Software Testing Techniques

Instructor: Dr. Jerry Gao

Software Testing Techniques

- *Software Testing Fundamentals*
    - *Testing Objectives, Principles, Testability*
- *Software Test Case Design*
- *White-Box Testing*
    - *Cyclomatic Complexity*
    - *Graph Matrices*
    - *Control Structuring Testing (not included)*
        - *Condition Testing (not included)*
        - *Data Flow Testing (not included)*
        - *Loop Testing (not included)*
- *Black-Box Testing*
    - *Graph-based Testing Methods (not included)*
    - *Equivalence Partitioning*
    - *Boundary Value Analysis*
    - *Comparison Testing (not included)*

Software Testing Fundamentals

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and coding.

Software testing demonstrates that software function appear to be working according to specifications and performance requirements.

<u>Testing Objectives:</u>

Myers [MYE79] states a number of rules that can serve well as testing objectives:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has high probability of finding an undiscovered error.
- A successful test is one that uncovers an as-yet undiscovered error.

The <u>major testing objective</u> is to design tests that systematically uncover types of errors with minimum time and effort.

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:

<u>Software testability</u> 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."

Test Case Design

Two general software testing approaches:

Black-Box Testing and White-Box Testing

Black-box testing:
knowing the specific functions of a software,
design tests to demonstrate each function and check its errors.

Major focus:
functions, operations, external interfaces,
external data and information

White-box testing:
knowing the internals of a software,
design tests to exercise all internals of a software to make sure
they operates according to specifications and designs

Major focus: internal structures, logic paths, control flows, data flows
internal data structures, conditions, loops, etc.

White-Box Testing and Basis Path Testing

White-box testing, also known as glass-box testing.
It is a test case design method that uses the control structure of the procedural design to derive test cases.

Using white-box testing methods, we derive test cases that
- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decisions one their true and false sides.
- Execute all loops at their boundaries and within their operational bounds.
- Exercise internal data structures to assure their validity.

<u>Basic path testing (a white-box testing technique):</u>

- First proposed by TomMcCabe [MCC76].
- Can be used to derive a logical complexity measure for a procedure design.
- Used as a guide for defining a basis set of execution path.
- Guarantee to execute every statement in the program at least one time.

Cyclomatic Complexity

Cyclomatic complexity is a software metric
-> provides a quantitative measure of the global complexity of a program.

When this metric is used in the context of the basis path testing, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program.

Three ways to compute cyclomatic complexity:

- The number of regions of the flow graph correspond to the cyclomatic complexity.

- Cyclomatic complexity, V(G), for a flow graph G is defined as
$$V(G) = E - N + 2$$
where E is the number of flow graph edges and N is the number of flow graph nodes.

- Cyclomatic complexity, V(G) = P + 1
where P is the number of predicate nodes contained in the flow graph G.

Deriving Test Cases

Step 1 : ***Using the design or code as a foundation, draw a corresponding flow graph.***

Step 2: ***Determine the cyclomatic complexity of the resultant flow graph.***

Step 3: ***Determine a basis set of linearly independent paths.***

For example,

path 1: 1-2-10-11-13
path 2: 1-2-10-12-13
path 3: 1-2-3-10-11-13
path 4: 1-2-3-4-5-8-9-2-…
path 5: 1-2-3-4-5-6-8-9-2-..
Path 6: 1-2-3-4-5-6-7-8-9-2-..

Step 4: ***Prepare test cases that will force execution of each path in the basis set.***

Path 1: test case:

value (k) = valid input, where k < i defined below.
value (i) = -999, where 2 <= I <= 100
expected results: correct average based on k values and proper totals.

Equivalence Partitioning

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

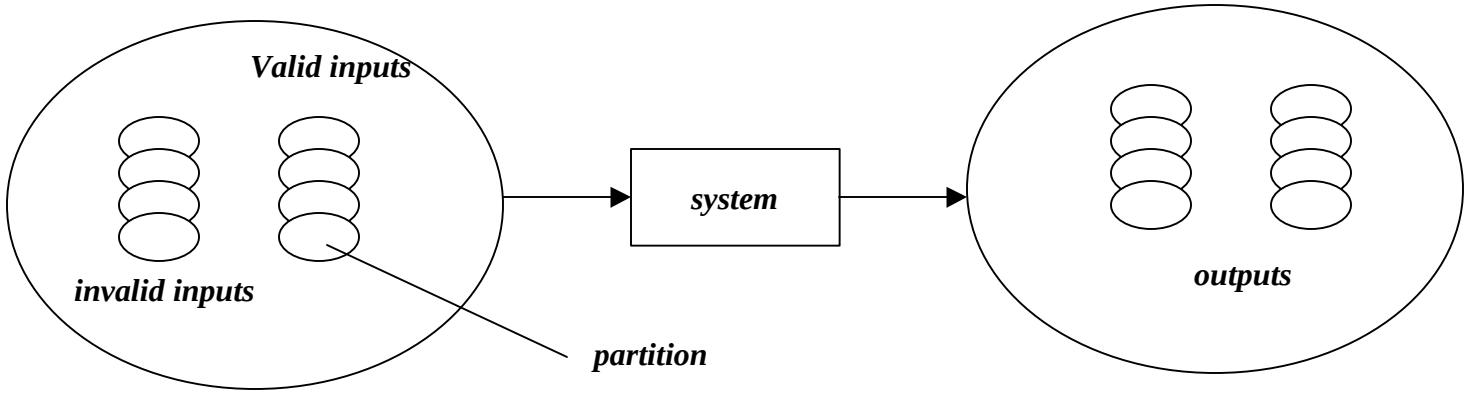
- *divide the input domain of a program into classes of data*
- *derive test cases based on these partitions.*

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.

<u>Examples:</u>

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