AC 2012-4602: IMPROVING THE STATE OF UNDERGRADUATE SOFTWARE TESTING EDUCATION

Prof. W. Eric Wong, University of Texas, Dallas

W. Eric Wong received his Ph.D. in computer science from Purdue University. He is currently a professor and Director of International Outreach in the Department of Computer Science at the University of Texas, Dallas. Prior to joining UTD, he was with Telcordia (formerly Bellcore) as a Project Manager for Dependable Telecom Software Development. Wong received the Quality Assurance Special Achievement Award from Johnson Space Center, NASA, in 1997. His research focus is on the technology to help practitioners develop high quality software at low cost. In particular, he is doing research in software testing, debugging, safety, and reliability at the application and architectural design levels. Wong is the Vice President for Technical Operations of the IEEE Reliability Society and the Secretary of the ACM Special Interest Group on Applied Computing (SIGAPP).

Improving the State of Undergraduate Software Testing Education

Software has become fundamental to our everyday life. Regardless of age, gender, occupation, nationality, etc., each of us depends on software in some way, either directly or indirectly. Yet software is far from defect-free and very large sums of money are spent each year only to fix and maintain defective software. According to a study by NIST in 2002[3], software bugs cost the U.S. economy an estimated $59.5 billion annually (about 0.6% of GDP). The same study also found that more than one third of these costs could be eliminated by an improved testing infrastructure. Furthermore, these estimates have not taken into account any potential deaths or catastrophic financial loss associated with the failure of mission-critical software. These figures would be much higher if the study were conducted today.

A Deficiency Needs to be Corrected

Software testing continues to be the primary approach used to ensure the development of high quality software. It is estimated that more than 60% of the cost of software development is spent on testing and debugging. However, a large part of the problem is not as much the amount of testing that is performed, as much as it is "who" the software is tested by, and "how" these *testers* do it. Most of the personnel responsible for software testing are software engineers with a very basic background in testing, mostly restricted to the application of a small set of testing tools. A simple knowledge of a few testing tools cannot hope to substitute for a strong foundation in software testing principles and methodologies. The fact is that a significant number of the people responsible for testing the software that we rely on are not adequately prepared for the task. If we were to trace this *deficiency* in software testing background back to its source, we would end up at the educational institutions that are responsible for teaching and training people to test software. Thus, if today's software testers are not sufficiently armed with the knowledge required to test software well, then it is most likely because they have not been adequately trained. ***This is one of the main root causes of the current state of software testing, and it is here that we need to begin to remedy the problem.***

Current Approach

The subject of software testing rarely appears in the undergraduate curricula, despite its well established place in classical computer science (CS) literature[2] and its extensive use in industry. Many academic CS programs only briefly cover software testing, limiting the topic to software engineering (SE) courses[1] that may not be mandatory for a CS degree.

According to a presentation at the Panel, *Teaching Software Testing: Experiences, Lessons Learned and the Path Forward*, from CSEE&T 2011[6], the number of undergraduate testing courses offered in the USA is around 30, but the number of undergraduate CS programs in the USA that require software testing is zero. One may argue that this statistic is based on a study of the published undergraduate CS curricula, the results of which may not be entirely accurate. Nevertheless, even if that is the case, it still provides a clear picture that very few (if any) CS undergraduates are properly trained in software testing before graduation.

Another argument is that undergraduate SE programs, following the SWEBOK[5], or the undergraduate SE curriculum recommended by the ACM and the IEEE Computer Society[4], generally do teach software testing. However, we must recognize two important facts: (1) most universities and colleges only offer undergraduate degrees in CS, not SE, and (2) for the majority of software engineers, if they have a Bachelor's degree, it is most likely in CS rather than in SE.

Besides, although some aspects of software testing may be covered, the actual application of testing practices is not explored in-depth during the undergraduate education. Therefore, offering a single elective SE-related course or covering the topic to some extent without providing opportunities for students to actually make use of the knowledge in different settings is not a good solution to the issue of software testing; these techniques require repeated practice before they become second nature.

Our Approach

Software testing is an extremely broad subject, and even a dedicated one-semester course cannot adequately cover all the important concepts and techniques with an appropriate level of detail, let alone a course with a more general learning objective. Instead of only briefly covering software testing (if at all) in one course, we need to teach this important topic from beginner programming classes (e.g., CS 1336 − Programming Fundamentals, CS 1337 − Computer Science I, and CS 2336 − Computer Science II at the University of Texas at Dallas), followed by intermediate courses (e.g., CS 3376 − C/C++ Programming in a UNIX Environment, and CS 4336 − Advanced Java Programming), to a dedicated elective (e.g., CS and SE 4367 − Software Testing, Validation and Verification) for more advanced techniques, and the final senior project (CS 4485 – the CS version of the capstone project course and SE 4485 – the corresponding SE version) which provides students with an in-depth, hands-on experience in all aspects of software engineering including how to effectively and efficiently test the software systems they produce. By the end of the semester students should have a working knowledge of each individual aspect of software engineering, and also have gained experience in how these aspects are related to, and depend on, one another in order to successfully develop a software system. Through this process, we can help students make software testing an integral part of their coding practice with the understanding that testing cannot just be added on to the software at the last minute after it is produced.

Currently, we are working on a TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Type II project funded by NSF to develop a set of instructional materials in the form of course modules, not confined to a particular technique or tool but generalized over different aspects of software testing.

We use a pedagogical model for teaching software testing at the undergraduate level with three important concepts: *many-to-many*, *minimally intrusive* and *non-restrictive*. Our model emphasizes a ***many-to-many*** relationship between courses and modules such that educational materials can be selectively applied to any appropriate courses in a ***minimally intrusive*** and ***non-restrictive*** way. A module can be used repeatedly in many courses but not necessarily in the same breadth or depth as there is no need to cover all its topics within each course that employs it, and a course can draw materials from multiple modules. Instructors have the flexibility to either follow the suggested teaching outline or use their own discretion to determine which of the topics are suitable, fine-tuning the course materials to make them more accessible and understandable to their students. This also increases the effectiveness of the modules and achieving the desired learning outcomes.

Seven Course Modules

The following is a description of seven course modules that are to serve as the instructional materials for teaching software testing in multiple CS and SE undergraduate courses. Also explained is the rationale behind the choice and design of each module, and the course(s) it might apply to.

Module 1 – Software Testing Fundamentals: *The must-knows of software testing*

This module covers concepts that are essential to establishing a firm foundation in software testing. Students are exposed to the idea of ***functional testing*** and how it can be applied at the unit or the system level, thereby also introducing them to the concepts of ***unit testing***, ***integration testing***, and ***system testing***. Although this module is intended for courses such as CS 1336, 1337 and 2336, the techniques should be revisited and expanded upon whenever appropriate.

Module 2 – Test Case Selection/Generation: *Where do test cases come from?*

This module covers materials on how effective test cases can be generated and why one test case might be better than another. Students at the lower level courses (such as CS 1336 and 1337) are introduced to the two most popular black-box requirements-based test generation techniques: *equivalence class partitioning* and *boundary value analysis*. Students at the intermediate and upper level courses (such as CS 2336, CS 3376, CS 4336, and CS/SE 4367) are introduced to more advanced test generation techniques such as coverage-based adequate test set generation, where the adequacy of a test set is measured against a criterion of interest. For example, the criterion can be a black-box approach based on the functional requirements such that every requirement has to be tested. It can also be input domain coverage-based testing, or a white-box approach such as controlflow-based code coverage (e.g., statement and decision coverage) and dataflow-based code coverage (e.g., c-use, p-use, and all-uses coverage). Tools are also introduced, wherever appropriate, to the students.

Other topics to be covered in CS/SE 4367 (at the instructor's discretion) include the following: ***mutation testing*** – a fault injection-based technique that introduces simple syntactical changes into the program, ***adaptive random testing*** – a technique to improve random testing by having test cases as evenly spread over the entire input domain as possible, test generation from finite-state models and formal specifications. Students are also to be exposed to state of the art techniques in automatic test generation and some of the advantages and disadvantages of each.

Module 3 – Regression Testing & Test Minimization/Prioritization: *Minimizing the expenses*

Regression testing also known as program revalidation, is a testing process intended to check that small changes made to one part of a program did not result in unexpected consequences in another seemingly unrelated part of the program. This module discusses techniques for selecting tests for regression testing. In particular, it focuses on test set minimization and test case prioritization in order to ***maximize coverage*** and ***minimize test redundancy.*** One way to do the minimization is to find a minimal subset of tests which gives the same coverage with respect to a pre-selected criterion (e.g., the same statement coverage or the same decision coverage) as the entire test set. For prioritization, test cases are ranked based on a suitable metric (e.g., based on the statement coverage of each test).

This module also discusses the potential weakness of using minimization and prioritization for selecting regression tests. Additional test selection techniques for regression testing are also covered. Module 3 is most suitable for inclusion in the advanced programming course (e.g., CS 4336) and the undergraduate testing course (e.g., CS/SE 4367).

Module 4 – Quality Software Testing Documentation: *Leave yourself more than a note*

This module covers software testing documentation standards and the importance of creating quality documents. Students are taught about the documents such as test plans, test requirements, test case specifications, transmittal reports, logs, etc.

The amount of documentation required depends on the course and this decision is left to the instructor's discretion. In lower level courses (e.g., CS 1336 and 1337), students are required to submit a basic test plan for some programming assignments including details such as a list of functionalities that need to be tested, and how equivalence class partitioning and boundary value analysis are used to help them generate test cases. Students in intermediate courses (e.g., CS 2336) will submit not just basic testing documentation but also test logs to ensure that each test case was properly executed and the result was logged. In the software testing course (e.g., CS/SE 4367), students learn about standards such as IEEE 829-1998 for Software Test Documentation and quality documentation practices such as version control, etc.

Module 5 – Advanced Software Testing: *A deeper understanding of software testing*

This module goes over advanced software testing techniques that are beyond the scope of the materials covered in the lower and intermediate level courses. Suggested topics include, but are not limited to, non-functional software testing such as performance testing (scalability, response time, etc.), usability testing, security testing; Web-based; interface and GUI-based testing.

It should be emphasized however that this module has many advanced materials which may not be suitable for all the students. We promote the idea of "***fit-for-purpose***" usage of the module by only selecting appropriate topics at the instructor's discretion for students in a specialized software testing course (e.g., CS/SE 4367), a senior software engineering project course (e.g., CS 4485 and SE 4485), or those who are taking independent study with a topic on software testing. The ultimate goal is to promote further research in software testing and to encourage students to pursue related studies in graduate school.

Module 6 – Efficient and Effective Testing Tools: *There is no need to do it all manually*

In addition to teaching students about the fundamentals of software testing, we also want to make sure the students are exposed to useful software testing tools that are used both in academia and industry. Once the strong background in software testing has been created, the use of tools is also important to reduce the manual labor involved. We will provide a set of appropriate tools for each testing technique and a description of the limitations as well as advantages and disadvantages of each tool. However, the choice of which tool(s) to be used is left up to the discretion of the instructor. This module can be used in courses such as CS 4336, CS/SE 4367, CS 4485, SE 4485, and others as appropriate.

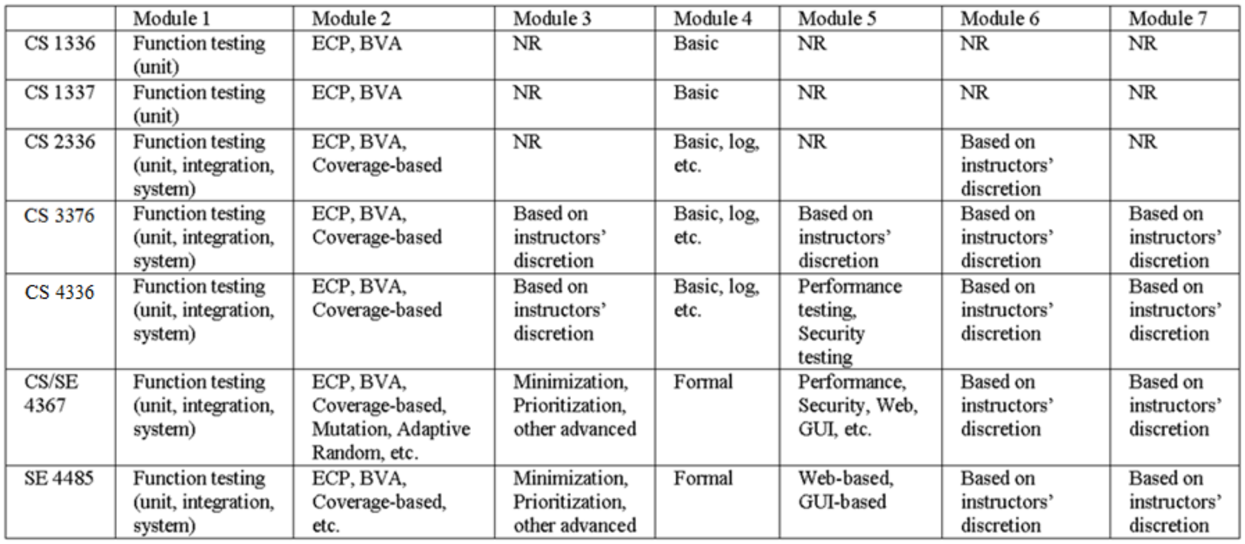
Module 7 – Integrated Solutions for Testing, Debugging and Profiling: *The wealth of dynamic information in a test case*

The dynamic information collected during test case execution can be used for several important purposes. Students learn about how runtime trace information collected in the form such as statement coverage reports can help programmers quickly find where the bugs are using state of the art ***fault localization*** techniques. The concept of ***code profiling*** is also explained in detail and students learn how to use tools to investigate program behavior in terms of performance analysis in order to understand which portions of code can be optimized.

Concepts from this module can be taught in upper level course such as CS 4336, CS/SE 4367, CS 4485 and SE 4485.

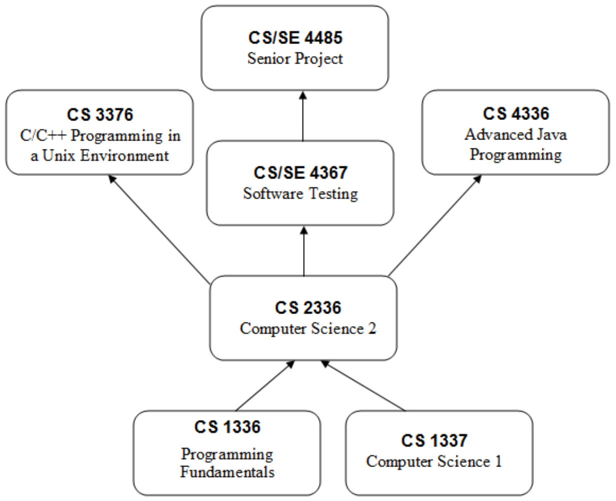
Relationship between Courses and Modules

Having described each module, we now use Figure 1 to illustrate how different courses can apply the same module and how a module can be used by different courses. "NR" implies "not required," and "based on instructors' discretion" implies that each instructor can determine, using their own judgments, which suggested topics from the modules are appropriate for their students. Also included in Figure 1 is the pre-requisite relationship between different courses that are discussed in this paper.



	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
CS 1336	Function testing (unit)	ECP, BVA	NR	Basic	NR	NR	NR
CS 1337	Function testing (unit)	ECP, BVA	NR	Basic	NR	NR	NR
CS 2336	Function testing (unit, integration, system)	ECP, BVA, Coverage-based	NR	Basic, log, etc.	NR	Based on instructors' discretion	NR
CS 3376	Function testing (unit, integration, system)	ECP, BVA, Coverage-based	Based on instructors' discretion	Basic, log, etc.	Based on instructors' discretion	Based on instructors' discretion	Based on instructors' discretion
CS 4336	Function testing (unit, integration, system)	ECP, BVA, Coverage-based	Based on instructors' discretion	Basic, log, etc.	Performance testing, Security testing	Based on instructors' discretion	Based on instructors' discretion
CS/SE 4367	Function testing (unit, integration, system)	ECP, BVA, Coverage-based, Mutation, Adaptive Random, etc.	Minimization, Prioritization, other advanced	Formal	Performance, Security, Web, GUI, etc.	Based on instructors' discretion	Based on instructors' discretion
SE 4485	Function testing (unit, integration, system)	ECP, BVA, Coverage-based, etc.	Minimization, Prioritization, other advanced	Formal	Web-based, GUI-based	Based on instructors' discretion	Based on instructors' discretion

(a) Many-to-many relationship between courses and modules



(b) Course pre-requisite relationship

Figure 1. The relationship between courses and modules, and the pre-requisite hierarchy between courses

Implementation and Findings from the First Project Year

We have adopted the approach described above using the course modules developed by our TUES project. During the first year, five faculty members and two graduate students in the CS Department at UTD participated. A sequence of four courses was selected: CS 1336, CS 1337, CS 2336, and CS 3376. Each course is a prerequisite of its successor with CS 1336 specially designed for students with no prior computer programming experience, and as such cannot be used to satisfy degree requirements for majors in CS or SE. For the first two courses, materials from a module on *black-box requirements-based testing* were used, while materials from an additional module on *white-box code coverage-based testing* were also used for the latter two.

A special lecture on these testing materials (ranging from 30 to 50 minutes) was given to students in each of these classes which did not cause any significant disruption of the course. The same concepts were also repeatedly explained by the instructors, whenever appropriate, throughout the entire semester. Overall, following the ***minimally intrusive*** concept discussed earlier in Our Approach, instructors did not find it to be overly difficult to include the testing materials in their courses, nor did it adversely affect their ability to adequately teach their own materials. PowerPoint slides of our testing lectures are available at the project website http://paris.utdallas.edu/CCLI. Not only students in the selected courses, but also others who did not attend the lectures in software testing can take advantage of this resource and continuously use it as a reference when testing their software.

The two graduate students as special teaching assistants (in addition to the regular TAs for those courses) provided extra recitation sessions and tutoring to students who needed additional help in understanding and applying the materials discussed in their classes.

Starting from Fall 2011, one additional faculty member at UTD and one at Collin County Community College have also adopted our approach and the course modules. This has an impact on students who take CS 3376 at UTD and COSC 2336: Programming Fundamental III – C++ at

Collin College. The collaboration between UTD and Collin is vital as the latter is a two-year community college in the Dallas metropolitan area serving about 53,000 credit and continuing education students each year and the first among the Texas community colleges to allow students to apply to a university pre-admission program, in which credit could be earned both at Collin and a major university at the same time. Many of its students, after one or two years of study, transfer to UTD for their Bachelor's degree, it is therefore extremely important that these students receive the same background during their freshman and sophomore years as our own students.

Students in these selected courses (each of which has multiple sections except for CS 3376 and COSC 2336) were required to turn in a test plan explaining how their programs were tested for at least one programming assignment. Questions on software testing were also included in the midterm and/or final exams to evaluate students' learning of the principles and techniques for software testing, discussed in the testing modules presented to them.

More than 800 undergraduates have benefited since the commencement of our project. From the test plans students submitted for their programming assignments, it is very clear that the majority of the students understood the equivalence class partitioning (ECP) and boundary value analysis (BVA), and were able to select appropriate inputs using these two techniques. This observation is also supported by the exam scores for the question on software testing.

An anonymous evaluation (using the questionnaire in Figure 2) was conducted at the end of each semester. Depending on the materials covered in each class, some questions were removed from the survey. Figure 3 and Figure 4 present the results based on the feedback from students in CS 1337 and CS 2336, respectively. Similar data was also obtained for other classes but not included due to the space limit.

Questions	Rating 4 – Strongly agree 3 – Agree 2 – Disagree 1 – Strongly disagree
1. You will test your programming assignments before you submit them	
2. It is important to conduct a good testing on your programs before they are submitted for grading	
3. The testing techniques discussed in class are appropriate for students in your class	
4. The testing techniques discussed in class are easy to use	
5. The testing module presented in class helps you better understand the *Equivalence Class Partitioning* technique	
6. The testing module presented in class helps you better understand the *Boundary Value Analysis* technique	
7. The testing module presented in class helps you better understand the *Statement and Decision Coverage* techniques	
8. The *Equivalence Class Partitioning* technique can help you better select test inputs from different parts of the input domain	
9. The *Boundary Value Analysis* technique can help you select input values to detect bugs at or near the boundaries of different equivalence classes	
10. The *Statement and Decision Coverage* techniques can help you select input values to cover statements and decisions that have not been covered, and increase the probability of detecting hidden bugs	

11. You are able to perform and demonstrate the *Equivalence Class Partitioning* technique	
12. You are able to perform and demonstrate the *Boundary Value Analysis* technique	
13. You are able to perform and demonstrate the *Statement Coverage* technique	
14. You are able to perform and demonstrate the *Decision Coverage* technique	
15. You will apply the testing techniques which you have learned to other programming assignments, whenever appropriate	
16. Software testing should become an integral part of a student's coding practice	

Figure 2: Questionnaire for students' survey

On a scale of 1 (strongly disagree) to 4 (strongly agree) with 3 as (agree), the average scores for understanding ECP and BVA are around 3.4 (with the majority either strongly agreeing or agreeing that our testing module presented in class helps them better understand software testing techniques), and for being able to apply these techniques to test generation are around 3.2. Although the difference is small, it seems that there are some students who could understand the principles of the techniques, but were not able to effectively use them. As heavily emphasized in our previous discussion, becoming skilled in any of these testing techniques requires repeated practice. This outcome supports our claim.

	4	3	2	1	
1	71	14	0	0	3.835
2	74	11	0	0	**3.871**
3	43	40	2	0	3.482
4	32	46	7	0	3.294
5	35	42	7	1	3.306
6	34	43	8	0	3.306
8	32	45	8	0	3.282
9	43	37	5	0	3.447
11	20	46	16	3	2.976
12	28	43	14	0	3.165
15	48	33	4	0	**3.518**
16	57	27	1	0	**3.659**

"It is important to conduct a good testing of your programs before they are submitted for grading."

"You will apply the testing techniques which you have learned wherever appropriate."

"Software testing should become an integral part of student's coding practice."

Figure 3. Quantitative feedback from students in CS 1337.
Questions 7, 10, 13 and 14 are not suitable students in CS 1337.

	4	3	2	1	
1	39	7	3	0	3.73
2	41	8	0	0	**3.91**
3	25	22	2	0	3.47
4	15	28	6	0	3.18
5	20	26	3	0	3.35
6	23	22	4	0	3.39
7	22	22	4	1	3.33
8	21	25	3	0	3.37
9	23	22	4	0	3.39
10	26	21	2	0	3.49
11	22	22	5	0	3.35
12	21	25	3	0	3.37
13	14	29	6	0	3.16
14	15	27	6	1	3.14
15	28	15	5	1	**3.43**
16	35	12	2	0	**3.67**

Students at this level are more mature, and can better demonstrate the testing techniques.

Figure 4. Quantitative feedback from students in CS 2336

Regarding the white-box code coverage-based testing techniques, since it is more advanced than ECP and BVA, it follows our prediction that students had more trouble understanding and applying statement and decision coverage. This is also supported by the results of our evaluation conducted at the end of the semester.

We also noticed that students in CS 3376 were more mature that those in CS1336, 1337 and 2336 at UTD and COSC 2336 at Collin, and could better demonstrate the testing techniques discussed in class.

Other important findings for all the sections include

- "It is important to conduct a good testing of your programs before they are submitted for grading" (Question 2) has a score of at least 3.87.
- "You will apply the testing techniques which you have learned wherever appropriate" (Question 15) has a score of 3.43 or higher.
- "Software testing should become an integral part of student's coding practice" (Question 16) has a score of 3.66 or higher.

Below is some qualitative feedback based on the written comments submitted by the students.

- From CS 1337 class: "I found that by the time I got into CS 1337, I was already practicing some of the techniques without knowing the technical reasons for them."

  This suggests that students seem to intuitively apply testing techniques in an ad hoc approach, but not in a systematic way with a complete understanding.

- From CS 2336: "I think that there should be a stronger emphasis on testing earlier in the learning process (e.g., CS 1336 & CS 1337). I also believe that testing should play a bigger part in programming courses in general."

  This shows that students realize the importance of software testing and agree that when they begin writing code for their programming assignments, testing should be an integral part of their practice.

- From CS 3376: "I feel that more time should have been given to this. Creating software is pointless if you don't know how to test it. The materials are very applicable to real world situations." "This knowledge helps me enforce my testing practices and make sure they are complete."

  This indicates that students at this level are already evaluating the real world applicability and usefulness of these techniques.

From the above data, it has been shown unanimously across all the sections of every class participating in our project that students understand the importance of software testing and intend to use the techniques to help them improve the quality of their programming assignments.

Additional Ongoing Assessments

To supplement our assessments measuring how effectively our approach has been implemented and how well our course modules have been put to use (such as the qualitative feedback from the students and the quantitative data collected from the anonymous survey, presented above), we also focus on evaluating whether our goals and objectives have been met. These metrics may include the number of students that keep up good testing practices in later years, or the number of errors made by students in programming assignments given in future courses.

In order to understand the impact of this project and rule out other alternative explanations, we will utilize direct observation methods and holistic rubrics to assess students learning outcomes. We will also compare the quality of software testing skills and knowledge of our participant study cohort group in Senior Design Project with non-cohort group's testing performance utilizing various statistical methods and models, e.g., *Repeated-Measures ANOVA, Structural Equation Modeling*. These comparisons will allow us to examine the effects of this project, and also provide evidence of its impact. Furthermore, we will track the progress of our then-alumni cohorts to evaluate the overall outcomes of the project. In addition, we will seek feedback from industry advisors on how our alumni cohort group performs software testing.

Conclusion

We have observed that many undergraduates do poorly on their programming assignments when they fail to adequately test their code. They run the programs on a few randomly selected data sets; things that are easy to type; numbers for which the results are easily calculated. They do not use a logical, common sense approach for testing their programs. This is most likely because students are not taught the strategies that we as instructors consider simple logic until they take a software testing related course. Such a course (e.g., CS/SE 4367 at UTD), if offered at all, is usually an upper-level course students take after they complete the basic programming sequence (such as CS 1336, 1337 and 2336).

In order to fix this problem, we emphasize that software testing principles and techniques should be covered at appropriate stages of the undergraduate CS and SE education. This should be done in multiple years and different courses from the freshman introductory programming class (instead of postponing until sophomore, junior, or even senior years) for the most fundamental testing techniques such as *Boundary Value Analysis* and *Equivalent Class Partitioning* to the capstone project class, which gives students an opportunity to apply their knowledge in software testing to a semester-long group-based project sponsored by our industry partners. Only by doing so will students adopt software testing as an integral part of their coding practice to effectively produce more reliable software.

The evaluation will be conducted continuously to monitor activities that involve project implementation for further refinements and continuous improvement. We will use quantitative and qualitative data, and direct indicators (e.g., number of instructors using our modules and number of students learning software testing before graduation). The overall impact of our approach will be evaluated after our student cohorts finish their college education and enter the workforce. This will be done through a longitudinal study by monitoring and tracking our then-alumni cohorts who attended classes covering software testing as undergraduates. We are confident that even a partial success will cascade into software development and manifest itself in the form of lower software defect rates and software maintenance costs.

Acknowledgment

This work is supported by the National Science Foundation's *Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics* (TUES) program (formerly *Course, Curriculum, and Laboratory Improvement* (CCLI) program) under Award No. DUE-1023071.

Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation.

References:

1. Janzen, D. and Saiedian, H., “Test-driven Development: Concepts, Taxonomy, and Future Direction,” *IEEE Computer*, 38(9): 43–50, September 2005.
2. Myers, G. J., Sandler, C. (revised by), Badgett, T. (revised by), and Thomas, T. M. (revised by), *The Art of Software Testing*, 2nd Edition, John Wiley & Sons, June 2004
3. National Institute of Standards and Technology, “*The Economic Impacts of Inadequate Infrastructure for Software Testing*,” NIST Planning Report 02-3, May 2002
4. Leblanc, R., Sobel, A., Diaz-Herrera, J. L., and Hilburn, T. B., “Software Engineering 2004 − Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering,” The Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery, August 2004
5. SWEBOK: *Guide to the Software Engineering Body of Knowledge* (http://www.computer.org/portal/web/swebok)
6. Wong, W. E., Bertolino, A., Debroy, V., Offutt, J., and Vouk, M., “Teaching Software Testing: Experiences, Lessons Learned and the Path Forward,” in *Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training* (CSEE&T 2011), Honolulu, Hawaii, May 2011