Internal Assessment Test 1 – March 2018

Scheme and Solutions

Sub:	Software Testing				Sub Code:	10CS842	Branch:	CSE
Date:	14-03-2018	Duration:	90 min's	Max Marks: 50	Sem / Sec:	CSE(8A,B & C)		OBE

1. (a) Why do we need to test software? Describe what a typical test case information should include. [05]

Scheme:
Testing definition – [02]

Solution:
There are two main reasons: to make a judgment about quality or acceptability and to discover problems. We test because we know that we are fallible **(capable of making mistakes or being wrong, capable of making mistakes or being wrong)** — this is especially true in the domain of software and software controlled systems.

Test Case structure [03]

Solution:
Test Cases

The essence of software testing is to determine a set of test cases for the item being tested. Before going on, we need to clarify what information should be in a test case.

The most obvious information is

Inputs: Inputs are really of two types:
1. Pre-conditions (circumstances that held prior to testing case execution)
2. Actual inputs that were identified by some testing method.

Outputs: again, there are two types:
1. Postconditions
2. Actual outputs.

Test case ID			
Purpose			
Preconditions			
Inputs			
Expected Outputs			
Post Conditions			
Execution History			
Date	Result	Version	Run By

Test Case template

The output portion of a test case is frequently overlooked because this is often the hard part. Suppose, for example, you were testing software that determined an optimal route for an aircraft, given certain FAA air corridor constraints and the weather data for a flight day. **How would you know what the optimal route really is?**

Responses to this problem.

- The academic response is to postulate the existence of an oracle, who "knows all the answers".
- Industrial response to this problem is known as Reference Testing, where the system is tested in the presence of expert users, and these experts make judgments as to whether or not outputs of an executed set of test case inputs are acceptable.

The act of testing entails establishing the necessary pre-conditions, providing the test case inputs, observing the outputs, and then comparing these with the expected outputs to determine whether or not the test passed.

(b) Define Error, fault, failure, Incident, Test, and Test case. Distinguish between bug and an Error. [05]

Scheme:
Error, Fault, Failure, Incident, Test, and Test case [1/2 x 6 = 3]
Differentiation: [02]

Solution:
Error: People make errors. A good synonym is "mistake". When people make mistakes while coding, we call these mistakes "bugs".
Errors tend to **propagate**; a requirements error may be magnified during design and amplified still more during coding.
Type:

1. **Error of Commission**
2. **Error of Omission**

Fault: A fault is the **result of an error**. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, hierarchy charts, source code, and so on. **"Defect"** is a good synonym for a fault; so is "bug". Faults can be elusive.
The fault of Omission: the Resulting fault is that something is missing that should be present in the representation or when we fail to enter the correct information. Faults of omission are more difficult to detect and resolve.
The fault of commission: occurs when we enter something into a representation that is incorrect.

Incident: An incident is a symptom(s) associated with a failure that alerts the user to the occurrence of a failure.

Test: A test is an act of exercising software with test cases. Testing is obviously concerned with errors, faults, failures, and incidents. There are two distinct goals of a test: either to find failures or to demonstrate correct execution.

Test Case: A test case has an identity, and is associated with a program behavior. A test case has a set of inputs, a set of expected outputs.

2. (a) Briefly explain Testing using Venn Diagram. [05]

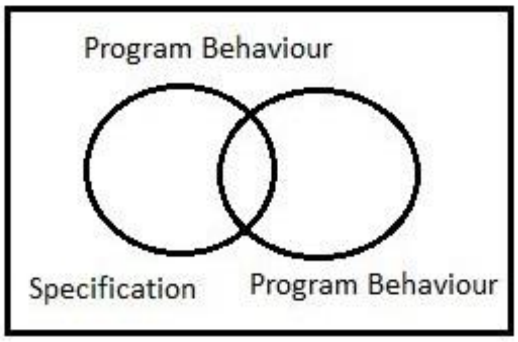
Scheme:
Diagram: [02]
Venn diagram explanation [03]

Solution:

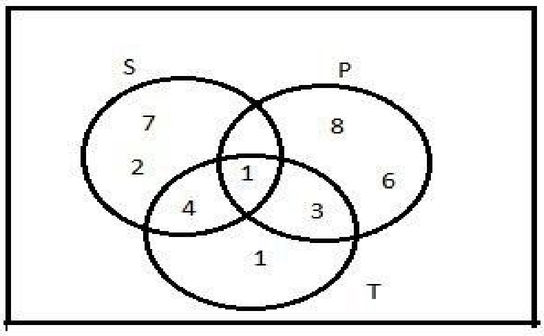
Testing is fundamentally concerned with behavior, and behavior is orthogonal to the structural view common to software (and system) developers.

A quick differentiation is that the structural view focuses on “what it is” and the behavioral view considers “what it does”. One of the continuing sources of difficulty for testers is that the base documents are usually written by and for developers, and therefore the emphasis is on structural, rather than behavioral, information. In this section, we develop a simple Venn diagram which clarifies several nagging questions about testing.

The figure shows the relationship between our universe of discourse and the specified and programmed behaviors.
Of all the possible program behaviors, the specified ones are in the circle labeled S; and all those behaviors actually programmed (note the slight difference between P and U, the Universe) are in P. **With this diagram, we can see more clearly the problems that confront a tester.**



What if there are specified behaviors that have not been programmed? In our earlier terminology, **these are faults of omission.**
Similarly, what if there are programmed (implemented) behaviors that have not been specified? These correspond to **faults of the commission**, and to errors which occurred after the specification was complete. **The intersection of S and P (the football-shaped region) is the “correct” portion, that is behaviors that are both specified and implemented.**



Specified, Implemented, and Tested Behaviors

S of specified behaviors, P of programmed behaviors and the set **T of Testing Behavior**.

We are already at a point where we can see some possibilities for testing as a craft: **what can a tester do to make the region where these sets all intersect (region 1) be as large as possible? Another way to get at this is to ask how the test cases in the set T are identified.** The short answer is that test cases are identified by a testing method. This Framework gives us a way to compare the effectiveness of diverse testing methods.

(b) Explain: i) Currency converter ii) Saturn wind shield wiper controller. [05]

Scheme:
Currency Converter [2.5]
Venn diagram explanation [2.5]

3. How to Identify Test Cases? Explain Functional Testing and Structural Testing and how they differ from each other. [10]

Scheme:
Test Case Identification basics [02]
Explain Functional and Structural Testing [4+4=8]

Solution:

Identifying Test Cases

There are two fundamental approaches to identifying test cases

Functional Testing: Functional testing is based on the view that any program can be considered to be a function that maps values from its input domain to values in its output range. This notion is commonly used in engineering when systems are considered to be "black boxes".
This leads to the term Black Box Testing, in which the content(implementation) of a black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs.

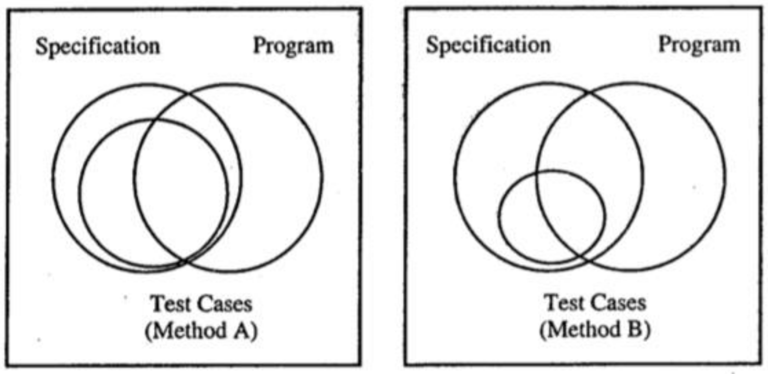
With the functional approach to test case identification, the only information that is used is the specification of the software.
There are two distinct advantages of functional test cases:

1. They are independent of how the software is implemented, so if the implementation changes, the test cases are still useful,
2. Test case development can occur in parallel with the implementation, thereby reducing overall project development interval.

Disadvantages:

1. There can be significant redundancies among test cases, and this is compounded by the possibility of gaps of untested software.

The figure shows the results of test cases identified by two functional methods. Method A identifies a larger set of test cases than does Method B. Notice that, for both methods, the set of test cases is completely contained within the set of specified behavior.

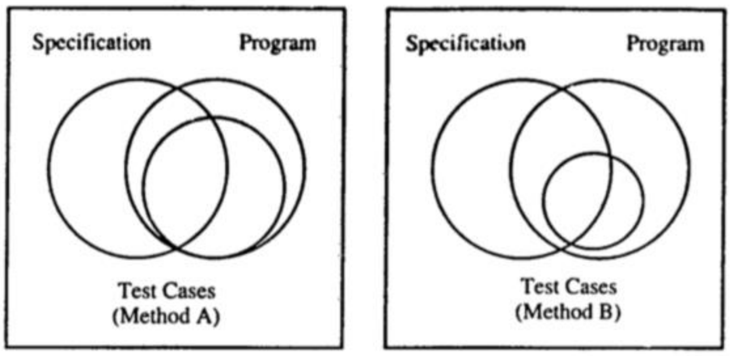


2. Since functional methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified.

1.4.2 Structural Testing

Structural testing is the other fundamental approach to test case identification. To contrast it with Functional Testing, it is sometimes called White Box (or even Clear Box) Testing.

The clear box metaphor is probably more appropriate because the essential difference is that the implementation (of the Black Box) is known and used to identify test cases. Being able to "see inside" the black box allows the tester to identify test cases based on how the function is actually implemented. Structural Testing has been the subject of some fairly strong theory. With these concepts, the tester can rigorously describe exactly what is being tested. Because of its strong theoretical basis, structural testing lends itself to the definition and use of test coverage metrics.



Notice that for both methods, the set of test cases is completely contained within the set of programmed behavior. Because structural methods are based on the program.

The Functional Versus Structural Debate Given two fundamentally different approaches to test case identification, the natural question is which is better?
Conclusion: Neither approach alone is sufficient. Both approaches are needed.

4. (a) What is boundary Value Analysis? Explain the Input domain of a function variables. [06]

Scheme:

Definition: Boundary Value testing [02]
BVT Input domain explanation [04]

Solution:

- Boundary value analysis is the best known **functional testing technique**.
- The objective of functional testing is to use knowledge of the functional nature of a program to identify test cases.
- Functional testing has focused on the input domain, but it is a good supplement to consider test cases based on the range as well.
- Boundary value analysis focuses on the boundary of the input space to identify test cases.
- The logic behind boundary value analysis is that errors tend to occur near the extreme values of an input variable.
- Programs written in non-strongly typed languages are more appropriate candidates for boundary value testing.

Input domain of function of variables

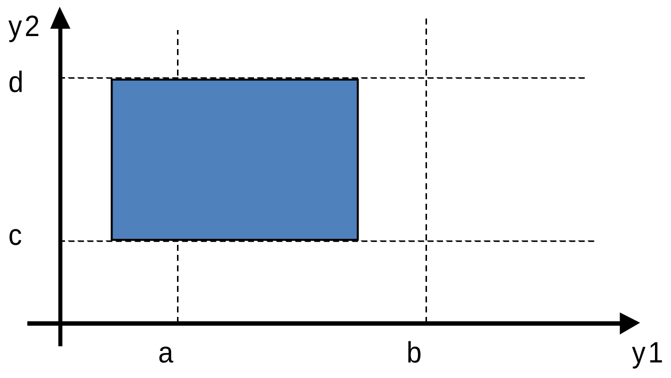
In our discussion we will assume a program P accepting two inputs y_1 and y_2 such that:

$$a \le y_1 \le b \text{ and } c \le y_2 \le d$$

Consider the following function:

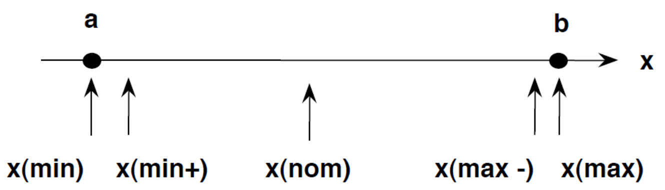
$$f(y_1, y_2), \text{ where } \quad a \le y_1 \le b \,, c \le y_2 \le d$$

Note: Boundary inequalities of *n* input variables define an *n*-dimensional input space:



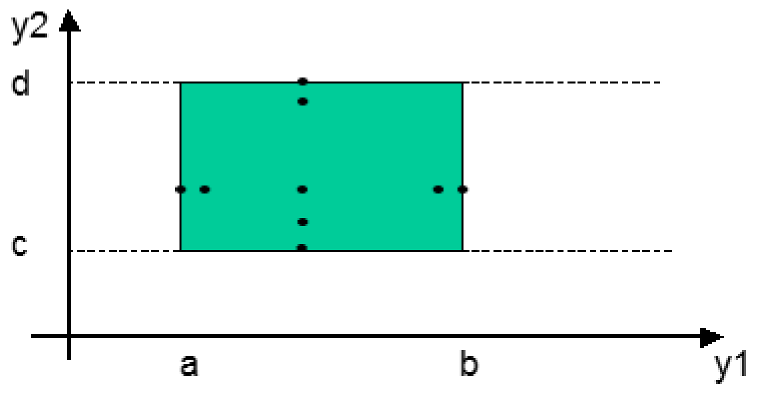
The basic idea in boundary value analysis is to select input variable values at their:

- Minimum (Min)
- Just above the minimum (Min+1)
- A nominal value (Nom)
- Just below the maximum (Max-1)
- Maximum (Max)



Boundary Value Analysis (BVA) for Program P

$$T = \{ <y_{1nom}, y_{2min}>, <y_{1nom}, y_{2min+}>, <y_{1nom}, y_{2nom}>, <y_{1nom}, y_{2max-}>, <y_{1nom}, y_{2max+}>, <y_{1min}, y_{2nom}>, <\ _{1nin+}, y_{2nom}>, <y_{1max-}, y_{2nom}>, <y_{1max}, y_{2nom}> \}$$



BVA for triangle problem is taken as 3 variable are in the range such that 1<= {a, b, c} <= 200.
Hence min = 1, min+1 = 2, nom = 100 (any value between 1 to 200) max-1 = 199, and max = 200. Below table shows the resultant **boundary value test cases**.

Table 1 Boundary Value Analysis Test Cases **Case**

	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles

(b) Write a short note on Limitations of Boundary value Analysis. [04]

Scheme:
One Limitation x 01 = [04]

Solution:

- Boundary value analysis works well when the program to be tested is a function of several independent variables that represent bounded physical quantities.

- A quick look at the boundary value analysis test cases for NextDate shows them to be inadequate. There is very little stress on February and on leap years, for example. The real problem here is that there are interesting dependencies among the month, day, and year variables.
- Boundary value analysis presumes the variables to be truly independent. Even so, boundary value analysis happens to catch end-of-month and end-of-year faults. Boundary value analysis test cases are derived from the extrema of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function, nor of the semantic meaning of the variables.

We see boundary value analysis test cases to be rudimentary, in the sense that they are obtained with very little insight and imagination. As with so many things, you get what you pay for.

The physical quantity criterion is equally important. When a variable refers to a physical quantity, such as temperature, pressure, air speed, angle of attack, load, and so forth, physical boundaries can be extremely important. (In an interesting example of this, Sky Harbor International Airport in Phoenix had to close on June 26, 1992 because the air temperature was 122 °F. Aircraft pilots were unable to make certain instrument settings before take-off: the instruments could only accept a maximum air temperature of 120 °F.) In another case, a medical analysis system uses stepper motors to position a carousel of samples to be analyzed. It turns out that the mechanics of moving the carousel back to the starting cell often causes the robot arm to miss the first cell. As an example of logical (versus physical) variables, we might look at PINs or telephone numbers. It's hard to imagine what faults might be revealed by PINs of 0000, 0001, 5000, 9998, and 9999.

5. What is Equivalence Classes? Demonstrate the weak & strong Normal equivalence class Testing with an example. [10]

Scheme:
Definition: Equivalence Class [2]
Weak Normal equivalence class [4]
Strong Normal equivalence class [4]

Solution:

- Equivalence Classes: Important aspect of equivalence classes is that they form a partition of a set, where partition refers to a collection of mutually disjoint subsets whose union is the entire set.
- Equivalence Classes has two important implications for testing: the fact that the entire set is represented provides a form of completeness, and the disjointness assures a form of non-redundancy. Because the subsets are determined by an equivalence relation, the elements of a subset have something in common.
- The idea of equivalence class testing is to identify test cases by using one element from each equivalence class.
- If the equivalence classes are chosen wisely, this greatly reduces the potential redundancy among test cases. In the Triangle Problem, for example, we would certainly have a test case for an equilateral triangle, and we might pick the triple (5, 5, 5) as inputs for a test case. If we did this, we would not expect to learn much from test cases such as (6, 6, 6) and (100, 100, 100). Our intuition tells us that these would be "treated the same" as the first test case, thus they would be redundant.
- The key (and the craft!) of equivalence class testing is the choice of the equivalence relation that determines the classes.
- To understand equivalence class, we need to make a distinction between weak and strong equivalence class testing which can be later compared with the traditional form of equivalence class testing.
- Suppose our program is a function of three variables, a, b, and c, and the input domain consists of sets A, B, and C. Now, suppose we choose an "appropriate" equivalence relation, which induces the following partition:

$$A = A1 \cup A2 \cup A3$$
$$B = B1 \cup B2 \cup B3 \cup B4$$
$$C = C1 \cup C2$$

Finally, we denote elements of the partitions as follows:

$$a1 \in A1$$
$$b3 \in B3$$
$$c2 \in C2$$

Weak Equivalence Class Testing

Note: Read 2 variable example to understand in page no. 90 & 91

With the notation as given above, weak equivalence class testing is accomplished by using one variable from each equivalence class in a test case. For the above example, we would end up with the following weak equivalence class test cases:

Test Case	a	b	c
WE1	a1	b1	c1
WE2	a2	b2	c2
WE3	a3	b3	c3
WE4	a1	b4	c2

This set of test cases uses one value from each equivalence class. We identify these in a systematic way, hence the apparent pattern. In fact, we will always have the same number of weak equivalence class test cases as there are classes in the partition with the largest number of subsets.

Strong Equivalence Class Testing

Strong equivalence class testing is based on the Cartesian product of the partition subsets. Continuing with this example, the Cartesian product A x B x C will have 3 × 4 × 2 = 24 elements, resulting in the test cases in the table below:

Test Case	a	b	c
SE1	a1	b1	c1
SE2	a1	b1	c2
SE3	a1	b2	c1
SE4	a1	b2	c2
SE5	a1	b3	c1
SE6	a1	b3	c2

SE7	a1	b4	c1
SE8	a1	b4	c2
SE9	a2	b1	c1
SE10	a2	b1	c2
SE11	a2	b2	c2
SE12	a2	b2	c2
SE13	a2	b3	c1
SE14	a2	b3	c2
SE15	a2	b4	c1
SE16	a2	b4	c2
SE17	a3	b1	c1
SE18	a3	b1	c2
SE19	a3	b2	c1
SE20	a3	b2	c2
SE21	a3	b3	c1
SE22	a3	b3	c2
SE23	a3	b4	c1
SE24	a3	b4	c2

Notice the similarity between the pattern of these test cases and the construction of a truth table in propositional logic. The Cartesian product guarantees that we have a notion of "completeness" in two senses: we cover all the equivalence classes, and we have one of each possible combination of inputs.
As we shall see from our continuing examples, the key to "good" equivalence class testing is the selection of the equivalence relation. Watch for the notion of inputs being "treated the same". Most of the time, equivalence class testing defines classes of the input domain. There is no reason why we could not define equivalence relations on the output range of the program function being tested, in fact, this is the simplest approach for the Triangle Problem.

6. Explain decision table structure and how to build it. Demonstrate decision table for the triangle problem with appropriate conditions and actions. [10]

Scheme:
Definition: Decision Table [2]
Construction [3]

Triangle problem solving [5]

Solution:

Decision tables are a precise yet compact way to model complicated logic. Decision tables, like if-then-else and switch-case statements, associate conditions with actions to perform.
Unlike the control structures found in traditional programming languages, decision tables can associate many independent conditions with several actions in an elegant way.

- Decision tables make it easier to observe that all possible conditions are accounted for.
- Decision tables can be used for:
  - Specifying complex program logic
  - Generating test cases (Also known as *logic-based testing)*
  - *Logic-based testing* is considered as:
  - structural testing when applied to structure (i.e. control flow graph of an implementation).
  - functional testing when applied to a specification.

Decision Tables – Structure

Conditions - *(Condition stub)*	Condition Alternatives – *(Condition Entry)*
Actions – *(Action Stub)*	Action Entries

Each condition corresponds to a variable, relation or predicate
Possible values for conditions are listed among the condition alternatives

- Boolean values (True / False) – Limited Entry Decision Tables
- Several values – Extended Entry Decision Tables
- Don't care value

Each action is a procedure or operation to perform
The entries specify whether (or in what order) the action is to be performed.

Decision Table Development Methodology

1. Determine conditions and values
2. Determine maximum number of rules
3. Determine actions
4. Encode possible rules
5. Encode the appropriate actions for each rule
6. Verify the policy
7. Simplify the rules (reduce if possible the number of columns)

Decision Table for the Triangle Problem

Conditions											
C1: a < b+c?	**F**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
C2: b < a+c?	**-**	**F**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**

C3: c < a+b?	**-**	**-**	**F**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
C4: a=b?	**-**	**-**	**-**	**T**	**T**	**T**	**T**	**F**	**F**	**F**	**F**
C5: a=c?	**-**	**-**	**-**	**T**	**T**	**F**	**F**	**T**	**T**	**F**	**F**
C6: b=c?	**-**	**-**	**-**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**
Actions	**How many Xs? 11**										
A1: Not a Triangle	**X**	**X**	**X**								
A2: Scalene											**X**
A3: Isosceles							**X**		**X**	**X**	
A4: Equilateral				**X**							
A5: Impossible					**X**	**X**		**X**			