Revista de Sistemas de Informação da FSMA
n. 17 (2016) pp. 40-43

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA PRINCIPAL

Teaching Good Programming Using Evaluation

Ricardo Linden, FSMA

***Abstract* — Programming is considered a fundamental skill for our technological, global world. Nevertheless, it is quite hard to teach and most universities face staggering rates of failure in programming disciplines. Besides, those who pass in these courses are programmers who deliver products below par. I propose some evaluation ideas in order to improve the results and diminish this problem.**

***Keywordss* — Teaching, Programming, Universities, Evaluation.**

I. Introduction

Nowadays, the world sees programming as a fundamental discipline. It is considered that no one can survive in the market without knowing how to make a computer work. This idea may seem preposterous to some, but it is a consensus that learning how to program computers is a remarkable skill that can be of use to any professional.

No matter where you stand on this divide, you will surely agree that Computer Science graduates must learn how to program. Even though it may be possible to work in very specific niches without ever making a more complex program, it is most likely that any graduate in this area must master this technique and be able to create good working programmers.

Therefore, it is our goal to make our students good programmers, who are equipped to be undeterred by highly complex environments while developing large pieces of software.

In order to be a good programmer means, our students should be able to develop programs that work correctly, are easily maintainable and user friendly. These ideas may seem obvious to anyone, but if you had spent some time looking at the products delivered by programmers, you will notice that unfortunately, this is not a truism that goes without saying.

Besides not imparting the correct techniques and behaviors, our classes do not teach to anyone. Watson and Li (2014) made a thorough study and verified that about 1 in 3 students fail in introductory programming classes. This number is lower than the usual anecdotes (and I am sure you have already taught a class with a higher failure rate), but it is still high enough to raise eyebrows. Hence, we should consider what we can do in order to improve these results.

Therefore, in this paper, I will propose some ideas. They are not panacea and should be taken into consideration with the necessary adaptations. Local realities and current practices change the nature of the problem and, consequently, the solution required. Nevertheless, this is a good start, with many good ideas that can be a starting point to improve the results from our classrooms.

II. Evaluation Model

Part of the blame for the students' failure relies on the model we use to evaluate them. Sometimes we are overwhelmed by the large classes that are common in our field and design some easy to grade tests, rather than others that would better evaluate their performance, but mostly, we are simply comfortable with the idea of having a large portion of our classes fail. Nevertheless, there are some ideas that could improve our results which we will discuss in this section.

II.1 Pair programming

The first idea we propose is that students should never code alone. This may seem to go against one of the principles of hands-on discipline, that is, everybody should be hand-on and do the work by themselves, but it looks like that only for those that are not familiar with pair programming.

The idea behind pair programming is that "all code to be sent into production is created by two people working together at a single computer". [2]. In our case, production means "delivered for evaluation".

Even though it may seem counter-intuitive, pair programming increases productivity because code tends to have more quality. In our case, we have the added benefit of having both students working in tandem to address each other faults, that is, when one makes a mistake, the other will have to correct it, based on the fact that his grade is also dependent on the result. With time, the error correction will cause the corrected student to learn the reason behind his/her mistakes.

Ricardo Linden is a Full Professor at FSMA, Macaé-RJ (Ricardo.linden@gmail.com)

This idea has the added benefit of dividing the correction load by half, which would be welcomed by any professor or TA in the area. Nevertheless, this is not the reason behind its adoption.

William et al [3] conducted a formal experiment with pair programming in their introductory programming classes and found out that students who practice pair programming perform better on programming projects and are more likely to succeed by completing the class with a passing grade.

Nevertheless, just adopting pair programming will not be enough to achieve good results if we don't adopt some relevant practices to boost its results. Prottsman [4] suggests some practices that are relevant for success, but two of them deserve comment here, which is to pair carefully and to switch often.

The concept of pairing carefully consists in choosing the pairs in order to maximize results. Based on this idea, students should not be allowed to choose freely their partners. Actually, pairing the best students with the ones in most need of help is usually a good practice, but only if done within the lab, where the professor can guarantee the participation of the lower performance students.

The students that need more help will be more easily identifiable as the term progresses. They should be discovered by their grades or their consistent inability to create functioning work products. Hence, monitoring performance is a necessity for those intending to maximize end grades.

The idea of switching gives us the advantage of creating a communication between different programming practices and always giving a new set of eyes to look for the mistakes a student tends to make.

Adopting these concepts, we together with the third practice proposed by Prottman (encourage respect), we get an additional benefit, that is to train our students to work in groups and to listen to other persons' opinions. This is a characteristic that is much appreciated by recruiters and workplaces in general.

II.2 – Test Cases

It is close to a consensus that programs cannot be graded binary (correct or incorrect). There is some nuance in how well the software developed by the students actually fulfills the goal of the exercise.

Usually, grades are defined in a somewhat arbitrary fashion. The professor defines some criteria on which part of the software is more important, which is less and gives points accordingly.

I would like to argue that this is not only quite arbitrary, but goes against the final goal of the software, which is to solve a specific problem. Software is never the goal by itself, but rather a tool to solve a problem.

Hence, evaluation would be much better served if it was performed through test cases. Students should be given all test cases their software is supposed to deal with, with the appropriate response and grades would be given by how well the software complies with the specifications.

The test cases should include a blanket covering of the required solution space and also, their choice should be explained to the students. The explanation is due because in a second moment (late in the first introductory course or early in the second), students should be required to develop their own test cases, attesting the extent to which their software was tested.

All this work means that students should be taught a unit testing tool together with their coding. Many would claim that this is a complication and in purely technical terms, it is, for there is another tool to learn. Nevertheless, giving your students strong testing skills makes them better programmers and ones who are readier for the job market.

The language in which the introductory programming course is defined is not a barrier to the adoption of the unit testing mentality, for there are unit testing tools for most of the main programming languages. There is, for instance, JUnit for Java, unittest for Python, PerlUnit for Perl, cUnit and cppUnit for C and C++ and many others. All share the main characteristics, differing only in their syntax and use.

A secondary benefit of adopting test cases as correction benchmark is that your grades will be fully objective, and there will be less whining at your doorstep.

II.3 – Coding Standards

Every software development company establishes coding standards. They range from the namespace of the variables and functions to the comments' style used. A classroom should have a coding standard, just like any other software development enterprise. This teaches students how to follow rules and also, many good practices in the industry.

I am not presenting here what your conventions practices should be. There are several books and papers on the issue. McConnel [5], for instance, devotes more than 1.000 pages to the issue (if you could get your students to read this book, surely they would learn a lot). The point here is to emphasize the need for a structured development and to establish guidelines. These will foster discipline and even facilitate the correction.

II.4 – Continuous Integration

There are a lot of benefits from adopting additional technologies in your classrooms, but usually they are too hard for the institution and/or the professor. Nevertheless, if they are achievable, you should consider the following.

First of all, you should consider the creation of a continuous integration environment. Continuous Integration is a software development practice where members of a team integrate their work often – usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced

integration problems and allows a team to develop cohesive software more rapidly [6].

Using this type of software, students are also learning new skills, which are nowadays considered of the utmost importance. Nevertheless, you need to implement a lot of software and maybe this will not be easy to do in your institution.

Continuous integration is a group coding technique, but there is no limitation that excludes single programmers from adopting them. Also, when you progress towards larger projects, the students will be ready to work together as a team.

If there is any kind of limitation in your institution, you should consider at least using public Internet repositories, such as Github for your assignments. This type of environment may be considered a fad by some, but is very important nowadays and also, if students make their work public, they can be downloaded and experience the taste of contributing to the programming community.

II. 5 – Participation in Coding Communities

There are many coding communities whose goal is to provide answers to questions other people may have. Most of the time, participation is voluntary and the person answering does not know the one who has the coding problem. An example of this community is Stack Overflow (http://stackoverflow.com).

One way to create an enterprising behavior and motivating students is to offer them rewards for participation in these sites.

In Stack Overflow, for instance, an answer can be upvoted by any participant, if he finds it good and helpful, and accepted by the answering party, if he considered that the answer solved his problem. The first gives the answering user 10 points and the second, 25 points.

You can give students extra points based on the number of points they amassed in a specific time frame (for instance, from the beginning of the semester until the midterm).

Answering questions encourage users to research, learn and experiment with code. It is usually quite addictive and beneficial for all parties involved. Therefore, this is an strategy that has low cost and can extend your classroom into the whole world.

III. Participation Model

The Hawthorne Effect, or observer effect, is quite known in Psychology. It states that individuals tend to modify or improve their behaviors when they are aware that they are being observed. The initial definition was that there would be “an increase in worker productivity produced by the psychological stimulus of being singled out and made to feel important” [7].

From this definition, we understand that a person becomes more productive when singled out. Personal experiments with a small number of classes have led me to conclude that this is also true when teaching programming.

It is obvious that with large classes, comprising 50 to 60 students, this might be a difficult feat to accomplish. Nevertheless, we are interested at this point in the bottom third of a class, for we assume that the others have learned well enough, given that they received a passing grade (we will address the issue of their quality below in this document).

The participation should be within the class framework, calling them out to help you with issues such as helping other students or finishing up a code you purposely left incomplete in the board.

Special care should be taken not to humiliate the students. Their mistakes should be corrected with delicacy and they should be helped to correct the assignments. Otherwise, this educational device will backfire.

A simple Excel spreadsheet containing the names of the bottom third of the class and the dates when they were last called to participate is enough to control this model.

A good technique consists in observing the coding pairs while they are working and write down in the spreadsheet the strengths of the bottom third students. There will always be issues they are good with. Hence, calling them out when these issues arise in class will make it easier for them to excel and achieve the goal of this didactic technique.

IV. Conclusion

Introductory programming classes are a daunting challenge for both the teachers and the professors. The failure rate, even though in average not as high as the anecdotes, is still high enough to raise eyebrows and leave us looking for improvements.

I consider that transforming you class into a software house, using common professional practices such as pair programming, coding standards and continuous integration can be beneficial not only for the current moment, where the students are learning to code, but also to improve their employability in the future.

Besides, common psychological techniques can also show to be beneficial. Engaging students lead to the Hawthorne effect, which can boost performance and increase student confidence and participation.

Most of these techniques are quite easy to employ and some of them, such as pair programming and the use of test cases, can reduce the total effort spent by a professor in these courses.

Even though the term is not widely used nowadays, we still face a crisis in software [8]. Hence, creating better and more conscientious developers in our classrooms may be a good way to diminish the nefarious effects that bad software causes in our businesses and in our society.

REFERENCES

[1] Watson, C.; Li, F. W. B., "Failure Rates in Introductory Programming Revisited", Proceedings of the 2014 conference on Innovation & technology in computer science education, pp. 39-44, 2014

[2] ExtremeProgramming.org, "Pair Programming" Available at http://www.extremeprogramming.org/rules/pair.html.

[3] William, L.; Wiebe, E.; Ferzli, M.; et al., "In Support of Pair Programming in the Introductory Computer Science Course", Computer Science Education, vol. 12, n. 3, pp. 197-212, 2010

[4] Prottman, K., "Three best practices for Pair Programming", ISTE, available at https://www.iste.org/explore/articleDetail?articleid=221, 2014

[5] McConnel, S., "Code Complete", 2nd Edition, Microsoft Press, USA, 2004

[6] Fowler, M.; "Continuous integration", web article available at http://www.martinfowler.com/articles/continuousIntegration.html, 2006

[7] McCarney, R; Warner, J.; Iliffe, S. et al, "The Hawthorne Effect: a randomised, controlled trial", BMC Med Res Methodol, vol. 30, n. 7, available at the address http://www.biomedcentral.com/1471-2288/7/30, 2007

[8] Rice, D., "Geekonomics: The Real Cost of Insecure Software", Addison-Wesley Professional, 2007