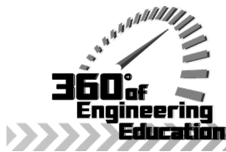

121st ASEE Annual Conference & Exposition
Indianapolis, IN
June 15-18, 2014

Paper ID #9465

Enhancing Computer Science Programming Courses to Prepare Students for Software Engineering

Dr. J. Jenny Li, Kean University

Prior to joining Kean as a faculty member last month, Dr. J. Jenny Li had been a research scientist at Avaya Labs, formerly part of Bell Labs, for 13 years. She is an experienced industrial researcher with more than 70 papers published in technical journals and conferences, and holder of over 20 patents with five pending applications. Her specialties are in automatic failure detection, with particular emphasis on reliability, security, performance and testing. Before Avaya, she worked at Bellcore (formerly Telcordia and now Applied Communication Science) for 5 years. She received her Ph.D from University of Waterloo in 1996.

Dr. Patricia Morreale, Kean University

Patricia Morreale is an Associate Professor in the Department of Computer Science at Kean University, Union, NJ.

©American Society for Engineering Education, 2014

Enhancing Computer Science Programming Courses to Prepare Students for Software Engineering

J. Jenny Li and Patricia Morreale
Computer Science Department
Kean University
1000 Morris Ave, Union 07083 NJ USA
{juli|pmorreale}@kean.edu

Abstract: Most Computer Science (CS) undergraduate programs include an introductory programing course intended to teach basic programming to students of various majors. Students from non-CS majors often find this course to be difficult and tedious, while CS-major students require the course to be challenging enough to establish a solid foundation for their future study of the major. We propose to introduce basic concepts of software engineering into such a course to make it easier for non-CS students to write meaningful programs and to prepare CS students for future software engineering courses. The two concepts are integrated development environment (IDE) and basic software testing. We observed the students' progress and found that on average students can program similar projects 80% faster after learning and using the two software engineering concepts.

1. Introduction

Introductory software programing is an important first-year course that brings students to the door step of the CS major, which we consider as a CS1 course based on the definition given in [1]. It is also a requisite course for many students majored in Science, Technology, Engineering and Math (STEM). The majority of the curriculum of this course is to teach a specific programing language without any introductory concepts of software engineering. For example, students might learn to write simple programs from the textbook using a text editor, but not know how to write realistic programs using an IDE and how to test the programs. Writing programs with a text editor and using a command line to compile the code is the main cause of the non-CS students' complaint about programming being too difficult and tedious. On the other hand, CS students might learn the syntax and semantics of Java or C/C++ to the extent that they can write code for simple problems, but not know how to test the resulting program. The lack of knowledge about basic testing concepts would trigger a scenario with inoperable code, such that the code submitted by students for the course project might look nice syntactically, but could not execute properly according to various test cases. The student code may work for one test case, but not for others.

Without any introduction to some basic software engineering concepts, such as proper usage of IDE and basic testing concepts, the student would not necessarily understand the benefit of IDE and the need for code testing and would not know how to test their code properly before submission. Their training in the programming course might even give them an unhealthy habit of focusing on text typing of programs without regard for proper support environment and adequate testing to insure the quality of their code, which will adversely hinder their future learning of software engineering in higher level courses.

Chen and Hall [2] proposed to apply software engineering to CS1. Their approach seeks to bring in to the course all major software engineering concepts from designing to testing. We found it very difficult to squeeze all the suggested content into the course due to our time schedule constraints. We can only introduce one or two basic software concepts with a total available lecture time of about 1 or 2 hours. We introduced the concept and usage of IDE to make programming straightforward to non-CS students and thrilling to CS students because they can write meaningful code so much faster. We also added basic software testing concept to the course to train students to start with proper software engineering practice of keeping testing in mind while programing. This gives the students advantages in preparation for future software engineering courses while still getting solid CS1 knowledge.

We understand that adding software engineering to CS1 is not a new idea, which has been proposed since 90's [3] [4] [5]. The goal of this paper is to take measurement of impact of introducing various software engineering concepts to help faculty decide what should be added under the tight schedule of the course. After this case study, we are quite convinced that IDE and basic testing concept should be included in CS1. The rest of the paper is organized as follows. Section 2 describes how IDE was introduced into the teaching curriculum and how the impact data was collected and analyzed. Section 3 presents the data collected to compare students' project speed before and after learning the concept of basic testing. Section 4 concludes through data analysis of the previous section that IDE helps to motivate students for programing and that students finish similar projects twice as fast after learning the basic testing concept and they are better prepared for later software engineering courses.

2. Teaching Integrated Development Environment (IDE)

Since the introductory programming course was offered to both CS-major students and other STEM major students, it has many sections. For the fall of 2013, we have 4 sections of the course, i.e. the course was offered to four different classes at four different time slots. Two of the classes did not have Integrated Development Environment (IDE) included in the curriculum, while the other two had. The students of the 4 classes were randomly selected and mixed with students of different majors. We can safely assume that the average academic ability of each class is very close, i.e. their difference is negligible.

We added the concept of IDE to the last two classes very early in the semester. We taught the concept and how to use existing popular IDEs in the second week of the lecture when arrays and methods were introduced. By the third week, when the students were able to write programs with methods, we allowed them to try out Eclipse as their IDE (any version of Eclipse) and write programs using Eclipse in a lab setting. By the fourth week, all students had chosen to install a version of Eclipse at their home machine to use it for their programming homework. Please note that we gave students options of whether to use IDE or use simple text editor with command line compiler and code execution. It turned out that 100% of the around 40 students in the two classes picked IDE.

For the first two classes, we didn't give students an option of using IDE. Many non-software engineering companies such as Google and Facebook in fact do not encourage the usage of software engineering tools. For example, in their recruitment job interviews of new college graduates, these companies require student candidates to write code on a text editor without supports from an IDE tool. However, most companies do use software engineering tools internally for software development.

We were not able to collect data to compare programming abilities of students from the 4 different classes. We did a very brief exit interview of randomly selected students from the 4 classes at the end of the semester. The interview questionnaire focuses on students' perception of whether any IDE should be introduced into the course curriculum and if it helps them in understanding the programming language. Most importantly, one of the questions is whether it helps them to stay in CS major or even switch from other majors to CS. The students found that IDE help them enforcing their knowledge of program language syntax and semantics. They are less likely to make mistakes and IDE is particularly useful to non-CS students who found programing to be more interesting after using IDE because it allows them to try out different ways of programming and learn or to catch up with programing skills. IDEs point out syntax errors for them to figure out how to correct on their own, even without help from a tutor or an instructor.

Overall students enthusiastically support the idea of introducing software engineering tools such as an IDE in the programming course. For example, one student from a class that does not use IDE even decided to retake the course so that she can get into one with IDE. However, as we discussed earlier, some non-software-engineering mature companies still require text editors in job interviews. So we encourage students to learn about how to code using text editor before starting with an IDE tool. The introduction of IDE is brief in this course and for this study. We focus most of our efforts on measuring the impact of introducing the testing concept, which is given in the next section.

3. Introducing Basic Testing Concepts

The key concept we introduced into the last two classes is basic testing. Before the concept was introduced, the goal of coding for students is to have the program run for one specific scenario. For example, a simple calculator program would only do basic arithmetic calculation without checking of input formats or unintended operations such as division by 0.

To overcome the issue of not taking testing into consideration in programming courses, we proposed to incorporate a lecture on basic testing into the programing course to enhance the existing curriculum. Basic testing was added in the beginning of the second half of the semester, by which time the students were able to or had written a real runnable program. At this point, the students can use either conventional approach or test-driven approach to start their next coding project and will be able to compare their outcomes. Through the comparison study, they will be able to better understand and appreciate software engineering testing concepts. Hopefully when they see the benefit, they will be more likely to take high level software engineering courses at the later stage of their study.

Because the testing was only introduced to the two classes with IDE experience, we were able to collect more data points to show students' progress after the learning of the testing concept. We first had students working on a project, *A*, without any discussion of testing concepts and we then introduced the concept of basic testing, followed by their work on another project, *B*.

3.1 Four Measurements

From each project, we collected the following time measurements:

1) Planning time: the duration from a student receiving the problem to the time (s)he starting to write code. Because these students haven't taken any software engineering courses, they don't have any formal knowledge of design and specification. They rely on the reading of the project description to understand the requirements of the project. So this measurement tells us how much time the students take to comprehend the project requirements and it is not the time for design or requirement analysis.

2) Coding time: the duration from when the student starts to write the code to her/his attempt to run the first usage scenario of the project. The definition of this one is critical to our analysis, which clearly divides up coding and testing time. We understand that at this stage that coding and testing might mingle together. Most developers might try to add more code after the first attempt of running the code. Since this is an introductory programming course, even though the projects are realistic and useful, they tend to be in the smaller size of less than 1000 lines of uncommented code, which justifies our assumption that students always finish up coding before starting to run the program for the first time.

3) Testing time: the duration for the student's first attempt to run the program to the time that the first usage scenario works on the program. After the first running of the program, the students might find that their program does not work correctly as planned for the first scenario. They then revise and retest the program until it runs the first scenario properly, which are all included in this test time measurement. Please note here the key word of "first scenario". Before learning testing concept, students tend to think that their programs work or are finished, as long as the main usage scenario can get through the program uneventfully. When the program is executed under some less common but possible scenarios, it often crashes. This leads to the next measurement of revision time.

4) Revision time: the duration from a working main scenario to the time until the program runs for all provided test cases. Since before learning of testing concepts, students might not consider other possible usage scenarios of the programs, they would not expect to include revision time during the planning. So we should see a larger improvement of this measurement before and after introduction of the basic testing concept. This is the key measurement we will study to show the impact of testing concepts.

We collected measurement numbers of all students from the two classes of students with IDE training and randomly selected 15, 5 with high Grade Point Average (GPA), 5 medium and 5 low. Table 1 in Appendix shows the raw data we collected with the 15 students. We found noticeable improvement in the measurements from project A to B, which will be explained in the rest of subsections.

3.2 Comparing Planning Time

Project A and B are quite similar. Project A is to write a program to help veterinarians maintain records of different types of pets, i.e. creating a Pet class with accompanying methods to maintain, store, retrieve, and sort pet records. Project B is to write a program to help realtors maintain property records using a Property class. The structures of the two programs are quite similar and most importantly the two projects have similar number lines of code, as well as similar number of function points. So the students' familiarity with the type of the project should help them reduce the time of the other three measurements from Project A to Project B. As it can be seen from the following Chart 1 that all students spent less time in understanding Project B because their previous experience with Project A.

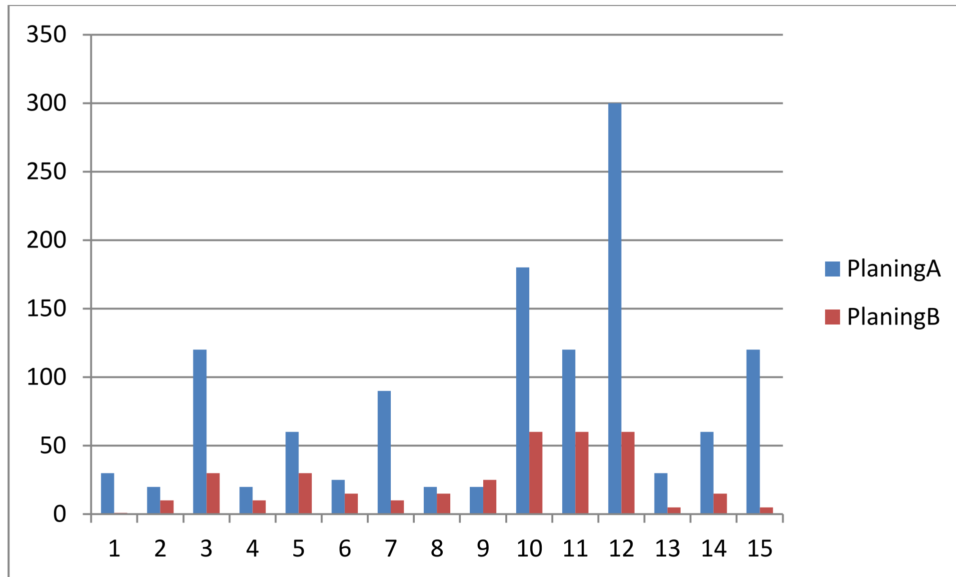

Chart 1. Planning time comparison of Projects A and B.

On the other hand, during the planning time of Project B, they would also need to come up with additional test cases besides the only one main usage scenario in Project A. To maintain our ability to gauge the impact of the second time familiarity to the students' speed in the three other measurements, we exclude the additional test-case generation time from the planning stage by providing those augmented test cases for Project B. In this way, we can use the ratio of the improvement with the planning time as a normalization factor when comparing the other three measurements which are important for the programming course and this study.

On average, the students spent 81 time units to plan for Project A and 23.4 time units for Project B. The ratio of improvement is 3.5 fold. We believe this ratio is an indicator of the impact of students' familiarity with this type of projects and it should be taken away from the other ratios when comparing other three measurements.

3.3 Faster Coding and First Test Case

The coding speed of the students has also improved greatly from project A to project B, which we believe is due to their increasing fluency with the programming language and the IDE tool introduced earlier in the course, as well as their familiarity with the project type. Chart 2 shows the comparison of coding time of the two projects.

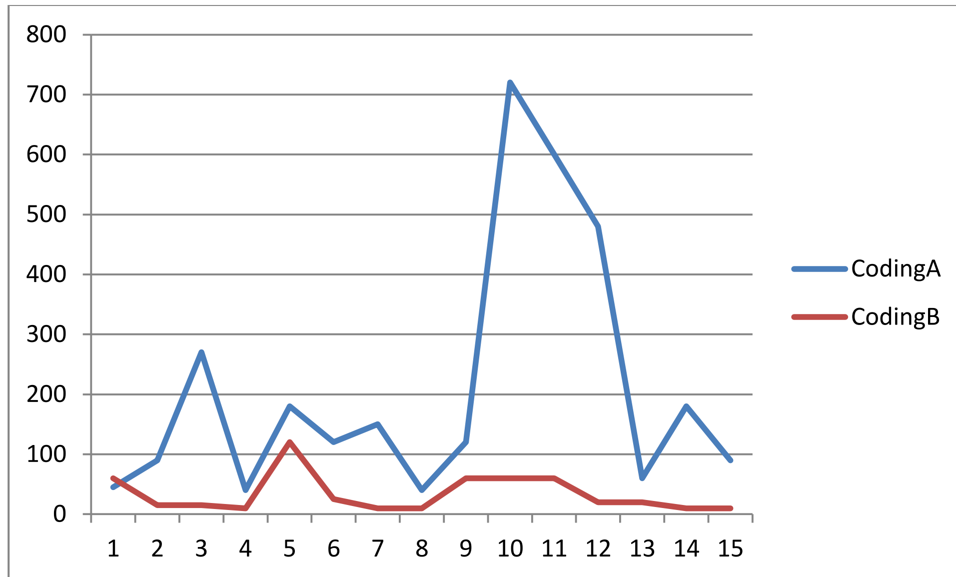

Chart 2: Comparison of Projects A and B coding time

The mean coding time is 212 for Project A and 34, an improvement of 6.2 times, of which 3.5 is caused by students' familiarity with the type of project. So the actual improvement of the students coding speed is 6.2/3.5=1.8 times as fast as the previous project. This number indicates that learning of the two software engineering concepts help student to gain knowledge of the programing language, which is the core of this programing course.

Another observation of the chart caught our attention, which is the change in the measurement variations for the two projects. It shows that the variations among students tend to even out towards the end of the course. Almost every student show an improvement of their programming speed, but the difference among them becomes less and less as the time goes. It seems to support the idea that it gets harder and harder to further improve programming speed once it gets to a certain point, which can be a subject for another study. It also seems to suggest that different levels of courses should be offered to various levels of students.

For the third set of measurements, the situations for ensuring proper execution of the first test case are identical in two projects. So we did not expect much difference between the two projects. It turned out the ratio of the time reduction is 2.2 for this set of measurements.

3.4 Reduction in Revision Time and Overall Project Time

The most important of the four measurements is the revision time because in Project A, students did not think of other test cases except the main usage scenario. This measurement is the indicator of how much testing concept or thinking of test cases before coding helps to reduce the test revision time, and thus overall project time. Chart 3 below compares the revision time of Project A and B.

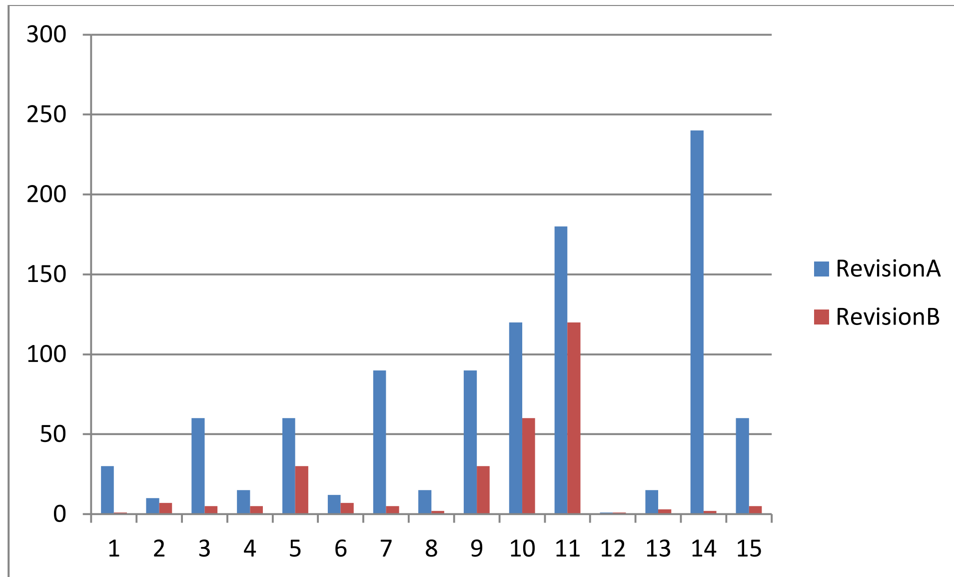

Chart 3. Comparing revision time of Project A and B

Chart 3 might look familiar to readers. Interestingly, the ratio of the time improvement for revision is identical to that of the planning time of 3.5. This reveals that the impact of learning basic testing is equivalent of previous experience with similar projects. This really makes sense to most experienced software engineers. Previous similar experience helps them prepare for how to test a project and vice versa. We see great reduction in revision time after students learned about basic testing, which is the result we expect and hope to validate through this study. The revision time was indeed reduced based on my collected data.

As to the reduction of the overall project finishing time, it has an improvement ratio of 4.4, which indicate a saving of 4.4/3.5=1.3, i.e. 30% reduction in total project finishing time excluding the factor of students' familiarity with the project. All these calculations and their meanings can be found in Table 1 of Appendix.

4. Concluding Observations

This paper reports on a trial of augmenting IDE tools and basic testing concept to the curriculum of an introductory computer science course with four different classes of students. Our data are collected mainly from the second two classes for the trial and they showed that the time to complete projects is reduced, students code fast with the help of IDE and test more effectively with the concept of thinking about likely test cases before coding. The data show an 80% reduction in coding time and 30% reduction in overall project time, a summation of all 4 measurements, planning time, coding time, testing time and revision time. Another advantage of this enhanced curriculum with testing concepts is better preparation of students for future software engineering courses such as software assurance and software testing.

We understand that our conclusion is based on only on one case study in one semester. One future research direction is to follow up with the students at their later stage of their undergraduate study to confirm that indeed more students have selected software engineering courses and they are better prepared for those courses.

The data we collected clearly indicate that IDE's and basic testing should be introduced into introductory programming courses. We hope that this conclusion will encourage computer science faculty to make the choice of augmenting briefly with IDEs and basic testing to their early programing courses. Since the study presented in this paper is straightforward with a limited number of randomly selected samples, we encourage others to repeat the experiment and to compare the impact of introducing other software engineering concepts into programming courses. We hope this paper will contribute to helping faculty make decisions on whether to add software engineering concepts into CS1 and CS2 courses and what software engineering concepts to be added to CS1 for improved student preparation and future success in the major.

5. References

[1] M. Hertz, "What do "CS1" and "CS2" mean?: Investigating differences in the early courses", *Proceedings of the 41st ACM Technical Symposium on Computer Science Education(SIGCSE10),* pp 199-203, New York, NY, USA, 2010.

[2] W. K. Chen and B. R. Hall, "Applying software engineering in CS1", *Proceedings of the 18th \ACM Conference on Innovation and Technology in Computer Science Education*, pp 297-302, New York, NY, USA 2013

[3] J. L. Gersting, "A software engineering "frosting" on a traditional CS-1 course", *SIGCSE'94*, pp 233-237, March 1994.

[4] S. H. Edwards, "Improving student performance by evaluating how well students test their own programs", *Journal of Education Resources in Computing (JERIC)*, Volume 3, Issue3, Sept 2003.

[5] M. H. Goldwasser, "A gimmick to integrate software testing throughout the curriculum. *SIGCSE'02*, 34(1) pp 271-275, February 2002.

6. Appendix

Table 1: The 4 sets of Project A and B time measurements including 15 randomly selected students

Student	PlanA	PlanB	CodeA	CodeB	TestA	TestB	ReviseA	ReviseB	totalA	totalB
1	30	1	45	60	20	1	30	1	125	63
2	20	10	90	15	25	10	10	7	145	42
3	120	30	270	15	30	5	60	5	480	55
4	20	10	40	10	10	5	15	5	85	30
5	60	30	180	120	1	1	60	30	301	181
6	25	15	120	25	60	10	12	7	217	57
7	90	10	150	10	27	5	90	5	357	30

8	20	15	40	10	20	5	15	2	95	32
9	20	25	120	60	30	15	90	30	260	130
10	180	60	720	60	60	60	120	60	1080	240
11	120	60	600	60	60	60	180	120	960	300
12	300	60	480	20	1	1	1	1	782	82
13	30	5	60	20	5	1	15	3	110	29
14	60	15	180	10	60	5	240	2	540	32
15	120	5	90	10	30	5	60	5	300	25
total	1215	351	3185	505	439	189	998	283	5837	1328
average	81	23.4	212.3	33.7	29.3	12.6	66.5	18.9	389.1	88.5
ratio	3.5	1	6.2	1	2.2	1	3.5	1	4.4	1