Evolving an Elective Software Testing Course:

Lessons Learned

Ed Jones

Florida A&M University

ABSTRACT

At Florida A&M University we offer an elective course in software testing once every other year. This presentation shares the objectives and methods used in this course, along with motivations for improving the course born from failed objectives and feedback from students and employers. This course is slated to become a regularly offered course in the software engineering track of the undergraduate curriculum.

The course is 80% practice, 20% theory. Laboratory assignments give students practical experience performing the detailed and mundane activities that are an essential, though unattractive, part of being a software tester. The first two-thirds of the course deal with fine-grained testing (functions). Students design test cases, write test drivers, and test scripts. Students complete test logs documenting test results, and evaluate discrepancies found in the test results. Decision tables are the specification formalism of choice. We use a systematic process for deriving black-box, boundary, and white-box test cases from the decision table. We teach students the theory of control-flow coverage analysis, and require them to use a coverage analysis tool. We use white-box coverage measures as the primary indicators of test thoroughness; we also use error-seeding to evaluate student solutions to testing problems. In the last one-third of the class, students adapt the techniques for small-grained testing to larger-grained software such as objects and applications. We complete the course with lectures on application testing using ad hoc techniques and systematic techniques such as operational profiles.

This presentation states the course ideal, along with the underlying principles about the testing experiences students need. An introspective summary of the course outcomes is given, along with areas for improvement for the next offering of the course. This paper contains compilation of ideas from previously published work.

The course website is at www.cis.famu.edu/~cis4932joe.

TABLE OF CONTENTS

1 Course Overview

COURSE DESCRIPTION: The purpose of this course is to build the skills necessary to perform software testing at the function, class and application level. Students will be taught concepts of black-box (functional and boundary) and white-box (coverage-based) testing, and will apply these concepts to small programs and components (functions and classes). Students will also be taught evaluative techniques such as coverage and mutation testing (error seeding). This course introduces the software engineering discipline of software quality engineering and to the legal and societal issues of software quality.

TEXTBOOKS:

- Hetzel, The *Complete Guide to Software Tes*ting, John Wiley & Sons.
- *Teach Yourself Unix in 24 Hours, 3rd Edition*, SAMS Publishing.

Though not used for lectures or homework, the textbook was recommended as a valuable professional resource due to the management perspective it presents.

1.1 Learning Objectives

The audience for the course is not software testers but software developers. What distinguishes the course approach is that is stresses the programming aspect of software testing. A goal is to enhance and expand students' programming skills to support activities across the testing lifecycle: C++ programming and Unix shell script programming to automate aspects of software testing.

Conceptual Objectives: Upon completion of this course, the student should understand

- the software test life cycle
- the relationship between testing, software quality and other verification techniques
- the theoretical limits of software testing
- concepts and techniques for black-box and white-box testing
- the SPRAE (specification-premeditation-repeatability-accountability-economy) framework for testing practice
- design patterns for test automation
- the challenges of object-oriented testing
- test coverage measures such as statement, branch, and path coverage
- management procedures for software testing.

Performance Objectives: Upon completion of this course, the student should be able to perform the following tasks

- Use an advanced Unix text editor (`emacs` or `vi`) to create and edit a C++ program
- Compile and execute C++ programs in a environment
- Write simple shell scripts with arguments and inputs
- Design functional and boundary test cases
- Devise and document a test strategy (plan)
- Develop manual test scripts
- Conduct testing according to a test script and document results
- Write test drivers to automate testing a function, object (class) or program (application)

- Write test scripts to automate the set-up of a test environment and execute test scripts
- Evaluate the results from a test session and write problem reports (test incidence reports).

1.2 Student Assignments

LABORATORY ASSIGNMENTS: There are 10-15 out-of-class assignments requiring the use of the Unix system for C++ and Unix shell script programming. One or more projects will be team projects. All work must be submitted to the class repository on the CIS Department's Unix network. Procedures for submitting assignments are posted on the course website.

HOMEWORK ASSIGNMENT SCHEDULE

hw#	DueDate	Assignment Title
1	8/28	Unix Lab 0 Profile
2	9/03	Test Lab 1A Blind Testing - Part I
3	9/09	Test Lab 1B Blind Testing - Part II
4	9/12	Unix Lab 1 Unix command overview
5	9/16	Test Lab 2 Test Documentation: scripts, logs
6	9/29	Test Lab 3 Software specification
7	10/10	Test Lab 4 Test Design, Implementation, Execution AUTOMATION
8	10/16	Unix Lab 2 Unix utilities
9	11/03	Test Lab 5 Unit Test Driver Construction (oracle driver)
10	11/10	Test Lab 6 White-Box Unit Testing
11	12/03	Test Lab 7 Class Testing via Test Driver
***	12/11	AMNESTY -- 2 assignments; may include a make-up test3 with repeat limit of 2. This score will replace the lower of test1 and test2.

I am not satisfied with the set of assignments given nor with student performance. It appears we need a dedicated laboratory with recitation sessions to ensure students complete assignments and achieve the educational objectives. In hindsight, clear statement of educational objectives for each assignment would benefit the student and improve outcomes.

In making the assignments I wanted to cascade experiences, where each assignment builds on skills used in previous assignments. The students did not always make the connection, possibly due to gaps in the lecture notes or reading assignments.

A major challenge is grading the homework. I made significant investment in automating the grading of assignments, for future reference. This investment left inadequate time for supervising students while completing the assignments and for giving directed, specific feedback to close the loop on the learning objectives.

EXAMS: There will be four (4) exams, each counting 10% of the overall grade, and a final exam counting 20% of the overall grade. One or more exams will be administered as an on-line test.

Two exams and the final were administered on-line. Performance on these exams was low, which is typical of short-answer tests where partial credit is not awarded. The online tests were randomly generated from a bank of short-answer questions. The advantage of on-line tests include (1) the potential for saving in-class time; (2) the ability to give students practice tests, encouraging the mastery of the essential material; and (3) reducing teacher workload; and (4) encouraging the evolution of a large test bank representing the body of knowledge. In reality, on-line test development requires more upfront work – which was not recovered for this incarnation of the course.

1.3 History

This is the fourth time this course has been taught. Previous terms were Summer 1999, Spring 2000 and Fall 2001. The basic course syllabus has remained fairly constant. Course notes and assignments have evolved. Though intended for students with a strong interest in software testing, the course is subscribed to by students who simply need an elective course. This reality has contributed to the direction in which the course has evolved.

2 Student Background

Eighteen students enrolled in the course. Students were in the course principally because they needed an elective course, not because they had a particular interest in software testing. Most were one or two semesters from graduation.

Elements of the ideal student background include:

- An interest in software testing
- Strong programming skills
- Scientific method
- Sophomore or junior classification
- Desire to seek internship in software testing

An emphasis in the course was the discipline associated with software testing.

3 Driving Principles

My philosophy on teaching software testing in the undergraduate curriculum has been published [1,2] and recommend that testing be integrated throughout the curriculum rather than concentrated in a one course.

3.1 A software testing lifecycle model

Table 1. Reference Software Testing Life Cycle

Phase	Activities	Products
Analysis	Define scope and strategy for testing. Refine specification.	*Test plan, refined specification.*
Design	Derive test cases. Organize test effort.	*Test cases, test data sets.*
Implementation	Develop machinery needed to conduct tests.	*Test procedures/scripts, drivers.*
Execution	Run test as controlled experiment. Capture test results.	*Test results.*
Evaluation	Verify results against expectations. Take follow-up action such as reporting, debugging.	*Test log, modified software.*

3.2 Experience objectives

The primary conceptual objective is that students understand that, like software development, software testing follows a process, and that they understand basic limitations of testing (exhaustive testing, uncertainty), standard techniques (the black-box /white-box dichotomy, equivalence-based methods, model based testing (e.g., control or data flow coverage). The experience objective of the course is that the student be able to perform testing tasks that span phases in the test lifecycle. Table 2 presents an experience/competency model discussed at length in [1]. Certain competencies, namely Test Specialist, require integration of skills from other competency areas. The purpose of the course is to give the student experience performing at least one task in each competency area.

Table 2 also suggests an architecture for a *competency-based* tester training program (which is beyond the scope of this course). The goal of such a competency-based tester training program is to give provide a resource to students who wish to purse testing more seriously. Each competency level is decomposed into a progression of achievement levels as discussed in [1]. Achievement of each level requires the mastery of concepts and the application of specific skills, which can be achieved through a combination of tutorials, completed examples, exercises and mini-projects, mentoring, and intervention. The implementation of such a training program is an ongoing project.

Table 2. Testing Experiences/Competencies Across the Testing Lifecycle

Competency Areas	Duties
Test Practitioner	Perform a defined test within an established test environment, and document results.
Test Analyst	Handle front-end of testing process. Determine testing needs. Assure that the specification is an adequate basis for starting the testing process.
Test Designer	Given a specification, design test cases using published and recommended techniques.
Test Builder	Construct machinery needed to run test cases.
Test Inspector	Verify that a testing task has been performed correctly according to standards and procedures.
Test Environmentalist	Set up and support the test environment.
Test Specialist	Perform a series of testing tasks spanning multiple phases of the test life cycle.

3.3 The SPRAE Software Testing Framework

SPRAE is an acronym for major principles of quality assurance that can be applied to software testing [1, 2]. In this section each symbol in the acronym is expanded. A course in software testing should include experience applying each of these principles.

- **Specification**. *A specification is essential to testing.* The specification states the expected behavior of software, and is the basis for testing activity. The simple example makes this point clear: In order to tell whether 11 is the correct output from a function taking inputs 3 and 7 requires that the tester know what the function is expected to do. The specification states the expected externally visible behavior of software. This principle can be stated alternatively as *No spec, no test.*
- **Premeditation.** *Testing requires premeditation,* i.e., a systematic design process. Testing is not a random activity, but is based upon principles and techniques that can be learned and practiced systematically. This principle suggests that there is a test life cycle that includes stages of analysis and design. The various test design techniques [4] are based on theoretical and empirical foundations.
- **Repeatability.** *The testing process and its results must be repeatable and independent of the tester.* The practice of testing must be institutionalized to the extent that all practitioners use the same methods and follow the same standards. In terms of the SEI CMM, the process must *repeatable* and *defined*, and practitioners must be trained.

- **Accountability.** *The test process must produce an audit trail*. Test practitioners are responsible for showing what they planned to do (premeditation), what they actually did, what the results were, and how the results were interpreted. These results must be reviewable by others. This principle may require the capture, organization and retention of large volumes of data.
- **Economy.** *Testing should not require excessive human or computer resources.* In order for a test process to be followed, it must be cost effective. Concern for cost provides a counterbalance to the previous elements of the SPRAE framework. The economy principle is often the single driver of an organization's test practice. Economy fuels the pursuit of automated testing tools and other "silver bullets."

The SPRAE framework represents a way of understanding why testing is a complex process with competing dimensions of theory and practicability. SPRAE is a tool for negotiating these complexities.

4 Assignments

Twelve laboratory assignments were given, three focused on Unix/environment skill development, and eight on software testing. The Unix emphasis recognizes the fact that testers need proficiency in using a programming environment for test automation and for storing and managing test artifacts. (A subliminal message of the course is that Unix is really an important skill in one's pursuit of a job.) The depth of treatment of Unix shell script programming was less this term than in previous terms: students were required to adapt an shell script that automated the testing of a small application (assignment #7).

A goal of the course was to cover all phases of the testing lifecycle and to give students experience applying each testing competencies at least once. It was felt that doing so would give students a sense of the breadth of software testing. Testing tools were not used; instead students built several principal tools. The rationale is that students should understand the underlying effort associated with testing in order to appreciate what problems the tools are solving and not solving.

Assignments can be characterized in terms of the type of skill students were expected to learn and apply. Table 3 summarizes the skills involved in each assignment

A. Observation Skills – ability to systematically explore software behavior and to describe observations in a generalized manner.
B. Specification Skills – ability to describe expected or actual behavior of software.
C. Programming Skills – ability to write code to satisfy a specification, or to build the machinery to automate testing.
D. Test Design Skill – ability to derive test cases following a systematic approach based on standard techniques.
E. Team Skill – ability to work in teams to perform testing task.

The appendices include specifications for six lab assignment to give insights into the test experiences students were given. One requirement for all assignments was that students follow standards for testing artifacts. Naming conventions were enforced to ensure that test artifacts could be located and verified for content and format. Assignments were progressive: artifacts from one assignment flowed into subsequent assignments. These appendices are expected to generate recommendations and discussion from workshop participants.

Table 3. Skills Required/Developed in Assigned Projects	
Assignment	**Primary skill set**
#1. Unix Lab 0 Profile	Programming. Orientation to programming/testing environment.
#2. Test Lab 1A – Blind Testing	Observation/Specification.
#3. Test Lab 1B - Blind Testing	Programming
#4. Unix Lab 1 Unix command overview	Programming.
#5. Test Lab 2 Test Documentation: scripts, logs	Documentation/Specification.
#6. Test Lab 3 Software specification	Specification. Transform natural language spec into lightweight formalism, decision table (DT)
#7. Test Lab 4 Test Design, Implementation, Execution AUTOMATION	Test Design from DT. Programming.
#8. Unix Lab 2 Unix utilities	Programming. Intermediate Unix features, intro to shell scripts.
#9. Test Lab 5 Unit Test Driver Construction (with oracle)	Programming.
#10. Test Lab 6 White-Box Unit Testing	Test Design. Hand instrumentation.
#11. Test Lab 7 Class Testing via Test Driver	Test Design. Programming. Object-oriented testing.

5 Course Reflection

The observations that follow regarding the nature of the testing course, its importance to the development of the student, and factors affecting student performance have been reported in [2]. Many of these finding are being corroborated by others [5, 6].

- Testing is a programming intensive activity. Students with deficient programming skills have a difficult time with the programming aspects of testing.
- Testing requires analytical skills and facility with mathematical reasoning tools such as decision tables and graphs.
- Students are surprised and annoyed by the amount of mundane record-keeping testing requires.
- The in-depth treatment of concepts and practices in this course gives students a decided advantage when entering the workforce. Several recent graduates have taken on leadership roles in software testing, depite being in entry-level IT positions.
- Students who wait to take the elective course lose numerous opportunities to learn and reinforce testing skills in contexts other than the "testing class."

These insights from earlier instances of this course motivated me to consider an incremental approach that disperses testing concepts and experiences across the entire undergraduate experience [1,2]. This finding is consistent with recent thinking. Edwards [6] recommends a test-first approach to computer science education to ensure programming and analytical skills are developed. Tool support for this approach has been proposed by Patterson et al [7] who have integrated JUnit into the BlueJ Java teaching environment. The skills that testers need are complementary to those needed by developers: it makes sense to develop both concurrently. Another justification for integration is simply that students tend to compartmentalize knowledge to a single course and do not transfer it to new situations. Christensen [5] endorses the radical position that a separate course sends the wrong message, that software testing is a separate from computer science, not an essential part of the body of knowledge for the computer scientist.

My experience is that the goal of integration, though laudable, is very difficult to achieve. In the near term, testing courses are necessary. The evolution of this course has moved in the direction of providing a broad coverage to round out the educational experience of all students, which differs from my preference that this course prepare students for advanced study and careers in software testing.

5.1 My Expectations

At the outset of the course I wanted to achieve certain things.

- Coverage of testing at all levels – development (unit), classes (reinforce programming concepts), applications (in-the-small). Did not cover applications in the large (e.g., websites or end-user applications).
- Students would appreciate the labor associated with testing and the required attention to detail. This would be achieved by doing some of the dirty work testers must do, the hardest of which is to keep good records.
- Students would learn a few basic concepts that would be reinforced by practical application exercises.
- Students would develop intuitive testing skills.
- A secondary goal was to take steps towards transforming this course into a distance or self-paced learning course by expanding the set of courseware (notes, examples, exercises, test anks).

5.2 Challenges

Educators are born with the drive of continual improvement, especially when they believe in the importance of the subject matter. The challenges I faced this offering the course are listed below.

- A good text book. Students associate learning with the experience of growing familiar with a good textbook. I did not find a text that matches the course content and instruction approach.
- Balancing theory and practice. My philosophy is that experience is the foundation for grasping theory. Students often separate doing from learning, as evidenced by exams and reflective discussions (e.g., what did you learn from doing assignment #5?).
- Providing a laboratory environment that supported supervised skill development.

- Engaging students in discussions, where they must use the language of testing.
- Providing timely and constructive feedback (a la apprenticeship).

5.3 Feedback

Due to time pressures only anecdotal feedback was gathered from this instance of the course.

- Students felt there was too much work.
- Some students discussed course in their interviews, where it was viewed as valuable by employers.
- A solid laboratory environment and assignments would help students learn.
- Students desired an opportunity for discussion assignments. They felt they were left to their own devices when carrying out assigned projects. (As usual, students seldom began working on assignments until they were almost due, making such discussions infeasible. A laboratory approach with mini-milestones may be more workable. I would want students to get used to talking the language of testing.)
- Students did not feel they were provided timely and constructive feedback (a la apprenticeship).

An encouraging source of feedback came from faculty colleagues who not only recognized the value of a testing course, but promoted the course to students.

5.4 Review of Outcomes

- Goal was to give experience of several of aspects of testing to provide a basis for careers or internships in testing.
- Formalisms are difficult for students. The decision table is simple but not easy for students to grasp. Formalisms are important because they are the basis for systematic analysis and test case design that are amenable to automation.
- My ability to provide feedback was limited by my workload. I used automated grading to reduce my workload and to provide consistent feedback to students. This does not substitute for one-on-one or group discussion. (Post mortems after assignments will be included in the next course offering.)
- Feedback from students who discussed this class with corporate interviewers (Microsoft, in particular) indicated that formalism is secondary to economical test case design (e.g., small but effective test sets).
    - Student – too much formalism
- The SPRAE testing framework was intended to give holistic view to software testing – testers must have knowledge and skill to perform a range of activities, along with certain disciplines for accountability and cost.
- The course provided a tool-poor environment, partly by design – to convey the detailed, repetitive and tedious nature of principled testing. I believe this experience will lead to the proper use and appreciation for test automation.
- The balance of experience and concept needs improvement. The issue of a textbook will be addressed by providing Powerpoint notes, on-line practice tests (study guide), with a complement of examples and exercises. (The problems assigned this term will become examples for the next instance of the course.)

- As far as developing "intuitive" testing skills, I am not sure that students really "got it." This could be due to the programming emphasis that may have obscured the testing emphasis. These are manifested most at the application level, which we did not cover. The intent of the blind-testing laboratory was to stimulate creativity, but the problems assigned were too simple! (An outside-in approach will be considered the next time the course is offered.)
- The secondary goal of transforming this course towards a distance or self-paced learning course was furthered. The set of courseware (notes, examples, exercises, testbanks) was extended and made more self contained and usable remotely.
- One can not accomplish everything in a single course.
- One third of the students showed genuine aptitude and interest in testing.

6 Improvements

The following improvements will be attempted the next time this course is offered. Resources that will be tapped to implement these changes include materials shared through this workshop, those available through the public domain (e.g., SourceForge) and materials shared with other educators.

- Improved lecture notes or text book.
- Testing in the large followed by testing in the small.
- Recitation/laboratory session for discussions and feedback.
- Automation to evaluation student work.
- Increased use of testing tools, custom and commercial.
- Accumulate a test bed of code/applications to be tested.

7 References

[1] Jones, E.L., "An Experiential Approach to Incorporating Software Testing into the Computer Science Curriculum," 2001 Frontiers in Education Conference (FIE 2001), Reno, Nevada, October 10-13, 2001, F3D7-F3D11.

[2] Jones, E.L., "Integrating Testing into the Curriculum -- Arsenic in Small Doses," *Proceedings 32nd Technical Symposium on Computer Science Education,* 21-25 February 2001, Charlotte, NC, 337-341.

[3] Jones, E.L., "SPRAE: A Framework for Teaching Software Testing in the Undergraduate Curriculum," *Proceedings of ADMI 2000,* Hampton, Virginia, 1-4 June 2000.

[4] Jones, E.L. and C. L. Chatmon, "A Perspective on Teaching Software Testing," ***Journal*** *of Computing in Small Colleges, Vol. 16*, 4, March 2001, 92-100.

[5] Henrik Bærbak Christensen, "Systematic testing should not be a topic in the computer science curriculum!" *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, June 30 - July 2, 2003 , Thessaloniki, Greece, 7-10.

[6] Edwards, Stephen H., "Rethinking computer science education from a test-first perspective," OOPSLA'03, October 26-30, 2003, Anahiem, California, 148-145.

[7] A. Patterson, J. Kolling, and M. Rosenberg, "Introducing Unit Testing with BlueJ," *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, June 30 – July 2, 2003, Thessaloniki, Greece, 11-15.

A Appendix. Assignment: Blind Testing

A.1 Problem Specification

```
------------------------------------------------------------------------------
Test Lab1 - "Blind" Testing
==========================

Part I: Due 9/2/03
You will be given a program to test. You will be given an executable file that
runs on the PC. You will also be given a general description of the program.
Your job is to determine the conditions under which the program works
correctly, and the conditions under which it does not work correctly. Write
up the results of your testing with enough detail to support your findings.

              Add01   Add02   Hello01   Hello02   Pay01   Pay02
   Lastname    A-D     E-I     J-M       N-Q       R-S     T-Z

Part II: Due 9/9/03
You must write an EQUIVALENT C++ program (that has the same behavior, warts and all).
When tested the same way you "blind tested" your implementation should produce the
same "actual outputs".

The programs:
    Add   - Add two numbers.
            Inputs: integer, number1
                    integer, number2
            Output: integer sum
    Pay   - Compute pay for hourly employee.
            Inputs: integer, hours worked
                    integer, hourly pay rate
            Output: integer, gross pay.
    Hello - Issue greeting to user.
            Inputs: user name (up to 12 characters)
            Output: greeting.

REPOSITORY USAGE:

(1) FROM public repository (TP)
    - Assignment: File that tells you YOUR assigned program.
    - Name:       hw_testlab01-assignments
    - Executable code to be tested:
      - Naming:  programname.exe
      - Example: add01.exe
    - Test report:
      - form:     frm_trp-testreport.txt
      - standard: std_trp-testreport.txt

(2) TO submission repository (TS)
    Part I:
        1. Test report (note the prefix "trp_"):
           - naming:  trp_programname.txt
           - example: trp_hello02.txt
              NOTE: You must rename the frm_trp-testreport file.
    Part II:
        1. Equivalent program:
           - naming:  programname.cpp
           - example: hello02.cpp
        2. Test report for YOUR implementation
           - naming:  trp_2programname.txt
           - example: trp_2hello02.txt
------------------------------------------------------------------------------
```

B Appendix. Assignment: Test Documentation

B.1 Assignment Specification

```
------------------------------------------------------------------------------
Lab2 - Test Documentation
=========================
You will formally document and test the program assigned below (based on your last name). You
will be given a specification for the program and a Unix executable file.

   Your Program  -->   Pay02  Pay01   Add02    Add01   Hello02   Hello01
   Your Lastname -->   A-D     E-I     J-M       N-Q       R-S     T-Z

Your job is to write a test script BEFORE you conduct the test. The script spells out what tests
you will perform and what you expect the results to be for each test case. The script also
includes instructions for setting up the test.  When you execute the tests, i.e., do the tests
you defined in the test script, you must check the result to determine whether the test case
passed, i.e, produced the expected result. The pass/fail status is maintained in the test log.

Samples of a test script and test log can be found on the Samples web page.

 1. TEST SCRIPT: defines the steps necessary to perform the test:
    a) Set Up:
           - getting the executable from the TP repository
           - copying the executable to the floppy
           - getting the a blank test log form from TP repository, or an
             existing test log from the TS repository
           - copying the test log to the floppy

    b) Running the test(s): step by step instructions for running the code under test and
       supplying test cases as input data. Include instructions for completing the test log entry
       for each step. (When a step fails, a note must be included giving more information about
       the nature of the failure, i.e., what was actually observed.)

    c) Instructions on how to verify the overall success of the test. (If one step fails, the
       test fails, and either the code under test or the test script must be modified, and the
       test script repeated.)

 2. TEST LOG: The primary documentation that testing has occurred is the Test Log. (In the future
    there will be various output files produced in the course of testing.)

    a) Test Log. Each time the test script is performed, a column of the test
       log must be completed.
       - each column must be labeled with the date (and time, if performed
         more than once on a given date)
       - Footnotes must be established for each test session. The first
         footnote identifies the tester. Subsequent footnotes are used to
         explain observed behavior upon failure. Failure footnotes look like:

             step xx: Pay = 10 (expected 20).
             step zz: no pay displayed (expected 400)

REPOSITORY USAGE:

(1) FROM public repository (TP)
    a) test script
       - standard:    std_uts-testscript.txt
       - form:        frm_uts-testscript.txt
    b) test log:
       - standard:    std_utl-testlog.txt
       - form:        frm_utl-testlog.txt
    c) Specifications for programs being tested.
       - file:        spec_blindtests.txt
    d) Executable for your program (Unix only).
       - file:        programname.run     (e.g., add01.run)
(2) TO submission repository (TS)
    a) test script: uts_programname.txt  (e.g., uts_add01.txt)
    b) test log:    utl_programname.txt  (e.g., utl_add01.txt)

------------------------------------------------------------------------------
```

C Appendix. Assignment: Software Specification based on Observed Behavior

C.1 Assignment Specification

```
------------------------------------------------------------------------------
Lab3 - Writing a Formal Specification
====================================


Write a formal specification for the unit under test. The specification states
clearly and unambiguously WHAT the unit does, based on your observations from
testing. The specification also states the CONDITIONS under which the unit
behaves in a particular manner. Preconditions state the criteria that must
be met by the inputs in order for the unit to have produce the correct result
(e.g., to take the square root the argument must be >0). The specification
must describe the "entire behavior" of the unit. Therefore, the specification
must also detail what happens when the preconditions are not satisfied.

You will develop the specification for the assigned program (based on
your last name). The description of the program is found in uut-list.html.


   Your Program  -->   cdi     pfg     mup       cls       ret     dup
   Your Lastname -->   A-D     E-I     J-M       N-Q       R-S     T-Z


REPOSITORY USAGE:

(1) FROM public repository (TP)
    a) Black-box schematic
          standard: std_bbs-bboxschematic.txt
          form:     frm_bbs-bboxschematic.txt
    b) Specification
          standard: std_spc-specification.txt
          form:     frm_spc-specification.txt
    c) Decision table
          standard: std_dtb-decisiontable.txt
          form:     frm_dtb-decisiontable.txt

(2) TO submission repository (TS)
     a) Schematic:      bbs_xxx.txt
     b) Specification:  spc_xxx.txt
     c) Decision table: dtb_xxx.txt

------------------------------------------------------------------------------
```

D Appendix. Assignment: Application Test Automation I (shell scripts)

D.1 Assignment Specification

```
-------------------------------------------------------------------------------------------
Test Design, Implementation & Execution
=======================================

The purpose of this assignment is to design functional and boundary test cases based on the
specifications developed in the previous lab. You will also develop simple shell scripts to
automate the execution of test cases.

Your program assignments remain the same as the previous lab.

   Your Program  -->   cdi     pfg     mup       cls       ret     dup
   Your Lastname -->   A-D     E-I     J-M       N-Q       R-S     T-Z
```

Part I. Manual Testing

```
Your job is to write a test script that contains the test cases. You can get the executable of
your assigned program from the public repository. Execute the manual test script and complete the
test log. Beware: There will be bugs in the executable.

REPOSITORY USAGE:

(1) FROM public repository (TP)
    a) test script
       - standard:    std_uts-testscript.txt
       - form:        frm_uts-testscript.txt
    b) test log:
       - standard:    std_utl-testlog.txt
       - form:        frm_utl-testlog.txt
    d) Executable for your assigned program (Unix only).
       - file:        programname.run     (e.g., add01.run)

(2) TO submission repository (TS)
    a) test script: uts_1programname.txt  (e.g., uts_1add01.txt)
    b) test log:    utl_1programname.txt  (e.g., utl_1add01.txt)
```

Part II. Automated Testing

```
Your job is replace the table in the test script for manual testing with instructions for
invoking the a shell script that executes the tests and records results to an output file.
Create a new test log for the automated test. NOTE: You will still have to compare expected and
actual results and note failures in the test log.

REPOSITORY USAGE:

(1) FROM submission repository (TS)
    a) test script: uts_1programname.txt
    b) test log:    utl_1programname.txt

    Hint: Copy each file into the corresponding utx_2programname.txt file.
          Then make simple editing changes.

(2) TO submission repository (TS)
    a) test script: uts_2programname.txt  (e.g., uts_2add01.txt)
    b) test log:    utl_2programname.txt  (e.g., utl_2add01.txt)
    c) test driver shell script:  utd_2programname.csh (e.g., utd_2add01.csh)
    d) test results file: utr_2programname.txt (e.g., utr_2add01.txt)

    e) BONUS:
       - test driver shell script:  utd_3programname.csh (e.g., utd_3add01.csh)
       - test log:                  utl_3programname.txt  (e.g.,utl_3add01.txt)
       - test results file:         utr_3programname.txt (e.g., utr_3add01.txt)
       - test data input file:      tdi_3programname.txt (e.g., tdi_3add01.txt)

-------------------------------------------------------------------------------------------
```

E Appendix. Assignment: Unit Test Automation I (C++ driver)

E.1 Assignment Specification

```
-------------------------------------------------------------------------------------
Test Lab5 - Unit Test Drivers (C++)
==================================

Purpose: The purpose of this assignment is to perform all the phases of the testing lifecycle for
your assigned software unit (this time a subprogram).

  1. Analysis - develop a valid/complete decision table capturing the function the unit performs;
        a) Create decision table, dtb_xxx.txt

  2. Design -- apply the test case design methods for functional and boundary testing.
        a) Complete the test design worksheet, tdw_xxx.txt showing how
           functional test cases and boundary test cases were derived.

  3. Implementation -- Create the "machinery" needed to execute tests.
        a) Create test data input (tdi) file containing the functional and
           boundary test cases -- put functional test cases first.
        b) Encapsulate the assigned module u_xxx.cpp into the test class f_xxx.cpp. Throughout
           this homework, keep the original code in a separate member function.
        c) Create the oracle-style C++ test driver for your assigned module.

  4. Execution  -- Run the test driver using the test cases in the tdi file, documenting results
     in the test results file, utr_xxx.txt. You will supply a different test execution id
     (00,01,02,...) each time you run the test driver. All test results will be stored in a
     single utr_ file, but each result will contain the execution ID. The utr_ file serves as
     a HISTORICAL test log!

  5. Evaluation -- Analyze test results, noting failures. In this assignment you will have to
     identify the bug that causes the software to fail. Please fix one bug at time, retesting
     after each fix to ensure that fixes are not having the negative effect, i.e., creating
     new bugs.

  6. Correction & Regression. Fix one bug at a time, then retest to confirm whether the fix
     worked without introducing other errors.
        a) Bug analysis -- document the hypothesis about what needs to be done
           to remove the bug. Use the bug-fix-log (bfl_xxx.txt) to document
           the failure, the bug fix hypothesis what needs to be changed);
        b) Apply the bug fix to the code; make a copy of the original code
           before making changes.
        c) Re-run the test and evaluate the results. Each time you run the
           test driver, use a different test run ID#.
        d) Record in the bug-fix-log whether the fix in fact removed the bug
           and/or whether additional bugs were introduced by the fix.

Your program assignments:
    Your Program  -->   mup     dup     pfg       ret       cls     ret
   Your Lastname -->   A-D     E-I     J-M       N-Q       R-S     T-Z

REPOSITORY USAGE:

(1) FROM public repository (TP)
    a) Sample source files shown in the lecture.
       add.cpp f_add.cpp u_add.cpp tdw_add.txt utd_add.cpp tdi_add.txt utr_add.txt bfl_add.txt

(2) TO submission repository (TS) [Note: xxx = name of  your function.]
    a) Function Decision Table:         dtb_xxx.txt.
    b) Test Case Design Worksheet file: tdw_xxx.txt
    c) Test Driver Input file:          tdi_xxx.txt.
    d) Corrected Encapsulated function: f_xxx.cpp file.
    e) C++ Test Driver source file:     utd_xxx.cpp
    f) Test results output files:       utr_xxx.txt
    g) Bug Fix Log file:                bfl_xxx.txt

-------------------------------------------------------------------------
```

F Appendix. Assignment: White Box Unit Testing

F.1 Assignment Specification

```
-------------------------------------------------------------------------------------------
Test Lab6 - White-Box Unit Testing
==================================

Purpose: The purpose of this assignment is to give you practice doing white-box testing. You will
         manually construct the control flow graph, instrument the source code to collect
         execution flow data, and perform the coverage calculations.

  1. Construct the control-flow graph for your unit xxx.
     a) Create file cfg_xxx.txt that contains the control flow graph. Use the file format below.

                  File format           Example
                ------------------    --------------
        line 1: #nodes               |  4
        line 2: edge1                |  1 2
        line 3: edge2                |  1 3
        etc.

  2. Edit the class for the unit under test, f_xxx.cpp as follows:
     a. Update the f_xxx class declaration.
        - Add a new method i_xxx that has the same signature as method xxx;
        - Add method void rv(int node); // Record Visit to node.
        - Add method void startTrace();
        - Add method void stopTrace();
        - Add private output file stream variable, wbtF, for the "white-box trace" file.
      b) Update the f_xxx class body:
        - Edit method void startTrace() so that it opens file "wbt_xxx.txt" in append mode.
        - Edit method void stopTrace() so that it closes the output file stream wbtF.
        - Edit method rv(int node) so that it writes a line to the output file. The line
           contains the name of the unit and the node number:  xxx node  (e.g., mup 7).
        - Copy the definition of method xxx(..), and rename it i_xxx(..).
      c) Instrument method i_xxx(..) by inserting compound statements and instrumentation probes
          (the call  "rv(node#);" statements) as shown in class.

  3. Verify that the instrumentation is correct by repeating the last set of black-box test
     cases. In the driver, change the function call from xxx to i_xxx.
     a) Use the same set of test cases in tdi_xxx.txt.
     b) Use a DIFFERENT test results file: 'utr_wbt-xxx.txt'.
     c) Start your test execution IDs at W01, then W02, etc. (W means "white-box").
     d) The same outcomes from the last test execution recorded in the utr_xxx.txt file should be
        obtained from execution W01. OTHERWISE, your instrumentation has changed the meaning of
        the program. If so, you must fix the instrumentation and repeat step 3.

  4. Analyze the results from white-box testing.
     a) For each trace in the wbt_xxx.txt file, manually trace out the traversal of the
          control flow graph. Accumulate markings for nodes visited and edges traversed.
     b) Store coverage results in file 'wbc_xxx.txt', in this format:

         xxx.x = NODE COVERAGE
         xxx.x = EDGE COVERAGE
         xxx.x = BRANCH COVERAGE
         xxx.x = PATH COVERAGE

REPOSITORY USAGE:

(1) FROM public repository (TP): NONE.

(2) TO submission repository (TS) [Note: xxx = name of your function.]

    a) Control flow graph:                    cfg_xxx.txt
    b) White-box trace results:               wbt_xxx.txt
    c) White-box coverage file:               wbc_xxx.txt
    d) Instrumented Encapsulated function:    f_xxx.cpp
    e) Test driver for white-box testing:     utd_wbt-xxx.cpp
    f) White-box test results file:           utr_wbt-xxx.txt
 -------------------------------------------------------------------------------------------
```

G Appendix. Assignment: Class Testing

G.1 Assignment Specification

```
------------------------------------------------------------------------------
Test Lab7 - Class Testing
=================================

Purpose: The purpose of this assignment is to give you practice applying black-box testing
         techniques learned for unit testing to testing a class.

Logistics: This is a team project, for teams up to 3 people. WE WILL NOT USE the team repository.
           You must inform me of the members of your team, then designate specific members to
           submit work on behalf of your team.

Details: The class you will test has a number of public and private methods. Testing this class
         requires that you understand what the class does and how the class is designed. You are
         expected to FIND and document the bugs in the "bug fix log." Your grade will be based
         on the skill you demonstrate in finding the bugs AND describing them -- what the
         shortcomings were AND what code changes are required to fix them.

         Your tests will be against object files simpleacct0x.o, so you will not see the source
         code. Each team must test 2 different object files.

         simpleacct:   01          02          03          04          05
                     ----------  ----------  ----------  ----------  ----------
         Teams:      01 06 03 08  02 07 04    03 08 05    04 01 06    05 02 07


         This is a team assignment to get to discuss and "talk shop" about your testing. You must
         Perform all stages of the testing lifecycle stages of:
         1.  Analysis - (decision table for each method)
         2.  Design - functional and boundary testing of the methods
         3.  Implementation - modify the given test driver and create the tdi_ file)
         4.  Execution - run the driver using the test cases (tdi_ file)
         5.  Evaluation - identify and document the bugs (use the bug fix log bfl_xxx.txt).

         I will ask you to give in-class presentations of your work, so be prepared to talk in a
         larger group about the testing you'll do on this assignment.

REPOSITORY USAGE:

(1) FROM public repository (TP): NONE.
    a) Class specification:    spc_simpleacct.txt
    b) Class declaration:      simpleacct.h
    c) Class object files:     simpleacct01.o, simpleacct02.o, simpleacct03.o,
                               simpleacct04.o, simpleacct05.o
    d) Class test driver:      ctd_simpleacct.cpp

 (2) TO submission repository (TS)
    a) Class Decision Tables:             dtb_simpleacct.txt.
    b) Class Test Case Design Worksheet : tdw_simpleacct.txt
    c) Class Test Driver Input file:      tdi_simpleacct.txt.
    d) Class Test Driver source file:     ctd_simpleacct.cpp
    e) Class Test Results output file:    ctr_simpleacct0#.txt
    f) Bug Fix Log file:                  bfl_simpleacct0#.txt


        NOTE: 0# = the version of simpleacct you tested.

------------------------------------------------------------------------------
```