LITERARY CRITICISM AND PROGRAMMING PEDAGOGY

Corey D. Schou
Associate Professor
Information Systems
Idaho State University
Box 4043
Pocatello, Idaho 83205

Roland Hord
Department of English
Idaho State University
Pocatello, Idaho 83209

ABSTRACT

If one grants certain similarities between the programming and writing processes, then the critical perspectives adopted by the readers of literature may be generalizable to the readers of program texts. M.H. Abrams' classification of literary critical theories as expressive, mimetic, pragmatic, or objective serves as an impetus for a discussion of the reading and writing of program texts. Recognition that program texts can be viewed from numerous perspectives may serve as a liberating force in programming pedagogy.

INTRODUCTION

> ...I look at software writing like authorship, like normal writing. You're trying to combine ideas and concepts in a way that will make other people think, that will be new and exciting.
> John Warnock, In Programmers at Work 1986

This paper is based upon a set of assumptions: that the programming process is similar to the writing process, that program texts are similar to literary texts, and that reading program texts is similar to reading literary texts. The product of the programming process is a program text, just as a literary or written text is the product of the writing process. More specifically, the program text is roughly equated with the program listing (of the source code), which would include internal documentation but exclude external (user) documentation, program output, and program specifications.

Thus, a programmer creates a program text just as a writer creates a literary text. And just as the writer writes for a reader, the programmer programs for a reader. However, note that the computer is not the reader; humans read program texts. This is a most difficult claim for many to accept, due in part to the frequent anthropomorphization of computer systems.

If one will grant that programmers produce program texts for readers, one may find that the practice of literary criticism has interesting ramifications for the pedagogy of programming languages.

M.H. Abrams[1] argued that theories of literary criticism discriminate four elements in discussion of a work of art, although any one theory tends to stress a single element over (and often to the exclusion of) the others:

> a critic may stress the role of the artist (writer), the role of the audience (reader), the role of the work (text) itself, or the role of the world that the work imitates.

Expressive theories of art emphasize the role of the artist in creating the work. Pragmatic theories emphasize the effects of works of art on an audience. Objective theories emphasize the self-containedness of a particular work of art. Mimetic theories emphasize the relationship of the work of art to the world.

Thus, Wordsworth's description of poetry as "the spontaneous overflow of powerful feelings" emphasizes the expressive nature of art, while Wilkie Collins' dictum, "Make 'em laugh, make 'em cry, make 'em wait," is a pragmatic approach to writing. The New Critics emphasized treating works of art as objective texts, criticizing approaches that emphasized historical or biographical considerations. Aristotle's definition of drama as the imitation of an action and H.D. Howell's characterization of the artist's task as "the truthful treatment of material" emphasize mimetic approaches to art.

These are not the only possible critical approaches to program or literary texts; Abrams' typology is simply a well-known and useful starting point for examining critical approaches to program texts and program pedagogy.

To appreciate this approach, one must accept that program texts--like literary texts--are written for people to read. Such a view is the subject of Steven Levy's anecdote about Jerry Sussman:

> Looking at Gosper's programs, Sussman realized an important assumption of hackerism: all serious computer programs are expressions of an individual. "It's only incidental that computers execute programs." Sussman would later explain. "The important thing about a program is that it's something you can show to people, and they can read it and they can learn something from it. It carries information. It's a piece of your mind that you can write down and give to someone else just like a book." Sussman learned to read programs with the same sensitivity that a literature buff would read a poem. There are fun programs with jokes in them, there are exciting programs which do The Right Thing, and there are sad programs which make valiant tries but don't quite fly.[10]

The remainder of this paper deals with how one might learn to read and treat program texts as one would read and treat a literary text and how that might affect the way in which one teaches programming.

PROGRAMMERS

Much programming pedagogy suppresses the expression of the programmer's individuality. The basic argument is that the less individualistic a program text is, the easier that program text is to maintain. This is especially true of large programming efforts. Thus programmers are encouraged to use library routines, to use self-documenting code, and to structure their programs as explicitly as possible. The egoless programming group, as advocated by Weinberg[3], is a manifestation of such a viewpoint. Because the program text is the product of a team, rather than an individual, such texts are thought to be easier to debug and to maintain, containing fewer individualistic coding characteristics which might decrease the ease of program text comprehension.

However, individualistic expression within a program is not at odds with the basic philosophy of structured programming or the use of self-documenting code; individualistic expression is not to be taken as synonymous with obscurity or lack of clarity.

Furthermore, not all programmers program in group or industrial environments, although many programming textbooks stress industrial programming techniques. It is the uncommon program text that prints only complete programs rather than illustrative lines of code taken from longer program texts. It is also the rare textbook that prints more than one program text to satisfy a program specification, to demonstrate the use of a programming technique, or to illustrate the development of program logic or data structures. The message to the student is that there is a best way to do things, not that the programmer ought to experiment in order to produce individualistic program texts.

Many introductory composition textbooks provide numerous examples of complete essays developed through use of the same rhetorical modes. Perhaps programming students need a second textbook--a programming composition textbook containing numerous program texts that could be read and used in the same way that the essay sampler is used by writing students. The use of student program texts may also facilitate the programmer's expression of individuality within program texts in much the same way that sample student essays facilitate the writer's expression of individuality within essays. In short, students need to read numerous program texts if they are to both recognize the similarities and perceive the differences between texts.

Many programming assignments are given in terms of what the results (output) should be, not in terms of what the formal characteristics of the program text should be. Often, students are required to hand in program listings only as a check that they actually wrote necessary program. In other words, only if the student's results are incorrect need the instructor give the program text more than a cursory glance.

Of course, not all program texts need extensive instructor comment any more than all written texts do. Many composition instructors rely heavily upon quantitative writing assignments, such as journals and free writing, which are designed to foster individuality and expressiveness. Such assignments develop the writer's facility to handle the language and to recognize topics and concerns of interest to him/herself. Similar assignments may be profitably used in programming courses.

Donald Murray[11] has argued that it is only through reading what one has already written that one discovers what it is that one wants to say. Janet Emig[3] has argued that writing constitutes a unique way of learning. Both writers clearly are emphasizing the nature of the writing process, rather than the text as a product. Programming assignments, however, that emphasize the product--the results or output--are likely to ignore important aspects of the programming process.

Young, Becker and Pike[14] and Flower and

Hayes[5,6] have found the pre-writing topics of discovery and invention fruitful areas of investigation in composition theory. However, too often, little emphasis is placed upon the heuristics of problem solving or algorithm formation within the introductory programming classroom. The well-intentioned instructor may eliminate expressive aspects of a text by overdefining the program requirements or by being overly generous in providing guidance on algorithm formation.

READERS

Since program statements take the form of commands, one would expect that programmers and programming instructors would be particularly aware of pragmatic criticisms of program texts. Yet few people, other than the programmer him/herself, read program texts. The assumption is that the commands are directed to the computer. This, however, cannot be since the computer cannot read the program text; computers execute programs. Perhaps, then, part of the difficulty in programming is in writing for an audience which does not seem to exist, or more correctly, one might say that programmers typically fail to write for any audience at all.

Part of the problem is determining how various audiences may be addressed within a program text. One indicator may be the remarks or comments that form internal documentation. Even the mere presence or absence of internal documentation, quite aside from the content of the documentation, may reveal the attitude of the programmer to the reader.

The programmer may also express him/herself or manipulate the reader through choice of program language, and within a language, through choice of commands/diction. Often, in a higher level language, one may either issue a simple command or call a machine language subroutine that performs the same function as the simple command; one's choice in the matter is seldom arbitrary. The programmer may also express him/herself or manipulate the reader through choice of data structures, number of program lines, length or complexity of line, or more commonly through program structure. While not an exhaustive list, these are options open to programmers that perceptive readers need to recognize in attempting to formulate a criticism of program texts.

The mere recognition that program texts are to be read by humans is a major step forward. The most recent proponent for reading program texts is Donald Knuth[9]:

> I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be *works of literature*. Hence, my title: 'Literate Programming.'
>
> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.
>
> The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style...[author's italics][9]

Unfortunately, Knuth's main concern is with documentation and the WEB programming language. Literate programming is useful because it calls attention to the reader of a literate program, just as any program text is written for a reader. Because novice programmers have much to learn before they can fully comprehend and appreciate program texts written for more critical audiences, they must be given opportunities to compare programs written for various audiences, much as beginning writers need to read both professional and student essays.

PROGRAM TEXTS

The critical approach that the reader of program texts may be least likely to entertain requires him/her to view a program text as a self-contained entity, quite complete unto itself. Usually the program critic wants to know if the program works or does what it's designed to do. Should the student's program supply the wrong output, the instructor must dissect the code, pinpointing the place(s) where the student has gone astray. Failure to produce a correctly working program is just that, a failure. The writing instructor, however, is able somehow to evaluate the text as a text, quite apart from the issue of whether or not the student followed the assignment, as irritating as that failure may be.

Here, one might also deal with certain aesthetic issues. Typically, good programs are described as those that first and foremost work correctly, that combine power and efficiency, and that possess a certain elegance. It has already been suggested that "work correctly" usually refers to program output, not the program text qua program text. One might consider a possible interpretation of "work correctly" to mean something like succeeds aesthetically.

Efficiency and power are frequently considered qualities of the program as executed on a particular machine. There is an interesting sense of power that has a meaning somewhat akin to 'breadth of treatment.' Shakespeare's works are powerful not only in their ability to move one emotionally but also insofar as they create their own world, be it the world of Elizabethan England or the world of the Shakespearean theater. A powerful program, in this sense, would be one that is both broad in scope, yet detailed in construction, dealing with a subject of importance.

Programs also possess beauty. In an interview, Gary Kildall, the developer of CP/M, was asked if he found anything aesthetically pleasing in his work:

> Oh, absolutely, when a program is clean and neat, nicely structured, and consistent, it can be beautiful. I guess I wouldn't compare a program with the Mona Lisa, but it does have a simplicity and elegance that's quite handsome. Stylistic distinctions of different programs are intriguing, very much like the differences art critics might see between Leonardo's Mona Lisa and a Van Gogh.[8]

Beauty and style are frequent topics of conversation among programmers, yet very little extended commentary has been written about beauty and style in program texts. More commonly one can find theorists and programmers speaking of elegance.

Paul Hide[7] defines an elegant solution to a problem as one that is both simple and ingenious, meaning not immediately obvious. "Elegance," as used by programmers, seems to combine elements of what literary critics have differentiated into a number of categories--style, wit, and perhaps to a degree, sublimity.

Hide also equates elegance with tricks of the trade, which he admonishes programmers to avoid: "Many of these are necessary, but where they are used purely to demonstrate the mental agility of the programmer they are not, and they should thus be avoided"[7]. If one is to consider, however, programs as texts to be read, then it might be reasonable to reject Hide's dictum because of the adverse effects it might have on programming style--much as if one were to have instructed the metaphysical poets to avoid metaphors. From an expressive perspective, one might question what the program text is for if it is not to demonstrate a certain amount of mental agility on the part of the programmer. From the reader's perspective, texts lacking such mental qualities will have little attraction. From a textual perspective, one must wonder what separates one text from another if it is not more than simply a difference in subject/topic.

REALITY

Certainly the most common critical approach to program texts has been some sort of mimetic critical approach, suggesting that the program text ought to imitate or reflect reality. Perhaps the most obvious example of such an approach would be the emphasis on self-documenting code and variable names: Naming a real number variable "Number1" is considered better programming practice than naming that same variable "Rambo" because the former somehow describes its content.

Data validation of a program is based upon a perceived relationship between the program text and mathematical computations. Thus the reader is to follow the text, supplying sample data for the appropriate variables. In the case that the reader arrives at what seems to be an erroneous result, the text receives critical attention. Certainly, the early literary realists did something similar when arguing that fiction should be the truthful treatment of life. Life, in that sense, formed the data of their novels.

Program verification is based upon a perceived relationship between the program text and mathematical proof. This relationship is not merely one of critical bent, for much of it is based upon the very nature of the programming language itself, much as the relationship between language and logic is not merely one of critical choice.

Dijkstra argues that a primary motivation for structured programming is the attempt to depict within a program text the relationship between the program text and a process:

> The moral of the story is that when we acknowledge our duty to control the computations (intellectually!) via the program text evoking them, that then we should restrict ourselves in all humility to the most systematic sequencing mechanisms, ensuring that "progress through the computation" is mapped on "progress through the text" in the most straightforward manner.[2]

Thus, program texts imitate reality, which according to Dijkstra is a process:

> ...whenever a programmer states that his program is correct, he really makes an assertion about the computations it may evoke.[2]

This perspective has dominated program criticism in recent years. The problem is not in what Dijkstra says, but rather in that it fails to acknowledge other possible critical perspectives.

CONCLUSION

Insofar as programmers have examined program texts critically, they have tended to adopt a viewpoint that stresses the relationship between the program text and the world. This is not bad, nor is it necessarily unusual in any way. Mimetic theories of literature, beginning with Aristotle, have long and often held sway in literary criticism.

Nevertheless, programming instructors might find it useful to examine program texts in the light of other critical approaches. Knuth's emphasis on writing 'literate programs' aimed at a particular audience is a positive step forward, but more needs to be done to help sutdents of programming recognize the expressive nature of the texts they write, to help them recognize conventions of program texts, and to aid them in developing an effective style.

When asked if studying computer science is the best way to prepare to be a programmer, Bill Gates, developer of Microsoft BASIC, replied, "No, the best way to prepare is to write programs, and to study great programs that other people have written"[4]. Gates' reply requires that one be able to distinguish great programs from their more mundane counterparts, and that requires that programmers develop critical reading skills

If one is willing to grant that writing and programming are similar processes, the instructor of programming may soon find use for numerous exercises now prevalent in the composition classroom, exercises such as free-writing and brainstorming, exercises in discovery and invention, exercises in the use and development of heuristics, and exercises in rhetorical and structural analysis to name but a few.

Works Cited

1. Abrams, M.H. The Mirror and the Lamp. Oxford University Press: NY, 1953.

2. Dijkstra, Edgar. "Notes on structured programming." In Structured Programming. Eds. O.--.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. Academic Press: NY, 1972, 1-82.

3. Emig, Janet. "Writing as a mode of learning." CCC, 18 (1977), 122-127.

4. Gates, Bill. "Bill Gates." In Programmers at Work. Ed. Susan Lamers, Microsoft Press: Redmond, WA, 1986, 70-90.

5. Hayes, John, and Flowers, Linda. "Problem-solving strategies and the writing process." CCC, 28, 4 (1977), 449-461.

6. Hayes, John, and Flowers, Linda. "The cognition of discovery: defining a rhetorical problem." CCC, 31, 1 (1980), 21-32.

7. Hide, Paul. "The design of algorithms." In Guide to Good Programming Practice. Eds. Brian Meeks and Patricia Heath. John Wiley & Sons: NY, 1980, 29-34.

8. Kildall, Gary. "Gary Kildall. "In Programmers at Work. Ed. Susan Lammers. Microsoft Press: Redmond, WA, 1986, 56-69.

9. Knuth, Donald. "Literate programming." The Computer Journal, 27,2 (1984), 97-111.

10. Levy, Stephen. Hackers: Heroes of the Computer Revolution. Ancor Press Doubleday: Garden City, NY, 1984.

11. Murray, Donald. "Internal revision: a process of discovery." In Research on Composing: Points of Departure. Eds. Charles R. Cooper and Lee Odell. NCTE: Urbana, ILL, 1978, 85-103.

12. Warnock, John. "John Warnock." In Programmers at Work. Ed. Susan Lammers. Microsoft Press: Redmond, WA, 1986, 40-55.

13. Weinberg, Gerald M. The Psychology of Computer Programming. Van Nostrand Reinhold Company: NY, 1971.

14. Young, Richard, Becker, Alton, and Pike, Kenneth. Rhetoric: Discovery and Change. Harcourt, Brace & World, Inc.: NY, 1970.