Workbook

Software Testing Strategies & Techniques
(SE – 481)

Name _____

Roll No _____

Batch _____

Year _____

Department _____

Workbook

Software Testing Strategies & Techniques (SE – 481)

Prepared by

Dr. Sh. M. Wahabuddin Usmani

Approved by

Chairman
Department of Computer Science & Information Technology

Table of Contents

PRACTICAL No: 01

TEST CASE

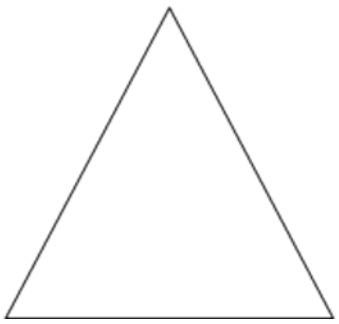
Not the same thing as test data

Test Cases require:

- input and output specifications
- statement of the function under test
- mapping to requirements

Example: A program that determines whether a triangle is isosceles:

- **function Is_Isosceles**
  (Side 1, Side 2, Side 3: in integer)
- **return boolean;**



Required:

A test case document of IEEE is given below. The students are required to fill it for the above example case. Students must specify the logic for the Test Cases they include or exclude.

Project Name
Test Plan

Table of Contents

History of Changes – VERY important to track history of changes to document, who made the changes, and why.

Version	Date	Change
First Draft		

Purpose

The purpose of these test cases is to verify changes to

Related Documents

Requirements Document and location (path)

Test Team

List Test Team members and phone numbers

Test Case Abbreviations

Req	Requirement that the test cases are validating (number / identifier)
Case	Test Case Number / Identifier
Action	Action to perform
Expected Result	Result expected when action is complete
P / F	Pass / Fail indicator. Checkmark = Pass. “F” = Fail
Notes	Additional notes, error messages, or other information about the test.

Items Not Covered by These Test Cases

List any functions or processes not being tested and why.

Bug Tracking

The “Insert Name Here” Database will be used to track defects found while performing the test cases. All defects will be logged as they are discovered. Defects will be assigned to Person A to fix, or to Person B to investigate.

Quality Control

The completed test cases will be reviewed to ensure that all cases were run; that all were completed successfully; and that any deviations from the test cases were noted accordingly. Each step should be marked as Passed or Failed. Failed cases should be marked with the date and time of the failure, and the associated test track number. When the failed cases is fixed, the date and time of the retest should be noted.

ROLE (Security Role)

Function (Section of Software being Tested)

Case	Action	Expected Result	P / F	Notes
Number				

This table is the most important section. In general, for the final turn-in, you should have requirements (system-level) tests and clearly show all outstanding defects.

PRACTICAL No: 02

Fault Injection Method

Below are described some of the newer techniques in the software testing field. Fault based methods include Error Based Testing, Fault seeding, mutation testing, and fault injection, among others. After briefly describing each of the 4 techniques, fault injection will be discussed in more detail.

a) ***Error based testing*** defines classes of errors as well as inputs that will reveal any error of a particular class, if it exists.
b) ***Fault seeding*** implies the injection of faults into software prior to test. Based on the number of these artificial faults discovered during testing, inferences are made on the number of remaining 'real' faults For this to be valid the seeded faults must be assumed similar to the real faults.
c) ***Mutation testing*** injects faults into code to determine optimal test inputs.
d) ***Fault Injection*** evaluates the impact of changing the code or state of an executing program on behavior of the software

These methods attempt to address the belief that current techniques for assessing software quality are not adequate, particularly in the case of mission critical systems. It is suggested that the traditional belief that improving and documenting the software development process will increase software quality is lacking. Yet, they recognize that the amount of testing (which is product focused) required in order to demonstrate high reliability is impractical. In short, quality processes cannot demonstrate reliability and the testing necessary to do so is impossible to perform.

Fault injection is not a new concept. Hardware design techniques have long used inserted fault conditions to test system behavior. It is as simple as pulling the modem out of your PC during use and observing the results to determine if they are safe and/or desired. The injection of faults into software is not so widespread, though it would appear that companies such as Hughes Information Systems, Microsoft, and Hughes Electronics have applied the techniques or are considering them. Properly used, fault insertion can give insight as to where testing should be concentrated, how much testing should be done, whether or not systems are fail-safe, etc

As a simple example consider the following code:

Original	**Fault Injected**
X = (r1 – 2) + (s2 – s1) Y = z – 1 … T = x/y	X = (r1 – 2) + (s2 – s1) **X = perturb(x)** Y = z – 1 … T = x/y If T > 100 then print ('WARNING')

In this case it is catastrophic if T > 100. By using perturb(x) to generate changed values of X (i.e. a random number generator) you can quickly determine how often corrupted values of X lead to undesired values of T.
The technique can be applied to internal source code, as well as to 3rd party software, which may be a "black box"

Required:

Students are required to take a function of their project assigned to them by their programming teacher and apply Fault Injection Method on the code and evaluate it.

It is also required that logic for selecting a particular function/statement for it be explained.

PRACTICAL No: 03

Error based Testing

Required: Apply error based testing method on a sample code and explain classes of errors

PRACTICAL No: 04

Fault Seeding Testing

Required: Apply Fault Seeding testing method on a sample code and explain the logic of seeding faults.

PRACTICAL No: 05

Mutation Testing

Required: Apply Mutation Testing method on a sample code and explain the logic of testing.

PRACTICAL No: 06

Testing GUIs

Graphical user interfaces (GUIs) present interesting challenges for software engineers. Because of reusable components provided as part of GUI development environments, the creation of the user interface has become less time consuming and more precise. But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.

Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI. Due to the large number of permutations associated with GUI operations, testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years.

Required: Study different automated GUI Testing tools and explain any one.

PRACTICAL No: 07

Syntax Testing

Introduction

This black box technique is based upon an analysis of the specification of the component to model its behaviour by means of a description of the input via its syntax. We illustrate the technique by means of a worked example. The technique is only effective to the extent that the syntax as defined corresponds to the required syntax.

Example

Consider a component that simply checks whether an input *float_in* conforms to the syntax of a floating point number, float (defined below). The component outputs *check_res*, which takes the form 'valid' or 'invalid' dependent on the result of its check.

Here is a representation of the syntax for the floating point number, float in Backus Naur Form (BNF) :

```
float = int "e" int.
int   = ["+"|"-"] nat.
nat   = {dig}.
dig   = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
```

Terminals are shown in quotation marks; these are the most elementary parts of the syntax - the actual characters that make up the input to the component. | separates alternatives. [] surrounds an optional item, that is, one for which nothing is an alternative. {} surrounds an item which may be iterated one or more times.

Test Cases with Valid Syntax

The first step is to derive the options from the syntax. Each option is followed by a label, in the form [opt_1], [opt_2], etc., to enable it to be identified later.

float has no options.
int has three options: nat [opt_1], "+" nat [opt_2] and "-" nat [opt_3].
nat has two options: a single digit number [opt_4] and a multiple digit number [opt_5].
dig has ten options: one for each digit [opt_6 to opt_15].

There are thus fifteen options to be covered. The next step is to construct test cases to cover the options.

The following test cases cover them all:

test case	float_in	option(s) executed	check_res
1	3e2	opt_1	'valid'
2	+2e+5	opt_2	'valid'
3	-6e-7	opt_3	'valid'
4	6e-2	opt_4	'valid'
5	1234567890e3	opt_5	'valid'
6	0e0	opt_6	'valid'
7	1e1	opt_7	'valid'
8	2e2	opt_8	'valid'
9	3e3	opt_9	'valid'
10	4e4	opt_10	'valid'
11	5e5	opt_11	'valid'
12	6e6	opt_12	'valid'
13	7e7	opt_13	'valid'
14	8e8	opt_14	'valid'
15	9e9	opt_15	'valid'

This is by no means a minimal test set to exercise the 15 options (it can be reduced to just three test cases, for example, 2, 3 and 5 above), and some test cases will exercise more options than the single one listed in the 'options executed' column. Each option has been treated separately here to aid understanding of their derivation. This approach may also contribute to the ease with which the causes of faults are located.

Test Cases with Invalid Syntax

The first step is to construct a checklist of generic mutations. A possible checklist is:

m1. introduce an invalid value for an element;

m2. substitute an element with another defined element;

m3. miss out a defined element;

m4. add an extra element.

These generic mutations are applied to the individual elements of the syntax to yield specific mutations. The elements, represented in the form el_1, el_2, etc., of the syntax for float can be identified from the BNF representation as shown below:

```
float  =  int "e" int.                          el_1   =  el_2 el_3 el_4.
int    =  ["+"|"-"] nat.                        el_5   =  el_6 el_7.
nat    =  {dig}.                                el_8   =  el_9.
dig    =  "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".  el_10  =  el_11.
```

["+"|"-"] has been treated as a single element because the mutation of optional items separately does not create test cases with invalid syntax (using these generic mutations).

The next step is to construct test cases to cover the mutations:

test case	float_in	mutation	element	check_res
1	xe0	m1	x for el_2	'invalid'
2	0x0	m1	x for el_3	'invalid'
3	0ex	m1	x for el_4	'invalid'
4	x0e0	m1	x for el_6	'invalid'
5	+xe0	m1	x for el_7	'invalid'
6	ee0	m2	el_3 for el_2	'invalid'
7	+e0	m2	el_6 for el_2	'invalid'
8	000	m2	el_2 for el_3	'invalid'
9	0+0	m2	el_6 for el_3	'invalid'
10	0ee	m2	el_3 for el_4	'invalid'
11	0e+	m2	el_6 for el_4	'invalid'
12	e0e0	m2	el_3 for el_6	'invalid'
13	+ee0	m2	el_3 for el_7	'invalid'
14	++e0	m2	el_6 for el_7	'invalid'
15	e0	m3	el_2	'invalid'
16	00	m3	el_3	'invalid'
17	0e	m3	el_4	'invalid'
18	y0e0	m4	y in el_1	'invalid'
19	0ye0	m4	y in el_1	'invalid'
20	0ey0	m4	y in el_1	'invalid'
21	0e0y	m4	y in el_1	'invalid'
22	y+0e0	m4	y in el_5	'invalid'
23	+y0e0	m4	y in el_5	'invalid'
24	+0yeo	m4	y in el_5	'invalid'

Some of the mutations are indistinguishable from correctly formed expansions and these have been discarded. For example, the generic mutation m2 (el_2 for el_4) generates correct syntax as m2 is "substitute an element with another defined element" and el_2 and el_4 are the same (int).

Some of the remaining mutations are indistinguishable from each other and these are covered by a single test case. For example, applying the generic mutation m1 ("introduce an invalid value for an element") by replacing el_4, which should be an integer, with "+" creates the form "0e+". This is the same input as generated for test case 11 above.

Many more test cases can be created by making different choices when using single mutations, or combining mutations.

Syntax Test Coverage

No test coverage measures are defined for syntax testing. Any measures of coverage would be based on the rules for generating valid syntax options and the checklist for generating test cases with invalid syntax. Neither the rules nor the checklist are definitive and so any syntax test coverage measures based on a particular set will be specific to that set of rules and checklist

Required: Considering BNF Example for phone numbers

special_digit ::= 0 | 1 | 2 | 5
other_digit ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
ordinary_digit ::= special_digit | other_digit
exchange_part ::= ordinary_digit
number_part ::=
phone_number ::= exchange_part number_part
*other _ digit*2
*ordinary _ digit*4

• Correct phone numbers:
– 3469900, 9904567, 3300000
• Incorrect phone numbers:
– 0551212, 123, 8, ABCDEFG

Write down test cases for correct and incorrect telephone numbers and also explain the basic testing strategy

PRACTICAL No: 08

Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E1 <relational-operator> *E2*

where *E1* and *E2* are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥.

A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬). A condition without relational expressions is referred to as a *Boolean expression.*

Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error.

The condition testing method focuses on testing each condition in the program. Condition testing strategies (discussed later in this section) generally have two advantages. First, measurement of test coverage of a condition is simple. Second, the test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program *P* is effective for detecting errors in the conditions contained in *P*, it is likely that this test set is also effective for detecting other errors in *P*. In addition, if a testing strategy is effective for detecting errors in a condition, then it is likely that this strategy will also be effective for detecting errors in a program.

A number of condition testing strategies have been proposed. *Branch testing* is probably the simplest condition testing strategy. For a compound condition *C*, the true and false branches of *C* and every simple condition in *C* need to be executed at least once.

Domain testing requires three or four tests to be derived for a relational expression. For a relational expression of the form

E1 <relational-operator> *E2*

three tests are required to make the value of *E1* greater than, equal to, or less than that of *E2* [HOW82]. If <relational-operator> is incorrect and *E1* and *E2* are correct, then these three tests guarantee the detection of the relational operator error. To detect errors in *E1* and *E2*, a test that makes the value of *E1* greater or less than that of *E2* should make the difference between these two values as small as possible.

For a Boolean expression with *n* variables, all of 2*n* possible tests are required ($n > 0$). This strategy can detect Boolean operator, variable, and parenthesis errors, but it is practical only if *n* is small.

Error-sensitive tests for Boolean expressions can also be derived. For a singular Boolean expression (a Boolean expression in which each Boolean variable occurs only once) with *n* Boolean variables ($n > 0$), we can easily generate a test set with less than 2*n* tests such that this test set guarantees the detection of multiple Boolean operator errors and is also effective for detecting other errors.

Tai suggests a condition testing strategy that builds on the techniques just outlined. Called *BRO* (branch and relational operator) testing, the technique guarantees the detection of branch and relational operator errors in a condition provided that all Boolean variables and relational operators in the condition occur only once and have no common variables. The BRO strategy uses condition constraints for a condition *C*. A condition constraint for *C* with *n* simple conditions is defined as (*D1, D2, . . ., Dn*), where *Di* ($0 < i \leq n$) is a symbol specifying a constraint on the outcome of the *i*th simple condition in condition *C*. A condition constraint *D* for condition *C* is said to be covered by an execution of *C* if, during this execution of *C*, the outcome of each simple condition in *C* satisfies the corresponding constraint in *D*.

For a Boolean variable, *B*, we specify a constraint on the outcome of *B* that states that *B* must be either true (t) or false (f). Similarly, for a relational expression, the symbols >, =, < are used to specify constraints on the outcome of the expression.

As an example, consider the condition

C1: *B1* & *B2*

where *B1* and *B2* are Boolean variables. The condition constraint for *C1* is of the form (*D1*, *D2*), where each of *D1* and *D2* is t or f. The value (t, f) is a condition constraint for *C1* and is covered by the test that makes the value of *B1* to be true and the value of *B2* to be false. The BRO testing strategy requires that the constraint set {(t, t), (f, t), (t, f)} be covered by the executions of *C1*. If *C1* is incorrect due to one or more Boolean operator errors, at least one of the constraint set will force *C1* to fail.

As a second example, a condition of the form

C2: *B1* & (*E3* = *E4*)

where *B1* is a Boolean expression and *E3* and *E4* are arithmetic expressions. A condition constraint for *C2* is of the form (*D1, D2*), where each of *D1* is t or f and *D2* is >, =, <. Since *C2* is the same as *C1* except that the second simple condition in *C2* is a relational expression, we can construct a constraint set for *C2* by modifying the constraint set {(t, t), (f, t), (t, f)} defined for *C1*. Note that t for (*E3* = *E4*) implies = and that f for (*E3* = *E4*) implies either < or >. By replacing (t, t) and (f, t) with (t, =) and (f, =), respectively, and by replacing (t, f) with (t, <) and (t, >), the resulting constraint set for *C2* is {(t, =), (f, =), (t, <), (t, >)}. Coverage of the preceding constraint set will guarantee detection of Boolean and relational operator errors in *C2*.

As a third example, we consider a condition of the form

C3: (*E1* > *E2*) & (*E3* = *E4*)

where *E1*, *E2*, *E3,* and *E4* are arithmetic expressions. A condition constraint for *C3* is of the form (*D1, D2*), where each of *D1* and *D2* is >, =, <. Since *C3* is the same as *C2* except that the first simple condition in *C3* is a relational expression, we can construct a constraint set for *C3* by modifying the constraint set for *C2*, obtaining {(>, =), (=, =), (<, =), (>, >), (>, <)} Coverage of this constraint set will guarantee detection of relational operator errors in *C3.*

Required: Write down a program to accept 03 numbers and print the largest, middle and smallest amongst them. Apply condition testing approach on the program. Write test cases as per the stated techniques above.