ENHANCING LEARNING SUCCESS IN THE INTRODUCTORY PROGRAMMING COURSE

Ardian Greca¹, Vladan Jovanovic², and James Harris³

Abstract - This paper is the follow up of a work-in-progress presented at FIE 2002. The goal of the work is to implement those XP practices that contribute to rapid feedback and learning. Grading is based on student’ achievement. However, it is very important to ensure that a student’s grade is directly related to course content and the amount they have learned. In this paper we present a detailed analyze of the method used along with the assessment of the final method. We also will discuss our evaluation criteria and present the results of evaluation.

Index Terms - C++ programming, introductory programming course, teaching method, XP practices.

INTRODUCTION

A number of computer and information science programs have C++ as a common language in their first programming course. Teaching and learning are two intricately related issues. While a good teaching approach can improve learning, improved learning can make teaching more effective as students become more successful. Thus, they both determine the success rate in introductory programming course. There are many factors that make this difficult. Some of these factors are:

- dealing with the computer mostly for the first time (not as users),
- writing codes for the first time,
- facing a new technology and environment,
- learning a complex formal language syntax,
- learning new and subtle concepts,
- facing a new work ethic,
- differing student backgrounds and abilities.

Over the past few years, many approaches and studies have been proposed and many others are ongoing. They cover a wide range, from ways to encourage and motivate students, determining what factors contribute most to success of introductory programming classes, and success and failure in the instruction of the introductory programming languages [1] – [7], [13]. Reference [1] has proposed an approach for assessing students on the skills they have mastered during the course. Authors state that beside factors that contribute to success such as comfort level and mathematics background are also a pedagogy that would require students to prove they had obtained specific skills [1]. However, this requires a tremendous effort from instructors and there is no guarantee that students will avoid monotony in the course.

The approach presented at [2], notes that traditional programming taught via syntax, has limitations in that students can easily get stuck, slowing their progress dramatically. Other researchers [3] have developed tutors for different topics in C++, such as tutors for using dynamic memory, pointers for indirect addressing, etc. The use of these tutors improves student’ performance, but they can be used only in upper level programming courses. However, others found that fundamental concepts must be introduced early in order to learn through redundancy [4]. This approach can help students to understand concepts, but needs more detailed analyses. On the other hand many researchers/educators are interested in adopting XP practices in the computer science curricula. Reference [13] has studied the nature of pair-programming and examined the ways such practice may enhance teaching and learning in computer science education. However, only pair-programming was analyzed and it was introduced only in upper level courses.

Over the past few years we also have been faced with many of the same problems and concerns discussed in the literature. So, our concern is how to improve and enhance success in introductory programming courses. Each of the approaches proposed so far have their advantages and drawbacks. Thus, based on the above observations and factors that contribute to success, an opportunity to develop new course pedagogy exist. In this context our approach is adopting those XP (extreme programming) practices that increase feedback and learning and assure that the student’s grade is directly related to learning. The different between this and other approaches is that we focus our research on introductory programming courses. To our best knowledge, the novelty of this work is introducing and adopting XP practices in introductory programming courses. The results obtained so far using this method have been promising.

DEVELOPMENT OF THE CONCEPT

Before introducing our approach, we explain how to prepare the background for applying XP practices. We offer an “introduction to programming” course three days a week. There are two academic hours for lectures and two for laboratories. During lectures concepts and coding examples are explained to students and then they are required to write code for problems based on the material covered. There are

¹ Ardian Greca, Georgia Southern University, Department of Computer Science, P.O. Box 8093, Statesboro, GA 30460 agreca@gasou.edu
² Vladan Jovanovic, Georgia Southern University, Department of Computer Science, P.O. Box 8093, Statesboro, GA 30460 vladan@gasou.edu
³ James Harris, Georgia Southern University, Department of Computer Science, Statesboro, GA 30460 jharris@gsu.cs.gasou.edu
approximately five quizzes, two projects and several homework assignments. During the semester each problem is carefully designed to make progress and continuity between the problem and topics. To provide experience with maintenance of larger programs, several labs are cumulative, i.e., resulting code from one lab is a starting point for the next lab. We also found that dividing the course into three parts can increase efficiency and productivity. It is shown in the literature [4] that concepts must be introduced early in order to learn through redundancy. We also believe that this can help prepare students for the basics of programming. In the first part basic concepts of C++ are explained in details with examples. This includes computer fundamentals, variables, arithmetic operations/operators, input/output, math functions, decision making/branching, looping, functions and arrays. In the second part, pointers and classes, constructors, destructors, and member functions are explained. The basic concepts obtained in the first part are used to understand and develop each of the topics covered in the second part. In the third part file, input/output, and miscellaneous topics are explained. We have tried several versions of mixing the last two parts. In each of the parts students will have the opportunity to revise and better understand concepts of C++. Supplementary to language concepts and syntax are a set of standardized examples and representative documentation using encompassing flowcharts, structure charts, class diagrams, decision tables (in some sections), all with support of a MS Visio Professional documentation drawing package.

A list of expected outcomes was decided at the departmental level and have been consistently used as part of a course syllabi for several years used by all sections.

CSCI 1301 General Learning Outcomes

- Ability to read and write short programs (main functions in C++).
- Ability to understand syntax/semantics, identify and fix syntax errors, compile, build and execute short (C++) programs using (Visual Studio) Integrated Development Environment.
- Ability to identify the need for, and declare and/or use constants, variables, comments, and preprocess- or directives.
- Ability to use input and output operators (<<, >>, cin, cout).
- Ability to write C++ code using built in arithmetic and library functions.
- Ability to select appropriate if, for, and while selection and iteration statements and combine them in a well structured manner.
- Ability to read and understand simple program logic by desk checking the code and establish trust in correctness of the program by evaluation program state from start to end without executing the live code.
- Ability to develop and communicate its understanding and descriptions of systems, problems, and requirements using text and diagrams (UML) for documentation.
- Ability to design logic for small programs/functions/methods and represent it graphically with flowcharts, VTOC type diagrams, or similar (Wardner, Jackson structured programming notations).
- Ability to read, write, and use functions in C++.
- Ability to identify the need for and use arrays and strings in C++ programs.
- Ability to understand and use text file I/O.
- Ability to understand how to read, identify, write and use test drivers and test cases for testing simple C++ classes and programs.
- Ability to read, recognize, write, and use class specifications (class headers), class implementations (.cpp code), use (create, manipulate, destroy) objects of those classes.
- Ability to understand the use as well as the ability to overload functions and operators.

CSCI 1302 General Learning Outcomes

- Ability to write recursive functions.
- Ability to create accessors and mutators.
- Ability to create a copy constructor.
- Ability to overload the assignment operator.
- Ability to create, use and understand abstract data types.
- Ability to create function templates and a class templates.
- Understanding of the “this” pointer.
- Understanding linear searching, binary searching and simple \(O(N^2)\) sorts.
- Understanding how to create a derived class.
- Understanding the relationship between the constructors in the base class and the constructors in the derived class.
- Understanding how public, private and protected scope modifiers work with respect to inheritance.
- Understanding overloading functions in the derived class.
- Understanding how to create polymorphic functions.
- Understanding how polymorphic functions can be used to localize the effects of changes to code.
- Understanding the notion of virtual functions and abstract base classes.
- Understanding structures and unions.
- Understanding how to create appropriately use libraries.
- Understanding the use of files, file streams, and file processing.
• Understanding the importance of testing and debugging.

Note that, not all of the sections used XP. This depended on the instructors involved. Those using XP coordinated their work in an experimental manner, varying practices systematically while using common elements of assessment such as exams etc.

XP is a simple set of common sense practices that, when used together, produces quality code and robust code and a clear view of its status. It can be used by small-to-medium-sized teams developing software. It has a wide range of practices, but we have been focusing only on those appropriate to small programming such as simple design, pair programming, collective code ownership, testing, (some) refactoring, and use of coding standard. The reminder of the paper provides explanations of each of these and how they were implemented in our courses.

Simple Design

We asked our students to start writing “simple” code. The students themselves decide what simple means. Then we instruct them what simplicity means. This is that their code should run all tests, that they expresses every idea they need to express, that is no duplicate code, and there are a minimum number of methods. Making the code simple also helps students track changes faster. Since students are learning syntax, writing simple code first and testing it makes it easier to determine errors. This helps them to better understand concepts.

Pair Programming

Pair programming has been discussed for a long time. XP uses pair programming to generate more code and higher quality code than the same two programmers working separately. Programming pairs help to correct each other. They discuss design and coding issues and this helps them to better understand what are they trying to do, and to sustain necessary and sometimes prolonged efforts in order to produce working programs. Pair programming helps especially when students have diverse backgrounds. However, there is a big drawback in that one student may do all the work and the other just stand by without even trying to understand what is going on. At this point, as early and as often as possible, the instructor should visit all pairs continuously and encourage and make sure that both are working on the assignment. This can be easily managed because all the lab assignments are done during laboratory time, where the instructor monitors students. For assignment such as projects and homework that must be done outside class meetings, other methods are used to make sure that each student understands the issue and is involved in the program. For example, some of the methods used are: immediately after the assignment is handed out to the instructor, a quiz is given with a question about the assigned program. If the student cannot demonstrate that they understood the assignment, their grades are lowered on both the quiz and assignment. This allows students to be involved actively in pair programming. Students were observed in lab and frequently asked to introduce changes and share debugging. Lab submission was live, in the presence of both partners, and probing questions, ad hoc tests etc., used to discern whether both partners in a pair have sufficient understanding of all the details and truly benefit from pair programming.

Collective Code Ownership

Anybody who sees an opportunity to add value to any portion of the code is required to do so at any time [8]. Collective code ownership is purported to prevent complex code from entering the system, and developed from the practice that anyone can look at the code and simplify it. The idea is simple, write code and you are the owner. If there is any need for changes you can go ahead and fix it. This is more preferable than using code written by someone else and if you need to change it you must discuss it first with the owner of the code. In industry this is used to prevent the chaos of creating unstable code. In XP, everybody takes responsibility for the whole of the system. This doesn’t mean that everyone knows every part equally well, although everyone knows something about every part [8]. In academia, since students in introductory programming courses are just beginning programming, it is much easier for them to modify their own code than to understand the code from others. On the other hand, different pairs can develop different codes for the same problem. This code varies from simple and correct code to complicated and sometimes not correct. Writing complicated code brings the possibility that students soon will forget their code and it will be very difficult for them to even understand their own code. Thus the collective code ownership becomes very important. We apply this as follows. First we ask different pairs to discuss their codes with each other, and then the instructor discusses the problem’s solution with each pair. This results in the best codes given as a solution to the problem, and the complicated and incorrect code is discarded. Students then become familiar with the best approaches and it is easier for them to modify their code as needed in the future.

Testing

Testing is one of the most important components in XP and also in learning programming. Successful testing helps students get confidence in their coding abilities. Before testing, students do not know the status of their programs. If there are some new changes required, students will be unsure if these changes will improve their code. We instruct students to test their code frequently. As soon as they perform a test, or they debug their code, the picture changes and they are more confident their code works. They can make changes as they are needed. It might look as if testing will slow their coding, but this is not necessarily true. Students spend much more time if there is no testing at all than if they test their
Refactoring

Refactoring is the process of improving the code structure while preserving its function. This problem is especially evident when functions are covered. In the beginning, it is hard for students to understand functions, for example, how to pass by value or by reference. But as soon as they understand these concepts, they tend to write everything as functions, leaving the main function as merely a set of function calls. This makes the code inflexible if changes or other additions to the code are required. Using refactoring students can change code without breaking it. Furthermore, each refactoring step is reversible, so they can try things and if they do not like the changes, they are easily reversed. This lets them experiment, with their code. Thus they can update their code anytime they think it is required. For example, if students are required to compute the total rainfall for 10 years, they could end up with something like this:

```java
for(i = 0; i < nYears; i++) {
   totrainfall += rainfall[i].rain();
}
```

This is a simple coded solution for the problem. It will take a few moments for students to determine what the code does, namely, compute the total rainfall. This can be expressed in a more compact way as:

```java
totrainfall = 0.0;
for(i = 0; i < nYears; i++) {
   totrainfall += rainfall[i].rain();
}
return totrainfall;
```

Testing first (including preparing the tests data) and refactoring reinforced each other. Cumulative labs are also an example of refactoring. The example of calculating simple statistics on an array of numbers was used incrementally to generalize the solution (from array of integers to one with doubles), analyze and improve algorithm for sorting, criticize existing (accidental algorithms) and device appropriate frequency and histogram display algorithms, reuse methods in developing more involved methods for variance and standard deviation on a basis of a mean, median and mode, and finally encapsulating developed methods into classes.

Coding Standard

This is the last practice, which is very helpful in finalizing the program. If the code is not standardized, then it will be very difficult to deal with the many different ideas that, in the end, serve the same purpose. Being familiar with your own code is not enough. You also need to fit your code with the code of others. This allows everyone to use each other’s code. This is important, since students write code in pairs. Although there is nothing new to keeping the coding standard at the desired level, we are stressing some of the key topics related with this. They are:

- **Indentation**. Using a consistent indentation format makes the code more readable.
- **Capitalization**. Using it makes the code more typical on its own.
- **Commenting**. Using comments makes the code more understandable. We have problems with this, since students frequently get lost, even on their own code. Thus, this should be very important component of programming.
- **Names**. Using meaningful names in the code helps student to communicate with each other. Writing names like “cat”, “dogs” are not good programming practice. Adopting practices used by Horstman at San Jose State was our collective recommendation to students.

RESULTS

Any one practice does not perform well on its own. Each requires the other to keep a balance. Figure 1 is a diagram that summarizes the practices. A line between two practices means that two practices reinforce each other. From this figure, it is obvious that the richness comes from the interaction between the parts.
We have been evaluating this method since Fall 2001 in some of our introductory C++ sections. Our long-term objective has been to evaluate whether using these practices helps to improve the student learning and assure that their grades are directly related to their learning. Class sizes vary from 15 to 30 students and lab components were required in all of the sections.

**Success Rate**

The results for Spring 2003, Fall 2002 when we used the XP practices more precisely compared with those of Fall 2001 where we had combined the XP practices with traditional ones were as follows:

<table>
<thead>
<tr>
<th>TABLE I</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE COMPARISON OF RESULTS</td>
</tr>
<tr>
<td>Baseline (no-XP practices)</td>
</tr>
<tr>
<td>Withdraws</td>
</tr>
<tr>
<td>F's</td>
</tr>
<tr>
<td>D's</td>
</tr>
<tr>
<td>C's</td>
</tr>
<tr>
<td>B's</td>
</tr>
<tr>
<td>A's</td>
</tr>
</tbody>
</table>

Baseline data are provided to show cause of our motivation and depth of potential. From this table we observe that the number of students who dropped the class did not change much after Fall 2001 when we started to implement XP practices (the difference is probably because the class size for Fall 2002 and Spring 2003 was higher). But compared with the baseline it decreased significantly. Also the number of students with A’s increased and the number of students that received D’s decreased significantly. The number of students who received B’s and C’s increased. Since the grade of each student was directly related to their learning, using XP practices gives us an improved pedagogy producing better results. As we can see from this table, the grades fell into a bipolar distribution. There are more students that failed the class and there are more students that got a B or a C in the class. This happened because students are working in teams (pair programming) so if they fail to complete the assignment they both fail the assignment. We are still working on how we can improve this. One solution, which we want to try, will be to have different combinations of students for each assignment.

The other issue that we wanted to stress is that students who received B’s had an average grade of 85 points with a deviation of about ±1, while students with C’s had an average of 75 with a deviation of about ±2. This means that grades are distributed in regions, so it will be very easy to distinguish between teams.

**Performance on Examinations**

On average, students in the XP sections performed better on the midterm exams and the final exam. Results are shown in Table 2.

<table>
<thead>
<tr>
<th>TABLE II</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE COMPARISON OF EXAM SCORES</td>
</tr>
<tr>
<td>Exam</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Midterm 1</td>
</tr>
<tr>
<td>Midterm 2</td>
</tr>
<tr>
<td>Final</td>
</tr>
</tbody>
</table>

Midterm exams are designed almost similar for XP and non-XP sections. It contains three parts: The first one deals with testing general knowledge, such as short answers (it is of 20% of the exam score). The second part tests understanding the programming structure such as programming outputs, segment codes correctness, etc (50%). And the third part deals with writing program codes (30%). Although results for sections were XP practices were used give better results, we cannot conclude that XP helped students to perform better on exams.

There are two factors that we think might influence these results. First, students in XP courses performed better in the second and third parts of the exam, while the others performed better in the first and second parts. Furthermore, among students who belonged to the same pair there was a difference in their performance. Some scored very low, while their partners very high.

A common final exam was given for two classes in Fall 2002. The results from each class showed that students assimilated most of the material covered. We believe that XP practices can prepare students to advance faster and learn more through challenging each other.

**Performance on Projects**

On average, students in XP classes performed better on each of the projects assigned to them. The mean for XP classes was 68 with a deviation of 18.7 for the first project and 87 with a deviation of 5.9 for the second project (these values are the average for both Fall 2002 and Spring 2003). While the mean for non-XP courses were 56 with a deviation of 21.4 for the first project and 69.3 with a deviation of 31.7 for the second project. We expected this to happen for projects since students work in groups and they could perform better than working alone.

**XP Programming and Learning**

All students in the XP classes showed a high level of interaction between each other during lab assignments. They
were discussing different questions with each other and usually answered these questions themselves. When they could not give an answer, they asked help from the instructor. However, the interaction with instructor was very short. It was however, much more effective than in the non-XP classes. Most of the discussions focused on advanced discussions rather than on the basic and elementary ones. All assignments were performed in time compared with extra time needed on some assignments when XP practices were not applied. When students worked alone they could get “stuck” very easily and waited a long time for the instructor to come and help them. In most of the cases the questions were very simple and often about syntax errors. In some cases students tried to get help from their nearby friends, but it seemed that they were too busy to help them or they were in different parts of the assignment so they could not get the needed help. However, there are also some cases when one student did all the assignment and the other one just stand by and did not understand much of the work. At this point the instructor intervened to make sure that both students were involved in solving the problem. Finally, the role of the instructor seems very important in applying XP practices. The instructor needs to explain and reinforce XP practices in regular basis in order to get students involved in team learning.

Teaching assistants, also, play an important role in XP programming. They must enforce it and remind students to apply and follow all XP practices. We also note here that XP programming is easier for teaching assistants because students do not rely on them as much as they could if working alone.

Student’s feedback

On the other hand, student opinion on using XP practices has been very positive. At the beginning students thought that working in teams can help them get good grades with little effort. It is common for students to have a friend solve the problem and submit the work together. But they very quickly understood that this was not the case and worked to complete the assignment successfully. The results of anonymous survey indicate that students like the method and their positive response increased from 60% in the beginning to 75% and 80% in recent semesters. Negative results centered around complaints such as “do not like the method”, “cannot understand the method” etc. Positive responses centered around answers such as “like the immediate feedback”, and “like the independent testing”.

CONCLUSIONS

In this paper we have proposed the use a number of XP practices to improve learning in a first programming course. A detailed analysis of our method is given. So far, the results have shown that both goals were met, i.e., to implement those XP practices that contribute to a rapid feedback and learning, and to have a grading based on the student’s achievements. Another important conclusion for this method is that it can be productive only if all the practices of XP are implemented correctly and continuously. Students like this method and as shown, it can produce effective learning. On the other hand, we need to gain more experience with this method. In the last two years some instructors also tried XP practices in teaching introductory Java, introductory VB, SQL programming in database classes, and student projects for a capstone Software Engineering experience. So far, over 300 students have been affected by XP practices at our school. Experimental results obtained so far have improved our practices and student success at the same time. Most students need continual feedback and must be monitored closely to prevent cheating. Minimizing, if not eliminating, cheating was one of the side benefits of XP use so far and a very positive outcome on its own. Thus, we plan to continue to apply this method in our programming classes and adopt it for other classes for which this method can be useful.

ACKNOWLEDGMENT

The first author gratefully acknowledge the Georgia Yamacraw initiative (http://www.yamacraw.org/) for supporting this work. Authors also, want to thank all anonymous reviewers for their valuable comments to improve the quality of this paper.

REFERENCES
