A SIMPLIFIED APPROACH TO TEST-DRIVEN DEVELOPMENT FOR THE FIRST PROGRAMMING COURSE

Christopher G. Jones, CPA/PhD, Utah Valley State College, jonescg@uvsc.edu

ABSTRACT

In industry experiments using test-driven development (TDD), some researchers report significantly increased code quality over traditional test-last approaches. Not surprisingly, information technology educators have begun to call for the introduction of TDD into the curriculum. Some early adopters have pioneered the use of TDD in advanced programming or software engineering courses. A few educators have experimented with the introduction of TDD as early as the beginning programming course. This paper examines various strategies for presenting test-driven concepts in an introductory software development course. Classroom experience with a simplified approach to TDD that doesn't require the use of an automated test framework is explored. Based on an end-of-course student survey, recommendations are made for integrating TDD into the first programming course.

Keywords: Test-Driven Development, TDD, Agile Methodologies, Unit Testing, Automated Test Framework, Test-First Design

INTRODUCTION

Proponents of Test-Driven Development (TDD) assert that commercial software defect rates can be reduced from 18% to 50 % when tests are written at the beginning rather than the end of the development cycle [10, 16]. TDD is one of the 12 key practices of Extreme Programming, a popular agile software development methodology [4]. In TDD, synonymously known as TFD (Test-First Design), TFP (Test-First Programming), and TDD (Test-Driven Design), software developers "test first, then code" [4, p. 9]. By focusing up front on the verification and validation of software requirements, the design organically evolves as new code is written to satisfy the failed tests [18].

With the promise of significantly increased code quality, information systems and technology educators have begun to call for the introduction of test-driven development into the curriculum [6, 8, 15, 18, 20]. The attempts to integrate TDD into the classroom vary [12, 13]. At the upper division level, Mugridge [17] experimented with TDD in his undergraduate software engineering course. Steinberg [21] recounted his experience with an Extreme Programming (XP) study group composed of computer science faculty and upper division students. Erdogmus et al. [8] conducted a controlled experiment of junior-level (third-year) students in an intensive Java course.

At the lower division level, Olan [18] recommends that IS educators "plant the seeds of unit testing" as early as the introductory programming course (CS1). To date, few educators have reported taking this step. Those that have, have had mixed results. Barriocanal et al. [3], for example, provided a custom testing framework, training sessions, and web-based documentation for his beginning programming course. Use of unit testing was optional for course assignments. Barriocanal et al. summarized their experience with TDD as follows: "The results of the experiment points out that a straightforward approach for the integration of unit testing in first-semester courses do[es] not result in the expected outcomes in term of students' engagement in the practice" [3, p. 125].

Marrero et al. [15] required students in an introductory Java course to submit a test suite ahead of the actual code submission. Student performance was low. "These results seem to indicate that the testing requirement hurt student performance rather than helping" [15, p. 6]. Elkner [7], on the other hand, reported that student enthusiasm for TDD was high. To avoid the overhead of an XUnit-style testing framework, Elkner [7], instead, provided students with an assertion macro [Syntax: `assertEquals (actual,expected,testmessage)`). "All the students were actively engaged in programming for the entire ninety minutes of class," wrote Elkner. "Several pairs kept working after the bell rang, working into their break" [7].

Although TDD purists would argue test-first design and automated testing frameworks are inseparable [2], the use of such frameworks can present serious pedagogical barriers for students. This paper examines various strategies for presenting test-driven concepts in an introductory software development

course, without the use of an automated testing framework. An experience report is provided on the helpfulness of several of these alternatives based on an end-of-course student survey in an introductory C# programming course.

FOUNDATIONAL CONCEPTS FOR TEACHING TEST DRIVEN DEVELOPMENT

Model curriculum guidelines for baccalaureate degree programs in both information systems and information technology acknowledge the importance of providing instruction in testing when discussing application programming and system development. IS 2002 [11] lists application development and system testing as part of the critical skill set expected of graduating seniors. At least three of the model course specifications address testing. The course specification for *IS 2000.5 Programming, Data, File and Object Structures* recommends coverage of program correctness, testing, and verification and validation methods [11, p. 21]. *IS 2002.8 – Physical Design and Implementation with DBMS* includes "unit testing, integration, and integration testing of the final system" [11, p. 30] and *IS 2002.9 – Physical Design and Implementation in Emerging Environments* asserts that "verification, testing and validation should be integral components of software quality assurance" [11, p. 31].

ABET draft curriculum guidelines for undergraduate baccalaureate degrees programs in Information Technology [19] address testing in two areas: (a) Fundamental Programming Constructs (PF2) and (b) Testing and QA (SIA5). A minimum of nine class hours is recommended for PF2, with some of that time devoted to the core learning outcome: "Design, implement, test, and debug a program" [19, p. 89]. Under System Integration and Architecture 5 (SIA5), a minimum of three class hours on Testing and QA are recommended. Core learning outcomes for System Integration and Architecture (SIA5) include coverage of "current testing standards", "techniques used in testing a system or product", and identifying "appropriate acceptance criteria" [19, p. 99].

Traditionally the concept of Software Testing is introduced in the beginning programming course. Testing is defined, the goals and roles of testing are discussed, and the linkage between testing and debugging explored. Students are often provided with rudimentary approaches to testing such as manually tracing the sequence of code execution (desk checking), program reviews by peers (walk-throughs), test cases, test data, and test suites [5]. Some introductory courses expose students to advanced software engineering testing practices by defining such terms as white-box and black-box testing, equivalence categories, and statement coverage [14].

While the basics testing principles alluded to above provide an overall perspective on the rationale for software testing, additional foundational knowledge is required in order for students to use a test-first approach to software design. At a minimum, students must be introduced to the notion of test design as a way to specify software requirements. This requires a change in paradigm. Rather than a traditional application development cycle of **Design-Code-Test**, the cycle is inverted. Development begins with testing as a means of expressing design, followed by just enough coding to satisfy the tests.

The application is then refactored to improve the design of the existing code [9]. Fowler defines refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [9, p. 53]. Under TDD, the application development cycle becomes **Test-Code-Refactor**. To operationalize refactoring, professional developers often employ refactoring patterns and a unit-testing framework to facilitate regression testing.

Adding automated unit testing and refactoring patterns to the already overcrowded list of learning units for a beginning programming course is a challenge. Another approach (one requiring more discipline on the part of the student) is manual testing with visual inspection of results. For traditional approaches to introductory programming, manual testing has been the norm, albeit after-the-fact, rather than before coding. What differs in a manual TDD approach is that formulating the tests, predicting the expected results, and writing the tests comes first rather than last.

In summary, the integration of TDD into the first programming course requires treatment of additional concepts and practices beyond those traditionally covered. Regardless of whether an automated or manual approach is used for testing, students must learn to: (a) design tests as a way to specify requirements, (b) partition the problem space into testable units, (c) write unit tests using assertions, (d) develop test data, (e) set up and tear down test data, and (f) refactor code without breaking the test suite. Using an automated approach to TDD with a unit testing framework, requires students to develop the following additional skills: (a) learn how to use the framework, (b) integrate the framework into the

development environment, (c) understand the limitations of the framework in testing user interfaces, (d) create mock objects when testing against external APIs, and (e) learn how to use automated refactoring tools. For a manual approach, students forgo the automated unit test framework, relying instead on visual inspection of test results and manual code refactoring.

EXPERIENCE USING A SIMPLIFIED APPROACH TO TDD

One of the biggest challenges facing beginning programming students is "getting their code to work". Students often complain about "not knowing where to begin" when starting an assignment. Rather than a systematic approach to problem solving, students often adopt a trial-and-error approach to coding. This translates into making random changes to code, all with the unfounded hope that the program will now compile error-free. Success becomes elusive; student retention an issue. The introductory programming course morphs into a gatekeeper, denying entry for many into the remaining courses in an information systems curriculum.

Low completion rates in introductory programming, specifically, and declining enrollments in the USA in computing disciplines, in general, is a troublesome combination. At Utah Valley State College (UVSC), the Information Systems & Technology Department (IS&T) is exploring a series of initiatives to increase student recruiting and retention, from academic program redesign to removing barriers to entry. As part of that effort, we began exploring the integration of test-driven development into the introductory programming course.

UVSC offers both a Bachelor of Science in Information Systems and a Bachelor of Science in Information Technology. Both programs have been designed in conformance with the latest curriculum guidelines of the Accreditation Board for Engineering and Technology (ABET) (1). Early coursework for both programs is shared. In the freshman year, students begin the curriculum with a course in Information Systems and Information Technology fundamentals. This is followed by the first programming course – Computer Programming for IS/IT I. C# is used as the primary pedagogical language of instruction throughout the curriculum, with exposure to other languages in the upper division courses. Early on, students are exposed to program development using a text editor and a command line compiler but soon migrate to Visual Studio 2005 for application development.

For beginning programming students in an information systems curriculum, unfamiliar with application development environments in general, even a simple text editor and a command line compiler can be daunting. Adding a unit testing framework to the mix, either standalone, as an IDE plugin, or built into the IDE, was considered too overwhelming for the first course. Instead a simplified manual approach to TDD was used. Fall 2005, students were introduced to test-first design as a way to approach application development. As early as assignment 1, students were given testing aids. As the course progressed, the level of testing aids was increased. By the end of the course, students were expected to be able to write test cases and visually inspect test results.

In the beginning of the course, students were provided with sample screen shots of the desired output (Figure 1). Classroom examples and in-class exercises focused on writing tests first before coding solutions. By the fifth exercise, students were constructing programmer-defined classes and testing them using test descriptions provided by the instructor. In the sixth exercise (conditional expressions), students were provided a test class with appropriate tests cases. By the ninth exercise (iterative control structures), a test framework was provided that included test descriptions but not the actual test code (Figure 2). Students were required to write the actual tests. As the complexity of the exercises increased, a sample executable was provided so that students could "test drive" the application before recreating the solution. In total, 15 exercises were assigned.

At the end course students were surveyed regarding their experience with the simplified test-driven development approach. Students were asked to rate the helpfulness of six different test aids using a five point Likert scale. All of the students completing the course (n = 17) participated in the survey. As Table 1 indicates, students found five of the six the test aids to be "very often" to "always" helpful. "Sample screen shots" received the highest rating at M = 4.71 on a five point scale. "Complete test case source code", "test data and expected results", "test case description", and "simple test framework" weren't far behind, clustering around the midpoint between "very often" and "always helpful". Only one test aid—"executables for test driving a sample solution"—was considered "sometimes helpful". None of the test aids was considered "never" or "rarely helpful."

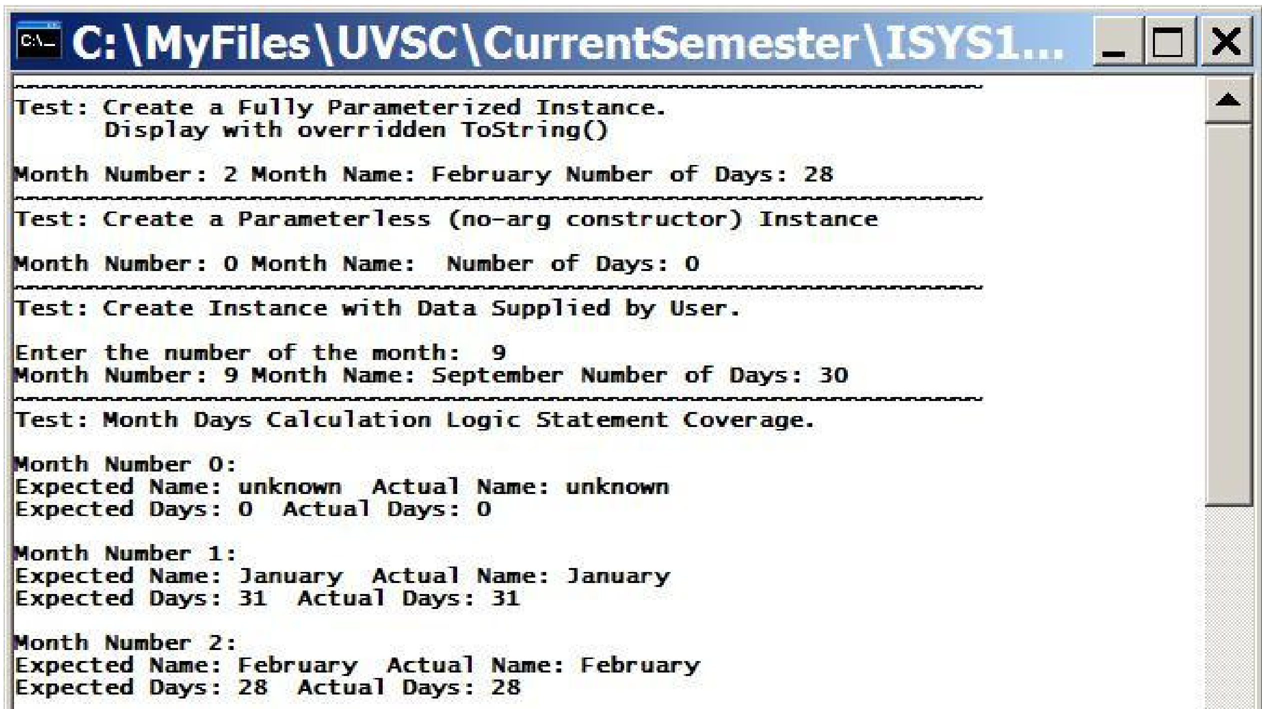

Figure 10. Sample Screen Shot of Test Cases

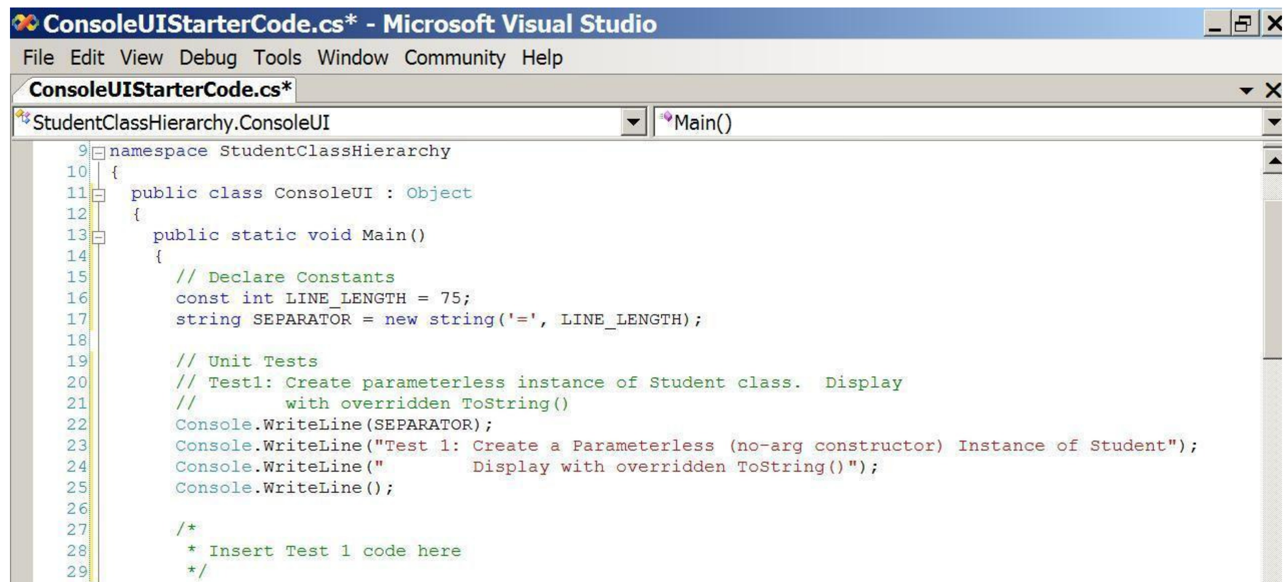

Figure 11. Sample Test Framework with Test Code Placeholders

Table 1. Helpfulness of Test Aid in Producing Stable Working Code

Testing Aid	*M*[a]	*SD*	*n*
Sample screen shots	4.71	0.47	17
Complete test case source code	4.56	0.73	16
Test data and expected test results	4.53	0.52	17
Test case description	4.47	0.87	17
Simple test framework (i.e., Test description with //ToDos)	4.47	0.62	17
Executable (.exe) for test driving sample solution	3.76	0.75	17

[a]Mean scores represent arithmetic averages based on the following Likert scale:
1 = never helpful, 2 = rarely helpful, 3 = sometimes helpful, 4 = very often helpful, 5 = always helpful.

In an open-ended survey question, students were asked to describe their successes and frustrations with the "test first, then code" approach. For the most part students expressed satisfaction with the approach and found it helpful. Students commented that test-first forced them to attack the problem step-by-step rather than all at once.

> "I have a nasty habit of trying to write a whole solution from the picture in my mind, rather than taking things step-by-step. When I started writing little snippets to satisfy a test, then move on, things went much faster, AND [*upper case in the original*] I believe it was easier for me to understand the concept I was learning."

> "One step at a time, making sure each part worked first, worked wonders. Once I figured this out it all became easier and less frustrating."

For students with previous programming experience, test-first required unlearning a design-code-test approach. Some struggled.

> "Test-first can be really frustrating to someone who's been coding for a while, because with more traditional methods, you can write code that suits your purposes rather than code the suits the test's purposes."

> "I'm somewhat of an experienced programmer. I often find the 'test-first' stuff annoying. However, for learning a new language, I found it to be extremely useful for anybody new to programming. I would suggest always following a TFD model."

Early in the course, students were asked to develop test code from the test case descriptions. This proved difficult for most. When, later, the source code for the test cases was supplied by the instructor, the testing process went smoother but not perfectly.

> "Having the example code was probably the most useful aid to me. It helped me to focus on 1 test at a time. Even my first few attempts at using the sample code weren't very successful as I tried to get ALL the code to work in my first compile attempt. I went through 2 or 3 assignments with test code before I realized that I needed to comment out the other tests until I got the first one done." [*capitalization and emphasis in original*]

Eventually students were weaned from the complete test source code to a test template. The template included test descriptions, source code to write the test header to the console, and placeholders for the test code the students were to write.

> "I liked not having all the test source code, but place holders helped to keep me from forgetting important test info. I liked when you gave the code with first programs then left just the placeholders later." [*emphasis in original*]

CONCLUSION

The test-first approach to software development has gained considerable attention in the academic community as a possible "silver bullet" for improving the code quality of information technology and computing science students. Experience reports provide anecdotal narrative on the benefits and challenges of integrating TDD into the curriculum,

whether in upper division software development courses or in the introductory programming course.

For beginning programmers, the overhead of an automated testing framework can be overwhelming. A simplified TDD approach using manual testing with visual inspection is an alternative. In this approach, the instructor begins the course by demonstrating test-first, providing the test cases, and supplying the test source code. As students progress, some of the pedagogical scaffolding is removed, with students provided a test template and asked to write the test code themselves. By the end of the course, students are expected to write the complete test code and generate test cases on their own. With the principles of test-driven development firmly in place, students are ready to move on to automated unit testing and refactoring in the intermediate programming course.

Although the simplified TDD approach appears to lead to increased use of TDD by students in the first introductory programming course, what remains to be seen is whether the testing aids result in improved code quality or are as effective as automated unit testing. This is a subject for further research.

REFERENCES

ftp://cseftp.uvsc.edu/IST/jonescg/pubs/SimplifiedTDD.pdf