Cpt S 422 - Software Engineering II

Assignment 7: Black-Box and Regression Testing

Assigned: Wednesday, November 13, 2013
Due: Wednesday, December 4, 2013 by midnight

I. Learner Objectives:

At the conclusion of this assignment, participants should be able to:

- Develop and apply black-box tests
- Implement regression testing

II. Prerequisites:

Before starting this assignment, participants should be able to:

- Construct technical documents
- Compose test cases and test suites for medium size applications
- Analyze a basic set of requirements for implementing and testing a solution to a problem
- Develop fault models from a specific software implementation
- Construct fault models based on a set of basic requirements
- Compose a concise README file
- Complete a detailed defect report according to class standards
- Summarize topics from sets 20, 21, & 22 of class notes including:
  - What is boundary value testing
  - What is equivalence class partitioning
  - What is black-box testing

III. Overview & Requirements:

For this assignment you may work with a partner of your choosing. You are going to perform black-box testing. This method of testing examines functionality of a unit, subsystem, or system as a whole. In particular for this assignment you will be developing tests at the system level.

A large part of black-box testing requires that you understand the problem domain well. I recommend that you run the program first to get acquainted with its behavior. You may find the program at:
http://eecs.wsu.edu/~aofallon/cpts422/assignments/TextSimpletron.zip

Part I:

Do NOT analyze the code for your first set of test cases. Use the requirements only, to generate your test cases. Place your test cases into a file called *validationTestCases.pdf*. You will need to ensure that there is traceability between your tests and the requirements. Aside from your tests cases in this document, also create a ***test matrix*** which lists each requirement as the header for a row in the matrix, and lists the kinds of inputs for each requirement as the header for each column. The matrix is used not only for traceability, but to show the thoroughness of your tests. For example:

	Alphanumeric Characters in Files Names	Special Characters in File Names	Floating Point Instructions in File
Open SML Files?	X	X	N/A
Close SML Files?	X	X	N/A
Support Integral Operations?	N/A	N/A	X

You are performing validation with these tests. The test *execution* from these test cases may be manual.

Simpletron Machine Language (SML) Program Requirements:

1. Must be able to open SML files
    a. SML files contain SML instructions and initial values for variables used with the program
2. Must be able to close SML files
3. Must be able to read SML files
4. Must perform the fetch/decode/execute instruction cycle on SML instructions
5. Must display program and data memory
6. Must display the instruction counter, instruction register, accumulator, operation code, and operand
7. Must allow for single step execution of instructions
8. Must load a SML program into memory
9. Must support the following integral operations:
    no operation, reading, writing, loading, storing, addition, subtraction, division, multiplication, modulus, exponentiation, branching, branching if result is negative, branching if result is positive, branching if result is zero, and halting
    - *no operation* (NO_OP) requires simulation time, but does not perform any operation: opcode 0
    - *read* (READ) gets input from the user through the keyboard and stores the value into data memory: opcode 10
    - *write* (WRITE) displays a value obtained from data memory to the screen: opcode 11

- *load* (LOAD) copies a value from data memory and places it into the accumulator: opcode 20
- *store* (STORE) copies the value in the accumulator and places it into data memory: opcode 21
- *addition* (ADD) computes the sum between two values: accumulator ← accumulator + data memory value: opcode 30
- *subtraction* (SUBTRACT) computes the difference between two values: accumulator ← accumulator – data memory value: opcode 31
- *division* (DIVIDE) computes the quotient between two values: accumulator ← accumulator / data memory value: opcode 32
- *multiplication* (MULTIPLY) computes the product between two values: accumulator ← accumulator x data memory value: opcode 33
- *modulus* (MOD) computes the remainder between two values: accumulator ← accumulator % data memory value: opcode 34
- *exponentiation* (EXP) computes the exponentiation between two values: accumulator ← accumulator ^ data memory value: opcode 35
- *unconditional branch* (BRANCH) causes the program to change flow: instruction counter ← data memory location: opcode 40
- *branch negative* (BRANCHNEG) causes the program to change flow if the accumulator < 0: instruction counter ← data memory location: opcode 41
- *branch zero* (BRANCHZERO) causes the program to change flow if the accumulator = 0: instruction counter ← data memory location: opcode 42
- *halt* (HALT) exits the program: opcode 43
- *branch positive* (BRANCHPOS) causes the program to change flow if the accumulator > 0: instruction counter ← data memory location: opcode 44

10. Must support user input through a keyboard
11. Must exit on an invalid operation and report the invalid opcode to a file called invalidOpcode.log

In a file called *testResults.pdf* report your findings for the following questions. Are any of your tests inconclusive? Do any of the tests fail? If so, do these failed tests detect failures that cause the program to become inoperable? Are some of the requirements ambiguous or too high level such that deriving test cases from them was difficult?

Part II:

Black-box testing is also a great way to get acquainted with a project. Now that you have some understanding of the project, fix the defects found with your previous tests.

Next, *modify* the program so that instructions may have access to data memory reaching 100 megabytes. Also, we have added new requirements to the project. The simulator must be able to perform shift left and shift right operations. Implement the

code for these new requirements and develop some new black-box tests. Place these additional tests in the *validationTestCases.pdf* document.

The test cases that original passed in the previous build may now be referred to as ***regression test cases*** forming a regression test suite. Rerun all tests for regression testing. Do your tests still pass? List your results in the same *testResults.pdf*. If your tests fail, fix any defects in the program.

Part III:

What kinds of ***acceptance tests*** could you perform? Acceptance tests include performance tests. Do you have any concerns about performance of the simulator as more instructions and memory is supported? Please list your responses to these questions in a file called *acceptanceTests.pdf*.

IV. Submitting Assignments:

1. Using the Angel tool https://lms.wsu.edu submit your assignment to your TA. You will "drop" your solution into the provided "Homework Submissions" Drop Box under the "Lessons" tab. You must upload your solutions as <your last name>_assignment7.zip by the due date and time.
2. Your .zip file should contain the following items:
    1. Simpletron source code with modifications and corrections
    2. *validationTestCases.pdf*
    3. *testResults.pdf*
    4. *acceptanceTests.pdf*

V. Grading Guidelines:

This assignment is worth 150 points. Your assignment will be evaluated based on adherence to the requirements. We will grade according to the following criteria:

- (40 pts) Simpletron source code with modifications and corrections
- (60 pts) *validationTestCases.pdf* – including test cases and test matrices developed for Part I and II
- (30 pts) *testResults.pdf* – including results from tests executed in Parts I and II
- (20 pts) *acceptanceTests.pdf* – including your conclusions about acceptance tests for this project