An Experience in Integrating Automated Unit Testing Practices in an Introductory Programming Course

Elena García Barriocanal
Computer Science Department. Polytechnic School
University of Alcalá. Ctra. Barcelona km. 33.600, 28871. Madrid, Spain
<elena.garciab@uah.es>

Miguel-Ángel Sicilia Urbán
Ignacio Aedo Cuevas
Paloma Díaz Pérez
Department of Computer Science, DEI Laboratory.
Carlos III University. Avda. Universidad, 30, 28911. Madrid, Spain
<msicilia@inf.uc3m.es, aedo@ia.uc3m.es, pdp@inf.uc3m.es>

Abstract
Unit testing is one of the core practices in the Extreme Programming lightweight software development method, and it is usually carried out with the help of software frameworks that ease the construction of test cases as an integral part of programming tasks. This work describes our first results in studying the integration of automated unit testing practices in conventional ‘introduction to programming’ laboratories. Since the work used a classical procedural language in the course’s assignments, we had to design a specific testing framework called tpUnit. The results of the experiment points out that a straightforward approach for the integration of unit testing in first-semester courses do not result in the expected outcomes in terms of student’s engagement in the practice.

Reviewed Papers

1. Introduction

Extreme programming [2] (XP) is a lightweight software development method organized around a set of practices and principles that have recently gained the attention of an important fraction of the software development community, possibly due to the promise of more flexibility for changes in customer requirements, improved code quality, and shorter development times. *Unit Testing* is one of its core practices, and XP fosters a disciplined approach to ‘white-box’ regression testing in the context of a *Continuous Integration* practice, in which coding assignments are broken up into small tasks, whose results are integrated daily in a collective code base.

We summarize the *Unit Testing* practice in the following principles (as described in [6]):

- Build the test as the code is developed (“test a little, code a little”), and not before of after.
- Run all the tests frequently, each time a change in code is released, and making sure that they run perfectly all the time.
- Consider test cases as code components that are deliverables, just as the program code itself.

Automated unit testing (AUT) practices are becoming widespread in the software development industry as a result of the influence of the XP development approach, since they’re required to effectively carry out daily regression testing.

In this work, we describe the early results of a teaching experiment in integrating AUT as part of a traditional “Introduction to Programming” course, intertwining test case construction with programming in laboratories and assignments, and raising test cases as a first-class deliverable. Although many programming language-specific freely available unit testing frameworks have appeared under the generic name of `xUnit` as in <http://xprogramming.org>, none of them apply to the classic procedural languages that we might use in many introductory programming courses. To overcome this lack, we have designed a unit testing framework for Borland’s old *Turbo Pascal* integrated development environment, which allows for testing subprograms (procedures and functions) rather than methods defined in classes (a testing framework called `dUnit` exists for the Object-Pascal language included in Borland’s Delphi tool, but our focus is not in object-oriented programming teaching in this article). Therefore, we can easily apply the approach taken here to courses using other procedural languages that we often use as introductory such as C or Modula-2.

This integration has the obvious benefit of reinforcing the learning of a subject included in curricula recommendations (for example, in [5], test case design is included in imperative-first introductory courses), but a deeper understanding of the potential benefits of testing frameworks in the classroom requires additional empirical evidence.

XP has received considerable attention from computer science educators [10] in search of benefits in the learning process that can result from XP practices. Related work include experiments that used AUT in conjunction with other practices in classroom settings, [1] or in supplementary courses [11], but our work focuses exclusively in introducing unit testing, so that its impact in learning can be examined in isolation from that of other XP practices. In addition, we can find in [9] a previous report on some potential benefits of unit testing in Java introductory courses.

2. Essential Automated Unit Testing Concepts

A unit test exercises "a unit" of production code in isolation from the full system and checks that the results are the expected ones. Its purpose is to identify bugs in the code under test before integrating the code into the rest of the system. The size of "a unit" to be tested depends on the size of the set of coherent functionality, and in practice varies from a class to a package when we use object-oriented programming languages like Java, or between a single procedure or function and a unit, when procedural languages like Turbo Pascal are used.

Software engineering describes several reasons for unit testing. Perhaps, the most relevant one is that the sooner the units are tested, the quicker and cheaper is to correct possible bugs and to demonstrate that the solutions works. However, testing units in isolation can be difficult. `xUnit` frameworks are the test harnesses that enable the automatic running of test cases, recording of results and reporting of errors for classes or code modules.

When using automatic unit testing frameworks, developers must write tests that check non trivial pieces of code that don't provide user-interaction. We can hierarchically structure these test cases forming 'trees of tests' (called *suites* in `xUnit` jargon) to facilitate the testing of related parts of the program. In addition, we must provide some kind of text or GUI-based tool to automatically run all or part of the tests of a program, reporting failures in a way that programmers can easily locate them in the code.

3. Integrating AUT in the Classroom

To find evidence about the usefulness of integrating AUT practices in the first course of computer science studies, we have planned and began to carry out a number of *design experiments* in the classroom. We followed the guidelines in [4]. To put it in context, we have to describe our mid-term research objectives in unit testing and education, which we can synthesize in the following three questions:

1. Do unit-testing practices in CS1 assignments and labs really improve *code quality*?
2. Do CS1 students *enjoy* writing test cases?
3. Do unit-testing practices in CS1 enhance the student's learning process?

The first and second hypothesis are the direct translation of two of the major claims of the XP *Unit Testing* practice, while the third is intended to assess if the integration is worthwhile from a pedagogical perspective. We arranged the methodology for our study in two phases. In the first phase, we will carry out experiments in both first and second semester programming classes. In the second phase, we will collect and analyze the results in search for correlations between the use of AUT practices and student scores (trying to answer to the third question). This paper reports the first experiment in the first phase, specifically intended to answer hypothesis two. The paper also describes some preliminary results and lesson learned about hypothesis one.

The participants in our experiment were one hundred students of a Computer Science introductory course at the University of Alcalá. The course contains a three-hours lecture and two laboratory hours each week, aimed at working the theoretical concepts and the practical matters, respectively. Around 75% of the students attended classes during a half-day session about how they can automate unit testing using testing frameworks. Documentation and examples about AUT practices were available through department web servers, and students were able to practice in labs all the key concepts. We previously explained to all students the essential theoretical concepts about unit test cases, just after they have reached basic procedural programming skills (typically, at the middle of the semester).

XP claims that programmers are supposed to enjoy writing testing units for their own code [3], considering unit tests as an integral part of their work. The procedure used for gathering evidence about this fact consisted in giving the student the freedom to decide on using our testing framework (in his/her assignments and exercises) or not. All of them knew that the use of the automatic testing framework was optional and they would not obtain higher score by merely using it. At the end of the semester, a simple Likert-scale questionnaire asked about their level of satisfaction with the technique. The sentences, which could be answered with a typical 1..5 scale (from 'strongly disagree' to 'strongly agree' as usual), were the following.:

- I had prior knowledge about XP and its practices.
- Writing testing cases was easy for me.
- Writing test cases helped me in making better code.

We included question one to be sure that the negative student perceptions about XP reported in senior students [8] was not present in our experiment.

As part of the course, students are required to develop a final assignment and they must pass an exam for the lab part. On the one hand, we used that final assignment to evaluate if we see an improvement in code quality when students use AUT. On the other hand, we can use exam scores obtained by students who used the framework in their assignment, compared to those who didn't, for measuring the improvement in knowledge about programming fundamental concepts when unit testing frameworks were used in practice.

Since the procedural language Pascal is used as the programming tool in the laboratory (more specifically, we use Borland's old integrated development environment, called Turbo Pascal), we have developed and documented a unit testing framework for this environment that is described in the following section.

4. The tpUnit Framework Design

We have developed a Turbo Pascal separately-compilable unit to isolate all the unit testing-related functions, so that students can reference them by including `tpUnit` in the appropriate reference clause. The most important definitions in the unit are the following:

```
type testCase = procedure;
procedure assert(exprResult: boolean;
        name: string; description: string);
procedure registerTest(test: testCase);
procedure runAllTests;
```

Some object-oriented unit testing frameworks like `jUnit` for the Java language rely on advanced language capabilities (i.e. reflection) to identify test methods by their name at run-time. Since old languages lack those capabilities, the programmer is responsible for registering test cases by invoking the register procedure `registerTest`, which uses a procedural-type parameter. Programmers must register all the test cases. Once tests are registered, they can run them by writing a main program. The `runAllTest` procedure executes all the previously registered tests, and reports failures in a similar manner as common `xUnit` testing frameworks do.

As we explained in previous sections, we can group test cases in suites. However, programming exercises in computer science introductory courses are so simple that usually do not require building suites that group them. However, suite trees can be created using `createTestSuite` and `createSubSuite` procedures, and test cases can be registered into a particular suite using `registerTestInSuite`. The test cases of a suite tree can run independently of other suites using `runAllTestsInASuite` procedure. A 'default' suite is available when a student calls the `registerTest` procedure.

Before registering and running test cases, they students should code test cases using procedures without parameters and without user interaction. Each test case verifies the correct working of a specific execution scenario for a subprogram. The `assert` procedure is used to check test case post-conditions. This procedure takes as parameters a boolean expression that should yield a 'true' value if the correct results are produced, the name of the subprogram that must generate the result (the tested subprogram) and the description that will be showed if the boolean expression is not satisfied (indicating that an unexpected error occurred, and debugging is required).

5. Results

Only about 10% of the students that were subjects of the experiment developed test cases in their assignments, so that the data gathered by the questionnaire is far from statistical significance. Despite this fact, the results of the satisfaction questionnaire were encouraging for further research: 70% of the students that answered the questionnaire found easy to use (4-5 points in the Likert scale) the Turbo Pascal testing framework, and all of they thought this practice improved their code quality (again, 4-5 points in the Likert scale). In addition, none of them had previous knowledge about `xUnit` or XP. They also carried out two simultaneous experiments to be able to compare the results. The first was in a very similar environment at Carlos III University; in this case, they used Java in laboratories in an 'object-first' approach for a first semester course using. They used the second through an internet asynchronous virtual environment in a first course in object-oriented programming for first course students. In both cases, results were even less significant in terms of student's engagement in unit testing practices, and thus served only as reinforcements of the results of the main experiment.

Overall, experiment results point out that hypothesis two does not apply to a traditional programming lab setting. (We note that empirical testing in professional development contexts did not validate this hypothesis.) In addition, perhaps further contextual elements are required in the classroom (beyond instructor advice and theoretical understanding of the benefits of the technique) to make students be motivated to write unit tests as they code. These results are in accordance to the recommended knowledge prerequisites in the SEI 'Unit Analysis and Testing module' [7] that indicates "the student of unit analysis and testing should, of course, have a solid background in programming". We think that the few students that used the AUT framework in their assignments had higher programming skills than the rest of the students in the sample, and this fact could justify the positive questionnaire results about their opinion about writing test cases. Not surprisingly, students that used the framework scored high in their assignment's evaluation. However, due to the low participation rates, we cannot draw any definitive conclusion about this fact.

In forthcoming evaluations, we expect to compare the results with second-semester students that should have acquired solid programming skills. We need a large number of automatically tested assignments to find evidence about the third hypothesis. That is, we need more information about the improvement in code quality when they use AUT. We are planning to introduce an additional motivational ingredient, like for example, an extra score in the course.

References

[1] Astrachan, O., Duvall, R. C., Wallingford, E. Bringing Extreme Programming to the Classroom. *Proceedings of XP Universe Conference'01, 2001*.

[2] Beck, K. *Extreme Programming Explained: Embrace Change*, The XP Series, Addison Wesley, 2000.

[3] Beck, K., Gamma, E. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[4] Brown, A. L. Design Experiments: Theoretical and Methodological Challenges in Creating Complex Interventions in Classroom Settings. *Journal of the Learning Sciences* 2(2), 141-178.

[5] IEEE/ACM Joint Task Force on Computing Curricula, Computer Science Curricula 2001 final report, 2001.

[6] Jeffries, R., Anderson, A. and Hendrickson, C. *Extreme Programming Installed*, Addison Wesley, 2001.

[7] Morell, L.J., Deimel, L.E. Unit Analysis and Testing, *SEI*

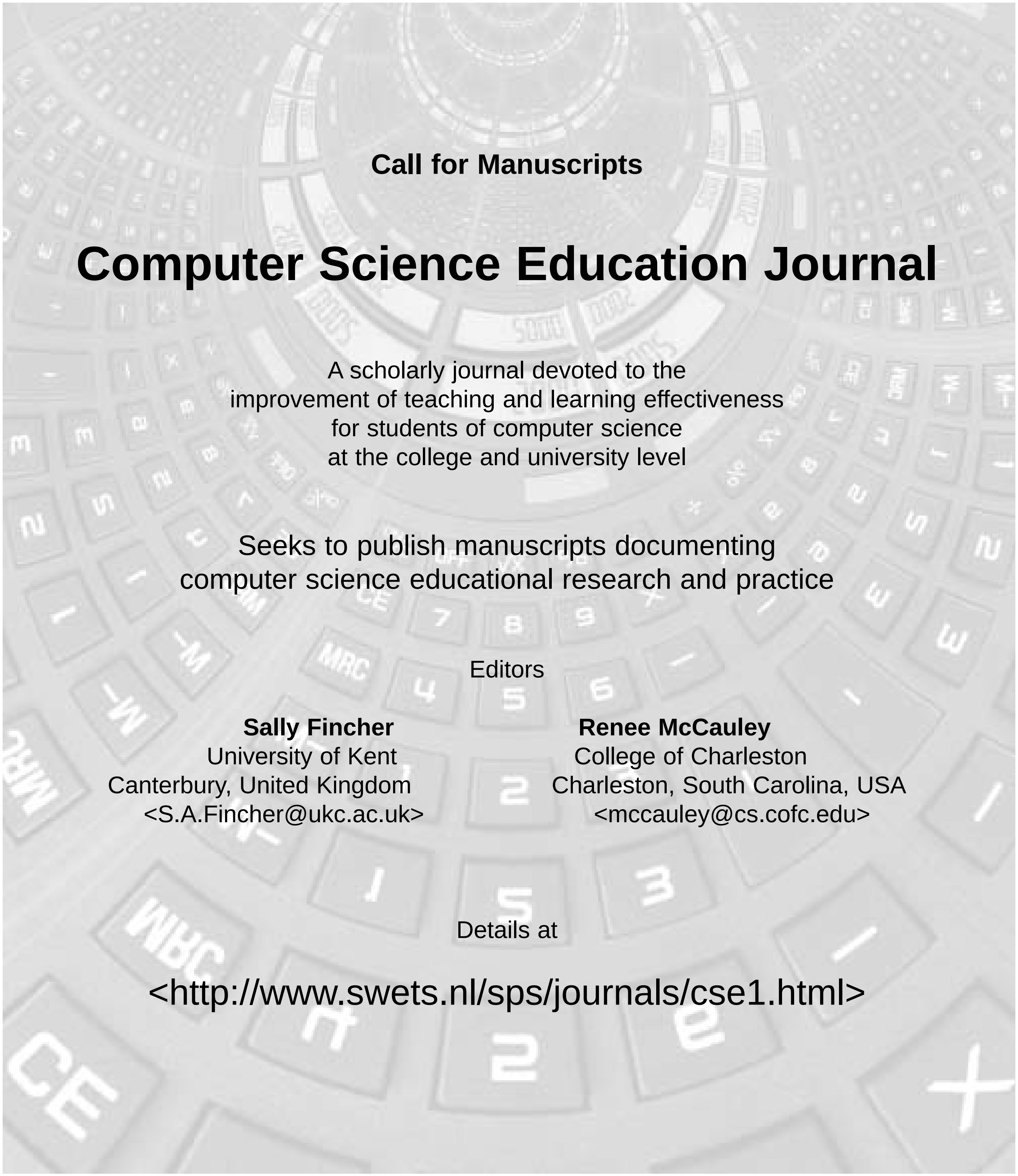
Curriculum Module, SEI-CM-9-2.0, 1992.

[8] Sanders, D. Student Perceptions of the Suitability of Extreme and Pair Programming. *Proceedings of XP Universe Conference'01*, 2001.

[9] Steinberg, D. H. The effect of Unit Tests on Entry Points, Coupling, and Cohesion in an Introductory Java Programming Course, *Proceedings of XP Universe Conference'01*, 2001.

[10] Williams, L. & Upchurch, R. Extreme Programming For Software Engineering Education?. *Proceedings of Frontiers in Education 2001*, 2001.

[11] Wege, C., Gerhardt, F. Learning XP: Host a Boot Camp. *Extreme Programming in Practice*. The XP series. Addison-Wesley, 2001.



Call for Manuscripts

Computer Science Education Journal

A scholarly journal devoted to the
improvement of teaching and learning effectiveness
for students of computer science
at the college and university level

Seeks to publish manuscripts documenting
computer science educational research and practice

Editors

Sally Fincher	**Renee McCauley**
University of Kent	College of Charleston
Canterbury, United Kingdom	Charleston, South Carolina, USA
<S.A.Fincher@ukc.ac.uk>	<mccauley@cs.cofc.edu>

Details at

<http://www.swets.nl/sps/journals/cse1.html>