Black-Box Software Testing (Part I)

Speaker: Jerry Gao Ph.D.

Computer Engineering Department
San Jose State University

email: jerry.gao@sjsu.edu
URL: http://www.engr.sjsu.edu/gaojerry

Presentation Outline

- Introduction to Black Box Software Testing?
  - Definition
  - Why Black Box Testing?
  - Testing Objectives and Focuses
- An Example
- Graph-based Testing Methods
- Equivalence Partitioning
- Boundary Value Analysis

Introduction to Black Box Testing

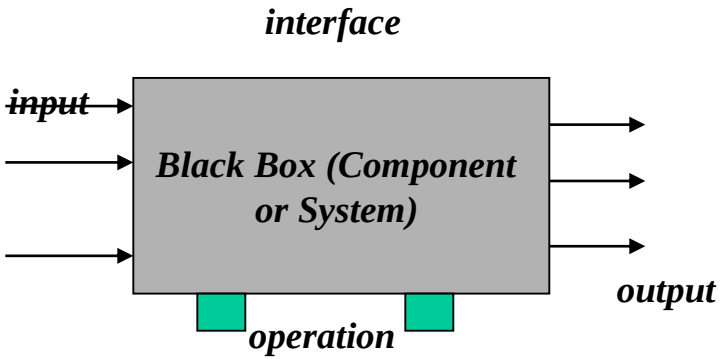
What is black box testing?

- Black box testing also known as specification-based testing.

- Black box testing refer to test activities using specification-based testing methods and criteria to discover program errors based on program requirements and product specifications.

The major testing focuses:

- ***specification-based function errors***
- ***specification-based component/system behavior errors***
- ***specification-based performance errors***
- ***user-oriented usage errors***
- ***black box interface errors***



Introduction to Black Box Testing

Under test units in black-box: ***Software components, subsystems, or systems***

What do you need?

- For software components, you need component specification, user interface doc.

-For a software subsystem or system, you need requirements specification, and product specification document.

You also need:

- ***Specification-based software testing methods***
- ***Specification-based software testing criteria***

- ***good understanding of software components (or system)***

An Example

Testing a triangle analyzer:

Program specification:

Input: ***3 numbers separated by commas or spaces***

Processing:

Determine if three numbers make a valid triangle; if not, print message NOT A TRIANGLE.

If it is a triangle, classify it according to the length of the sides as scalene (no sides equal), isosceles (two sides equal), or equilateral (all sides equal).

If it is a triangle, classify it according to the largest angle as acute (less than 90 degree), obtuse (greater than 90 degree), or right (exactly 90 degree).

Output: ***One line listing the three numbers provided as input and the classification or the not a triangle message.***

Example:	***3,4,5***	***Scalene***	***Right***
	6,1,6	***Isosceles***	***Acute***
	5,1,2	***Not a triangle***	

An Example

Functional Test Cases:

	Acute	*Obtuse*	*Right*
Scalene:	*6,5,3*	*5,6,10*	*3,4,5*
Isosceles:	*6,1,6*	*7,4,4*	*1,2, 2^(0.5)*
Equilateral:	*4,4,4*	*Not possible*	*Not possible*

Functional Test Cases:

Input	*Expected Results*
4,4,4	*Equilateral acute*
1,2,8	*Not a triangle*
6,5,3	*Scalene acute*
5,6,10	*Scalene obtuse*
3,4,5	*Scalene right*
6,1,6	*Isosceles acute*
7,4,4	*Isosceles obtuse*
1,1,2^(0.5)	*Isosceles right*

An Example

Test cases for special inputs and invalid formats:

3,4,5,6	*Four sides*
646	*Three-digit single number*
3,,4,5	*Two commas*
3 4,5	*Missing comma*
3.14.6,4,5	*Two decimal points*
4,6	*Two sides*
5,5,A	*Character as a side*
6,-4,6	*Negative number as a side*
-3,-3,-3	*All negative numbers*
	Empty input

An Example

Boundary Test Cases:

(1) Boundary conditions for legitimate triangles

1,1,2	*Makes a straight line, not a triangle*	
0,0,0	*Makes a point, not a triangle*	
4,0,3	*A zero side, not a triangle*	
1,2,3.00001	*Close to a triangle but still not a triangle*	
9170,9168,3	*Very small angle*	*Scalene, acute*
.0001,.0001,.0001	*Very small triangle*	*Equilateral, acute*
83127168,74326166,96652988	*Very large triangle, scalene, obtuse*	

Boundary conditions for sides classification:

3.0000001,3,3	*Very close to equilateral, Isosceles, acute*	
2.999999,4,5	*Very close to isosceles*	*Scalene, acute*

Boundary conditions for angles classification:

3,4,5.000000001	*Near right triangle*	*Scalene, obtuse*
1,1,1.41141414141414	*Near right triangle*	*Isosceles, acute*

Software Testing Principles

Davids [DAV95] suggests a set of testing principles:

- All tests should be traceable to customer requirements.

- Tests should be planned long before testing begins.

- The Pareto principle applies to software testing.
- 80% of all errors uncovered during testing will likely be traceable to 20% of all program modules.

- Testing should begin "in the small" and progress toward testing "in the large".

- Exhaustive testing is not possible.

- To be most effective, testing should be conducted by an independent third party.

Software Testability

According to James Bach:

***Software testability** is simply how easily a computer program can be tested.*

A set of program characteristics that lead to testable software:

- Operability: "the better it works, the more efficiently it can be tested."
- Observability: "What you see is what you test."

- Controllability: "The better we can control the software, the more the testing can be automated and optimized."

- Decomposability: "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

- Simplicity: "The less there is to test, the more quickly we can test it."
- Stability: "The fewer the changes, the fewer the disruptions to testing."
- Understandability:"The more information we have, the smarter we will test."

Equivalence Partitioning

***Equivalence partitioning** is a black-box testing method*

- *divide the input domain of a program into classes of data*
- *derive test cases based on these partitions.*

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input domain.

An equivalence class represents a set of valid or invalid states for input condition.

An input condition is:

- *a specific numeric value, a range of values*
- *a set of related values, or a Boolean condition*

Equivalence Partitioning

***Equivalence partitioning** is a black-box testing method*

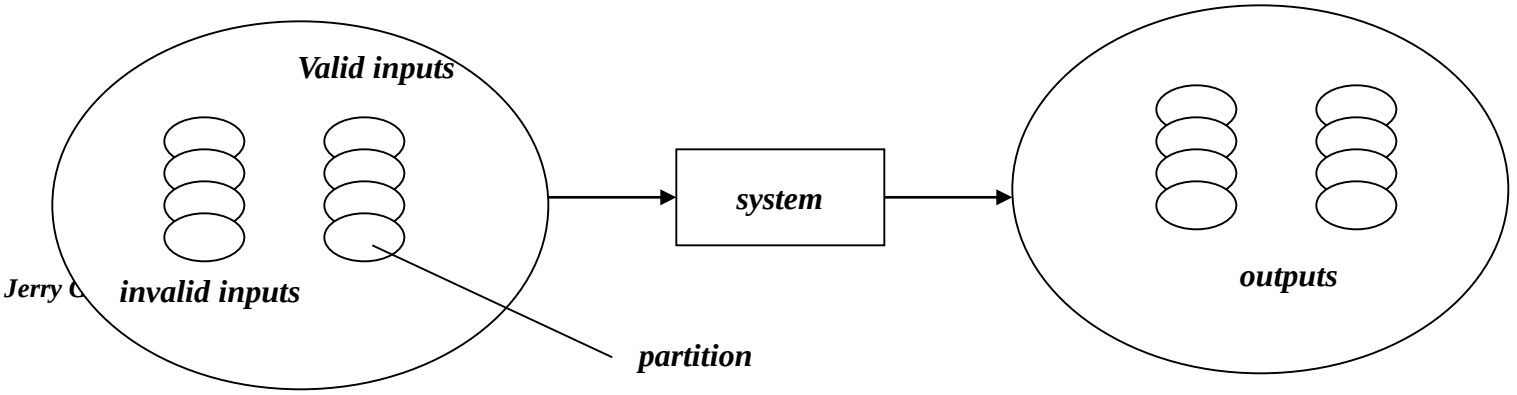
- *divide the input domain of a program into classes of data*
- *derive test cases based on these partitions.*

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input domain.

An equivalence class represents a set of valid or invalid states for input condition.

An input condition is:

- *a specific numeric value, a range of values*
- *a set of related values, or a Boolean condition*



Equivalence Classes

Equivalence classes can be defined using the following guidelines:

- If an input condition specifies a range, one valid and two invalid equivalence class are defined.

- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

- If an input condition specifies a member of a set, one valid and one invalid equivalence classes are defined.

- If an input condition is Boolean, one valid and one invalid classes are defined.

Examples:

area code: input condition, Boolean - the area code may or may not be present.
input condition, range - value defined between 200 and 900

password: input condition, Boolean - a password nay or may not be present.
input condition, value - six character string.

command: input condition, set - containing commands noted before.

Boundary Value Analysis

Boundary value analysis(BVA) *- a test case design technique*
- complements to equivalence partition

Objective:

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

Guidelines:

- If an input condition specifies a range bounded by values a and b, test cases should be designed with value a and b, just above and below a and b.

Example: Integer D with input condition [-3, 10],
test values: -3, 10, 11, -2, 0

- If an input condition specifies a number values, test cases should be developed to exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

Example: Enumerate data E with input condition: {3, 5, 100, 102}
test values: 3, 102, -1, 200, 5

Boundary Value Analysis

- Guidelines 1 and 2 are applied to output condition.

- If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary

Such as data structures:

- array *input condition:*
empty, single element, full element, out-of-boundary

search for element:
- element is inside array or the element is not inside array

You can think about other data structures:

- list, set, stack, queue, and tree

One-Dimensional Domain Bugs in Open Boundaries

An Open Domain (A):

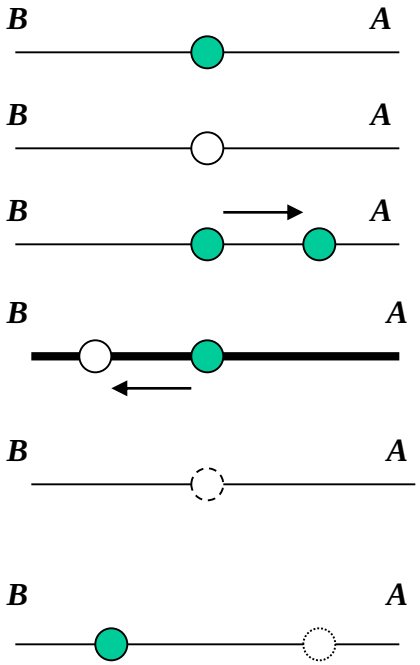
Closure Bug:

Boundary Shifted Right:

Boundary Shifted Left:

Missing Boundary:

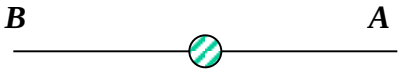
Extra Boundary:



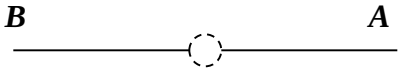
If the domain boundary is open, an off point is a point near the boundary but in the domain being tested.

One-Dimensional Domain Bugs in Closed Boundaries

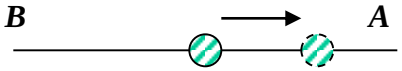
A Closed Domain (A):



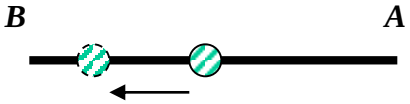
Closure Bug:



Boundary Shifted Right:

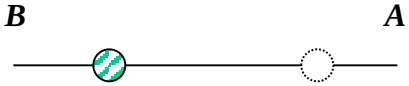


Boundary Shifted Left:



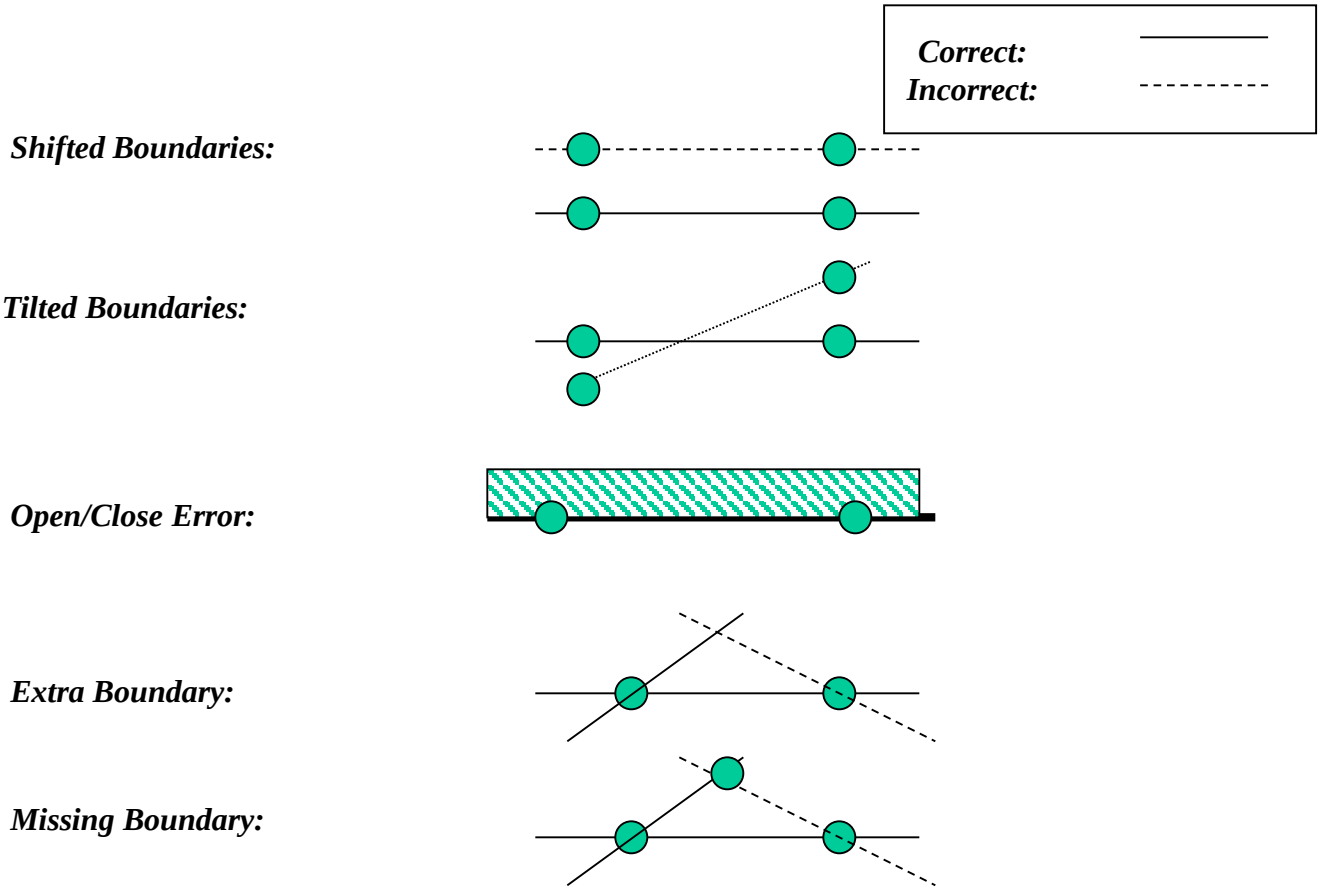
Missing Boundary:

Extra Boundary:



If the domain boundary is closed, an off point is a point near the boundary but in the adjacent domain.

Generic Domain Bugs in One-Dimensional Domains



An Example

Given a function module which implements function Z= F(X, Y), which defined as follows:

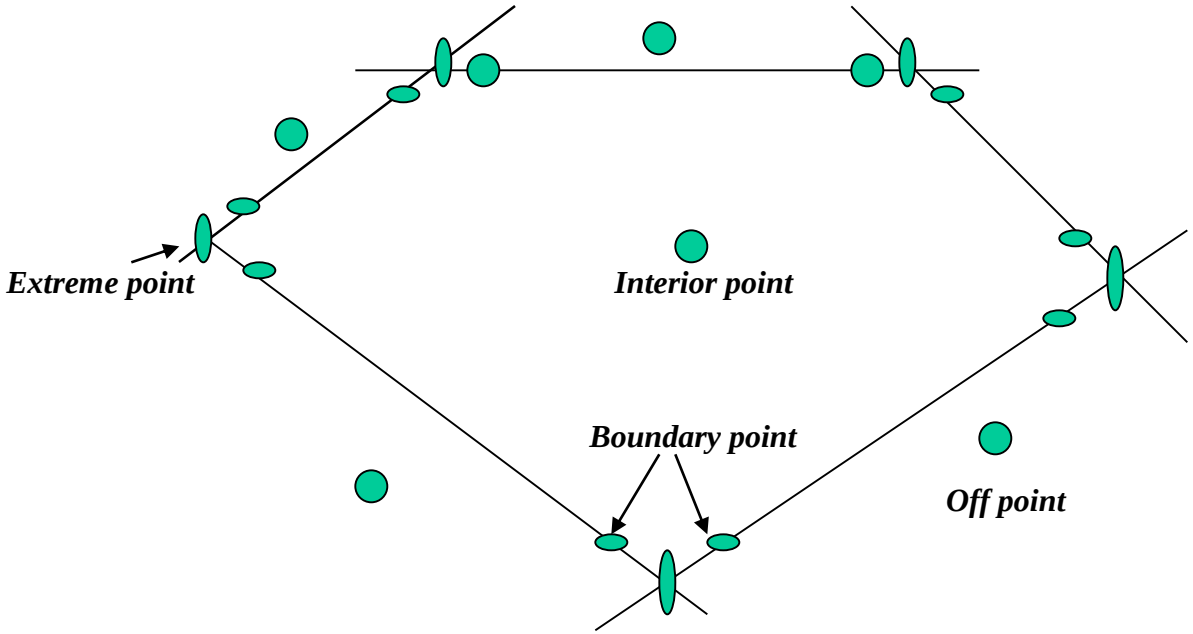
Z = F(X,Y), where X and Y are integer parameters for F. The detailed definition is given below.

```
        --
        |       X + Y          when 10<=X<=20, and 12<=Y<=30
Z =     |       X - Y          when 0<=X<10, and 0<=Y<12
        |       0              under other conditions
        --
```

Please answer the following questions:

- (4%) Identify the equivalence classes in [X, Y]. (hints: Considering Z as a function F with X and Y input variables.)
- (4%) List your test cases in [X, Y] based on the equivalence classes.
- (4%) Perform boundary value analysis, and list all boundary conditions for X and Y.
- (3%) List your test cases in [X, Y] based on boundary value analysis.

Domain Boundary



Two-dimension Domain Boundary

Testing Two-Dimensional Domains

- Closure bug.

For example, using a wrong operator (for example, x >= k when x > k is intended or vise versa). This bug could be detected due to the testing of different boundaries or trying interior and off points.

- Shifted boundary:

For example, a boundary is shifted due to the use of an incorrect constant in a predicate, such as x+y>=17 when x + y >=7 was intended. The off point catches this bug.

- Titled boundary:

A tilted boundary occurs when coefficients in the boundary inequality are wrong. For example, 3x+7y > 17 when 7x + 3y > 17 was intended. Testing different domain points can detect the bug.

- Extra boundary:

An extra boundary is created by an extra predicate. Try different boundary points can detect this bug.

- Missing boundary:

A missing boundary is created by leaving a boundary predicate out.