Unit testing as a teaching tool in higher education

Canek Peláez[1a]

[1]Facultad de Ciencias, Universidad Nacional Autónoma de México, Av. Universidad Nº 3000, CU, 04510, México DF, México

Abstract. Unit testing in the programming world has had a profound impact in the way modern complex systems are developed. Many Open Source and Free Software projects encourage (and in some cases, mandate) the use of unit tests for new code submissions, and many software companies around the world have incorporated unit testing as part of their standard developing practices. And although not all software engineers use them, very few (if at all) object their use. However, there is almost no research available pertaining the use of unit tests as a teaching tool in introductory programming courses. I have been teaching introductory programming courses in the Computer Sciences program at the Sciences Faculty in the National Autonomous University of Mexico for almost ten years, and since 2013 I have been using unit testing as a teaching tool in those courses. The intent of this paper is to discuss the results of this experience.

1 Introduction

In Software Engineering, unit testing is used to individually test small units of code to determine if they do what they are supposed to do. Although unit testing can be used in almost any programming methodology, it is generally associated with Test Driven Development (TDD) and Extreme Programming (XP), specially since the publication of the Manifesto for Agile Software Development at the start of the century [13].

In TTD unit tests are actually written before the actual application code, and the latter is supposed to pass the former [4]. No new code can be introduced into the code base without first introducing its corresponding tests, and any change performed on previous code needs to pass existing tests. If changes to previous code result in new functionality, new unit tests for the new functionality should be provided first.

[a] Corresponding author: canek@ciencias.unam.mx

An important feature of modern unit testing, is the use of frameworks that execute the tests automatically, and allow to check the results for each unit. The most important family of frameworks are collectively called xUnit, which follows the design and structure from SUnit, written for SmallTalk by Kent Beck, one of the creators of Extreme Programming [3]; almost all major programming languages have an xUnit framework available, and in some cases, several.

While there are still objections to some of the recommended practices of Extreme Programming [5], the use of unit testing has had almost unanimous approval. Many software companies nowadays mandate the use of unit testing in their developer practices, and three of the bigger software companies in the world, Google, Apple, and Microsoft, all of them offer unit testing frameworks and resources for third party developers ([9], [2], [14]). Google in particular even organizes a conference devoted to testing: The Google Test Automation Conference (GTAC [10]).

In the world of Open Source it is easier to measure the use of unit testing in a project, since its source code is available for everyone to see. Just to give an example, the two largest desktop environment in Linux, GNOME and KDE, also have special frameworks for unit testing, and they encourage the contribution of unit tests by members of their communities [8;12].

In education, unit testing has become a topic of many Computer Science programs, usually in the Software Engineering syllabus, and almost invariable as a subject, not as a tool used by the professor or the teaching assistants. There has been some experiments that tried to integrate Automated Unit Testing practices in introductory programming courses, but again the emphasis was on teaching the students how to write unit tests, not as a tool for the teacher [6;7].

After getting my bachelor's degree in 2002, I worked for several corporations for a few years before starting my graduate studies. At the time, TDD and XP where just gaining attraction, and the generalized use of unit testing was in its infancy. I started to use unit testing first in a now defunct company that provided voice recognition services, and then I kept using them in my own personal projects and introducing them to other companies I worked with.

Since 2005 I imparted introductory programming courses as a lecturer, but I stopped after starting my PhD in 2008, and I did not returned to teaching at my university until 2013. It was at this time that I started using unit testing as a teaching tool.

In the last two years I have taught 7 introductory programming courses using unit testing, and this paper presents the results I have encountered while doing it, in the form of the performance of each group (grade average, number of students that passed the course, etc.), and interviews I conducted to interested students. By its very nature, this is not a comprehensive study: It covers only the courses I myself have imparted, and the answers given to me by the students that voluntarily accepted to be interviewed.

However, I think the use of unit testing as a teaching tool has had a very clear and beneficial impact in the way I teach introductory programming courses, and I believe it would be interesting to see other educators trying this approach, and to hear about their experiences.

In the following section I will describe the Computer Sciences program in my university and the two introductory courses I am basing my research on; next I will explain how do I use unit testing for grading students in those two courses; then I will show the concrete results of using unit testing in 7 courses for the last two years; and finally I will relay the opinions of the students that agreed to be interviewed for this paper. In my conclusions I will argue why I think the use of unit tests in introductory programming courses is a good idea, not only in higher education.

2 Computer Sciences in the Faculty of Sciences, UNAM

The Computer Sciences program in the Faculty of Sciences of the National Autonomous University of Mexico has two introductory programming courses: Computer Sciences Introduction in the first semester, and Data Structures in the second. The former, as its name indicates, is an introduction to Computer Sciences, but one of the primary objectives of the course is for students to learn how to program. We currently use the Java programming language, and there is an important emphasis in teaching Object Oriented fundamentals and principles. The second semester course covers data structures, algorithms, and computational complexity basics, again using the Java programming language, and there is a strong programming approach to the syllabus: Students are not only supposed to study and learn about and use the covered data structures and related algorithms, they must also write a working implementation of each one.

The rest of the four year program has several programming courses, and in all of them teachers can assume that the students know how to program using the Java language by then, although they can use different languages if they consider it necessary.

Both courses are given all semesters, and by several teachers; usually, students taking Computer Sciences Introduction on an even semester, or Data Structures on an odd one, is because they failed to pass the course the first time: We call a student that not taking a course for the first time a repeater. This affects the overall quality of a group of students; it is normal for a group consisting mostly of repeaters to have a lower average in its final grades, to have more students dropping out of the course, and to have more students with a failing grade. Also, groups of repeaters are usually smaller than normal groups.

We grade using grades NP (which means that the student dropped out of the course) and 5 to 10; a grade of 5 is a failing grade, and 6 to 10 represent passing grades from worst to best. The first semester students are randomly assigned to the available courses, but they can decide with which professor to take a course starting in the second semester.

3 Using unit tests to help grade students

My courses are graded by exams, quizzes, projects, and practices. Projects and practices cover the practical 50% of the student grade, and exams and quizzes the theoretic 50%. Practices (around 10 of them per semester) account for only 20% of the final grade the student gets, but it is the most visible and urgent part of the course: Students basically have a practice to complete all the weeks of the semester. The practices also help them to finish their projects, and those account for 30% of the final grade.

To help the students with their practices, there is a computer laboratory assigned to each course, which consists of two hours a week where the students have the laboratory exclusively for themselves, and a teaching assistant is available to review some topics and to answer questions.

The practices all follow the same format: The student downloads a compressed file with the practice files; all the necessary classes are usually included, but all or most of the methods and constructors are just stubs that the student must write in order for the practice to compile and work properly. Also included are unit tests (written by me, as are the practices) for all the methods the student needs to write, and in principle the grade of the student depends on the number of unit tests her practice passes: If her practice passes all the unit tests, then she will get top grades. Since we are using the Java programming language, we use Ant [1] to compile the practice, and JUnit [11] as framework for our unit tests.

There are some variations, of course. For example, in Computer Sciences Introduction one practice is specifically given so the students can learn how to use control structures (if, switch, for, while, do … while), by implementing the methods of a linked list structure. The

following practice is dedicated to recursion, so the students must take the code they wrote for the previous practice, and remove all instances of for, while, and do … while, while still passing all the unit tests (which do not change from the previous practice). In this case the teaching assistant responsible for grading the practice must personally review that the students did not use any iterating control structure.

Similarly, in Data Structures the teaching assistant must personally check that the algorithms implemented by the students in each data structure comply with the required time and space complexities, so just checking that the unit tests pass is not enough.

From my perspective as a teacher, using unit tests has greatly simplified my courses, and has allowed me to cover more topics than I was able to when I was not using them. When a student asks for help because she cannot pass a particular unit test, I am able to quickly determine where the problem is, usually by the unit test itself. And for the teaching assistant that grades the practices, his work is also greatly reduced: In some cases, all he has to do is to check that the students are not copying from each other.

The course must tightly follow the practices in order for the professor to be able to cover the theory while the students are trying to implement the practice. This gives the course a very rigid structure, and does not allow for much flexibility; the same thing happens with the practices themselves: Since the unit tests require very formal interfaces, the students cannot diverge from the given design of the practices. This is a real issue; but for the very first two introductory programming courses, I believe is not a serious one.

4 Concrete results of using unit testing as a teaching tool

From August 2013 to January 2014 (what we call the semester 2014-1) I taught Computer Sciences Introduction in a group with 36 students. Of those 5 dropped out, and of the remaining 31 the grade distribution was the following:

Table 1. Grades Computer Sciences Introduction 2014-1

Grade	Quantity
5	5
6	6
7	4
8	7
9	5
10	4

The grade average of the group was 7.42, and 26 students in total got a passing grade (83.87% of the students that did not dropped out).

In the same semester, I taught Data Structures in a group with 24 students. This group consisted mostly of repeaters, and 5 of them dropped out of the course.

The grade average was 6.36, and of those who did not dropped out, 11 (57.89%) got a passing grade. The following semester (2014-2, from February 2014 to July 2014) I taught again the same courses, but this time Computer Sciences Introduction consisted mostly of repeater students, while in Data Structures were mostly regular students. Also, many students that decided to take Data Structures with me had taken Computer Sciences

Introduction with me the previous semester, so they already had experience working with unit tests provided by the teacher.

Table 2. Grades Data Structures 2014-1

Grade	Quantity
5	8
6	3
7	3
8	3
9	2
10	0

In Computer Sciences Introduction the group consisted of 19 students, of which only one dropped out. For the remaining 18, the grade distribution was the following:

Table 3. Grades Computer Sciences Introduction 2014-2

Grade	Quantity
5	12
6	1
7	2
8	1
9	1
10	1

The grade average was 5.94, and only 33.33% of the students got a passing grade. For Data Structures the results were more positive: The group consisted of 46 students, and of those only one dropped out. Of the remaining 45, the grade distribution was the following:

Table 4. Grades Data Structures 2014-2

Grade	Quantity
5	8
6	4
7	6
8	8
9	11
10	8

The grade average was 7.75, and of those that did not dropped out, 37 (82.22%) got a passing grade. It is by far my most successful group since I started using unit testing as a

teaching tool, and it was at this point (summer of 2014) when I started to think about publishing these teaching experiences.

The next semester was 2015-1, from August 2014 to January 2015. From the introductory courses I only taught Data Structures; this was because the students I taught Computer Science Introduction in 2014-1 and Data Structures in 2014-2, asked me to teach the next programming course in the program (Modeling and Programming). I felt a strong connection with this group, and so I yielded and did not taught Computer Science Introduction in 2015-1. For Modeling and Programming I made my students write their own unit tests, and I will teach the course again next semester, so I am planning to eventually publish a follow up paper on that subject. The Data Structures group in 2015-1 had 33 students, most of them repeaters, and none of which dropped out. The grade distribution was the following:

Table 5. Grades Data Structures 2015-1

Grade	Quantity
5	13
6	3
7	10
8	5
9	1
10	1

The grade average was 6.42, and 20 (60.60%) of the 33 students got a passing grade. In this semester (2015-2) I taught Computer Sciences Introduction and Data Structures again. Computer Sciences Introduction consisted of 40 students, most of them repeaters, of which 2 dropped out. Of the remaining 38, the grade distribution was the following:

Table 6. Grades Computer Sciences Introduction 2015-2

Grade	Quantity
5	20
6	0
7	4
8	5
9	7
10	2

The grade average was 6.61, and of the students that did not dropped out, 18 (47.36%) got a passing degree. For Data Structures in 2015-2, the group consisted of 39 students, most of them regular (non-repeaters), of which 6 dropped out. For the remaining 33, the grade distribution was the following:

Table 7. Grades Data Structures 2015-2

Grade	Quantity
5	7
6	0
7	6
8	7
9	6
10	7

The grade average was 7.79, and of the students that did not dropped out, 26 (78.78%) got a passing grade. While not as successful as my Data Structures group in 2014-2, it should be noted that the students in this group had never used unit tests before.

As I mentioned above, the groups consisting mostly of repeating students performed worse and got more failing grades. This was also the case when I was not using unit testing as a teaching tool, and is also true in general in the Sciences Faculty in general and the Computer Sciences program in particular.

What is interesting to me is that before using unit testing, in groups with mostly regular students I usually got results similar to that of the groups with mostly repeaters when using unit testing. And when I got groups with mostly repeaters before I started using unit testing, the results where usually even worse. This is also the norm for other teachers for the same courses: As a student that did not took Computer Sciences Introduction with me, but it did with Data Structures (using unit tests), told me about his Computer Sciences Introduction group: “it was a brutal massacre” (only about 12 of 40 passed the course). Similar comments are not difficult to hear about the two first introductory programming courses.

I want to attribute the better performance to my use of unit testing in Computer Sciences Introduction and Data Structures; but it could just be that I had luck and got better students the last couple of years. I do not know; that is why I want to share this experience, in the hope that other educators would try a similar approach, and then we could exchange information to know if it is really a good idea.

5 Students opinions

After two years using unit testing as a teaching tool I am convinced that it is a good idea to do so. However, and although my students seemed to mostly agree with me, I had never actually asked them. So I conducted a series of interviews with several of my current and former students, to try and gain knowledge about their actual opinion. This interviews were all voluntary, and just a handful of students took the time to answer my questions. By no means are the students that participated in this a representative sample of neither my groups nor the student population as a whole.

The interviews where informal in nature, and although I asked some questions to all the students, I would not qualify them as a questionnaire. They were more in line to conversations where they were able to express their opinions about the introductory programming courses where I used unit testing.

All of the interviewed students agree that the use of unit testing in the courses they took was a good idea. They all graded their use in or above 8 (in a 1 to 10 scale: 1 being a really bad idea to use unit testing and 10 being a really good idea to use them), and some of them

even graded it at 10. Interestingly, some students that took Data Structures with me using unit testing, but did not took Computer Sciences Introduction with me (and therefore did not used unit testing) said that they thought it would be a bad idea to use unit testing in the first introductory programming course. They said it would probably confuse the beginner programmers. However, those students that used unit testing in Computer Sciences Introduction said it was a good idea.

Most interviewed students said that the biggest disadvantage of unit tests is that sometimes they do not cover all the possible cases. I take full responsibility on this, since I wrote the tests; but it is a self-resolving problem since every time a taught a course, the students reported to me when the tests where wrong or incomplete, and I then could fix them. And it is a known issue with unit testing: It is also code, so it can also have bugs.

Some students mentioned that a possible problem of using unit tests is that a student could get used to trying to pass the tests, instead of really understanding and solving the problem. This is a real issue, and I do not know how to handle it exactly, although I believe, as a former professional programmer, that if a programmer writes code that does what it is supposed to do, then that is a good job, no matter how the programmer got to write that code.

A couple of students said that sometimes reading the unit tests would give them the necessary information to write the corresponding code. This is also true, but I do not think this is a problem: Again, if the student is able to write correct code, I do not think we should care too much about how she got to do it.

In general, most of the interviewed students are enthusiastic about unit tests, and they want to learn how to write their own unit tests and how to use them in their personal projects.

6 Conclusions

As I said in the introduction, the results presented here are by definition not comprehensive: They consist of only 7 courses, all imparted by me, and the students that agreed to be interviewed are not a representative sample. However, I have found that the use of unit testing has simplified the practical part of my introductory programming courses, and that my students find them useful and helpful.

As a teacher, unit testing allows me to set a clear and formal set of rules of how practices should be written and graded, and they facilitate how they are graded and how to help students when they find an error in their coded and do not know how to correct. They also allow students to find themselves the problems in their code, and introduce them (from the very beginning) to the use of unit testing. Hopefully this will help them to write their own unit testing in the future.

The use of unit testing is not without its own problems. They stunt the student's imagination, and it is possible that they make things look easier than they really are. They are by definition inflexible and rigid, and do not help students to design by themselves the solution to a problem. However, I am convinced that their benefits outweigh their problems, and that most introductory programming courses should try to use them.

Even more, I think unit testing could be a useful teaching tool in middle and perhaps even in basic education. Most advanced countries are discussing the idea of teaching programming to kids and adolescents, and in those cases, the teacher could use unit testing to detect problems from students without them actually knowing about the unit tests. I have never taught a course in basic or middle education, but it is an experiment that it is perhaps worth trying.

References

1. Apache Ant, last accessed June 26, 2015, <http://ant.apache.org/>
2. Apple Testing Basics, last accessed June 26, 2015, <https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/testing_with_xcode/testing_2_testing_basics/testing_2_testing_basics.html>
3. Kent Beck, Extreme Programming Explained: Embrace Change, The XP Series, Addison Wesley, 2000.
4. Kent Beck, Test-Driven Development: By Example. Addison-Wesley, Boston, MA. 2003.
5. Gerardo Canfora, Aniello Cimitile, Felix Garcia, Mario Piattini Corrado, Aaron Visaggio, Evaluating Advantages of Test Driven Development: a Controlled Experiment with Professionals, In Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, pp. 364-371, ACM New York, NY, USA 2006.
6. Stephen H. Edwards, Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance, In Proc. of the Int'l Conference on Education and Information Systems: Technologies and Applications (EISTA'03), Orlando, Florida, USA, 2003.
7. Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, Paloma Díaz Pérez, An Experience in Integrating Automated Unit Testing Practices in an Introductory Programming Course, ACM SIGCSE Bulletin, Volume 34 Issue 4, pp. 125-128, December 2002, ACM New York, NY, USA.
8. GNOME Developer: Unit Testing, last accessed June 26, 2015, <https://developer.gnome.org/programming-guidelines/stable/unit-testing.html.en>
9. Googletest, last accessed June 26, 2015, <https://code.google.com/p/googletest/>
10. Google Test Automation Conference, last accessed June 26, 2015, <https://developers.google.com/google-test-automation-conference/>
11. JUnit, last accessed June 26, 2015, <http://junit.org/>
12. KDE TechBase, Unittests, last accessed June 26, 2015, <https://techbase.kde.org/Development/Tutorials/Unittests>
13. Manifesto for Agile Software Development, last accessed June 26, 2015, <http://agilemanifesto.org/>
14. Microsoft Developer Network: Unit Test Basics, last accessed June 26, 2015, <https://msdn.microsoft.com/en-us/library/hh694602.aspx>