WRITING ACROSS THE COMPUTER SCIENCE CURRICULUM

Harriet J. Fell Viera K. Proulx John Casey
College of Computer Science, Northeastern University
Boston, Massachusetts 02115
fell@ccs.neu.edu vkp@ccs.neu.edu jcasey@ccs.neu.edu

ABSTRACT

At our university, as at many others across the country, there is a movement to integrate the common core subjects with the disciplinary studies. While in the past writing has been a domain of English departments, the new trend is 'writing across curriculum'. It is clear that effective written and oral communication skills are critical for the successful computer professional.

We present suggestions for writing assignments that complement the main themes of computer courses from introductory to advanced levels. While some of these have appeared in the literature, others are new. We report on our experience with these assignments and reflect on how they enhance the computer science curriculum.

INTRODUCTION

A computer scientist must be able to clearly express his or her ideas. A computer only does what it is programmed to do, so it is the programmer's task to cover all contingencies. It has been shown numerous times, e.g. [1], that understanding of the problem and the ability to talk about it come hand-in-hand. In addition, a computer scientist does not work in a vacuum. The need to communicate one's ideas to coworkers, clients, and supervisors is becoming critical for success in the field. The need for communication skills is not limited to the field of computer science, as can be seen from the recent emphasis in colleges on 'writing across curriculum'.

Our university recently developed a plan [5] to provide a common experience for students in all disciplines. It includes the following directive:

These skills--effective thinking, effective communication, information literacy, and interpersonal skills--cannot be learned in a vacuum.

SIGCSE'96, 2/96 Philadelphia, PA USA.

Rather, they are best acquired while learning information and ideas across a broad range of arts and sciences courses and in-depth, within a specific discipline.

The outcomes can and should be integrated with the study of specific subjects and explicitly identified by instructors as course objectives. General education and study in the major will become inexorably intertwined.

In response to this directive, we have expanded many existing assignments to include writing components. This forced the students to reflect on the concepts explored in the assignments and they often produced better programs as a result of this reflection.

We start by summarizing the types of computer science writing assignments reported in the recent literature. We then describe the types of writing assignments used in our computer science courses, the rationale behind them, and the impact on students performance. We include samples of student's writing and point out ways in which we plan to expand our writing requirements.

SUMMARY OF WRITING IN CS COURSES

In the past, writing in the computer science curriculum has been restricted to specific courses. Students wrote extensive documents in the software design courses and wrote reviews and essays in courses that focused on social issues related to computing. Some programs offered a separate technical writing course, often taught by the English department faculty. Finally, a capstone of many programs was some form of senior presentation - a project, a thesis, or a report. Clearly, most of the writing was some type of paper or essay. Several recent papers that emphasize the importance of writing in computer science give as examples only this type of writing [2, 11].

In [11] we find a shocking statement that in a group of about 80 seniors in a Programming Languages course over one quarter of the students had never written a term paper. Gross-Davis [4] stresses that "improving your students writing is your responsibility." She suggests that you regularly assign brief writing exercises and describes many possible assignments for science classes. Most of her suggestions involve writing essays of at least one page in length.

In recent papers we see the emergence of a new type of writing in computer science courses. Typically these assignments are only a part of a larger exercises and the goal is to help students focus on the main concepts and reinforce the topics related to the exercise. One new type of exercise is the design and analysis of quantitative experiments [6, 8]. Students form a hypothesis, construct a model and make a prediction, design an experiment and collect data, and analyze the results. The second type of writing assignment appears in closed laboratories for introductory courses [7]. The goal is to reinforce and give a discipline-specific context to written and oral communication skills by including at the end of a lab written summation reports and replies to guided questions. Another method is to ask students to analyze other student's user documentation for clarity and completeness or analyze another student's design [9]. Additionally, S. Shum's students [10] use WEB (software that manages the source code and documentation interleaving) for documenting their programs.

We need to prepare students for more extensive writing projects by including simple, short writing components as an integral part of early programming and lab assignments. In addition to improved communication skills, students are led to reflect on the lessons learnt in these assignments which leads to deeper understanding of the new concepts. We show, how a series of short, simple writing assignments can be integrated into all computer science course and comment on the multiple benefits of this approach.

WRITING IN CS1 AND CS2

The first computer science course usually focuses on the skills needed to clearly express an algorithm for solving a given problem. Writing is often limited to adding comments to "document" their code. We have found many other opportunities for students to practice their writing skills by extending existing assignments. These assignments make it clear to students that the ability to communicate in writing is integral to success in computer science. We state briefly each assignment or type of assignment and follow with comments on its goals and its place in the overall first year curriculum.

- **Describe as clearly as you can the rules for a board game a card game, or a team sport with which you are familiar.**

  We gave students this assignment as a part of simple diagnostics to evaluate their learning and communications style. This assignment provided a foundation for class discussion about the need for precise formulation of each algorithm before a program is written. A short sample of one student's report and the open questions it provokes are shown here:

  The game of golf is a sport for the individual where a series of different sized and shaped clubs are used to strike a small ball into a hole distances ranging from 300-600 yards. The simple object of the game is to utilize the least amount of strikes in dropping your ball into the hole. Usually 18 holes are played, but variations are acceptable. The player resulting with the least score when the entirety of holes is played wins.

  Problems for discussion:
  - Do the players take turns?
  - Do they all start each hole at the same time?
  - What is the field surface?
  - Can one player's ball hit a ball of another player?
  - What happens if it does?

- **Describe your computing environment. What is the make and model of your computer? What appears on the screen?**

  In the first lab students learn about the Macintosh computer, networks, hard drive, floppy disks, folders, mouse, projects, and see the first program. By asking them to write down what they observe at each step and identify various objects and events, they are forced to reflect and remember what they learned.

- **Describe, in a few sentences, the behavior of a small application program.**

  In several labs, students were presented with a compiled, working solution to the programming assignment. The programs were typically interactive and graphics based. Students were asked to describe the functionality of a program before they set out to write code. This helps students develop the vocabulary and style they will later need to prepare functional design documents.

- **Describe the purpose of a complex piece of code.**

  We all learn a lot from working examples. Some of the code (especially the code that interacts with the operating system) is too complex to explain in detail in the introductory classroom. Nevertheless, our students were able to add a button to a Macintosh program simply by following an example. They started with reading existing code that installed several buttons. At each critical point they were asked to describe the functionality of that section of code. This forced them to practice another aspect of the work of computer scientist often neglected in our classrooms - reading and understanding existing code as a preparation for making modifications.

  When our honors students were first asked to summarize the functionality of procedures in a "fatbits"

module used in a "Game of Life" program they were to modify, their initial attempts were surprisingly poor. A simple procedure like:

```
procedure getplace (var f: fatbits;
                    var mousespot: point;
                    var row, col: integer);
 begin
  with f, bounds, mousespot do
   begin
    row := (v - top) div cellsize;
    col := (h - left) div cellsize;
   end;
 end;
```

resulted in answers like:

Procedure getplace takes in variables f, mousespot, row, and col. With f, bounds, mousespot it assigns the value of (v - top) divided by cellsize to the variable row.
. . .

It was important here to grade and comment on the students' work. In time, students learned to abstract and describe the purpose of code rather than just say the code in English.

- **Answer questions in the lab report form about your approach to solving a particular problem.**

  The goal of these exercises was to guide student through the program development process and to point out potential pitfalls. The questions were similar in style to those that one might expect to hear from an experienced tutor. By requesting written answers, students were forced to think about the issues and clearly explain their solution and the reasoning behind it. When student programmed the animation of a rolling ball, they were asked about the starting and ending conditions, and how would the size of the radius affect the program. When students were asked to program a movement of a fish through an underwater cave, they first had to answer questions that helped them avoid getting trapped in infinite loops and dead ends.

- **Write a paragraph comparing the performance of two similar programs.**

  The two programs 'miniPaint' and 'mightyPaint' were examples for students to emulate in their assignment. Students had to play with both versions, compare their different features, and - hopefully - use their observations to design their own version of the program. In the future, we also plan to ask students to write a short 'user's manual' for their program - how to select an option, how each option works, how to exit from the program. The goal of this assignment is to prepare the student for more detailed user and product evaluation documents they will write in later courses and on the job. Part of a student's response is shown here:

DIFFERENCES BETWEEN MINIPAINT AND MIGHTYPAINT

There are four main differences between the two programs:

1) MightyPaint has a color menu added on the right-hand side that allows the user to change the foreground color of the objects drawn. The chosen color is displayed in a space in the left-hand menu area.

2) MightyPaint lets the user know which task has been chosen by inverting its icon until the task has been completed (i.e. the object has been drawn).

3) MightyPaint allows to user to interactively select the size of the object by pressing the mouse down to select the first point and dragging the mouse to the second point where the user then releases the mouse button to choose the final dimensions of the object. During the dragging process the program lets . . .

- **Analyze the graphic output of a mapping program.**

  Students wrote a program that displayed in color the magnitude of values of a function of two variables. They were asked to describe the results - identify the minima, the maxima, the saddle points, valleys and ridges. Often, students write a program and produce some output without reflecting on the meaning and correctness of the results. The writing assignment made them look back and demonstrate an understanding of the concept they just programmed.

- **Explain how your program could be modified to include additional features.**

  This kind of document is often included in the software design process. It assures, that the project makes allowances for the addition of features not included in the original design. Students in the first semester may be asked to do similar planning for the future. They may not be ready to complete a more complex project, but by learning to plan for it they learn valuable software design lesson early in their studies. By documenting the potential expansion they not only improve their communication skills, but also increase their understanding of the project they have been working on.

- **Design and describe an experiment to empirically evaluate algorithms.**

The writing we describe here has been implemented at several colleges [6, 8]. The goal is to teach students how to design an experiment to collect meaningful data. The second part below follows up with analysis of the results.

Our students learn about many sorting algorithms by observing and experimenting with an interactive animation where the array values are represented as a bar chart and items that are being compared or moved are highlighted. In addition, the same algorithms appear in another application that allows students to perform timing tests on arrays of different sizes, with random, sorted, or reversed initial data values. Students are asked to write a design document for experiments that will evaluate and compare the performance of several of these algorithms. Some algorithms take too long and the tests cannot be carried out as proposed. In other cases, the running times of some algorithms for small size arrays are so small, that there is no differentiation between them. Students learn what is needed to design experiments that produce useful results.

- **Analyze the results of an experiment to empirically evaluate algorithms.**

  This is the other side of the previous example. Students learn that the measure being used must be able to observe the differences. (If all algorithms take zero or one tick to run, nothing is learned.) Students produce a report that describes both the algorithm design, the results - both in tabular and in graphical form - and comments on the meaning of the results. In this case we are looking for a comprehensive presentation of their findings. Students accept this challenge and produce impressive reports. Again, this is a first glimpse on the type of communications students will be required to produce in their professional lives.

- **Evaluate a class or lab.**

  This technique has been promoted several times in the past [7]. Students are asked to answer a series of short questions about the lab they did, or about the lecture they just heard. By doing so they are forced to focus on the main points and reflect on the lessons learned.

- **Take class notes and prepare them for distribution.**

  Each CS1 student had to take notes for one class period and prepare them for (electronic) distribution. We supplied a notes sample set in Microsoft Word. Students were asked to follow the sample format. They were expected to include correct explanations, examples, and working code. We feel that this is an excellent exercise for the students but it is hard for the teacher to grade and carry out successfully. While in most of the exercises described earlier the teacher needs to provide only a minimal feedback, few students are able to compile error-free comprehensible notes from a lecture. For this exercise to benefit the whole class, the notes need to be produced in a timely fashion and edited carefully so they are free of bugs and confusing statements.

WRITING IN THE ADVANCED COURSES

There are many opportunities to practice different types of writing throughout the upper level computer science courses. Fekete [3] described a variety of weekly writing assignments in a Computer Organization course.

Our effort at this level is in its infancy. We started by collecting reports from all instructors on the use of writing assignments in their courses. After compiling the responses we plan to develop, promote, and advertise the best methods and encourage other faculty to emulate them in their teaching.

The instructors reported using the following types of writing assignments in the more advanced courses:

- **Design, perform, and analyze a quantitative experiment .**

  This was similar to what was already described in the introductory course section. It is important to realize, that experimental design and analysis should be a thread that reappears in several courses throughout the curriculum.

- **Write a report describing, analyzing, or giving a critique of a feature, an algorithm, some part of a system.**

  Again we see that this is a recurring theme. At this level, the critique can include a deep technical study related to the course - for example, students were asked to express their opinion on a cryptosystem proposed by the government and include supporting technical arguments. While the good students wrote very good reports the poor writers' papers were incomprehensible, with lots of theorems and lemmas and without any reasoning.

- **Keep logs, progress or activity reports.**

  The instructors were most enthusiastic about this type of writing. The assignments are short and a good instructor can provide very fast feedback. In addition, both the instructor and the student can see what parts of the project consumed the largest amount of time. In the first courses this reinforced the importance of time

spent in design. In real life, many professionals need to report on their use of time, so again, this is a preparation for future employment.

- **Write a detailed program analysis: line-by-line or by annotating all functions.**

  Again, this technique has been already seen in the CS1 and CS2 courses. The goal is to teach students how to read programs carefully, either so they can explain their functionality, or in preparation for modification of existing code.

- **Explain the meaning of mathematical ideas in prose.**

  Many highly skilled programmers have a very difficult time grasping the precise meaning of mathematical theorems that appear in the analysis of algorithms course. In a statement 'for every c there exists n such that...' they need to see why are we 'choosing' c, what kind of c should we choose to test the 'limits' of the statement, where does the 'there exists n...' come from, etc. Those of us raised on a solid mathematical diet know how to read this type of statement. But most of our undergraduates need help in understanding the real meaning of these theorems. When students are asked to write prose instead of mathematical formulas, it is easy to see their misconceptions and address them in class or in tutoring sessions.

- **Write essays from a personal perspective.**

  This type of writing has been documented before - it may relate to career choices or to social and ethical issues in computing.

Almost all instructors who responded to this informal survey commented that writing component should be a part of every computer science course and were looking for an 'official mandate' for doing it. They all felt it is important to send a message to students that writing is an integral part of the profession and needs to be practiced at a number of different levels.

CONCLUSION

Our goal is to prepare computer science students for their professional careers. The ability to communicate in writing is critical to success in the field and should be practiced throughout the years students spend learning technical aspects of the field. We hope that writing practice will become an integral part of every course.

BIBLIOGRAPHY

1. Benezet, "An Experiment in Manchester Schools", (details to be included later)

2. D. Deremer, "Improving the Learning Environment in CS I - Experience with Communication Strategies", SIGCSE Bulletin, 3(25), September 1993, ACM Press, pp. 31-35.

3. A. Fekete, "Enhancing Generic Skills in the Computer Organization Course", 26th SIGCSE Technical Symposium, Nashville, TN, March 1995 (SIGCSE Bulletin 1(27), 1995), ACM Press, pp. 273-277.

4. Barbara Gross-Davis, *Tools for Teaching*;, Jossey-Bass Inc., 1993, San Francisco.

5. A. Leskes and the A.C.E. Committee, "Plan for a Common Academic Experience", xxx University document, September 1994.

6. T. K. Moore, A. G. Rich, and M. R. Wick, "Scientific Investigation in a Breadth-First Approach to Introductory Computer Science", 24th SIGCSE Technical Symposium, Indianapolis, IN, March 1993 (SIGCSE Bulletin 1(25), 1993), ACM Press, pp. 63-67.

7. B. C. Parker and J. D. McGregor, "A Goal-oriented Approach to Laboratory Development and Implementation", 26th SIGCSE Technical Symposium, Nashville, TN, March 1995 (SIGCSE Bulletin 1(27), 1995), ACM Press, pp. 92-96.

8. R. Rasala, V. K. Proulx, and H. J. Fell, "From Animation to Analysis in Introductory Computer Science", 25th SIGCSE Technical Symposium, Phoenix, AZ, March 1994 (SIGCSE Bulletin 1(26), 1994), ACM Press, pp. 61-65.

9. J. Robergé and C. Suriano, "Using Laboratories to Teach Software Engineering Principles in the Introductory Computer Science Courses", 25th SIGCSE Technical Symposium, Phoenix, AZ, March 1994 (SIGCSE Bulletin 1(26), 1994), ACM Press, pp. 106-110.

10. S. Shum, "Using Literate Programming to Teach Good Programming Practices", 25th SIGCSE Technical Symposium, Phoenix, AZ, March 1994 (SIGCSE Bulletin 1(26), 1994), ACM Press, pp. 66-70.

11. H. G. Taylor and K. M. Paine, "An Interdisciplinary Approach to the Development of Writing Skills in Computer Science Students", 24th SIGCSE Technical Symposium, Indianapolis, IN, March 1993 (SIGCSE Bulletin 1(25), 1993), ACM Press, pp. 274-278.