for doing this with Java programs. My hope is this leads to fewer discrepancies between a program's documentation and its actual behavior.

I hope that by creating specifications, people will think more about the problem than they would otherwise. I see far too much of what I call "programming by permutation," the process of starting with a somewhat-related solution and modifying it ad infinitum until a partially-working solution is hit upon. The quick turn times of modern language implementations certainly encourage this behavior – however I do not advocate the return to batch jobs on cards to solve this problem. I believe there are better responses to this problem, some of which are outlined below.

2.1.2 Solutions

A major part of creating a solution is actually knowing that a good solution has been found. Good solutions are those that create the correct answers under a wide variety of inputs, and also behave rationally under erroneous inputs. Here, the design of tests is a major issue.

Test design drives program design. When one thinks about what should be produced, how to test for the correct production of it, and what to do when the unexpected occurs, one views a program differently. A lot of emphasis should be given to system, integration, and unit testing. This also removes the "all or nothing" aspect of programming. If one can show that a program works for a particular set of input, or on a particular platform, at least partial credit can be assessed for the work. Often errors can be much more easily located by others once a version exists that works in a particular environment. Again, it is my belief that tests, and their results, should be included in the program source. I require students to design tests, predict the outcome, and then show which passed or failed by capturing the output and including it in comments in the source. For unit tests, assertions are an easy way to test for pre- and post-conditions.

2.2 Complexity

Complexity is probably the major issue facing software engineers today. Driven by ever-increasing hardware capabilities and the fact that features sell software, systems are evolving into ever more complex beasts. Complexity has far outstripped our ability to understand the details of most software systems. Indeed, a constant pedagogical issue is how to create a "real world" (complex) environment without burying the essence of what we are trying to teach. We are not the first to have to deal with complexity, and indeed computer science's approach to the problem mirrors that of other disciplines.

To again be self-referential, consider this paper. There are a number of different points I would like to make, and a number of different approaches I could take. I could state them all with as much brevity as possible in one paragraph, an option that holds a lot of appeal. This, however, would probably not communicate all I would wish, and many nuances would probably be lost. I am lucky in that many smart people before me have created a method for communicating these sorts of ideas: the technical paper. There are titles, abstracts, sections, citations, etc., that provide the framework on which to hang my ideas. Going from a single paper's organization, we have proceedings, books, collections, libraries, etc., all of which are organized to both hide the complexity of any one aspect from the casual user, while making everything accessible to all.

Computer science's approach to complexity has been similar. Starting with the ability to create sub-programs and currently culminating with Object-Oriented Design, we have created structures for both hiding irrelevant, and sharing appropriate, information. With these goals in mind, I strongly suggest to students that they keep any sub-program under 66 lines, including comments. I believe one cannot keep more details than are contained in those 66 lines in one's head at a time, and therefore no function should be longer than this. Biology's role in this would be interesting to study. I also believe that one should not create more than three levels of nesting – again because of a human's inability to understand more. Note, as with most everything computer today, it is not the machine's inability to deal with complexity, but ours.

2.3 Communication

In most every aspect of life, communication is very difficult – often the most difficult element. It is no different when creating programs. Computer programs will most often be used by others, much later, and for purposes not originally intended. Note that "by others" often means by oneself, after the several weeks it takes to forget what a program did or how it worked. Therefore, communication, via documentation, is critical for software to be used, and re-used.

A program, first and foremost, must communicate how it works, even at the grossest level, for anyone to use it. Therefore, usage statements, documentation about input, output, platforms, etc., are necessary for a useful program. Examples abound where good, or at least available, documentation have helped a system become substantially more successful. UNIX's online manual pages and Microsoft's Help files are but two examples. Both point to the need for online documentation, as a book is often in exactly the wrong place when needed. This, for me, points out the need for documentation to be included in the source itself, making it available at all times.

It is also the case that no one person can create a software system these days. Gone are the days when a single person can create software that many will purchase. Now, large multi-functional teams must be assembled in order to create the large systems that are in demand. In these, communication is paramount. Even if one is able to create a wonderful sub-system, it will not be used if it is not advertised. It is also the case the maintenance is a major cost for any successful software project. Communication about the program to those who maintain it can reduce the cost of fixing bugs or adding functionality.

Returning to the issue of communicating to a later version of yourself, I require that students create a thorough bibliography for their programs. I suggest that most are not born with the knowledge they use to create a program. Instead, they probably used a number of resources in the creation of a program, and therefore need to cite them. Such a record also forms a diary on their learning process which can be quite beneficial. A difficulty one faces when reexamining a program at a later date is ascertaining where the information used to create the program came from. I suggest the students put book references (including page numbers), lecture notes, web sites, etc., in their programs. I also suggest that by not doing so, they are plagiarizing their sources, an ethically indefensible position.

3. WHAT CAN BE DONE

How does one integrate these ideas into programming assignments? The approach I have taken is to require the following in comments at the beginning of each program: statement of purpose, author(s) (if there is more than one author, give the percentage of effort by each), date, email address(es), revision number, description of input and output, how to use, assumptions on data, exceptions, major algorithms and ADTs, citations, and system test design and results. Before each function, I require: purpose, pre and post conditions, exceptions, arguments, and unit test design. In order for students to learn what good design looks like, I pass out examples that I consider high quality. I encourage students to do their designs before they start coding. One could force this issue by collecting designs separately and ahead of the assignment submission and only allowing code to be added after. I have not chosen to do this; my hope is that the students will sooner or later realize that they save time by designing first. It also seems the case that most get into computer science to code – to quash this urge completely could result in diminished enthusiasm, something not beneficial to our profession.

In order to assure that they design and document at all, one must usually use a stick instead of a carrot (this goes for industrial practice too, in my experience.) Our stick is grades. I score programming assignments based on a 40%/30%/30% method for design/test/running. This mirrors the percentages often used in industry, whether for the waterfall model or for individual EVO/spiral iterations. It does not take advanced math for a student to realize that a well-documented solution with no code at all can earn a "C", whereas perfectly-working code with no design earns an "F". While this may seem backwards at first, it is based on the beliefs that understanding the problem, dealing with complexity, and communication between humans, are the difficult issues, not creating code. A good design can stand on its own, independent of the implementation languages. It is also the case that tools exist that can take specifications and create programs from them [3, 4].

4. CONCLUSIONS

This paper is an opinion, or position, paper, not a more traditional research and report paper. It is intended to generate discussion on what is difficult for students to learn, and how to help them overcome those difficulties. This paper's assumption is that writing code is neither difficult nor the point of computer science. Knowing *what* code to write, and how to write it, is the hard part. Providing communication between the developer and the users (or other developers) is critical to creating programs that both run well and will be used. We need to encourage our students to compose programs instead of hacking code.

5. REFERENCES

[1] Donald E. Knuth. Literate programming. The Computer Journal, 27(2):97–111, May 1984.

[2] Javadoc Tool Home Page http://java.sun.com/j2se/javadoc/index.html

[3] Mark Pollack. Code Generation Using Javadoc. Javaworld, August 2000. http://developer.java.sun.com/developer/technicalArticles/Programming/Javadoc/

[4] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic Code Generation from Design Patterns. IBM Systems Journal, Vol. 35, No. 2, 1996. http://www.research.ibm.com/journal/sj/budin/budinsky.html