Software Engineering

(6) Testing

Sokendai / National Institute of Informatics
Fuyuki Ishikawa / 石川　冬樹
f-ishikawa@nii.ac.jp / @fyufyu
http://research.nii.ac.jp/~f-ishikawa/

大学共同利用機関法人 情報・システム研究機構
国立情報学研究所
National Institute of Informatics

TOC

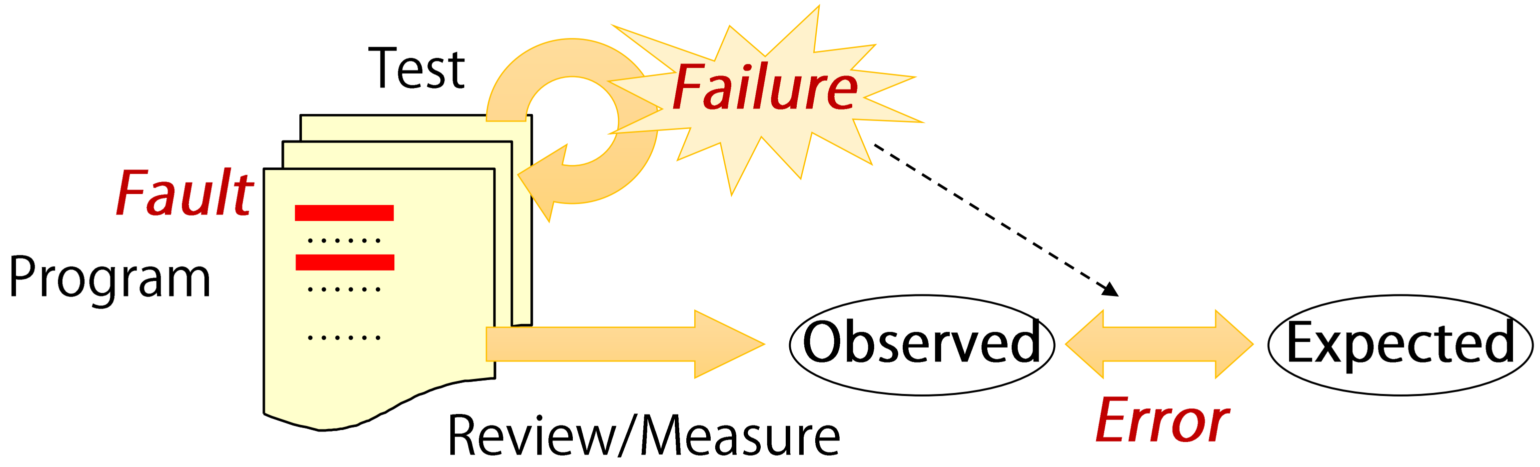
- <u>Overview</u>
- Overview of Each Test Phase
- Whitebox Testing
- Blackbox Testing
- Combinational Testing

Testing

- Testing

  Analyze the target software and try to cause failures for detecting bugs (defects, faults)

  - The most practical way for V&V on the program code
  - Cannot ensure to detect all the bugs



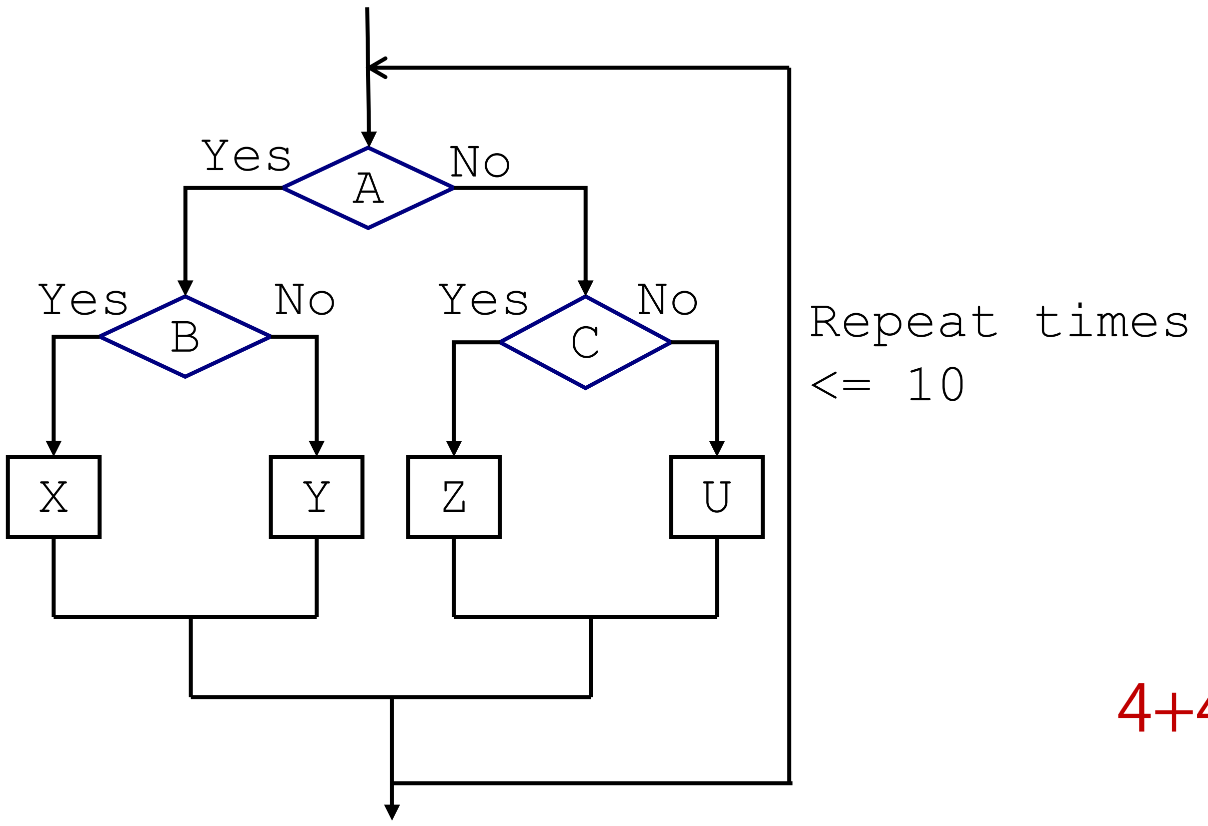
Terminology

- Error（エラー）: difference between the theoretical correct value/condition and the computed/observed/measured one
- Fault（障害，バグ，不具合）： wrong steps, process, or data definition in the program code
- Failure（故障）: state in which required functions are not provided

Different terminologies/translations exist,
e.g., “fault” in “fault-tolerance” is translated as 「故障」

Question: what are “good” tests?

- Example: how many executiosn required to try all the possible execution paths?



$4+4^2+4^3+\cdots 4^{10} \geqq 10^6$

Question: what are “good” tests?

- Achieve the “best” with the minimum effort, assuming we cannot be perfect
  - Example: compare two test suites for a program expected to calculate the absolute value of the input

```
if (x>=1) return x  else return -x
```

Test suite 1

1. input x = 3, check the result is 3
2. input x = 5, check the result is 5

Test suite 2

1. input x = 3, check the result is 3
2. input x = -3, check the result is 3

Myers Triangle Problem

- Define a test suite for the following target problem
  - Reads three integer values from the console
  - Outputs a type of triangles with the side lengths of the values: “regular triangle,” “isosceles triangle,” or “scalene triangle”

Myers Triangle Problem: checklist (1)

1. Included a case for a valid scalene triangle?
2. Included a case for a valid regular triangle?
3. Included a case for a valid isosceles triangle?
4. In #3, included at least three cases with different orders (regarding the position of the two same values)?

   e.g., (3, 3, 4) (3, 4, 3) (4, 3, 3)
5. Included a case in which one of the value is 0?
6. Included a case in which one of the value is negative?

Myers Triangle Problem: checklist (2)

7. Included a case in which one of the values equals to the sum of the other two values?

   e.g., (1, 2, 3)

8. In #7, included at least three cases with different orders? (regarding the position of)

   e.g., (1, 2, 3), (1, 3, 2), (3, 1, 2)

9. Included a case in which one of the values is more than the sum of the other two values?
10. In #9, included at least three cases with different orders?

Myers Triangle Problem: checklist (3)

11. Included the case with all the values as 0?
12. Included a case with a non-integer value?
13. Included a case with a wrong number of inputs?
14. Defined the expected outputs for all of the test cases?

Myers Triangle Problem: Summary

- Each of the different test cases (except for the last item) correspond to different types of possible faults
- 7.8/14

  (Probably improved now?)

➡ Need systematic principles/methods to derive such test cases

Other Topics in Testing (1) Termination Criteria

- Naive termination criteria
  - "When we ran out the time/budget": easy but not essential
  - "When we didn't find any bug": may lead to "weak" tests with less bug-finding capability
- Statistics and heuristics
  - e.g., comparison with known average values for detected bugs for the size of the target program
  - e.g., convergence in the number of detected bugs in a certain time period

Other Topics in Testing (2) Mental Aspects

- Principles based on the mental aspects
  - Define the expected outputs beforehand
  - Not test the program of your own
  - Make test cases for wrong inputs
  - Investigate also there is no unintended behavior
  - Not expect “there should not be any error”
  - Expect more errors in the part where many errors were found
  - Not criticize the coder
  - Think testing as creative challenges

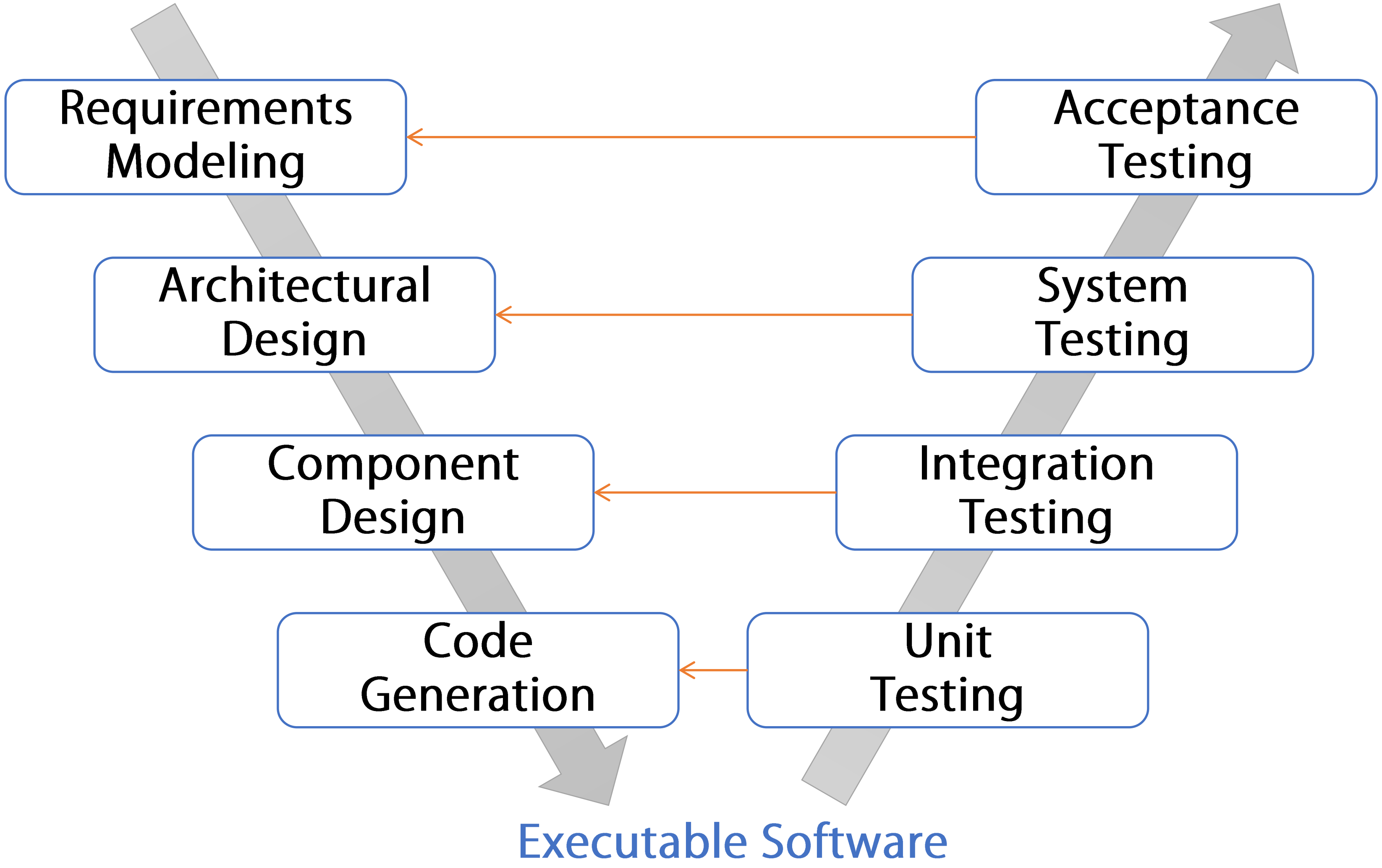
Classification (1) Whitebox/Blackbox

- Whitebox testing
  - Design tests by considering the internal structure of the program code
    - e.g., make tests for both of the branches in an if-else statement
- Blackbox testing
  - Design tests only by considering the specification
    - e.g., make tests based on use cases

Classification (2) Phases

- Unit Testing（ユニットテスト・単体テスト）
  - Target small components such as methods
- Integration Testing（結合テスト・統合テスト）
  - Target combination of (already tested) components
- System Testing（システムテスト）
  - Target system elements such as network, hardware, and database
- Acceptance Testing（受け入れテスト）
  - Target actual operation including user satisfaction, load, long-term operation

(Review) The V Process Model



[C. Bucanac, 1999]

Regression Testing

- Regression Testing（回帰テスト）
  - Check whether something is worse than the previous version
  - Background: it is very typical that a fix for a certain function leads to failures in other parts
    (Japanese engineers often say 「デグレ」 for degradation)

TOC

- Overview
- <u>Overview of Each Test Phase</u>
- Whitebox Testing
- Blackbox Testing
- Combinational Testing

Basic Concept: Test Driver and Stub

- We want to focus on SUT (System Under Test) in each test by excluding possibilities of bugs in other parts
  - Test driver: invoke the SUT and observe the outcome
  - Test stub: provide pseudo functions for components that have not been implemented or tested

Test driver

```
p = ...
result = m1 (p);
assert result==5;
```

Target program

```
int m1 (int x){
   ...
   m1_sub(...)
   ...
}
```

Test stub

```
m1_sub(int n){
  switch (n){
    case 0:
    return 2;
    case 1:
    return 3;
    ...
```

Unit Testing

- Focus on specific small components by making use of test drivers and stubs
- Often use popular XUnit frameworks
  - Junit, CppUnit, PHPUnit, unittest (Python), …
  - Define each test case separately, and execute them again and again

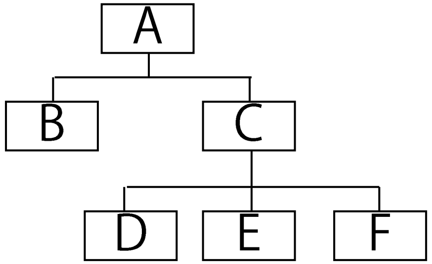
```
public class TestCase1
    extends TestCase{
  public void testFun1(){
    ...
    assertEqual( x, 3 );
  }
}
```

```
suite.addTest(new TestCase1());
suite.addTest(new TestCase2());
...
suite.run(result);
...
```

(now tools automatically collects defined tests in a project)

Integration Testing

- Gradually integrate and test combinations of components
  - Otherwise, it's hard to identify the bug (big-bang)
  - Incremental Testing
  - Top-down vs. bottom-up: bottom-up is easier to do in paralell with coding but may encounter large rollbacks as the key function at the top level is tested last



Top-down
1. A and B (with C stub)
2. A, B, and C (with D, E, F stubs)
3. A. B, C, and D (with E, F stubs)
4. …

Bottom-up
1. C and D (with E, F stubs)
2. C, D, and E (with F stub)
3. C, D, E, and F
4. …

System Testing and Acceptance Testing

- Various system-level aspects
  - Stress, performance, volume, usability, security, compatibility, portability, document-understandability, …
- Acceptance testing uses actual users or data

TOC

- Overview
- Overview of Each Test Phase
- **Whitebox Testing**
- Blackbox Testing
- Combinational Testing

Coverage

- Coverage（カバレッジ，被覆率）：

how many “elements” were covered by tests?

- Example: if (P and Q) then … else …
  - Cover branches (then, else)

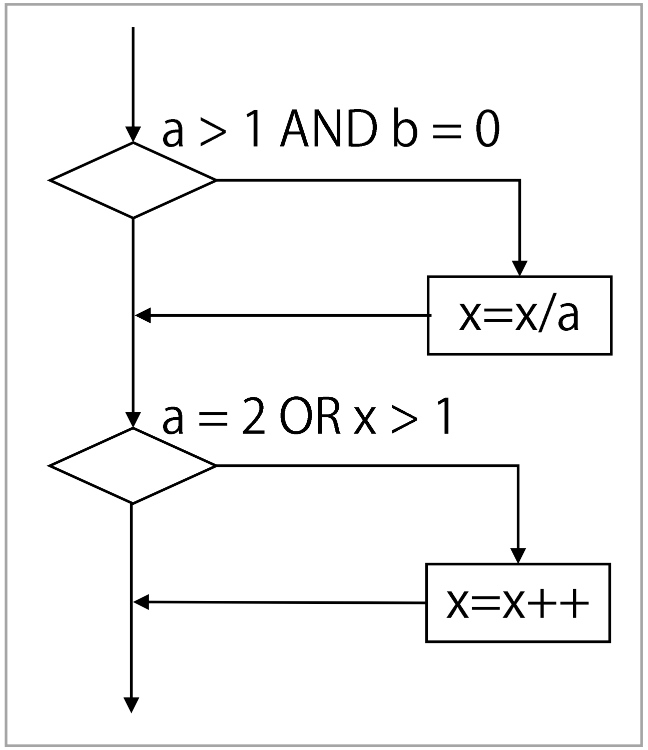
    → (P, Q) = (true, true), (false, true) covers 2/2
  - Cover conditions of P and Q (true, false)

    → (P, Q) = (true, true), (false, false) covers 4/4

Statement Coverage

- Statement Coverage（命令網羅, C0)
  - Each statement was executed at least once

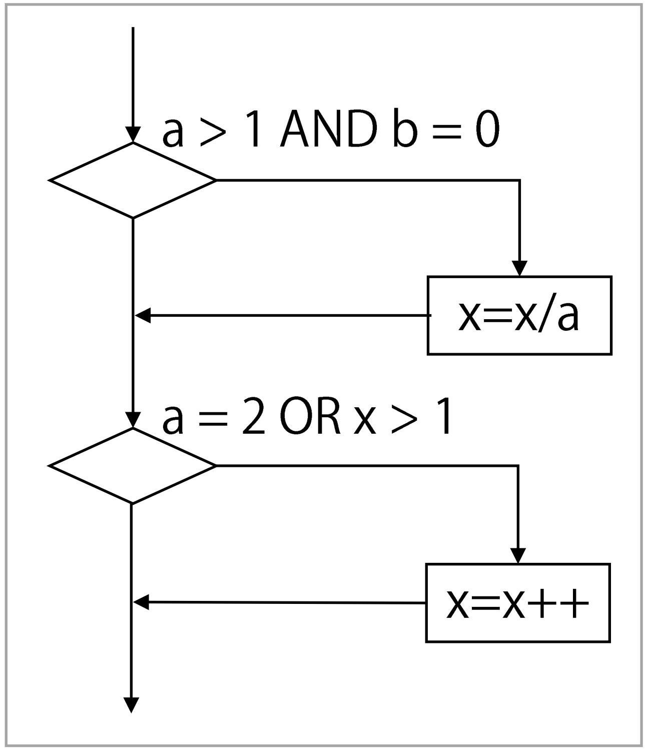
e.g., input (a, b, x) = (2, 0, 3)



Branch Coverage

- Branch Coverage / Decision Coverage
  （分岐網羅・判定条件網羅, C1）
  - Each statement was executed at least once

e.g., input (a, b, x) = (3, 0, 3), (2, 1, 1)



Note on Statement Coverage and Branch Coverage

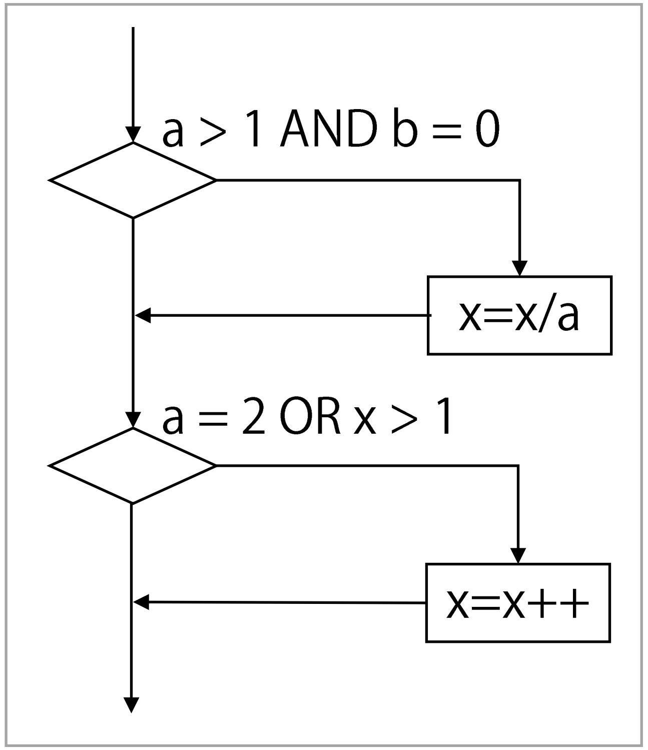
- We often think branch coverage subsumes statement coverage
- Strictly speaking, there are some situations where a test suite satisfy the branch coverage but not the statement one
  - If there is an unreachable part of the code (this is considered as a bug or undesirable)
  - If there are many entries for the program

Condition Coverage

- Condition Coverage（条件網羅, C2）
  - Each possible outcome of individual conditions was exposed at least once

e.g., input

(a, b, x) = (1, 0, 3), (2, 1, 1)



Note on Branch Coverage and Condition Coverage

- There are some situations where a test suite satisfy the condition coverage but not the branch one
  - e.g., for "if P AND Q"

    (P, Q) = (true, false), (false, true)

Multiple-condition Coverage

- Multiple-condition Coverage（複数条件網羅）
  - Each possible combination of possible outcomes in each branch was exposed at least once

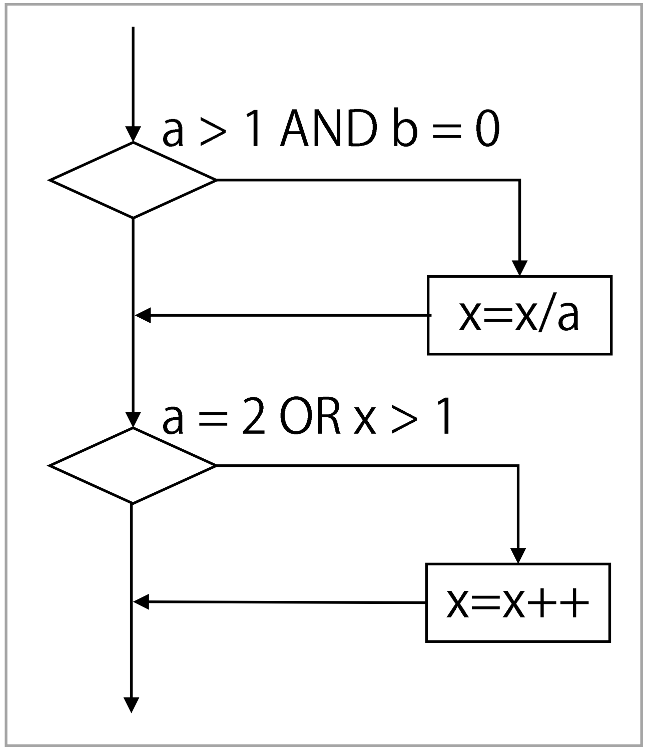
e.g., (a, b, x) =

(2, 0, 4) → (T-T, T-T)

(2, 1, 1) → (T-F, T-F)

(1, 0, 2) → (F-T, F-T)

(1, 1, 1) → (F-F, F-F)



MC/DC

- MC/DC（Modified Condition/Decision Coverage）
  - Branch coverage
  - Condition coverage, but "condition covered" means "each condition solely affects the branch decision"
- ⇨ i.e., we don't think "unused condition value" as "covered"

> Example: if (P and Q) then … else …
> → (P, Q) = (true, true), (false, false) → 4/4 condition values covered?
> → Q=false was not actually used!
>
> → We should have (P, Q) = (true, true), (true, false), (false, true)

Practices of Coverage

- 100% is often difficult
  - There may be impossible combinations of condition values
  - It is very difficult to derive a test suite
  - ➡ Thresholds are often defined in each company, e.g., 85%
- Complex coverage criteria are more difficult to achieve and costly to evaluate
  - Branch coverage (C1) is a modest standard?
- Safety-aware domains require MC/DC such as avionics

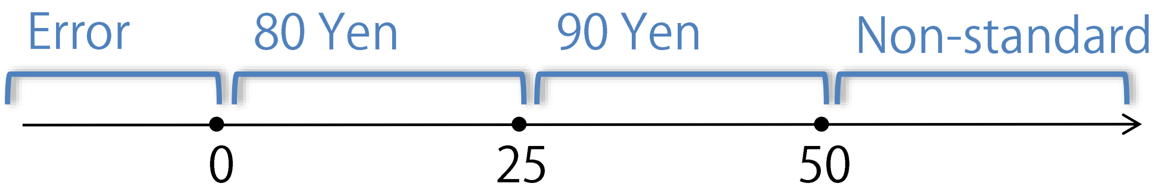
TOC

- Overview
- Overview of Each Test Phase
- Whitebox Testing
- **<u>Blackbox Testing</u>**
- Combinational Testing

Equivalence Partitioning

- **Equivalence Partitioning（同値分割）**：
  Make classes (groups) of inputs that lead to specific types of behaviors
  - We should have at least one test case for each equivalent class
  - Example: price calculation for standard mail

Weight	Price
<= 25g	80 Yen
<= 50g	90 Yen



Necessary test cases (example)

-5g, 10g, 35g, 80g

Equivalence Partitioning: Guidelines

- Guidelines for partitioning
  - An input has range of values or ranges for num. of values
  - ➡ Class for effective inputs, one for too small, one for too large

  - An input has enumeration of possible values
  - ➡ Classes for each value, one for invalid value

  - An input has a condition to satisfy
  - ➡ Class for valid inputs, one for invalid inputs

Equivalence Partitioning: Example

- Example: compiler function to handle array declaration part
  - Num of arrays: 1, 1+, none
  - Length of array name: valid, 0, too long
  - Array name: alphabets, with numbers, with other characters
  - Array dimension: valid, 0, too many
  - Num. of elements in each dimension: valid, negative, too many

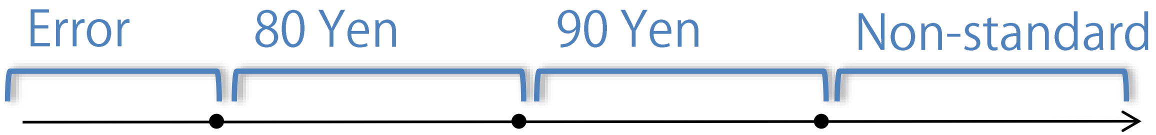
    specified, not specified

    specified as const, specified with int variable, …

Boundary Value Analysis

- Boundary Value Analysis（境界値分析）
  - Use boundary values in equivalence classes
  - Example: price calculation for standard mail

Weight	Price
<= 25g	80 Yen
<= 50g	90 Yen



Necessary test cases

0g, 1g, 25g, 26g, 50g, 51g

Boundary Value Analysis

- Example: sort and print of exam. results
  - Num. of questions: 0, 1, upper-bound, uppor-bound+1
  - Num. of students: 0, 1, upper-bound, uppor-bound+1
  - Sorting results: all the same, all different
  - Deviation scores: “one score 0 and the others 100” (max. deviation), “all the same scores” (deviation 0)
  - Num. of pages: 0, 1, upper-bound, uppor-bound+1
  - …

TOC

- Overview
- Overview of Each Test Phase
- Whitebox Testing
- Blackbox Testing
- <u>Combinational Testing</u>

Combinational Testing

- Some bugs lead to observable failures only for a certain combinations of multiple factors
  - Web application: OS type/version, browser type/version, browser plug-in version, …
- There is a logical reason why the specific combination does not work but it is very hard to know that before we actually encounter and investigate the failure
  - i.e., we cannot say “we don’t need tests for this combination”

Combinational Testing

In general,

- When we have *n* aspects （factors, 因子） and each can take *a* possible values (levels, 水準)

→ We have a^n combinations

  - 4 factor, 3 levels for each factor: 81 combinations
  - 10 factor, 3 levels for each factor: 59049 combinations

➡ How can we get effective tests with a smaller number of combinations?

Pair Construction

- One idea:

  test all the pairs of values from two factors

  - Just a heuristic, not without any theoretical gurantee
  - Past statistics showed more than half (sometimes 80%) faults could be detected with this strategy
    (faults with one aspect and faults with two aspects are dominant)

Pair Construction

- Example
  - Factor: A, B, C
  - Level for each factor: 0, 1
  - For each of (A, B), (B, C), and (C, A), we cover (0, 0), (0, 1), (1, 0), (1, 1)

	A	B	C
Test case 1	0	0	0
Test case 2	0	1	1
Test case 3	1	0	1
Test case 4	1	1	0

Construction of Test Suite

- Approach 1: prepare tables like the previous one and apply them to the give problem

  http://neilsloane.com/oadir/

➡ Orthogonal Arrays（直交表）
  - For each column pair (factor pair), all the pairs appear the same number of times
- Approach 2: make an algorithm to generate test suites
  - Exhaustive search does not scale
  - Typically use heuristics or meta-heuristics

Orthogonal Arrays: Parameters

- Parameters of orthogonal arrays
  - The num. of factors c (appears as columns)
  - The num. of levels: n
  - The num of test cases (appears as rows)

Notation
$L_x(n^c)$

0	0	0
0	1	1
1	0	1
1	1	0

$L_4(2^3)$

0	0	0	0
0	1	1	1
0	2	2	2
1	0	1	2
1	1	2	0
1	2	0	1
2	0	2	1
2	1	0	2
2	2	1	0

$L_9(3^4)$

Orthogonal Arrays: Examples

0	0	0	0	0	0	0
1	1	1	1	1	1	0
2	2	2	2	2	2	0
0	0	1	2	1	2	0
1	1	2	0	2	0	0
2	2	0	1	0	1	0
0	1	0	2	2	1	1
1	2	1	0	0	2	1
2	0	2	1	1	0	1
0	2	2	0	1	1	1
1	0	0	1	2	2	1
2	1	1	2	0	0	1
0	1	2	1	0	2	2
1	2	0	2	1	0	2
2	0	1	0	2	1	2
0	2	1	1	2	0	2
1	0	2	2	0	1	2
2	1	0	0	1	2	2

$L_{18}(3^7)$

0	0	0	0	0
1	1	1	1	0
0	0	1	1	1
1	1	0	0	1
0	1	0	1	2
1	0	1	0	2
0	1	1	0	3
1	0	0	1	3

$L_8(2^4 4^1)$

0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Cover triples
such as (0, 0, 0), (1, 0, 1)

Orthogonal Arrays: Application

- Application example
  - A: {IE,FF,CH}, B: {ON, OFF}, C: {win,mac,lin}

0	0	0	0
0	1	1	1
0	2	2	2
1	0	1	2
1	1	2	0
1	2	0	1
2	0	2	1
2	1	0	2
2	2	1	0

$L_9(3^4)$

→ *Remove the 4th column unnecessary*

0	0	0
0	1	1
0	2	2
1	0	1
1	1	2
1	2	0
2	0	2
2	1	0
2	2	1

→ *Replace with actual levels*

IE	ON	win
IE	OFF	mac
IE	?	lin
FF	ON	mac
FF	OFF	lin
FF	?	win
CH	ON	lin
CH	OFF	win
CH	?	mac

→ *Fill unused levels with arbitrary values*

IE	ON	win
IE	OFF	mac
IE	ON	lin
FF	ON	mac
FF	OFF	lin
FF	OFF	win
CH	ON	lin
CH	OFF	win
CH	ON	mac

Orthogonal Arrays: Characteristics

- Basically, we cannot remove rows when we customize
  - Probably violating the "all the pairs" constraint
  - Difficult to handle exception or inhibition, i.e., we want to exclude a certain combination
    - What if we cannot include "IE-mac" in the previous example
- All the pairs appear the same number of times
  - Not the minimum to cover all the pairs
  - But covers many triples, quadruples, ⋯ thanks to the symmetry

All-Pair Method

- All-Pair method
  - Finds a combination by an algorithm to cover all the pairs
  - Example for A: {0, 1}, B: {0, 1}, C: {0, 1, 2}

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0
0	0	2
1	1	2

We don't have
0 1 2
1 0 2
compared with
orthogonal array

All-Pair Method: Characteristics

- By good algorithms
  - Much less test cases than orthogonal arrays with symmetry
    - Example: for 100 factors and 2 levels, 101 test cases by orthogonal arrays but 10 by an all-pair algorithm
  - Allows for customization such as inhibition
  - Various approaches
    - Greedy search by generating rows one by one
    - Meta-heuristics such as genetic algorithms
    - Update on existing tables
    - …

Example of Tool

- PICT
  - https://github.com/Microsoft/pict/blob/main/doc/pict.md

Logical Combination

- Combinational Testing: without assumptions on logical dependencies between factors
  - No over-confidence on “
- If the logical relationship is clear, we can just organize it to design the tests
  - Decision tables
  - Cause effect graphs

	Rule 1	Rule 2	Rule 3	Rule 4
Condition				
Married	Y	Y	N	N
Student	Y	N	Y	N
Action				
Discount	60	26	50	0

Summary

- Testing
  - Core of V&V activities
  - Cannot be perfect and explore the cost-effectiveness by trying to efficiently expose hidden defects
  - Employs different approaches for different phases and objectives