Programming is Writing: Why Student Programs Need to be Carefully Read

Gary T. Leavens, Albert L. Baker, Vasant Honavar,
Steven M. LaValle, Gurpur Prabhu
TR #97-23
December 1997

Keywords: software, programming, writing, student ratio, teaching assistant.

1994 CR Categories: K.3.2 [*Computers and Education*] Computer and Information Science Education — Computer science education.

Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Programming is Writing: Why Student Programs must be Carefully Read

Gary T. Leavens* Albert L. Baker Vasant Honavar
Steven M. LaValle
Gurpur Prabhu
226 Atanasoff Hall
Department of Computer Science
Iowa State University, Ames, IA 50011-1040 USA

December 19, 1997

Abstract

Teaching a student to write computer programs well is much like teaching a student to write English prose well. That is, although a program must be correct in every last detail, achieving correctness is only half of the task. The other half consists of intangible factors such as clarity, organization, conciseness, maintainability, etc. Although these factors cannot be automatically measured, they have a large economic impact, because a major cost of software development is the time spent by other people reading programs to validate, maintain, and enhance them.

To teach these intangible factors, student programs must be *read* by a skilled programmer. Furthermore, grades for programs must be partly based on these intangible factors. Completely automatic testing and grading of student programs by machine not only ignores these intangible factors, but fosters the attitude that such factors are unimportant. When programs are automatically tested and not read, students come to believe that functional correctness is all that matters. They tend to write programs by making changes in an initial attempt at a program until it "works." The result is students who cannot write programs well.

From this analysis and our experience, we conclude that enough human resources, such as teaching assistants, have to be made available for programming courses to ensure that there is adequate time for careful reading of student programs.

1 Introduction

Computer programmers are in demand because programming is highly-skilled work and there are too few programmers. Computer programming is a precise and technical skill that cannot be learned without extensive training and practical experience. This is because computers are extremely literal-minded when they carry out their instructions (as described further below). Because employers are reluctant to hire people who have no training, students seek formal training in programming. This creates a high demand for computer programming courses, as evidenced by the sustained high enrollment since the late 1970s [8].

*Leavens's work was supported in part by NSF grant CCR-9503168

In a typical research university, however, high demand for courses and limited resources leads to problems. In this report we take our own university and department as a case study of such problems, which we believe illustrate the problems facing computing departments everywhere. We focus on a vital resource: the course staff, including professors, teaching assistants (TAs), and other humans (such as lab monitors) who assist students in ways that machines cannot. (These TAs are not necessarily graduate students; undergraduate students have been successful TAs in some introductory courses [10]. Similarly, the professors need not be primarily involved in research; dedicated instructional staff is used in many highly-rated computing research departments [7].)

In our particular case, as at most research universities, it is the number of TAs (and the ratio of students to TAs) that varies most rapidly when resource problems occur. Hence, we take the ratio of students per TA as a prime indicator of the course staffing.

At a college where there are no TAs, the resource problems would show up in the number of students per staff member. Since this is primarily a case study, however, we use "TA" below to mean the people who are hired to do grading in a course, regardless of whether they are undergraduates, graduate students, instructors, or professors.

What happens when the demand for programming courses increases, or when the number of TAs available for such courses declines? Two responses to such a dilemma are possible:

1. one can assign more students per TA, or

2. one can limit enrollments to maintain the student to TA ratio.

Clearly, neither of these responses is ideal. (We reject as harmful to students the possible responses of intentionally failing large numbers of students early in a course or teaching so poorly that they become discouraged.) Ideally, more money could be used to pay for more TAs, thus maintaining an acceptable ratio of students per TA.

However, universities change slowly, and while demand for computer programming courses has risen dramatically in the past few years, during this time many universities have seen flat or declining budgets. Our own department has been fortunate to have a fairly steady budget over the past decade, but is experiencing an increase in enrollment. Hence, like many other computing departments, we have been faced with the dilemma described above.

In the past our department has responded to this dilemma by assigning more students per TA in our programming courses. Whereas we once typically had about 25 students per TA in our courses, this semester (Fall 1997) we have 50 students per TA.

In this report, we explain why increasing the number of students per TA beyond a limit of about 25 is a mistake and a serious disservice to students. In Section 2, we explain the intangible factors that good programs should exhibit and describe their economic importance. This section concludes by drawing the analogy of writing programs to writing English prose. In Section 3, we explain the necessity of having students write programs in programming courses, and why it is difficult to have TAs carefully read student programs if the number of students per TA rises much beyond the limit of 25, at least in introductory courses. In Section 4 we describe our observations of how students program when their programs are automatically graded by machine and not carefully read by humans. Finally, in Section 5 we summarize this argument and offer some concluding remarks.

2 Programming is Writing

To the outside observer (who is not a practicing mathematician or computer scientist), programming seems like something akin to solving a mathematical problem. Memories of algebra or trigonometry may lead one to believe that the process of getting the answer is not as important as geting the right answer. Nothing could be further from the truth.

Writing a program is not at all like finding a number that solves some problem. Even if the task is to find a specific number, writing a program is more like writing a chapter in a textbook that tells people how to solve such problems. The programmer does not, personally, find the number, but writes instructions for finding the number.

This analogy between a computer program and a textbook chapter is enhanced by a closer look at programs. A computer program is a list of detailed instructions for a machine, together with some associated "comments." The instructions themselves make up an *algorithm*; the algorithm is like the rules for carrying out some task and formulas that may be found in a textbook chapter. However, unlike a textbook chapter written in English, algorithms expressed in a computer program are completely formal; that is, they have a mathematically described syntax and semantics, and must be 100% syntactically correct in every detail. The comments (or "documentation") are like the remarks in a textbook; they provide additional explanation and motivation, history, descriptions of purpose, or overviews.

Of course, if only one problem is to be solved or one number is to be found, then usually a computer program is not needed. The analogy to a textbook chapter holds again, in that programs usually have to be able to solve all (or many) instances of a particular type of problem instead of just one instance.

Although programs are a means to instruct a computer, humans have to read them as well. Indeed, in the software industry, the human readers of a program are just as important as the machine. This is critical for several reasons:

- Programs are often written in teams, and team members need to understand some of each other's code.

- Debugging a program (getting it to work correctly) also requires reading. Here the reader is often the program's author; the difficulty of finding bugs in a program [2, Chapter 13] shows the difficulty in reading programs carefully. Even the program's own author will have difficulty in reading a program that is unclear, poorly organized, or poorly documented.

- Programs are often read during "code walkthroughs." Here the readers are programmers other than the program's author, who read the program carefully to validate its correctness [2, 9].

- Programs are read by "reusers," people who wish to use or adapt the code for another purpose.

- Perhaps the most important reader of a program is the maintenance programmer. This is, very often, a different person than the program's author. The maintenance programmer has to understand a program to fix or enhance its functionality. Often the author of the program is not available to answer questions.

Because about half of the cost of a program is spent in its maintenance phase [1, page 18], maintenance costs have an enormous economic impact. "Studies have shown that 30-90% of software expenditure is spent on maintaining existing software [12, 14]" [11, page 66]. "Software engineers generally agree that the total cost of maintenance is *more* than the cost of development of software" [6, page 14].

"Studies have also shown that maintenance programmers spend about half of their time studying the code and related documentation. This has led Standish [12] to conclude that the cost of comprehending a program is the dominant cost of a program over its entire life cycle" [11, page 66].

Thus, training students to write a program so that it is clear, concise, well organized, well documented, etc., is vital for the economic health of the software industry. Hence, it is necessary to emphasize these aspects of writing programs when training students.

In summary, writing computer programs is much like writing a textbook chapter to instruct (a computer) to solve a collection of problem instances. For good economic reasons, programs must be written so that they can be easily read by others.

3 Too Many Students Leads to Less Careful Reading

Since writing programs is a skill, it has to be practiced to be learned. Who would imagine that an English course in technical writing could be taught without writing assignments? In a computer programming course, it would be equally absurd to try to teach programming without having students write programs. Typically many programs are assigned over the course of a semester, so that the students are more or less constantly writing programs.

What students write must be carefully read, as feedback is necessary to develop writing skills. In an English course, the writing assignments are carefully read by qualified instructors. To allow sufficient time for careful reading, English departments limit the number of students that are graded by each instructor. The same should hold for courses in computer programming.

In the following we again use our own department as a case study, and specialize it to our own introductory (200-level) undergraduate courses.

3.1 How much time TAs have available

One way to get a handle on the amount of time available for reading student programs is to estimate the time available for the task. In our department, TAs are usually required to attend classes, teach a discussion section, hold office hours, and meet with the professor. We estimate that TAs spend at least 8 hours per week, T_{fixed}, on such fixed activities (see Table 1).

The remaining time for which a TA is hired we call $T_{available}$. That is $T_{available}$ is the budgeted time, T_{budget}, minus T_{fixed}. In our department, where $T_{budget} = 20$, $T_{available}$, is 12 hours per week. Dividing $T_{available}$ by the number of students in a class gives the available time per student per week.

$$T_{available}/student = (T_{budget} - T_{fixed})/size(class) \tag{1}$$

For $T_{budget} = 20$, this equation is displayed in Figure 1. The following are example data points, in terms of minutes per week available for grading (and other activities, such as helping students outside of office hours):

- for 20 students, $T_{available}/student = 36$ minutes,
- for 25 students, $T_{available}/student = 29$ minutes,

Hours per Week	Activity
3	attending lectures
1	teaching discussion section
1	preparing for discussion section
2	office hours (meeting with students)
1	meeting with professor
8	total = T_{fixed}

Table 1: Hours per week for fixed activities.

- for 30 students, $T_{available}/student = 24$ minutes,
- for 40 students, $T_{available}/student = 18$ minutes,
- for 50 students, $T_{available}/student = 14$ minutes.

3.2 How much time TAs spend grading

We did an informal survey of our TAs, and asked them how long they spent grading programs. The survey was responded to by 9 TAs out of about 30; 8 were TAs for undergraduate courses, one for a graduate course. All of the undergraduate courses whose TAs responded to the survey were introductory (200-level) programming courses. Note that all of the undergraduate courses for which TAs responded have about 50 students per (20 hour a week) TA.

This survey revealed that, on the average, TAs spent about 8 minutes per program doing grading, with a range from 4 minutes (for 3 page programs) to 19 minutes (for 12 page programs with extensive specifications). This high value was for the one graduate course, where the programs consisted of both code and extensive specifications (formalized documentation).

To focus on the undergraduate courses, we disregard the one graduate course; these 8 responses are the bulk of our survey, and give us data for this critical stage in the development of programmers.

For the undergraduate courses, the TAs reported that they all spent less than 10 minutes (on average) grading time per program. The average was about 7 minutes per program. The average length of such a program was 3.25 pages. This means that TAs are currently spending about 2 minutes per page to grade each program. This group of TAs estimated that they spent about 6 minutes on the average reading programs, with time estimates ranging from 1 minute (for 2 page programs) to 12.5 minutes (also for 2 page programs). At the low end, the TA commented that he only read the summary information in the program, not the code, at the high end the TA commented that the estimate of 12.5 minutes was for a program with bugs. This wide variation in reading times is in accord with theory, which says that algorithmically deciding any non-trivial property of a program from its text (even whether the program halts) is impossible [13, 5].

Recall that for the undergraduate courses, the class size was about 50 students per TA. At that class size, TAs have a maximum of about 14 minutes per student per week to spend grading. Yet the TAs responding to the survey reported that they only spent half of that time on the actual act of grading programs. Presumably, the TAs spent some of their other time doing other things, such as grading non-program problem sets, helping students outside of

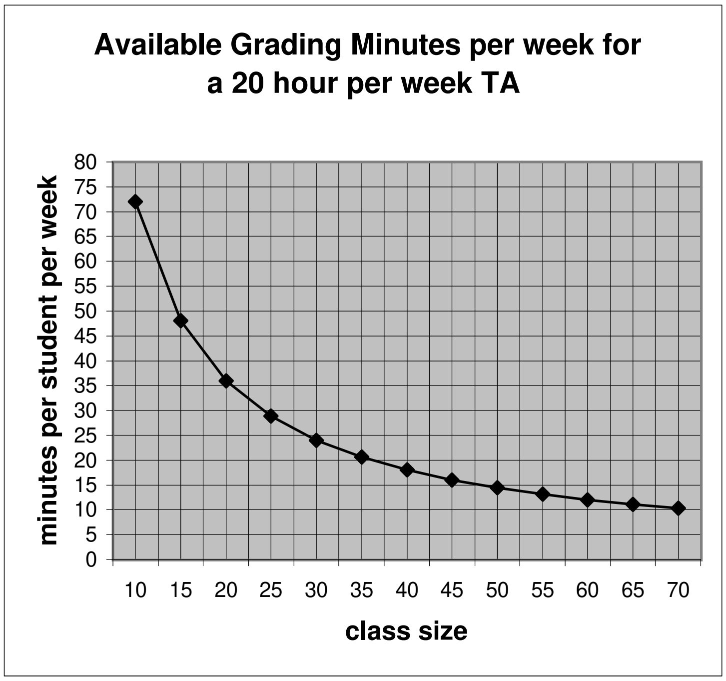

Figure 1: Class size vs. time available after fixed commitments are subtracted for a 20 hour per week TA.

office hours, preparing test cases and grading scripts, recording grades, answering student email, studying the material in the course, helping make up test questions, preparing homework problem sets or files, and so on.

Suppose we modify the equation (1) to reflect the reality that the time spent grading a student's program is only 1/2 of the time available. (In our experience, the amount of time needed for other activities that consume available time is also roughly proportional to the size of the class.) Then we get the following equation that can be used to estimate the time TAs will spend grading programs based on class size.

$$T_{grading}/student = 1/2 \times (T_{budget} - T_{fixed})/size(class) \tag{2}$$

For $T_{budget} = 20$, this equation is displayed in Figure 2. The following are example data points, in terms of minutes per week that this model predicts a 20-hour a week TA will be spent grading:

- for 20 students, $T_{grading}/student = 18$ minutes,
- for 25 students, $T_{grading}/student = 14.5$ minutes,
- for 30 students, $T_{grading}/student = 12$ minutes,
- for 40 students, $T_{grading}/student = 9$ minutes,
- for 50 students, $T_{grading}/student = 7$ minutes.

3.3 How much time is enough?

Now that we have a way to predict how much time will be spent grading based on class size, we can consider whether it is enough for a given class. There are

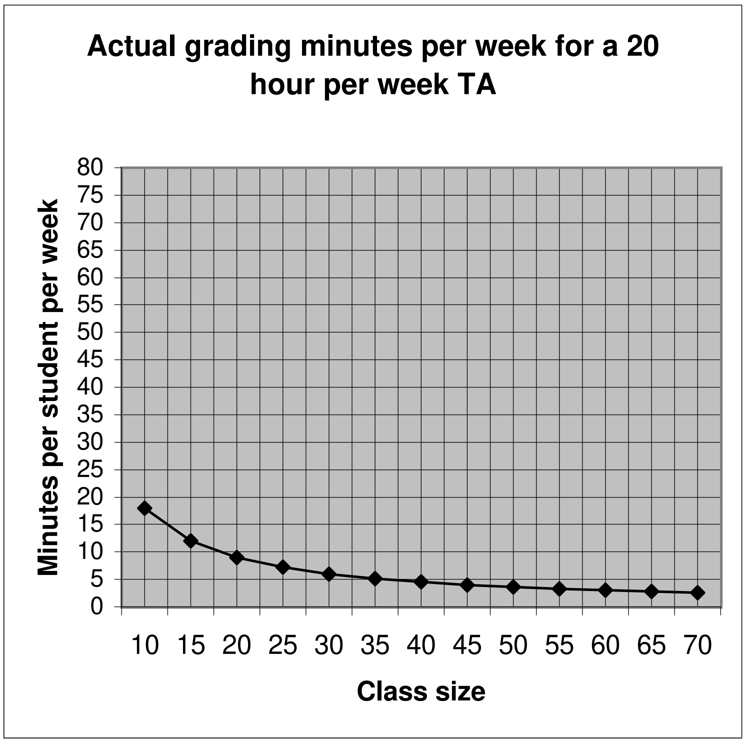

Figure 2: Class size vs. predicted actual time spent grading for a 20 hour per week TA.

several factors to consider in trying to decide how much time is enough to allow for reading and grading programs:

1. How long TAs need to spend reading each page to grade carefully, including reading the programs to grade the important intangible factors,
2. How many pages of program text each student generates per week.
3. How many students the TA must grade for.

The difficulty of reading a given page of code, as noted before, varies with the subject matter and with the thoroughness of the grading on the intangible factors. Our empirical data of 2 minutes per page reflects a reading of programs that is not careful and does not pay much attention to the intangible factors. All of the undergraduate courses surveyed use a grading script to test program code; only 1 of the 8 TAs for these courses seems to read the programs, and none of these TAs seem to grade on the intangible factors.

We are not sure how much more time per page should be spent to give careful attention to the intangible factors. An experienced TA and instructor, Clyde Ruby, says that “most of the time needed for careful grading is needed to write careful feedback” for the students about the intangible factors (personal communication). The issue is how to explain what the student should do to correct the problem; often this means giving an example of how to do it better, and explaining the problems with the student’s way of doing things.

We estimate that if TAs spent about 4 minutes per page of program text, then they would be able to also grade based on the intangible factors, and also write feedback about them for the students. This estimate is also reasonable because TAs will still have to do what they are currently doing to judge the correctness of programs, and that they need to spend time reading the programs to grade the intangible factors. However, more research is needed to determine if 4 minutes per page is accurate.

It is unlikely that the number of pages of code an average student generates per week is a constant, regardless of the level of the class. Studies by Boehm and others [1, Section 26.5] show that, for example, language experience makes a great deal of difference in the time needed to do coding and testing. This would indicate that the practice of increasing the size of programs as the semester goes on (or for later courses) is reasonable.

The number of pages students write per week, however, is only partly controlled by the instructor in a course. Students seem to find ways to make even short programs into long ones. Even professionals vary considerably in the length of programs they produce for the same task; Boehm reports that size of code for the same problem varies by individual with a factor of 5. He also reports that the time needed to code the same problem varies by a factor of 18 [1, page 447]. It may be, however, that this can be somewhat alleviated by teaching students the value of conciseness and basing part of their grade on conciseness.

Nevertheless, we can use our estimate that a rate of 4 minutes per page must be allowed to grade carefully, and the average size of programs in our introductory level courses to estimate the amount of time needed per week for careful grading. Since the average size of a program for these courses is 3.25 pages, assuming that one such program is due per week (or that if a program is not due, a test with the same amount of code is given), then the TA needs $4 \times 3.25 = 13$ minutes per student per week. Using Equation (2), this means that, for a 20 hour per week TA, we would recommend a maximum grading responsibility of about 27 students.

For $T_{budget} = 20$, this number of pages of program per student per week that can be graded carefully (at 4 minutes per page) is displayed in Figure 3. The following are example data points, in terms of minutes per week that this model predicts a 20-hour a week TA can grade:

- for 20 students, 4.5 pages per student per week,
- for 25 students, 3.6 pages per student per week,
- for 30 students, 3.0 pages per student per week,
- for 40 students, 2.3 pages per student per week,
- for 50 students, 1.8 pages per student per week.

Assuming that our average amount of code, 3.25 pages per week, is valid for such introductory courses, one can use this to derive a useful rule of thumb. The rule is that it takes about 15 minutes to carefully grade each student's programs per week. Hence we predict that a TA can carefully grade about 4 students per hour.

3.4 Grading crises and possible responses

What happens when a TA has more pages of programs to read than he or she has time to read carefully? The TA will either: do a less careful job of reading programs, or, if required to read them carefully, will start to fall behind in grading. Both situations lead to lack of feedback for students. Moreover, because the semester has a definite end-point, TAs must finish grading by the

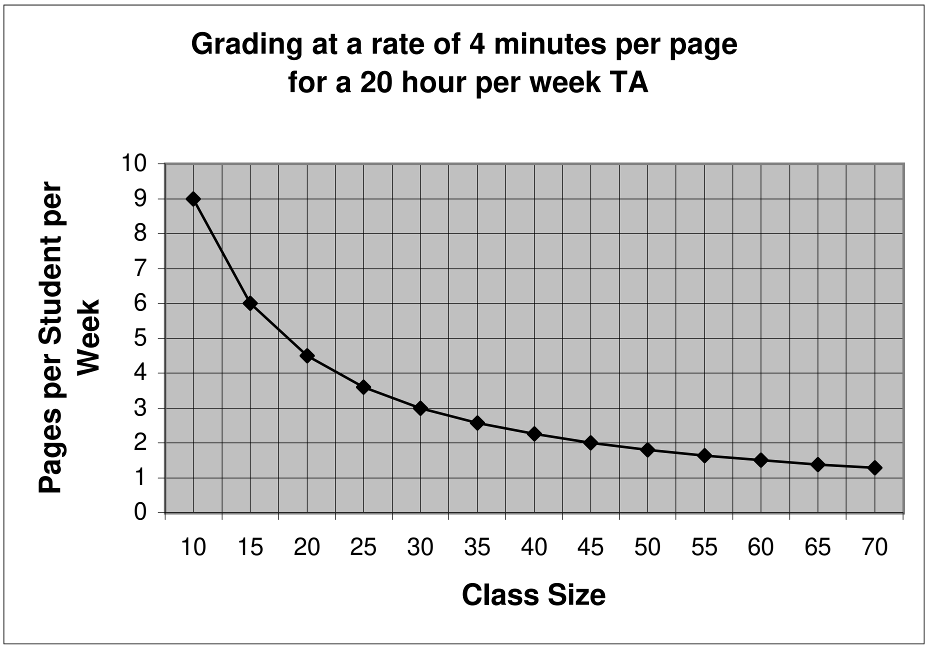

Figure 3: Class size vs. number of pages a 20 hour per week TA can carefully grade, assuming that careful grading takes 4 minutes per page.

end of the semester. A TA who falls behind in reading students programs must therefore, at some point catch up. Inevitably, to catch up, the TA must read the programs less carefully. So in either case programs will not be read carefully.

There are several responses to the crisis that happens in a course when the TAs cannot keep up with doing a careful job of reading student programs.

- The amount of programming homework can be reduced for students. However, this leads to lack of skills on the part of the students. (On the other hand, this may be better than the alternative of automatic grading, which we consider at length below.)

- Some of the assigned homework can be left ungraded. The idea here is to not tell the students which of their homework will be left ungraded. However, since a typical programming course has only 10 to 12 programs that constitute the homework for a class, this is not a viable option. Leaving some of the homework ungraded also leads to poor satisfaction levels on the part of students, lack of feedback, and poor teaching evaluations.

  Some of us have tried giving "suggested practice" problems to students, but students are overworked and tend to just ignore anything that is not graded.

- Group programming projects can be used instead of individual assignments. While this is a good idea in upper-level courses, it does not always work in introductory courses. (Some of us have tried that, but the programs in introductory courses are typically too small for group work.) Furthermore, group projects require more effort on the part of the staff, and tend to generate the same amount of code as would be generated for individual programs, which does not realistically lessen the reading burden. So, while group assignments may be good for other reasons, they do not themselves solve the grading crisis.[1]

- Programs can be automatically graded by running test cases. The idea is that students electronically submit their programs, and the TA can have the machine run test cases over these programs. The scores (how many test cases pass), can be used to generate a grade, without a human ever reading the program. However because machines cannot automatically grade on the intangible factors of writing programs, these factors are completely ignored. We consider the problems caused by this at length below.

- Some programs can be carefully graded, such as those early in the course, and automatic grading can be used later in the course. This would avoid some of the problems with automatic grading discussed below, but by assumption it would be difficult to get students timely feedback on the hand graded programs in the case we are considering.

Automatic grading of student homeworks is an attractive option to both TAs and professors when there are more than about 25 students per TA in a class. This is because, according to our model, the TA will be willing to spend less than 15 minutes per student per week to spend grading. At 50 students, the TA will be willing to spend about 7 minutes per week per student grading, which does not allow time for careful reading to grade on intangible factors. Indeed, it barely allows enough time for automatic grading (at 2 minutes per page) for our average program at this level. Furthermore, the work tends to become tedious with so many students.

Automatic grading does not seem to suffer the disadvantages of the other responses to the grading crisis. It promises to relieve the tedium of checking programs for correctness, and to do a more thorough job of checking for correctness than most TAs [8]. If (some of) the test cases used in automatic grading are made available to students, then students can also get immediate feedback on their programs, by running the test cases themselves, which is good.

Automatic grading also seems to promise reduced costs for teaching students. Once designed, it would seem that an automatic grading system would allow virtually unlimited ratios of students to TAs. Unfortunately, our experience is that automated test systems for computer programs require a great deal of effort in designing test cases. Furthermore, errors in student programs often lead to problems in testing of their programs; hence, in our experience, a TA must constantly attend the running of an automated testing program. When a student's program has a problem in automated testing, often the program must be read to assign a grade; according to our survey, this will take as long if not longer than the automatic grading procedure for that program.

Another problem with programs that do automated grading is that they work best with highly constrained, batch-mode programming problems. If one wishes to teach students about how to design to incomplete requirements, larger systems, or graphical user-interfaces, then automated grading systems become impractical.

Finally, even advocates of automated testing systems in programming courses only advocate the use of such systems to help decide the part of a program's grade that is based on correctness.

[1] One alternative to group programming is a "pairwise exchange," in which two students each write a program, exchange the code, and edit the other person's code to make some change or enhancement. This would tend to emphasize the ideas of readabilty in programming. However, it would not solve the grading problems, as each student would still be writing each program. Furthermore, it could easily lead to more time spent grading, because grading would have to weigh both the original programs exchanged and the relative merits of the maintenance efforts.

> "We emphasize, however, that this testing tool does not necessarily compute scores or grades, nor does it reduce the human judgement involved in evaluating students' work (which includes not only the program's correctness, but also its adherence to the principles of good design, its documentation, and perhaps its user interface or the student's own choice of test data)" [8, page 382].

Automatic grading, as opposed to automatic testing as a supplement to careful reading, would ignore the intangible factors in programs. As we explain in the next section, it is our experience that this causes very severe problems for student learning. Unfortunately, these problems are not immediately apparent.

4 Automatic Grading and Lack of Understanding

What is the response of a rational and often overworked, busy student to automatic grading of his or her programs? First, such a student quite sensibly focuses on doing the minimum needed to get the desired grade in the class. In a class with totally automatic grading, no (other) human will look at the student's programs; hence the student just focuses on the program's correctness. This focus on correctness means that the intangible factors are ignored. After all, why put effort into writing clear, concise, well documented programs if no one is going read them? Thus the first effect of automatic grading is that students do not learn how to write programs for human readers; they ignore the economically important intangible factors.

The second effect of automatic grading is more subtle. Because the student does not have to worry about clarity, organization, and documentation in his or her programs, the student spends less time planning and organizing. The lack of up-front organization and planning in particular is evident from our experience with students in classes where TAs do not have time to carefully read programs and grade on the intangible factors. Typically, a student writing a program sits down in front of a computer, and begins typing with minimal planning. Then the student tries to test the program, discovers errors ("bugs"), and starts to try to fix them ("debugging"). The process of debugging, however, is hampered by the lack of clarity and good organization in the program. (We have seen this problem almost every time we try to aid students in debugging. While they are focused on fixing minor bugs, there are major problems with the overall clarity and organization of the program that, in addition to being the source of as yet undiscovered errors, make debugging nearly impossible.)

As the deadline for when a program is due nears, the typical student response is not to question their method of writing programs. On the contrary, it is often a desperate attempt to fix the program by a process of almost random changes. We have all had the experience with students doing this, and for many students, this is their normal way of writing programs. The complaint "I tried everything I could think of and it still doesn't work" means that the student has tried a large number of minor changes to the details of the program, without trying to alter its basic structure. In essence, the student is using a "generate and test" method of programming, where he or she generates programs, and then tests them to see if they happen to work.[2] Eventually the student gives up in frustration, seeks help from the TA or professor, or in some cases, copies a working program from a classmate.

[2] Writing a program to generate programs by such methods is a different matter; this is similar to genetic programming.

While it may be possible for students to learn small nuggets of information using the "generate and test" method of programming, continued use of this method spells their doom as programmers. In addition to the frustration and the time needed to use the "generate and test" method, there are two fundamental problems with it. First, the "generate and test" method does not help students learn how to write programs well, because the students never learn how to generate correct programs in the way that experts do: by forming a plan, and mentally checking the plan, revising it as needed, and only then writing the program. When an expert finds a bug, he or she is likely to go back and question the overall plan. This process of refining plans, and learning how to write programs quickly and correctly is the opposite of the "generate and test" method. It engenders deep learning of concepts, tactics, and strategies. By contrast, the "generate and test" method is more like playing a video game, in which the student notices what happens to work. Often students do not even stop to find out why their program works when it does. Because they are not directly refining their mental models, students who use the "generate and test" method of programming only learn how to write programs slowly, if at all.

Second, and more importantly, the "generate and test" method does not scale; that is, it just does not work for programs that are larger than a page or two in length. Oversimplifying, and ignoring feedback gathered from testing small segments of code[3], we can see this as follows. Suppose that in a program of 10 statements, each may have 2 "sensible" variants; in this way one can generate 10^2 (100) programs. Assuming it takes a minute to test each generated program, and assuming that only one of these is correct, then it would take the student, on the average about an hour ($10^2/2$ minutes) to find a correct solution. (This estimate matches our experience with students who use this method, and complain about how long even short programs take to write.) However, in a program of 100 statements, if each has 2 "sensible" variants, students would average about 83 hours ($100^2/2$ minutes) to find a correct solution. There is not enough time in a semester to do the same thing with 1000 statements (which would take a year). (There is barely enough time in a lifetime to write a 10000 statement program in this way.) Because it does not scale, the "generate and test" method is not practical. Students who attempt to use it in industry will soon be out of a job.

We would like to think that when students are faced with the failure of the "generate and test" method would learn better methods for writing programs. Surely some students do learn better methods. However, teaching students better programming methods requires two things:

- having sufficient staff with enough time available to help students in the critical moments when they are both frustrated enough to learn a better way of programming and not so frustrated that they give up, and
- reinforcement of the lessons of planning and organization by grading programs partly on issues other than correctness.

In the end, it is less time consuming to teach students how to design and organize programs clearly as a class than to do it one-on-one with each student. Teaching these lessons one-on-one is very time consuming, because it involves reading programs carefully, getting students to see the value of advance planning, and correcting the problems with the intangible factors in their writing.

Furthermore, for such lessons to sink in, they must be reinforced by grading, which means that TAs have to carefully read student programs to grade on

[3] Of course, students should be encouraged to get feedback from small segments of code; doing so is the basis for good modular design of a program.

these factors. In a class with totally automatic grading, however, there is little incentive for students to make a fundamental change in their writing method; instead, it is all too easy for them to learn (and for professors and TAs to teach) a quick technical nugget of information, which lets them generate fewer programs (by lowering the average number of variants they have to generate and test) while keeping their "generate and test" method of writing programs.

By contrast, when students know that their programs will be read carefully by a TA, and graded on the intangible factors that reflect advance planning (clarity, organization, documentation, and test or verification plans) they are forced to think and plan (to some extent) before they program. In such a situation correcting students who fall into the "generate and test" pattern is much easier, because the grading system emphasizes and reinforces the value of advance planning. In addition, because the students are more involved with the writing of their programs, they learn more and increase their understanding as the programming assignments become more complex.

In summary, the lack of planning fostered by totally automatic grading leads to student frustration and ultimately to lack of learning even how to write correct programs. Since totally automated grading of student work causes such problems, we agree with Kay, *et al.*, that automatic testing should be used only as a supplement to human judgement [8]. Such human judgement, of course, comes from carefully reading programs.

5 Conclusions

As a department, we first looked for problems with our teaching of programming when we began to see evidence that our students were not learning how to program well. One piece of evidence was that our students seemed to be less capable programmers when they reached our upper-division courses. Other evidence came from selective employers (such as Microsoft), who began to question the programming skills of some of our seniors.

In investigating these problems, we have come to believe that a major reason for these problems is that student programs are not being read carefully during grading. Because of this, too much emphasis has been placed on correctness issues, which, although they are of first importance, should not completely displace the other intangible factors. This seems to be the reason why so many of our students have been using the "generate and test" method of programming. As we described, this method of programming simply does not work.

Our first recommendation, therefore, is that the teaching programming, especially in introductory classes, must emphasize and grade based on the intangible factors (clarity, organization, etc.). While automatic testing of programs is useful, it must be used as a supplement to careful reading of programs. As we described above, heavy reliance on automatic testing to assign grades leads to all the problems we have experienced.

Our second recommendation is that, at least in introductory programming courses, class sizes should be limited to about 25 students per TA. This assumes that the TA is hired for 20 hours per week. (A TA hired for 10 hours per week, can handle 10 students if they do not teach a discussion section, or 5 students if they do teach a discussion section. For such a TA, the number of students that can be taught is quite sensitive to the fixed overhead.) Based on our survey and analysis, this would allow TAs time to spend about 4 minutes per page reading programs, which would allow them to grade both correctness and the intangible factors carefully. Automatic testing can be used, but will not reduce the time needed to grade a page of program text at this level. If more homework (or tests) than about 3.25 pages per week were assigned, then fewer students would be able to be graded carefully by a single TA.

More research needs to be done to estimate acceptable ratios of students to TAs in higher-level undergraduate and in graduate courses.

More research is also needed to estimate acceptable ratios for instructors or professors who handle an entire course, both teaching and grading. It may be that our basic model of 4 minutes per page holds even for instructors or professors who are grading courses. However, one should note that the fixed time demands on an instructor or professor who is both grading and lecturing in a course are much higher than for a TA. Clearly, such a person would not be able to teach and grade 25 students with 20 hours per week. If the fixed time needed for overhead involves 6 hours of course preparation work, the number would work out to more like 1 student! This is in accord with our own experience that just teaching a programming course, without grading, takes up about 20 hours per week. To ensure adequate time for grading 12 students, such a person would have to spend about 23 hours per week on a course, or about 26 hours for 25 students. While more data are needed for such situations, administrators need to recognize that they cannot just increase the number of students per instructor or professor without affecting quality.

As the number of students per TA increases, something has to give. With large enough ratios, considerations of giving a quality education to students take second place to administrative tasks, such as simply assigning grades to students. As we described above, the use of automatic grading, while it seems to solve the short term problem of assigning grades, leads to lack of understanding, as students focus on correctness issues only, ignore the intangible factors that are important in programming, and use the "generate and test" method. All of this leads to student frustration, and eventually to students leaving computing. (Or, perhaps worse, to poorly trained students who get jobs and make a bad impression on employers.)

As teachers, we believe strongly in student learning and in maintaining the quality of our instruction. Like others [3, pages 41–42] [7], we have noted the high correlation between quality instruction in programming and sufficient human resources. Both our experience and the analysis described show that quality instruction suffers when the ratio of students to TAs begins to exceed 25. When the ratio reaches the point where automatic grading must be used the student learning suffers greatly. These problems are compounded if they are found in the introductory courses, which should lay a foundation for skills in writing programs [4].

We therefore recommend that a limit of 25 students per TA be enforced, at least in introductory programming courses. We believe it is better to teach fewer students well, if need be, than to not teach any students well, and to cause so much frustration, wasted time, and lack of understanding.

Acknowledgements

Thanks to Harry Brearley, Mary Jo Brearley, Don Heller, Janet Leavens, Clyde Ruby, and Akhilesh Tyagi for their ideas and comments on an earlier draft.

References

[1] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[2] F. P. Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley Publishing Co., Reading, Mass., 1975.

[3] R. Dawson and R. Newsham. Introducing software engineers to the real world. *IEEE Software*, 14(6):37–43, Nov/Dec 1997.

[4] N. E. Gibbs. The SEI education program: The challenge of teaching future software engineers. *Communications of the ACM*, 32(5):594–605, May 1989.

[5] D. R. Hofstadter. *Gödel, Escher, Bach : an Eternal Golden Braid.* Basic Books, New York, N.Y., 1979.

[6] P. Jalote. *An Integrated Approach to Software Engineering.* Springer-Verlag, New York, N.Y., 1991.

[7] D. G. Kay, J. Carrasquel, M. J. Clancy, E. Roberts, and J. Zachary. Managing large introductory courses. *SIGSE Bulletin: The Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, 29(1):386–387, Mar. 1997.

[8] D. G. Kay, P. Isaacson, T. Scott, and K. A. Reek. Automated grading assistance for student programs. *SIGSE Bulletin: The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, 26(1):381–382, Mar. 1994.

[9] H. D. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, Sept. 1987.

[10] E. Roberts, J. Lilly, and B. Rollins. Using undergraduates as teaching assistants in introductory courses: An update on the Stanford experience. *SIGSE Bulletin: Papers of the 26th SIGCSE Technical Symposium on Computer Science Education*, 27(1):48–52, Mar. 1995.

[11] S. Shum and C. Cook. Using literate programming to teach good programming practices. *SIGSE Bulletin: The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, 26(1):66–70, Mar. 1995.

[12] T. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, Sept. 1984.

[13] A. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2:153–163, 1937.

[14] Y. Wu and T. Baker. A source code documentation system for Ada. *ACM Ada Letters*, 9(5):84–88, Jul/Aug 1989.