# Testing First: Emphasizing Testing in Early Programming Courses

Will Marrero
wmarrero@cs.depaul.edu

Amber Settle
asettle@cs.depaul.edu

School of CTI
DePaul University
Chicago, IL 60604

## ABSTRACT

The complexity of languages like Java and C++ can make introductory programming classes in these languages extremely challenging for many students. Part of the complexity comes from the large number of concepts and language features that students are expected to learn while having little time for adequate practice or examples. A second source of difficulty is the emphasis that object-oriented programming places on abstraction. We believe that by placing a larger emphasis on testing in programming assignments in these introductory courses, students have an opportunity for extra practice with the language, and this affords them a gentler transition into the abstract thinking needed for programming. In this paper we describe how we emphasized testing in introductory programming assignments by requiring that students design and implement tests before starting on the program itself. We also provide some preliminary results and student reactions.

## Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education

## General Terms

Experimentation

## Keywords

CS1, CS2, TDD, testing

## 1. INTRODUCTION

A common challenge among those who teach introductory programming courses, especially courses in which Java or C++ is used, is that there is insufficient time to solve problems that are large enough to demonstrate the benefits of object-oriented design. This challenge is clearly present in the introductory programming sequence at DePaul University's School of Computer Science, Telecommunications, and Information Systems (CTI). Almost all information technology students, both undergraduate and graduate, take a two quarter sequence in introductory programming in Java. At CTI, it is especially important to address this challenge because of the large number of students in less technical information technology degrees that do not go on to advanced programming courses and who many not experience first hand the benefits of object oriented programming. Additionally, there is pressure to cover a large number of topics in these initial two courses because of the diversity of the students in these courses. Testing is one of the topics that can easily be pushed aside in order to cover more Java features, especially when the academic year is divided into ten week quarters as it is at CTI. In this paper, we describe how we have emphasized testing in our introductory Java courses, and the benefits that have resulted from this approach. Below, we describe some of the challenges that we have tried to address. In the rest of the paper, we provide background on our introductory programming sequence, we provide examples of our testing first methodology, and we conclude with a summary of our experiences while using this strategy.

### 1.1 Introductory programming challenges

While teaching the introductory Java courses, we have identified challenges that we want to address. Some of these arise from the great diversity of students in our introductory courses and their various educational goals. However, most if not all of these challenges are common across all introductory Java courses.

There seems to be a discrepancy between the importance that educators place on testing and the actual testing that students do during their education. Recently, the ACM and IEEE have pointed out the importance of testing in their final report on computing curricula [6]. This is by no means a new observation. In 1977, Alford, Hsia, and Petry argued the importance of introducing and using software engineering techniques (including testing) in the introductory programming courses [1]. In 1978, Schneider suggested ten essential objectives of an initial programming course of which two were devoted to debugging and testing [7]. Despite the consensus that testing is an important component to all programming, including introductory programming, the importance of testing is often overlooked in introductory programming courses. Sometimes, students will be required


*ITiCSE'05,* June 27–29, 2005, Monte de Caparica, Portugal.

to submit a test suite with their assignment. Often these test suites are incomplete, incorrect, or even missing.

A second challenge is the difficulty that many students face when they first encounter the abstraction of classes and objects in Java. Initial programming assignments often require students to implement small, but complete and self-contained, applications. It is not too difficult for them to understand how the program they write is required to behave. However, as the topics of classes and objects are discussed and students are asked to develop classes that provide some service as opposed to complete applications, students often struggle trying to understand what is required of them. It seems that for many, the transition from a program as providing a solution to a problem to a program as providing a useful abstraction, is a difficult one. This difficulty often manifests itself in the solutions the students submit as well as in the test suites they write for their assignments.

## 1.2 Emphasizing testing

We propose to address these problems by placing a greater emphasis on testing in introductory programming courses. We have done this by designing programming assignments in such a way that students focus on testing up front, either while working on the assignment or before working on the assignment. This idea of thinking about testing before writing code is often referred to as test-driven development (TDD) by advocates of Extreme Programming (XP). This notion has also been advocated by educators, who have proposed various ways of implementing this methodology in the classroom. Implementation suggestions have varied from providing tools to help both students and graders measure test coverage [3], to providing a tool that tests each student assignment submission with each student test submission [5], to organizing exercises so that test writing can precede code writing [2]. We have mostly opted for the simplest technique (requiring test submissions ahead of the actual code submission) because we want to minimize instructor overhead, and because we would like to emphasize testing in introductory courses. The details of our assignments are given in section 3 of the paper.

We have made this change with two goals in mind. The first goal is to have students place a greater emphasis on testing with minimal added work for the instructor and minimal extra time spent in class on the topic of testing. This is especially important for students who need to see testing but do not have a separate advanced course that covers testing. The second goal (and the one that motivated this approach in the first place) is to provide students with an easier transition to classes and objects. The idea of a service class and a tester or client class provide a concrete introduction to modularity. Furthermore, having students focus on code that *uses* a class should help them to get a better idea of what the requirements of the class are. In other words, by using the class, students work on something concrete that helps them to understand the abstraction.

# 2. BACKGROUND

To understand how and why we have integrated testing into the introductory Java courses CTI, one needs to understand the role that such courses have in the curriculum both at the undergraduate and graduate level.

The School of CTI was established in 1996, growing out of the Department of Computer Science in the School of Liberal Arts and Sciences at DePaul University. CTI is now the largest and most diverse institution for information technology education in the United States. CTI offers nine Bachelor of Science degrees and ten Masters of Science degrees, as well as two Bachelor of Arts degrees and several joint degrees with other Schools at DePaul [8]. Over 40 percent of all graduate students in Illinois studying information technology are enrolled at CTI.

The primary programming languages taught at CTI are Java and C++. The majority of degrees require students to take a two quarter Java programming sequence, which includes CSC 211: Programming in Java I and CSC 212: Programming in Java II. Six of the undergraduate degrees in CTI require that students take both CSC 211 and CSC 212, and one requires students to take only CSC 211. Five M.S. degrees require students to take both CSC 211 and CSC 212, four offer at least one track that requires two quarters of Java, and one degree requires only CSC 211. This means that students from seventeen different programs enroll in the introductory Java sequence. While many of these students will go on to take more advanced Java courses such as data structures and software engineering, students in two degrees are only required to take one quarter of Java and students in two additional degrees do not take any Java classes beyond CSC 212 [8]. For these students, it is crucial that instruction about testing be provided as early as possible in the Java sequence.

The first course in the Java sequence, CSC 211, takes a balanced approach between introducing objects early and teaching more procedural concepts. The first five weeks of CSC 211 cover variables, data types, expressions, control structures, and the use of some pre-defined classes. Right before or immediately after the midterm, students begin to learn how to define their own classes, and then move on to cover arrays, and to touch on event-driven programming and graphical user interfaces. In the beginning of CSC 212, students learn more advanced array topics such as sorting and multidimensional arrays. The rest of the course includes event-driven programming and graphical user interfaces in greater depth, inheritance and polymorphism, exceptions and I/O streams, recursion, and the use of fundamental data structures such as lists and stacks. Additionally, while not part of the course description in the catalog, one of the learning goals of CSC 212 is that students be able to write a set of tests to validate the operation of a given class. During the time period relevant to this paper, CSC 211 and 212 used two different textbooks [4, 9], and both take an objects first approach.

Finally, CTI offers a very large number of its courses in a distance-learning format. Each distance-learning section is associated with a traditional section of the course that meets one a week in the evening. Various aspects of the lecture are recorded via CTIs Course OnLine (COL) system. In particular, not only is the audio and video of the instructor recorded, but also all writing on the whiteboards as well as everything presented on the classroom projector. Students in the distance-learning sections can then watch these recordings on their personal computers at their leisure. Both traditional students and distance-learning students submit their assignments via the COL system, typically with the same submission deadlines. Since the authors taught courses with associated distance-learning sections, the testing-first approach was applied to these students as well.

# 3. ASSIGNMENTS AND STUDENT REACTIONS

As CTI began to embrace the objects first approach to introductory programming, we began to notice the difficulty that students have with the idea of classes as abstractions. Many students already struggle with procedural programming when asked to develop a full (but simple) application. When asked to develop a class that can provide some service, many students struggle trying to understand what is being asked of them. This difficulty is manifested in both the solutions submitted by the students as well as by the test suites that they provide along with the class. This initial observation, together with insufficient coverage of testing in class lectures, led us to adopt the strategy of placing a much greater emphasis on testing of assignments. We now describe some of the assignments that students were required to complete as well as how we placed a greater emphasis on testing.

## 3.1 Testing in Java I

Although students in the first Java programming course already face two significant tasks, learning to program and learning the intricacies of the Java language, we hypothesized that introducing a third task, namely testing, would ultimately help them. We tested this hypothesis using an assignment which involved designing and writing a test program for a class followed by the class implementation itself.

Students in two sections, one a traditional section and one a distance-learning section, of a fall 2004 CSC 211 course were asked to write an AirTemp class. This class is intended to represent a temperature reading taken on the planet Earth. The students were given a detailed API for the class which included a constructor, an equals method, and two accessors to retrieve the temperature, one in the Fahrenheit scale and one in the Celsius scale. Students were provided with a compiled implementation of the class and asked to create a test application for the AirTemp class. The main task of the assignment was to create the class itself. The test application was due two days before the AirTemp class.

Table 1 provides average scores for a number of assignments, including the AirTemp class. The data in the first two rows correspond to sections taught in the fall of 2004. The first row corresponds to a traditional class while the second corresponds to a distance-learning section. In both sections, the testing first approach was used. The student performance on the AirTemp assignment, the fifth of the quarter, was poor relative to earlier assignments. In both sections student performance on the testing application and AirTemp class were not significantly different.

This was the first assignment involving the implementation of a class, a task that is typically difficult for Java I students. In order to gauge the impact of the testing emphasis, it is helpful to compare the fall 2004 sections with previous sections taught by the same instructor. The lower three rows of Table 1 provide data for sections of the same course taught by the same instructor during the fall 2003. The bottom row corresponds to a distance-learning section, while the other two rows correspond to two different traditional sections. All three sections were given as an assignment the task of writing only the AirTemp class. They were provided with a test application written by the instructor. Those courses were given one extra assignment, so that the AirTemp class was the sixth assignment. In none of these sections was the AirTemp assignment the lowest scoring assignment.

These results seem to indicate that the testing requirement hurt student performance rather than helping. There are several possible explanations. It may be that the fall 2003 students were overall stronger than the fall 2004 students. To rule out this possibility, one can look a the scores in the fourth column of Table 1 which corresponds to the assignment after the AirTemp assignment and which students in all sections had in common. The similarity in these scores would also rule out the possibility that the student grader was more stringent in the fall 2004 quarter than the student grader in the fall 2003 quarter. Comparisons between midterm exam scores in the last column also yield similar results.

We suspect that students were overwhelmed when they were asked to complete an assignment with two parts, both of which were new to them. This assignment was the first time students were asked to implement a separate class and it was the first time students were asked to write a test suite. Testing and class writing are topics that students do typically struggle with and perhaps some initial practice is necessary before students can begin to do it well. If this is the case, then this additional practice early on could be very helpful and would lead to higher scores on later exercises involving testing. In particular, on the final exam during the fall 2003 quarter, every student was asked a question requiring them to write a client program to test a class completed as a part of the exam. Students did poorly on this question, despite having seen numerous such programs throughout the quarter. We intend to have such a question on the final exam during the fall 2004 quarter in order to test our hypothesis that the emphasis on testing on assignments would lead to students having better testing skills at the end of the quarter.

## 3.2 Testing in Java II

In the second quarter of Java students begin to write more significant programs. From the start they are expected to create and read programs that involve multiple classes and files. It is natural to begin to more seriously address the issue of testing in this course. To do this we developed several assignments that emphasized testing. We describe a representative selection of the assignments and the associated student outcomes below.

### 3.2.1 A Hand class for blackjack

In one instructor's CSC 212 course in the fall of 2003, students had to develop a blackjack application across four assignments. One of the assignments involved developing a Hand class that could be used to represent either the player's hand or the dealer's hand in a game of blackjack. Students had already developed a Card class and were given a correct implementation of the Card class. The design of the Hand class was discussed during the lecture and students were given a skeleton of the class and asked to supply code for the various methods

In order to stimulate deeper thought about testing and about the requirements of the Hand class, students were presented with a testing competition. Students were informed that their test suites would be executed on their own Hand class as well as on the implementations submitted by other

| *testing first?* | *section* | *prior grades* | *AirTemp grade* | *next grade* | *midterm* |
|---|---|---|---|---|---|
| yes | fall04 live | 87%, 79%, 77%, 77% | 60% | 76% | 86% |
| | fall04 dl | 91%, 83%, 62%, 86% | 55% | 75% | 88% |
| no | fall03 live1 | 94%, 99%, 88%, 87%, 75% | 78% | 79% | 85% |
| | fall03 live2 | 84%, 103%, 94%, 92%, 87% | 86% | 83% | 86% |
| | fall03 dl | 86%, 101%, 85%, 80%, 78% | 96% | 72% | 77% |

**Table 1: Student scores for the AirTemp class in Java I**

students and would earn extra credit for finding errors in other students' code. Because the class was small (four students), this could be done manually. Attempting such a competition with a larger class would have been infeasible, even with an automated tool such as the one presented in [5] since the grader would have to identify if the source of any errors was the test code or the class code. The average score on this assignment (after factoring out the extra credit) was 92%. The average scores earned on the three earlier programming assignments were 93%, 99%, and 108%, but the two highest scores included extra credit points. It is useful to compare these scores with a different section taught by the same instructor in the spring of 2004. This later section was much larger and so it was not possible to conduct the testing competition. Students in this section earned an average score of 79% on the Hand class assignment. Students in this section had the same set of earlier assignments and received scores of 63%, 79%, and 88% on those assignments. The significant difference in all the scores can be explained at least in part by the fact that the lower scoring class consisted of all undergraduate students while the higher scoring class consisted of all graduate students who were completing a prerequisite in their graduate program.

While it is difficult to draw a conclusion from the scores alone, it does seem that this assignment has provided a significant step toward achieving our initial goals, namely a greater focus on testing on the part of the students as well as a gentler introduction to classes, abstractions, and requirements. We are very encouraged by the added engagement with testing demonstrated by the students in the testing competition as well as their greater appreciation of the need for precise requirements. These students asked the instructor to clarify various points of the requirements so that they could write more thorough tests. For example, they asked the instructor to clarify how to score a hand that contains an ace. They also asked the instructor to clarify what the upper limit on the number of cards in a hand and the upper limit on the number of aces in the shoe were. These questions were not asked in the other section. Also, a number of students in the section without the competition made the incorrect assumption that the getScore method would only be invoked on a hand with two cards. Ultimately, the tests submitted for the Hand class competition were more thorough than the tests submitted by the same students on other assignments. These tests were also more thorough than the tests submitted for the Hand class by students in the section that did not have a competition.

### 3.2.2 A GradeBook class

Students in three different sections of one instructor's CSC 212 class were also given an assignment that required them to implement a GradeBook class. The GradeBook class had to provide methods to add student names and assignment names as well as actual grades. Once again, all students were asked to submit both a solution and a test suite; however, in the last section students were required to submit the test suite two days before the GradeBook class was due. The average score in the first two sections were 79% and 99%. Again the first score was from an all undergraduate class while the second score was from an all graduate class. The average score in the section that required a test suite first was 89%. This last section consisted of both graduate and undergraduate students.

As in the blackjack Hand class, a number of questions were generated by students who first had to submit a test suite. Students asked what the behavior of getGrade and setGrade should be if the grade book did not contain a matching student or assignment. These students also asked if students and assignments could be added to the grade book after some grades had already been inserted. Students that were not asked to submit the test suite first did not ask these questions.

A better measure of the impact of the testing first strategy is student performance on the final exam. The last two sections had a question on the final exam which asked students to provide a test suite for an IntegerSet class. The class in which testing was emphasized had an average score of 70% on this final exam question while students in the previous section had an average score of 47% on the same question. It should be noted that this improvement occurred despite that fact that the better scoring section covered the same material in only 5 weeks during the summer and so had significantly fewer programming assignments.

### 3.2.3 An Applet for MouseEvents

During the fall 2004, students in a traditional and distance-learning section of CSC 212 were asked to write two applets with listeners for MouseEvents. The first applet displays a colored dot when the user clicks the mouse in the window. Initially the dot is red and centered around position (50, 50) in the window. As the user clicks in the window, the dot is re-drawn centered around the location where the user clicks and is re-drawn in a randomly chosen color with a randomly chosen size. The second applet displays the word "UP" when the mouse is in the window but the mouse button is not pressed, the word "DOWN" when the mouse is in the window and the mouse button is pressed, and the word "OUT" when the mouse is not in the window. Both applets were modifications of examples given in class, and students were given access to web pages containing the .class file for the solution to each applet for a demonstration of the applet's behavior.

Since the assignment did not involve the development of an independent class, it was not possible to ask them to

create a client program for the assignment. Instead, two days before the applets were due, students were asked to create a document describing the purpose and function of each method needed for both applets, and a plan for testing that each method performs as expected.

Students did quite well on this assignment. Students in the traditional section earned 75% on the assignment, which is the fourth highest of the seven assignments during the quarter. Students in the distance-learning section performed even better, earning 84% on the assignment, which was the second highest assignment and only slightly lower than the Java I review assignment given during the first week of the quarter. It was interesting for us to see that students performed well despite the fact that the testing portion of this assignment was more abstract and more vaguely described than the client programs mentioned above.

# 4. CONCLUSIONS AND FUTURE WORK

While the quantitative results for the testing assignments do not indicate uniform improvement, we believe the emphasis on testing has been qualitatively beneficial for students in the introductory Java courses. First, the test suite provides some kind of transition from writing self-contained applications to writing classes providing some service. Many students struggle with the process of abstraction that is involved in designing and implementing classes. When writing the test programs or documents first, students can momentarily ignore the daunting task of implementing the class and can concentrate on the concrete task of using a class. By focusing on the use of the class, students are better able to reason about the required behavior of the class and to then write code that tests that required behavior. These assignments also forced the students to put into practice good software engineering principles early in the curriculum. They clearly forced students to put more effort into testing. But as the significant increase in the number of questions regarding the assignment demonstrates, these assignments also forced students to realize that the high level problem description was insufficient to arrive at a solution. Students were forced to consider how someone is allowed to interact with their class and students began to appreciate the importance of clear requirements.

From our experience, there are two recommendations we have for instructors interested in trying this approach. First, make the testing portion of the assignment required. On a CSC 212 assignment from the fall 2004 quarter which is not described in this paper, one of the authors made the testing portion, a pair of files that was to be processed by the regular part of the assignment, extra credit. Only two students out of thirty participated in the testing portion of the assignment. Clearly, this did not produce any benefit for the class as a whole. Second, students in introductory courses need to be provided with the .class file for the instructor's solution when creating their testing document. One author did not do this for the Java II students, and the students complained that this forced them to complete the assignment early in order to have the class that was to be tested done. It was inconceivable for them that one could write a dummy version of the class in order to develop the testing program.

We feel that there are several areas for future work. It would be interesting to more tightly couple the testing on assignments and the midterm and final exams. This would provide a second way to determine if students are benefiting from assignments that take a testing first approach, beyond comparisons of assignment performance. It would also be helpful to analyze the student evaluations of courses that use the testing first approach and compare them with similar courses taught by the same instructor to see if student satisfaction was impacted by the approach.

# 5. REFERENCES


[1] M. Alford, P. Hsia, and F. Petry. A software engineering approach to introductory programming courses. In *Proceedings of the 7th SIGCSE Technical Symposium on Computer Education*, pages 157–161, 1977.

[2] J. Bergin, J. Caristi, Y. Dubinsky, O. Hazzan, and L. Williams. Teaching software development methods: The case of extreme programming. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 448–449, 2004.

[3] S. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 148–155, 2003.

[4] A. Gittleman. *Computing with Java: Programs, Objects, Graphics, Second Alternate Edition.* Scott/Jones Publishers, 2002.

[5] M. Goldwasser. A gimmick to integrate software testing throughout the curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 271–275, 2002.

[6] The Joint Task Force on Computing Curricula. *Computing Curricula 2001 Computer Science*, December 2001. Available at http://www.computer.org/education/cc2001/cc2001.pdf.

[7] G. M. Schneider. The introductory programming course in computer science – ten principles. In *Papers of the SIGCSE/CSA Technical Symposium on Computer Science Education*, pages 107–114, 1978.

[8] School of Computer Science, Telecommunications, and Information Systems, DePaul University. *http://www.cti.depaul.edu*, 2004.

[9] C. T. Wu. *An Introduction to Object-Oriented Programming with Java, 3rd Edition Update (Java 1.5 Update).* McGraw Hill, 2004.