An Environment for Training Computer Science Students on Software Testing

Jim Collofello and Kalpana Vehathiri
Department of Computer Science and Engineering, Arizona State University
Tempe, Arizona 85287 collofello@asu.edu kalp@asu.edu

Abstract - **Software testing is essential to ensure software quality. On most software projects testing activities consume at least 30 percent of the project effort. On safety critical applications, software testing can consume between 50 to 80 percent of project effort. There is a vast amount of information available on software testing techniques and tools and some universities offer software testing courses at the advanced undergraduate and graduate level. Software testing must also be stressed in beginning courses and even in the high schools when students are first learning to program. It is also important in these early years to instill a respect for software testing and some interest in possible testing careers. This paper describes a project to develop a learning and training environment that enables students to develop the knowledge and skills to perform requirement based testing. The target audience for the environment are beginning programming students at either the high school or university level. The software training environment consists of web based instructional materials as well as a testing simulator which enables students to actually test software programs. This paper describes the educational objectives of the test training environment, its implementation and results from utilizing the environment with beginning programming students.**

Index Terms – software testing, training simulator, web-based training

BACKGROUND

Software testing is a costly and time-consuming process but is essential if a high quality product is to be produced. The importance of software testing and the establishment of good testing skills must begin as early as possible in computing curricula complementing programming skills. Various techniques exist for test design and execution [1-4] Some tests do not involve running the system at all; while other tests require inside knowledge of how the system works [5]. Some tests are run step by step by the tester (manual test execution or testing on a test bench), while in other cases a computer does the test execution, though skilled test engineers must perform the test design and result interpretation (automated testing or testing using simulators) [6]. The skilled test professional achieves the optimum return on the testing investment by selecting the appropriate test techniques.

Software testing is a time-honored approach for evaluating the software in terms of correctness, robustness, efficiency, functionality, and ease of use. The two crucial questions about software correctness are: (a) what exactly is the software supposed to do? and (b) is it doing exactly what it is supposed to do? The former is performed by the verification step of testing and the latter is performed by the validation step of testing [7]. Correctness is relative to a specification. Specification based testing performs the task of checking the code for correctness [8], [11]. The concept is to develop test cases to check if the responsibilities stated in the specification, the preconditions and the postconditions, are being fulfilled by the designed code. Other approaches to check correctness are: program proving [9] and static testing [10]. The task of program proving is to use mathematical proofs to prove that computer programs are consistent with their specification of their intent and purpose. The task of static testing is to inspect the programs with a list of predefined properties or structures to detect possible defects.

The two major categories of testing techniques are referred to as black box and white box testing. Black box testing is the verification of an item by applying test data derived from the specified functional requirements without considering the underlying product architecture or composition. Test case selection is based on an analysis of the component's specification without reference to its internal working. Black box testing is also called functional testing. White box testing is the verification of an item by applying test data derived from analysis of the item's underlying product architecture and composition. Test case selection is based on an analysis of the internal structure of the component. White box testing is also known as structural testing. Testing is done using knowledge of the internal working of the system, namely the program logic. The test training environment described in this paper was targeted at developing requirements based testing skills. In particular, the specific competencies to be developed included equivalence partition testing, boundary value testing and cause-effect testing.

A review of the published testing research identified numerous tools which support requirements based testing. AQTest, Test Mentor, Jtest, JUnitX, JtestCase, MockObjectes, JProbe and JMSAssert were examined. These tools provide support for capturing test cases, executing test cases, reporting errors, identifying code anomalies and assessing code coverage. The tools, however, provide little assistance to

0-7803-9077-6/05/$20.00 © 2005 IEEE October 19 – 22, 2005, Indianapolis, IN
35th ASEE/IEEE Frontiers in Education Conference
T3E-6

students trying to learn a particular testing method and obtaining feedback on their success. The goal of this research was to build upon the test automation concepts incorporated in these production testing tools and incorporate them into an environment in which students could learn about testing through feedback on the effectiveness of their testing.

OVERVIEW OF THE TEST TRAINER TOOL

The test trainer environment consists of both web-based test training materials and a test trainer tool which provides students with the opportunity of actually practicing their testing skills and obtaining automated feedback to assist in the learning process. The web-based training was developed as slide shows using Macromedia® DreamWeaver® and hosted on the web space provided by the school. The study material provided a walk through of testing. The material included graphic examples to provide a visual representation of the testing mechanism. The modules provided a walk-through of the basics of testing, namely the objectives of testing, the different models of testing – black box and white box, and the mechanism of test case selection namely boundary value analysis, equivalence partitioning, and cause effect. The mechanisms were explained with examples. Any queries could be sent to the web space owner by clicking on the *Contact Us* link. The tutorial is expected to take about 30 minutes to complete. The remainder of this section will address the features and design of the test trainer tool.

Test Trainer Functions

From a student perspective, the test trainer tool enables a student to download the specification of a program to test and to submit test cases to the tool. A test case consists of both inputs and expected results. After testing is complete, the student can view the results of the program's execution and indicate which tests passed and failed. Feedback is then provided to the student as to the number of errors which were detected, the number missed and the completeness of their testing in terms of test conditions executed. The student can then choose to continue to test or utilize the "learn by feedback" features. These features enable the student to view exactly which requirements were executed by their tests and which were not as well as providing them insight on how they might have detected missed errors. The effects of redundant tests are also shown.

From an instructor perspective, the test trainer tool enables an instructor to enter a program to test and its requirements into the tool. Templates are provided for structuring the requirements to facilitate parsing of input variables. The instructor can also enter assertions to identify errors and requirement conditions to be tested as further defined in the tool architecture section.

Test Trainer Architecture

To support the student and instructor functions, the high level architecture of the test trainer consists of four major components:

1. graphical user interface
2. parser
3. executer
4. metrics generator

The Graphical User Interface was implemented using the Java swing package. The event driven programming model was used for invoking tool functionality. A subset of the model view controller architecture was implemented. The tool environment was designed to contain a menu bar and two-editor panes, namely the requirement specification pane and the output pane. The menu bar acts as the controller section. The selection of menu options by the user acts as the events, which trigger the tool operation .The menu item selections by the user, causes the change in the views namely data being loaded onto the requirement specification pane, and the output pane contents. The model is the .Java file and the .txt file which gets manipulated in the tool environment.

The Parser was designed to analyze a requirement's text document and the assertion enabled Java code entered by the instructor. The Java language built-in parser *Java.io.StreamTokenizer* was used in the tool design. The tool assumes the existence of a predefined grammar, which provides a well-defined syntax for representation of the requirement text file and the Java program. The parser facilitates the conversion of the input files to a format in order to analyze their contents for conformance to their grammar specification. Two types of parsing were employed namely text parsing and Java file parsing.

Text file parsing was performed to identify the input variables, the output variables, and their data types for the program under test. The text parsing separates the requirement specification text from the variable "info" section. The details from the info section namely the number, type, and name of the variables are extracted to generate the test case input dialog box. Based on the variable types, the parsing functionality creates appropriate space in memory, to store the test case inputs against the appropriate variable list. The plain text English statements stated in the requirements are extracted and loaded in the requirements pane.

Source file parsing was done to create new Java source code and modify the existing Java source code. The code produced is compliant with the Java SDK1.4 so that it can be compiled by the Java Virtual Machine to the corresponding byte code. The test case inputs entered by user are stored in a text document. The parser extracts the test case inputs entered by the user to create the Java code called inputs Java code. The parser then works on the functional specification namely the assertion enabled Java file, to create a new assertion enabled Java file, which is a combination of input Java code and old assertion Java code. The goal of parsing is to create another copy of the original Java program with the input command line reader statements replaced with the test case

inputs entered by the user through the GUI. Thus, the values for the input variables are assigned to be equivalent to the test case inputs and the new copy of the program is executed under the tool environment. Figure 1 shows the text parser operation and Figure 2 shows the Java parser operation.

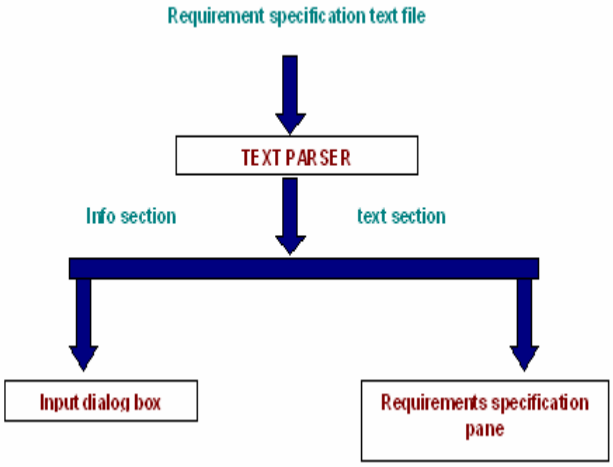

FIGURE 1: TEXT PARSER

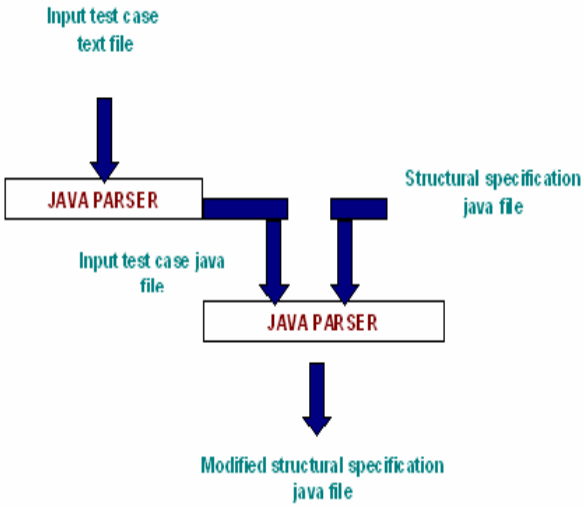

FIGURE 2: JAVA PARSER

The Executer provides the capability of creating a runtime environment within the tool environment. This enables the manipulation and execution of the assertion enabled programs based on test case inputs. The logic is to execute one program within the body of the other. A Java file called *input.Java* is created corresponding to the test case inputs. The execution call for this Java file is embedded within the main function section of the modified assertion enabled Java program, which corresponds to the requirement specification. The execution of *input.Java* causes the result to be dumped on the standard output, which in turn acts as the input to the modified assertion enabled Java program. This execution call for the modified assertion enabled code is embedded in the tool environment, which causes it to run under the entered test case inputs causing assertions to be thrown on the output pane of the GUI design of the tool. Figure 3 describes the execution sequence followed by the runtime environment controller.

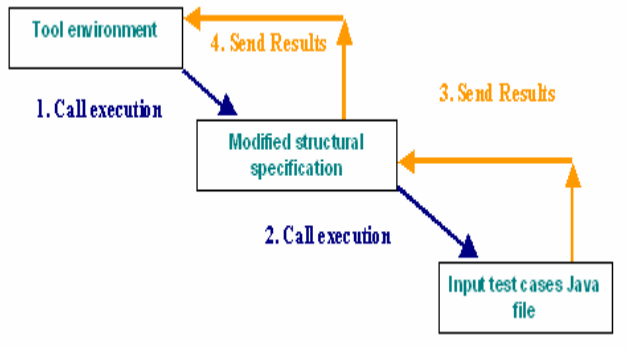

FIGURE 3: EXECUTION SEQUENCE

The Metrics Generator component supports a cumulative test case tracking approach. At any point after testing, the feedback option generates the test cases covered and the percentage of test case completeness, thus helping and guiding students in their test case selection. After each test case execution, the effects of test case inputs entered so far (from the start of testing on the current requirement specification to the current point) for the program are displayed. The "Learn through Feedback" option is implemented as a combination of an accumulator and a comparator. The accumulator sections cumulatively stores the traced assertions generated on each program execution. The comparator section compares the accumulator contents against the internally stored complete assertion list and provides the difference as a percentage of completeness. Since the type of data to be traced is not known to the tool environment, the object data type is used. The object is made to retain its contents across class invocations, thus producing the cumulative effect. Figure 4 shows the implementation of the "Learn through Feedback" option.

One of the metrics calculated is test completeness.The completeness metric is used to check if every possible input combination has been tested. The metric is designed to provide a feedback on what the students tested against what the students should have tested. The metric provides the percentage of flow coverage and also the list of all possible test cases that would have constituted a complete test case set. Programmatically, test case completeness is checked on the basis of the embedded assertions present in the modified structural specification. The concept is to include at least one assertion per requirement and one assertion at every possible program flow path. Presence of all the assertions in the program output indicates that every possible flow path has been covered and also every requirement has been tested. A percentage of the traced assertions to the list of all the assertions in the program are used to provide the test case effectiveness.

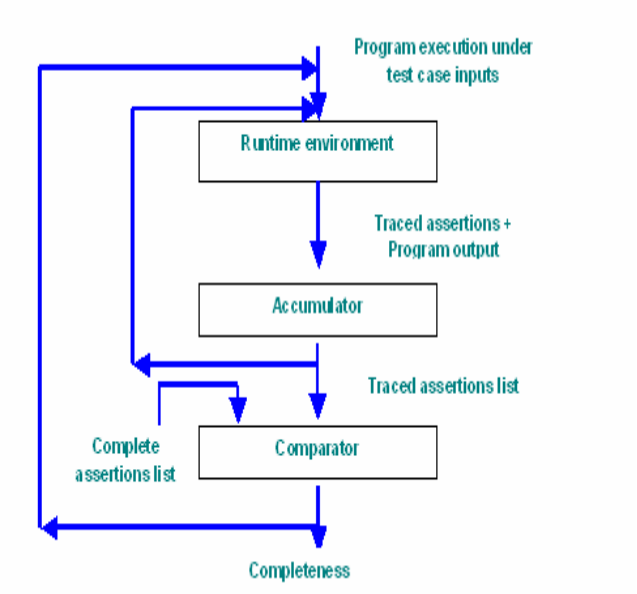

FIGURE 4: FEEDBACK

Another metric calculated is the correctness metric which checks if the output predicted by a user coincides with the actual program output. The metric focuses on the student's understanding of how the program would behave and what the program would generate if the given test of test case inputs is provided. If the student's statement of expected output matches the actual program output then the correctness metric becomes 100 percent. During the test case input collection phase, the expected output entered by the user is internally stored. After execution of the program under the test case input, the results of execution are compared against the stored value and the result of comparison is tracked. The process is repeated for each test case. The cumulative result then becomes the correctness metric.

A fault detection metric is also calculated which checks if the test cases entered by the user are efficient in uncovering program errors or faults. The program faults are indicated by special assertions, which have the keyword "Error" embedded in them. After each program execution under test case inputs, the assertions traced are compared with the fault assertions. If the traced assertion list contains every fault assertion, the tool predicts that all faults have been uncovered with 100 percent effectiveness. The test case number, test case inputs, and the actual output that uncovered the faults are displayed along with the count of the faults embedded in the program.

EXPERIENCE WITH THE TRAINING ENVIRONMENT

To evaluate the effectiveness of the training environment, a case study was performed with a group of students from our first programming class CSE 100. A pre-test was first performed in which the students were given two testing problems, one for equivalence partitioning with boundary value testing and the other for cause effect with boundary value testing. The students developed test cases for the given problems which were evaluated on the basis of completeness and effectiveness. Completeness was defined in terms of the percentage of requirements which were addressed by the tests. Effectiveness was defined to address the value added by each test case.

During the next phase of the case study, the students were taught about testing and the three testing techniques addressed by the investigation (cause-effect, equivalence partitioning and boundary value). A web-based presentation incorporated as a part of the test training environment was utilized. The material included graphic examples to provide a visual representation of the testing mechanism. The modules provided a walk-through of the basics of testing, namely the objectives of testing, the different approaches to testing (black box and white box), and the equivalence partitioning, cause-effect and boundary value testing techniques. The techniques were explained with examples. Any queries could be sent to the web space owner by clicking on the *Contact Us* link. The students took about thirty minutes to complete the tutorial.
During the next phase the participants were introduced on the testing training tool. After a demonstration of the tool was provided and the students familiarized themselves with the tool, assessment on the tool began. Each student was asked to test a sample program. The test cases entered by the student to test the sample program generated tool metrics reflecting the completeness and effectiveness of the tests. The option "Learn Through Feedback" guided the students through the testing process by providing test completeness at various stages of tool training. The feedback option, when selected, provides the following: the count of the relevant test cases entered by the user until that point, the total number of test cases that account for completeness of testing and the percentage of the completeness achieved by the test cases entered by the student. The option "View Execution Result" provided information about the inputs and the assertions traced by them. The students were able to analyze the effect of their inputs on the test case effectiveness metric.

After the training was completed, the students were given two additional testing problems of equivalence partitioning with boundary value and cause effect with boundary value testing to perform a post-test assessment. Once again the test cases were evaluated on the basis of completeness and effectiveness. During the pre-tests, the completeness metric was 61% for equivalence partitioning with boundary value testing and 62% for cause effect testing with boundary value testing, while in the post-tests the completeness metric was 97% and 91% respectively. The effectiveness metric for the pre-tests was 8.3% for equivalence partitioning with boundary value testing and 2.2% for cause effect with boundary value testing, while in the post-tests the metrics were 88.6% and 89.5%, respectively illustrating that the use of redundant test cases by the students was reduced drastically. As a consequence, the case study showed that the test trainer environment helps in teaching and training students in the field of specification based testing.

FUTURE RESEARCH

The training tool is presently designed for students new to the field of testing. The tool would be more effective and would

serve a broader audience if adaptive learning is incorporated. Based on the student's response the tool could choose more stringent metrics and increase the complexity of the problems.

The current automated feedback provided by the tool is based on test case input completeness. If the tool is equipped with a more advanced feedback system for evaluating the quality of tests, it will train students more effectively. The trainer environment can be designed to provide feedback on the effectiveness of testing based on its comparison to the minimal complete test case set.

REFERENCES

[1] W. Hetzel, "A complete guide to software testing", QED. Information Sciences, Inc, 1984.

[2] J. B. Goodenogh, S. L. Gerhart, "Towards a Theory of Test Data Selection", IEEE Transactions. Software Eng., Vol. SE-1, No. 2, pp. 156-173, June 1975.

[3] B. Beizer, "Software testing techniques", Van Nostrand Reinhold, 2nd edition, 1990.

[4] D. S. Rosenblum, "A Practical Approach to Programming with Assertions", IEEE Transactions. Software Eng., Vol. 21, No. 1, pp. 19-31, January 1995.

[5] M. J. Harrold, G. Rothermel, "Performing data flow testing on classes", ACM SIGSOFT Software Engineering Notes, Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software Engineering, Vol. 19 Issue 5, pp. 154-163, December 1994.

[6] E.G Barriocanal, M.A Sicilia Urbán, I.A Cuevas, P.D Pérez, "An experience in integrating automated unit testing practices in an introductory programming course", ACM SIGCSE Bulletin, Vol. 34, Issue 4, December 2002.

[7] W. R. Adrion, M. A. Branstad, J. C. Cherniavsky, "Validation, Verification and Testing of Computer Software", ACM Computing Surveys (CSUR), Vol. 14, Issue 2, June 1982.

[8] Y. Chen, R. L. Probert, D. P. Sims, "Specification-based regression test selection with risk analysis", Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research, September 2002.

[9] R. L. London, "Program verification", Research Directions in Software Technology, Peter Wegner (Ed), The MIT Press, pp. 302-315, 1979.

[10] M. E. Fagon, "Advances in Software Inspections", IEEE Trans. Software Eng., Vol. SE-12, No. 7, pp. 744-751, July 1986.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, MA, 1995.