A PERSPECTIVE ON TEACHING SOFTWARE TESTING

Edward L. Jones and Christy L. Chatmon
Department of Computer and Information Sciences
Florida A&M University
Tallahassee, FL 32303
850-599-3050
{ejones,cchatmon}@cis.famu.edu

ABSTRACT

This paper addresses the issue of how to approach the teaching of software testing. Relative to its importance in the software industry, software testing receives very little attention in the undergraduate curriculum. It is not practical to offer a separate course in software testing, so relevant test experiences need to be given throughout core courses. This paper presents a software testing framework identifying five essential principles that motivate the types of testing experiences a student should gain from the curriculum. The framework is illustrated using a simple test problem.

1 INTRODUCTION

How should educators approach teaching the basics of software testing? This is the issue discussed in this paper. Software testing is important to the software industry, with testing accounting for over 50% of the cost of software. Software testing is expected to play an even more prominent role as software complexity and the legal ramifications of poor quality continue to increase [1].

There is no lack of research in software testing but, according to Osterweil, the chasm between research and practice is so wide that it blocks needed improvements in both communities [1]. Software testing research must be less academic and more practical (palatable) to industry consumers [2]. Finally, educators must do a better job equipping students with skills and attitudes for dealing effectively with software quality concerns.

Although testing has been a part of classical computer science literature for decades [3], the subject is seldom incorporated into the mainstream undergraduate experience. One radical approach is a quality-centric curriculum [4] based on Humphrey's personal and team software processes. A more practicable approach is to give every student a common set of testing experiences from core courses. What remains is a way to identify a minimal set of testing experiences.

A holistic approach is required [5]. A premise of this work is that students who learn to test become better software developers: testing forces the integration and application of the

JCSC 16, 3 (March 2001)

software development skills of analysis, design and implementation. Another premise is that repeated exposures are essential to learning. Addressing the issue of software testing by offering an elective course is only a partial solution. In *The Art of Software Testing,* the definitive work on software testing, Myers identifies attitude (psychology) and economics as major obstacles to testing [3]. Students must face these issues of testing in the normal course of software development, not in the isolation of a separate testing course.

2 THE SPRAE FRAMEWORK

SPRAE is an acronym that encodes five principles that are essential to the practice of software quality assurance: s*pecification, premeditation, repeatability, accountability and economy*. This section introduces these principles. Taken as a whole, the SPRAE framework gives insight into the management, technical, and legal/ethical aspects of software quality assurance [6].

- **Specification**. *A specification is essential to testing.* 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 draws attention to the fact that there is a test life cycle, such as that shown in Table 1. This principle also recognizes the need for multiple test design techniques [7] based on theoretical and empirical foundations.
- **Repeatability**. *The testing process and its results must be repeatable and independent of the tester.* Testing should be institutionalized: 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 adequately.
- **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 in a form others can review. The accountability principle requires the capture, organization and retention of large volumes of data.
- **Economy.** *Testing should not require excessive human or computer resources.* Unless a test process is cost effective, it will not be followed. 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.

The SPRAE framework gives insight into the attitudes and skills needed to be a test practitioner. As discussed in section 4, the framework also identifies experiences students need to receive during undergraduate study to be equipped to deal effectively with the quality aspect of software careers.

Table 1. Test 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 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 AN EXAMPLE – FUNCTION TESTING

In this section we illustrate the use of SPRAE to test a simple function. Pay, which computes the weekly gross pay for an hourly employee. Pay takes two arguments and returns a single real number. The testing problem is this: Does Pay always produce the correct answer? The test process has the following steps.

1. Develop a set of test cases;
2. Develop a test script defining a repeatable process for building and executing the program against the test cases;
3. Execute the function for each test case in the test set, and
4. Verify that for each test case the computed pay matches the expected pay.

3.1 Specification

A specification is a statement of what the function does. The specification is required to determine the expected result portion of a test case. A specification may take various forms. We will illustrate three: natural language narrative and semi-formal narrative with pre- and post-conditions (in this section); and decision table (in the next section).

Specification #1 (natural language narrative): Compute pay for an hourly employee, given the hours worked and hourly pay rate. Compute overtime at 1.5 times for hours over 40.0.

A set of test cases for this specification shown in Figure 1. One test case covers overtime, the other does not.

Figure 1. Pay Test Set #1

Rate	10	10
Hours	40	50
Expected Pay	400.00	550.00

This specification is simplistic, as it does not consider practical limits such as a maximum hourly pay rate or the maximum number of hours an employee can work per week (it can never exceed 168 hours). A more complete specification follows.

Specification #2 (semi-formal narrative): Compute pay for an hourly employee, given the hours (Hours), and hourly pay rate (Rate). Compute overtime at 1.5 times for hours in excess of 40. Return -1 when Hours or Rate is invalid.

Preconditions: (0 £ Hours £ 112) and (5.15 £ Rate £ 20)

This specification requires a more systematic process for test case design.

3.2 Premeditation

This section illustrates a systematic process for designing functional test cases based on the specification. When the specification is simple, an intuitive process of deriving test cases is often adequate. In general, however, a systematic, repeatable process is needed to translate the specification into test cases. It is also highly desirable that this process be amenable to automation. We illustrate this process using Specification #2. The narrative specification will be converted to a logic-based specification, a *decision table* (Figure 2). A decision table is a formalism amenable to straightforward test case generation. The upper part of the decision table defines the implicit and explicit binary conditions stated in the specification; the lower (shaded) portion specifies the computation rules defined in the specification. The columns define the different behaviors of the function. For a given column, the combination of condition values ('Y' or 'N') selects the computational rule(s) marked by 'X'.

Figure 2. Decision Table for Pay

Hours ³ 0	Y	Y	N	-	-	-	-	-		
Hours £ 112	Y	Y	-	N	-	-	-	-		
Rate ³ 5.15	Y	Y	-	-	N	N	-	-		
Rate £ 20.00	Y	Y	-	-	-	-	N	N		
Hours £ 40	Y	N	Y	N	Y	N	Y	N		

Pay = Hours * Rate	X									
Pay = Rate * [Hours + 1.5 (Hours – 40)]		X								
Pay = -1			X	X	X	X	X	X		
Function Behaviors	1	2	3	4	5	6	7	8		

Each column in the decision table defines a distinct behavior of the function determined by the combination of input conditions under which a subset of computation rules is selected. The test case design rule, “define one test case to satisfy each behavior,” results in a minimal number of test cases that treat each different behavior of the function. The focus on distinct behaviors is an example of an *equivalence partitioning* testing strategy [7]. The resulting test cases are shown in Figure 3.

Figure 3. Pay Test Set #2

	Test Case							
Test case #	1	2	3	4	5	6	7	8
Rate	10	10	10	10	2	3	40	30
Hours	40	50	-3	120	10	50	10	50
Expected Pay	400.00	550.00	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00

3.3 Repeatability

Testing is repetitive in nature. During initial testing, the goal is to achieve a "clean" test run, i.e., execute an entire test set without error. When testing uncovers an error, the cycle *"fix the error(s), then retest"* is followed. When significant changes are made, it may be necessary to do *regression testing*, where previously conducted tests are repeated to ensure that errors were not introduced by the changes. Both these situations require that the same test cases be executed. Repeatability is ensured only when test products–test data, test results, and the description of the test process–are governed by a carefully controlled process.

The primary test product for ensuring repeatability is the *test script*. The test script has two major parts: the test build, and test execution. The goal of the test build step is to ensure that the correct version of the software is tested: copies of controlled versions of the source code are retrieved and compiled to build the executable program, and required test data files are retrieved. The test execution part of the test script gives explicit instructions for executing the program on each test case. When the test requires a human in the loop, the test execution script is a detailed, step-by-step presentation of test cases, as shown in Figure 4.

Figure 4. Test Script Execution Steps (Program Pay)

	User Action		Expected Result
Step	Enter Rate =	Enter Hours =	Pay =
1	10	40	400.00
2	10	50	550.00
3	10	-3	-1.00
4	10	120	-1.00
5	2	10	-1.00
6	3	50	-1.00
7	40	10	-1.00
8	30	50	-1.00
9	END		

3.4 Accountability

Executing the test script produces test results that are saved in the test environment. During the *Test Evaluation* stage of the test lifecycle, test results are inspected for discrepancies between expected and actual results. Each apparent discrepancy is recorded in the *test log* (see Figure 5) in sufficient detail to determine the cause of the discrepancy. Discrepancies may result from (a) the wrong expected result is stated; (b) an error in the driver or data file; or (c) errors in the function under test.

The collection of all the products of the test lifecycle provides an audit trail that answers the basic questions, "W*hat was attempted?", "Who performed the tests, and when?", "What were the results?",* and *"What were the errors?"* These questions can not be answered unless a controlled process for storing, retrieving and updating test products is in place. Testing has the mandate of keeping sufficient records to answer these questions. The software engineering discipline of configuration management is an essential part of testing. The principle of accountability ultimately addresses issues of professional responsibility and, in the event of software induced catastrophes, professional liability.

Figure 5. Test Execution Log (Pay)

	Test Identification								
Step	01	02	03	04	05	06	07	08	09
1	P	P	P						
2	P	P	P						
3	P	P	P						
4	P	P	P						
5	F	P	P						
6	P	P	P						
7	P	P	P						
8	P	F	P						

END		
...		
Discrepancy Log		
TestID	Step	Description
01	5	16.50 returned. Expected –1.
02	8	222000.00 returned. Expected –1.

3.5 Economy

Economy addresses cost effectiveness. There is a need to work smart rather than work hard. Limitations in testing are accepted, some theoretical, others practical. The impossibility of exhaustive testing is an accepted theoretical result. The practical question, "How much testing is enough?" is often resolved by answering the question "How much resource (time, labor, computer) is available?"

The economy principle motivates practices that lower the cost of performing tasks in the test process. The obvious choices, to delete, skip or curtail steps in the process, should be resisted in favor of employing better test design techniques and using test automation. Repetitious tasks are prime candidates for automation. The principle of economy is best illustrated by the use of a test driver to automate the execution of test sets. Test drivers read test cases from a test data file, executes the program being tested, and writes results to the test results file, thus reducing the labor of executing tests and documenting the results. Test drivers ensure repeatability of tests, and provides the audit trail required by the accountability principle.

Test automation is a major thrust of current testing research [2,7]. Opportunities for automation occur throughout the test process. Often, the benefits of test automation extend beyond economy, and encompass repeatability, accountability. Opportunities for automation increase when formalisms are used to specify the software being tested.

4 APPLICATION OF THE SPRAE FRAMEWORK

In this section we show how the SPRAE framework can be used to derive a minimal set of test experiences to sprinkle throughout core courses. The software testing life cycle shown in Table 1 has the same phase names as the development life cycle. It should not be surprising that the skills required for software testing are complementary to development skills: learning a testing skill enhances the mastery of the related development skills. Table 2 lists typical test-relevant experiences students should gain at points in the curriculum. The following holistic objectives govern the mapping of these experiences to specific courses.

1. Each core course incorporates at least one testing activity.
2. The student gains experience in every phase of the testing life cycle.
3. Each testing activity incorporates one or more SPRAE principles.

4. The student gains experience applying each of the SPRAE principles.

Table 2. Test Activities in Core Courses

Test Phase	SPRAE Principle	Relevant Experiences
Analysis	S	Read and understand specification
	SP	Convert specification to a formalism, such as decision table
	SP	Identify deficiencies in specification
Design	P	Select test case design technique (white/black box)
	P	Design test cases using selected technique
	PE	Optimize test cases
	PRE	Define test plan
Implementation	PRAE	Develop test script
	PRAE	Develop test driver
Execution	RA	Perform a scripted test
	A	Document test results
	AE	Use a test drivers
Evaluation	A	Document program errors from test results
	PA	Debug test script
	PA	Debug program given description of error

5 EXPERIENCE TO DATE

The SPRAE framework has been in use in our department for the nearly two years, playing a pivotal role in three on-going activities [5,6,8,9]:

- Training of students in the software TestLab.
- Insertion of testing technology into the curriculum.
- Refinement of an elective course in software testing.

The software TestLab is an environment in which students learn the practice of testing and transfer practices into courses. As a part of their training, test lab students refine and develop sample test artifacts for insertion into selected courses. An on-going technology-insertion project involves treating the programming assignment task as a software testing problem: a SPRAE based testing life cycle is followed to develop automated graders for student programs [9]. Finally, an effort is underway to extract topics from an elective course in software testing and to package them as instructional modules for use in other courses.

6 CONCLUSIONS

Although it is impractical to teach every student all there is to know about software testing, it is practical to give students experiences that develop attitudes and skill sets need to address software quality. We have shown that SPRAE provides a framework for addressing the technical and managerial issues of software testing. We have demonstrated a goal driven approach to identifying desired test experiences and mapping them to courses in the curriculum. Our experience, to date, is that students benefit from the injection of even small doses of testing into existing courses.

7 REFERENCES

[1] Osterweil, L., et al, Strategic Directions in Software Quality. *ACM Computing Surveys 28,4* (December 1996), 738-750.

[2] Whittaker, J.A. What is Software Testing? And Why is It So Hard? *IEEE Software 17, 1* (January 2000), 70-79.

[3] Myers, G.J. *The Art of Software Testing*, John Wiley & Sons, 1976.

[4] Hilburn, T.B., and Towhidnejad, M. Software Quality: A Curriculum Postscript? *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education* (May 2000), 167-171.

[5] Jones, E.L.. Software Testing in the Computer Science Curriculum -- A Holistic Approach. *Proceedings 4th Australasian Conference on Computing Education,* (4-6 December 2000), Melbourne, Australia, 153-157.

[6] Jones, E.L. The SPRAE Framework for Teaching Software Testing in the Undergraduate Curriculum. *Proceedings of Symposium on Computing at Minority Institutions,* (June 1-4, 2000), Hampton Virginia.

[7] Zhu, H., Hall, P.A.V., and May, J.H.R. "Software unit test coverage and adequacy," *ACM Computing Surveys, Vol. 29*, No. 4, (December 1997) 366-427.

[8] 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, to appear.

[9] Jones, E.L., "Grading Student Programs - A Software Testing Approach," *Journal of Computing in Small Colleges 16,* 2, January 2001, pp. 185-192.