CS 6340 – Fall 2009 – Problem Set 6 **Name**_____

Assigned: October 7, 2009
Due: October 15, 2009

At the beginning of class on the due date, submit your neatly presented solution with this page stapled to the front (100 pts). As with the previous assignments, you are to do this assignment individually.

Part 1

Your new position as Test Manager requires that you establish a set of requirements that developers will use for unit testing of the software that they write. Before you establish these requirements, you want to assess the fault-detection ability, expense, tool availability, etc. of various techniques that have been proposed in the literature. To do this, you will use a program, which we'll call **tritype**, that has the following requirements specification

> **tritype** takes as input three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

You are to do the following:

1. Use the specification to develop a set of test cases (a test suite) for **tritype** using two black box testing methods (both described at the end of this document):
   - Equivalence Partitioning
   - Boundary Value Analysis

2. Create a file of test cases (reason for test (in quotes), inputs, expected outputs) that consists of one test case per line; the number of the test case will be the line number in the file. For example, suppose I created two test cases:
   Test Case 1: isosceles 2 2 3 isosceles
   Test Case 2: equilateral 4 4 4 equilateral
   The file should contain
   "isosceles" 2 2 3 isosceles
   "equilateral" 4 4 4 equilateral

3. Send the test cases to Raul (raul@cc.gatech.edu), and he'll send you the **tritype** program for the second part of the assignment. Let him know whether you want the C version or the Java version.

Part 2

Using your copy of the implementation of **tritype** (either in C or in Java), perform the following activities:

1. Measure the *Statement Adequacy* of the test suite you developed in Part 1.
2. Measure the *Multiple Condition Adequacy* of the test suite you developed in Part 1.
3. Create the control-flow graph for **tritype**
4. Using the control flow graph for **tritype**, compute the cyclomatic complexity of **tritype**.
5. Measure *Basis Path Adequacy* for the test suite you developed in Part 1.

For any of 1,2,4, or 5, you can (a) perform the tasks manually, or (b) automate the process by writing a program to perform the task or by finding an existing tool that will do it for you.

At the beginning of class on the due date, submit the following:

- Your test suite (typed).
- Statement Adequacy: a list of the statements and, for each statement, an indication of whether the statement was covered; a statement of the coverage achieved by your test suite.
- Multiple Condition Adequacy: a list of the conditions required for multiple condition coverage and, for each condition, an indication of whether the condition was covered; a statement of the coverage achieved by your test suite.
- The control-flow graph for **tritype.c**.
- Basis Path Adequacy: The cyclomatic complexity, along with an explanation of how you got it; a statement of the coverage achieved by your test suite.
- A discussion of the results of your experiment (e.g., difficulty of measuring adequacy, expense of measuring adequacy, inadequacy of your initial test suite, ability of the various techniques to help uncover errors).
- A decision about what requirement(s) you will establish for your developers, and why you made this decision. To help with your decision and discussion, you can select a type of software that your company develops.

Y *is required to compute* **Z**

Therefore, a transitive relationship has been established between **X** and **Z**:

X *is required to compute* **Z**

Based on this transitive relationship, tests to find errors in the calculation of **Z** must consider a variety of values for both **X** and **Y**.

The *symmetry* of a relationship (graph link) is also an important guide to the design of test cases. If a link is indeed bidirectional (symmetric), it is important to test this feature. The UNDO feature [BEI95] in many personal computer applications implements limited symmetry. That is, UNDO allows an action to be negated after it has been completed. This should be thoroughly tested and all exceptions (i.e., places where UNDO cannot be used) should be noted. Finally, every node in the graph should have a relationship that leads back to itself; in essence, a "no action" or "null action" loop. These *reflexive* relationships should also be tested.

As test case design begins, the first objective is to achieve *node coverage*. By this we mean that tests should be designed to demonstrate that no nodes have been inadvertently omitted and that node weights (object attributes) are correct.

Next, *link coverage* is addressed. Each relationship is tested based on its properties. For example, a symmetric relationship is tested to demonstrate that it is, in fact, bidirectional. A transitive relationship is tested to demonstrate that transitivity is present. A reflexive relationship is tested to ensure that a null loop is present. When link weights have been specified, tests are devised to demonstrate that these weights are valid. Finally, loop testing is invoked (Section 16.5.3).

16.6.2 Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an *input condition*. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [BEI95]. An *equivalence class* represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a *range*, one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific *value,* one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a *set,* one valid and one invalid equivalence class are defined.
4. If an input condition is *Boolean,* one valid and one invalid class are defined.

As an example, consider data maintained as part of an automated banking application. The user can "dial" the bank using his or her personal computer, provide a six digit password, and follow with a series of keyword commands that trigger various banking functions. The software supplied for the banking application accepts data in the form:

area code—blank or three digit number
prefix—three digit number not beginning with 0 or 1
suffix—four digit number
password—six digit alphanumeric value
commands—"check," "deposit," "bill pay," etc.

The input conditions associated with each data element for the banking application can be specified as:

area code:	input condition, *Boolean*—the area code may or may not be present input condition, *range*—values defined between 200 and 999, with specific exceptions
prefix:	input condition, *range*—specified value > 200 with no 0 digits
suffix:	input condition, *value*—four digit length
password:	input condition, *Boolean*—a password may or may not be present input condition, *value*—six character string
command:	input condition, *set*—containing commands noted above

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item could be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

16.6.3 Boundary Value Analysis

For reasons that are not completely clear, a greater number of errors tend to occur at the boundaries of the input domain than in the "center." It is for this reason that *boundary value analysis (BVA)* has been developed as a testing

technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [MYE79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a *range* bounded by values a and b, test cases should be designed with values a and b, just above and just below a and b, respectively.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Guidelines 1 and 2 are applied to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying the guidelines noted above, boundary testing will be more complete, thereby having a higher likelihood for error detection.

16.6.4 Comparison Testing

There are some situations (e.g., aircraft avionics, nuclear power plant control) in which the reliability of software is absolutely critical. In such applications redundant hardware *and* software are often used to minimize the possibility of error. When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification. In such situations, each version can be tested with the same test data to ensure that all provide identical output. Then all versions are executed in parallel with real-time comparison of results to ensure consistency.

Using lessons learned from redundant systems, researchers (e.g., [BRI87]) have suggested that independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer-based system. These independent versions form the basis of a black-box testing technique called *comparison testing* or *back-to-back testing* [KNI89].

When multiple implementations of the same specification have been produced, test cases designed using other black-box techniques (e.g., equivalence partitioning) are provided as input to each version of the software. If the out-