

# On Criteria for Grading Student Programs\*

James W. Howatt  
Department of Computer Science  
St. Cloud State University  
St. Cloud, MN 56301

jhowatt@StCloudState.edu

**Introduction.** For many of us, the initial experience of grading programs brings a quick realization of the need for objectivity and consistency. When grading, we should consider only the program before us and the requirements that led to it. We should not be swayed by comparisons with other programs, by distracting characteristics of the code, or by past interactions with the coder. Also, we should focus on the same concepts to the same degree in every program, otherwise grading becomes nonuniform (a fact that students see quickly). This is especially important when several teaching assistants are grading for the the same course. To help us achieve this needed consistency and objectivity, we developed set of grading guidelines. Originally defined in 1989, they have evolved into the criteria described here.

Since we were reasonably sure that others had recognized the same needs, we looked to the literature for some guidance. Our search revealed few papers. Miller and Peterson [3], in 1980, developed grading guidelines for the reasons described above. But, their paper focuses on statistically validating their criteria rather than on developing them. In 1983, Hamm and his colleagues [1] identified factors they incorporated in an automated method for grading, but they, too, failed to describe their selection process. In 1988, Olson [4] compared analytic and holistic methods of grading. Although he clearly defined his criteria for each category, he said little about his rationale for selecting them. This paper takes a different slant. Instead of defining criteria for further study, we focus on the criteria themselves.

The criteria defined here are less dichotomous than Olson's. If criteria are too analytic, they assess only program "mechanics"; if they are too holistic, they may slight important concepts. We combined his two approaches by defining analytic categories, but within each, setting holistic criteria.

The following paragraphs define the current version of our criteria, describe the relative ranking of each, discuss how they evolved, and describe our experience using them.

**Grading Criteria.** The following six criteria guide our evaluation of student programs.

**Program Execution.** Programs should compile and run cleanly, and produce correct results. This criterion covers the usual characteristics of output, including values, formatting, and

---

\*This paper appeared in *SIGCSE Bulletin*, (26, 3) pp. 3-6.

completeness.

**Specifications.** Programs that produce correct output sometimes fail to satisfy other assignment specifications. This criterion targets nonfunctional aspects of an assignment. Typically, we find programs that do not contain required language constructs, that are improperly modularized, or that request different input or differently formatted input than that specified.

**Program Design.** Programs that satisfy specifications and produce correct output can be “designed at the keyboard” instead of thoughtfully planned. This criterion focuses on the amount and quality of planning reflected by the code. Prime topics include program structure, modularity, and algorithm and data structure selection.

**Coding Style.** Even well-designed programs can be difficult to read and understand. This criterion spotlights the mechanics of coding, including variable and subprogram naming, use of language constructs and capabilities, and use of white-space.

**Comments.** We distinguish comments from style because we consider them “outside” the program, like footnotes in papers. Although we encourage code that needs little commenting (as suggested by Ledgard [2]), students must include administrative information in the main program, provide pre- and post-conditions for each subprogram, and describe, briefly, each logical paragraph of code. Comments should enhance understanding of the code and must be correct with respect to the code.

**Creativity.** If a program exactly satisfies the criteria described above, it does not necessarily deserve an “A”; it provides only what is expected. True “A” work provides more. Thus, this criterion. Students can earn “creativity” points by devising particularly interesting solutions, by making code more robust than required, by presenting output in a particularly informative fashion, or by simply writing in a style that makes the program a pleasure to read.

**Ranking the Criteria.** After selecting the criteria, we had to decide how to apply them. This was tougher. Which of the criteria are more important than the others? By how much? After much reasoning and a little trial and error, we settled on the following.

Problem solving is a thread through all computer science courses. Design is a major tool in problem solving. So we ranked “Program Design” at the top.

At the beginning of each course, we emphasize that student time is best spent designing a good solution. We describe the steps they should follow: understand the problem (we get the occasional program that solves a different problem), develop a high-level solution, refine that solution and before coding, ensure that the solution completely and correctly solves the problem. Periodically, we restate this, emphasizing how much better their programs will be and how more easily they can write their programs if they follow these steps.

Incorrect results and unsatisfied requirements make even a well-designed program useless. So after design, we placed “Program Execution” and “Specification Satisfaction”. We ranked the two equally because they are not totally independent. A program that fails to produce correct results also fails to satisfy the specification. A program that completely satisfies a specification normally

produces correct results. By ranking these after design, we hope to show that producing correct results is a product a good problem solving process.

After well-designed and correct, we put “Coding Style”. Students must learn to use a language appropriately and to express algorithms clearly. They should treat programs like other written documents. Punctuation, paragraphing, proper use of grammatical elements and all the other concepts they learn about writing papers apply equally to programs. We sometimes even suggest that they write successive drafts of their programs, each version cleaning up the previous.

We also discuss how style affects software maintenance costs and emphasize that costs can be reduced by making software easier to understand. Having the students modify a style-poor program helps drive this point home.

After style, we placed “Commenting”, further emphasizing that design and style are more important to program understanding. We encourage the use of short informative comments and code that speaks for itself. We also stress that content is more important than format, but that poorly formatted comments can “hide” the code.

At the bottom, we put “Creativity”. Most students need to focus more on developing well-designed, understandable, correct solutions than on creatively extending solutions.

The weight we give to each criterion varies by course. If the emphasis is on problem solving and algorithms and data structures, then design receives most emphasis. In language-specific courses, where the central issue is appropriate use of language capabilities, style is heavily weighted. Regardless, design is never worth less than any other criterion. Students must be encouraged to think before they code.

We were unsure, at first, how to weight creativity. Miller and Peterson [3] scored it as 20% of the program grade. But, since we wanted little emphasis on it, we weighted it at 10%.

**Evolution of the Criteria.** We started with just three categories: Execution, Design and Documentation. The first included both execution and adherence to specifications. But to the students, execution implied only results. Therefore, we split the criterion to make the concepts distinct.

The second originally covered both design and coding style. However, the students tended to skip the design, evidently thinking that if they wrote well-styled programs, the criterion was satisfied. After grading many ill-designed programs, we split style and design, clearly distinguishing preparation from implementation.

In the third, we originally included internal comments, external writeups, and style characteristics that affect program readability. After we defined the separate style criterion, we included all language characteristics there. Later, we defined separate guidelines for external documentation. Since only comments remained, the criterion was renamed.

We added the creativity criterion in the Fall of 1993 to distinguish truly outstanding programs from simply sufficient ones.

We never change the criteria in the middle of a course, but we do “tweak”, and sometimes radically alter, them between courses to incorporate our experience using them.

**Applying the Guidelines.** On the first day of class, the students receive the form shown in Appendix I. We explain each criterion and the rationale behind it. Students see the relevant properties of good programs and the relative value of each. They don’t always agree with the properties, but they do know, up front, what we expect from them. We remind them to use the handout to evaluate their programs before submitting them.

The students receive a grading form (see Appendix II) for each program they submit. While specific errors are marked on the program listing, the form shows the larger areas that may need attention. In the “Remarks” part, we comment on larger concepts or on the program as a whole.

We admit an uneasiness about giving students credit for programs that run incorrectly, or that don’t even compile. But, often the causes are minor, and the students deserve credit for what they do right. Programs containing major errors are also usually ill-designed, stylistically poor and uninformatively commented. The criteria allow those programs to be heavily penalized.

We expected complaints about the creativity criterion, but the students seemed to accept it with few reservations. Of course, they want specific definitions of creative work, but we try to say no more than the hints given in the grading criteria (Appendix I). To earn these points, most students add more error checking than required. While this is laudable, we would like to see additional effort in other areas as well. We may eliminate creativity as a separate criterion, and incorporate it into the other five.

**Conclusion.** After we began using the criteria, we graded more objectively and with more consistency. We were better focused on the areas we had determined to be important, and were far less distracted by very good or very poor characteristics of individual programs. Students also seem satisfied with the grading results; their questions about our grading dropped significantly.

We found, not surprisingly, that specifying what we expect from the students as they design and write their programs, as well as what we expect from ourselves when we grade those programs, has had a distinctly positive effect. The students can see and incorporate the concepts that improve their skills (and their grades), and graders can uniformly assess how well they did it.

## References.

1. Hamm, R. Wayne, Kenneth D. Henderson, Marilyn Repsher and Kathleen Timmer, “A Tool for Program Grading: The Jacksonville University Scale”, *SIGCSE Bulletin* (15, 1), 1983, pp. 248-252.
2. Ledgard, Henry and John Tauer, *Professional Software, Volume II: Programming Practice*. Menlo Park: Addison-Wesley. 1987.
3. Miller, Nancy E. and Charles G. Peterson, “A Method for Evaluating Student Written Computer Programs in an Undergraduate Computer Science Programming Language Course”, *SIGCSE Bulletin* (12), 1980, pp. 9-17.
4. Olson, David M., “The Reliability of Analytic and Holistic Methods in Rating Students’ Computer Programs”, *SIGCSE Bulletin* (20, 1), 1988, pp. 293-298.

## APPENDIX I

### CSCI 283 – Programming in C Fall Quarter, 1993 Program Grading Criteria

#### A. Program Design (25%)

<u>Rating</u>	<u>Criteria</u>
5	Solution well thought out
3	Solution partially planned
1	<i>ad hoc</i> solution; program “designed at the keyboard”

#### B. Program Execution (20%)

<u>Rating</u>	<u>Criteria</u>
5	Program runs correctly
3	Program produces correct output half of the time
1	Program runs, but mostly incorrectly
0	Program does not compile or run at all

#### C. Specifications Satisfaction (20%)

<u>Rating</u>	<u>Criteria</u>
5	Program satisfies specifications completely and correctly
3	Many parts of the specification not implemented
1	Program does not satisfy specification

#### D. Coding Style (15%)

<u>Rating</u>	<u>Criteria</u>
5	Well-formatted, understandable code; appropriate use of language capabilities
3	Code hard to follow in one reading; poor use of language capabilities
1	Incomprehensible code, appropriate language capabilities unused

#### E. Comments (10%)

<u>Rating</u>	<u>Criteria</u>
5	Concise, meaningful, well-formatted comments
3	Partial, poorly written or poorly formatted comments
1	Wordy, unnecessary, incorrect, or badly formatted comments
0	No comments at all

#### F. Creativity (10%)

0 to 5 points to programs that usefully extend the requirements, that use the capabilities of the language particularly well, that use a particularly good algorithm, or that are particularly well-written.

APPENDIX II

CSCI 283 -- Programming in C  
Fall Quarter, 1993  
Program Grade Sheet

Student: \_\_\_\_\_ Program: \_\_\_\_\_

Criteria Grades (0 to 5 points each):

(A) Program Design: \_\_\_\_\_ x 5% = \_\_\_\_\_

(B) Program Execution: \_\_\_\_\_ x 4% = \_\_\_\_\_

(C) Specifications Satisfaction: \_\_\_\_\_ x 4% = \_\_\_\_\_

(D) Coding Style: \_\_\_\_\_ x 3% = \_\_\_\_\_

(E) Comments: \_\_\_\_\_ x 2% = \_\_\_\_\_

(F) Creativity: \_\_\_\_\_ x 2% = \_\_\_\_\_

Late Submission Penalty: \_\_\_\_\_

Overall Program Grade: \_\_\_\_\_ Total % = \_\_\_\_\_

Program's Point Value = \_\_\_\_\_

Your score = \_\_\_\_\_

Comments: