Software Verification & Validation

Natthapong Jungteerapanich

Handout 3

Acknowledgement

- Some slides in this lecture are adapted from
  - **Paul Ammann and Jeff Offutt's** slides for their textbook **"Introduction to Software Testing"**
  - **Lee Copeland's** "**A Practitioner's Guide to Software Test Design**". Artech House, 2004.

Functional Testing

- **Step 1 : Identify testable functions**
  - Non-OOP Languages: Functions
  - OOP Languages: Methods

- **Step 2 : Find all the parameters**
  - Non-OOP Languages: Parameters of functions
  - OOP Languages: Parameters of methods and relevant data members in the class (because the computation of the method may depend on the values in some data members of the object)

Example: Obesity Calculator v1.0

- Function **obesityCalc(int w)** return **obesityLevel**
  - **Precondition**: **w** is an integer of a weight in Kg, -1000 ≤ w ≤ 1000
  - **Postcondition**: **obesityLevel** is a string of the obesity level, determined from **w** according to the table below.

w	obesityLevel
< 0	Invalid weight
0 – 40	Extremely underweight
41 – 50	Underweight
51 – 60	Normal
61 – 80	Overweight
> 80	Extremely overweight

Input Domains

- How should we design the test cases?

- It is **not** practical to test **all** the possible input value,
  i.e. w = - 1000 … 1000.

- The input domain of a program or a function contains all the possible inputs for that program.

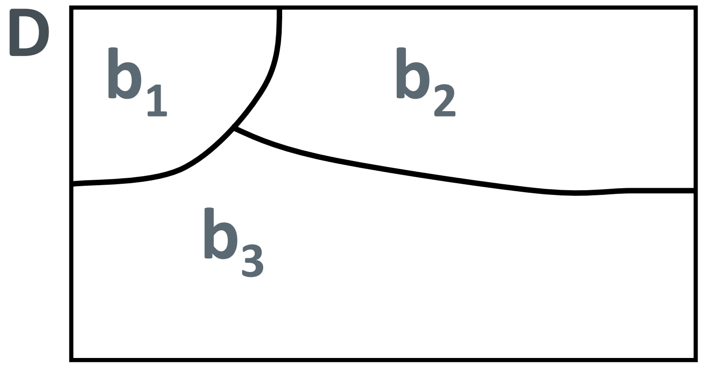
- For even a small program, the input domain can be very large.

- To design a test suite, we have to choose finite sets of values from the input domain.

Input Space Partitioning

- Domain for each input parameter is partitioned into blocks.
- At least one value is chosen from each block.

- Suppose *D* is the input domain.
- A partition scheme *q* defines a set of blocks: $b_1, b_2, \ldots b_n$
- The partition should satisfy two properties :
  1. together the blocks *cover* the domain *D* (complete)
  2. blocks must be *pairwise disjoint* (no overlap)



Input Space Partitioning

- **Step 3 : Model the input domain**
  - An **input domain model (IDM)** represents a partitioning of the domain.
  - The domain is the set of all possible input.
    - For functions or methods, an input can be represented as a tuple of values for the parameters.
  - There are usually more than one way to partition the domain, depending on a chosen characteristic of the input.
  - Each characteristic partitions the domain into sets of blocks. Each block represents a set of values.
  - For example, the input domain of a function **search(int[] arr, int x)** which returns the first position of **x** in **arr** may be partitioned according the following characteristics:
    - *“arr is an empty array or not”*
    - *“x occurs in arr or not*"
    - *“x occurs in arr as the first element, not the first element, or does not occur”*

Two Approaches to Input Domain Modeling

1. **Interface-based approach**
   - Develops characteristics directly from individual input parameters
   - Simplest application
   - Can be partially automated in some situations

2. **Functionality-based approach**
   - Develops characteristics from the functionality of the function or method
   - Typically from the information in the specification
   - Harder to develop—requires more design effort
   - May result in better tests, or fewer tests that are as effective

Interface-Based Approach

- Consider each parameter in isolation
- This is an easy modeling technique and relies mostly on syntax
- Some domain and semantic information won't be used
  - Could lead to an incomplete IDM
- Ignores relationships among parameters

Interface-Based Approach

- Partitioning based on the types of the input variables
  - **Integers**:
    - Positivity: negative, zero, positive
    - Odd or even
    - Prime or composite
  - **Floating points:**
    - Positivity: negative, zero, positive
    - Decimal places: 1, 1/10, 1/100, …
  - **Characters:**
    - Case: uppercase letters, lowercase letters, digits, symbols, …
  - **Strings**:
    - Length: empty string, strings of length 1, string of length 2 or more
    - Member: partitions of characters as above
  - **Lists (or arrays)**:
    - Length: empty list, lists of length 1, lists of length 2 or more
    - Member: partitions of the domains of the lists' members
  - **Pointers**:
    - Nullness: Null pointer, Non-null pointer
    - Type: partitions based in the type of pointed data

Interface-Based Approach

- In the Obesity Calculator example, there is only one input variable.
  - Partitioning characteristic: “**Positivity of w**”

Block no.	Input
1	w < 0
2	w is 0
3	w > 0

Functionality-Based Partitioning

- Partitioning based on the specification of the function
- We may determine the partitioning from the following information in the specification of the function
  - the precondition
  - the postcondition
  - the exceptional cases given in the specification of the function

Functionality-Based Partitioning

- To determine the partitioning from the postcondition, we first partition the possible output of the functions and then determine the corresponding partition on the input:
  - In the Obesity Calculator example, there are 6 possible output values. We then partition the input so that the inputs values which produce the same output are grouped together.

Block no.	output	input
1	Invalid weight	< 0
2	Extremely underweight	0 – 40
3	Underweight	41 – 50
4	Normal	51 – 60
5	Overweight	61 – 80
6	Extremely overweight	> 80

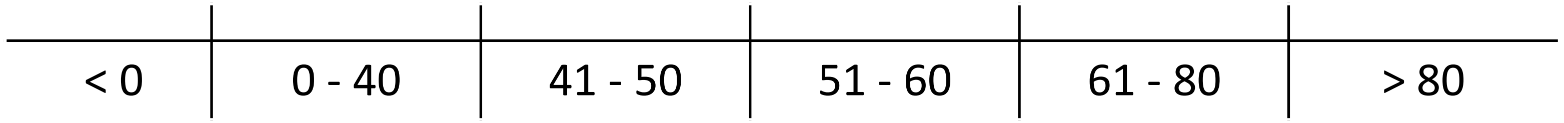
Input Space Partitioning

- **Step 4 : Identifying values from each block**
  - Choose appropriate values from each block

Test case no.	input (w)	Output (obesityLevel)
1	-10	Invalid weight
2	20	Extremely underweight
3	45	Underweight
4	55	Normal
5	70	Overweight
6	90	Extremely overweight

Choosing Test Cases: Boundary Values

- Errors often occur at boundary values of each block. So we should include the test cases for the boundary values.
  - Boundary values: -1, 0, 40, 41, 50, 51, 60, 61, 80, 81
  - Interior values: -10, 20, 45, 55, 70, 90



Test case no.	input (w)	Output (obesityLevel)
1	-10	Invalid weight
2	-1	Invalid weight
3	0	Extremely underweight
4	20	Extremely underweight
5	40	Extremely underweight
6	41	Underweight
…	…	…

Strategies for Identifying Values from Each Block

- Strategies for identifying values :
  - Include valid, invalid and special values
  - Sub-partition some blocks
  - Explore boundaries of domains
  - Include values that represent “normal use”

Interface vs Functionality-Based

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Interface-Based Approach

Characteristics :

- list is null (block1 = true, block2 = false)
- list is empty (block1 = true, block2 = false)

Functionality-Based Approach

Characteristics :

- number of occurrences of **element** in list: **0, 1, >1**
- **element** occurs **first** in list: **true, false**
- **element** occurs **last** in list: **true, false**

Example: Obesity Calculator v2.0

- Function **obesityCalc(int w, int h)** return **obesityLevel**
  - **Precondition**:
    - **w** is an integer of a weight in Kg, -1000 ≤ w ≤ 1000
    - **h** is an integer of height in cm, -1000 ≤ h ≤ 1000
  - **Postcondition**:
    - **obesityLevel** is a string of the obesity level determined from **w and h** according to the following table.

Example: Obesity Calculator v2.0

Obesity Chart						
	< 110	110 - 160	161 - 170	171 - 185	185 - 200	> 200
< 0	invalid	invalid	invalid	invalid	invalid	invalid
0 – 40	invalid	euw	euw	euw	euw	invalid
41 – 50	invalid	normal	uw	euw	euw	invalid
51 – 60	invalid	ow	normal	uw	euw	invalid
61 – 80	invalid	eow	ow	normal	uw	invalid
81 – 100	invalid	eow	eow	ow	normal	Invalid
100 – 110	invalid	eow	eow	eow	ow	Invalid
> 110	invalid	eow	eow	eow	eow	invalid

Example: FindTriType

- **Function FindTriType(int side1, int side2, int side3)**
  return string triType
  - **Precondition: side1**, **side2**, and **side3** are three integers, representing the length of each side of a triangle
  - **Postcondition: triType** is a string of the type of the given triangle. Possible values are:
    - "scalene“
    - "isosceles“
    - "equilateral",
    - "not a valid triangle"

Interface-Based IDM – FindTriType

- FindTriType had one testable function and three integer inputs

First Characterization of FindTriType's Inputs

Characteristics	b1	b2	b3
q_1 = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
q_2 = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
q_3 = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

- A maximum of 3*3*3 = 27 tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests …

Interface-Based IDM – FindTriType (*cont*)

Second Characterization of FindTriType's Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "Refinement of q_2"	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "Refinement of q_3"	greater than 1	equal to 1	equal to 0	less than 0

- A maximum of 4*4*4 = 64 tests
- This is complete because the inputs are integers (0 . . 1)

Possible values for partition q_1

Parameter	b_1	b_2	b_3	b_4
side1	5	1	0	-5

Functionality-Based IDM – FindTriType

- First two characterizations are based on syntax–parameters and their type
- A semantic level characterization could use the fact that the three integers represent a triangle

Geometric Characterization of FindTriType's

Characteristics	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

- Oops … something's fishy … equilateral is also isosceles !
- We need to refine the example to make characteristics valid

Correct Geometric Characterization of FindTriType's Inputs

Characteristics	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

Functionality-Based IDM – FindTriType (*cont*)

- Values for this partitioning can be chosen as

Possible values for geometric partition q_1

Input	b_1	b_2	b_3	b_4
(side1, side2, side3)	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Functionality-Based IDM – FindTriType (*cont*)

- A different approach would be to break the geometric characterization into four separate characteristics

Four Characteristics for FindTriType

Characteristic	b_1	b_2
q_1 = "Scalene"	True	False
q_2 = "Isosceles"	True	False
q_3 = "Equilateral"	True	False
q_4 = "Valid"	True	False

- Use constraints to ensure that
  - Equilateral = True implies Isosceles = True
  - Valid = False implies Scalene = Isosceles = Equilateral = False

Input Space Partitioning

In case there are more than one characteristics:

- **Step 5 : Choose how to combine values**
  - A test input has a value for each parameter
  - One block for each characteristic
  - Choosing all combinations is usually infeasible
  - Coverage criteria allow subsets to be chosen

Choosing Combinations of Values

- The most obvious criterion is to choose all combinations …

All Combinations (ACoC) : All combinations of blocks from all characteristics must be used.

- Number of tests is the product of the number of blocks in each characteristic :

$$\prod_{i=1}^{Q}(B_i)$$

- The second characterization of TriTyp results in 4*4*4 = 64 tests – too many ?

ISP Criteria – Each Choice

- 64 tests for FindTriType is almost certainly way too many
- One criterion comes from the idea that we should try at least one value from each block

Each Choice (EC) : One value from each block for each characteristic must be used in at least one test case.

- Number of tests is the number of blocks in the largest characteristic

$$\mathrm{Max}_{i=1}^{Q}(B_i)$$

For TriTyp: 2, 2, 2

1, 1, 1

0, 0, 0

-1, -1, -1

ISP Criteria – Pairwise

- Each choice yields few tests – cheap but perhaps ineffective
- Another approach asks values to be combined with other values

Pairwise (PW) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

- **Number of tests is at least the product of two largest characteristics**

$$(\mathrm{Max}_{i=1}^{Q}(B_i))^2$$

For TriTyp:	**2, 2, 2**	**2, 1, 1**	**2, 0, 0**	**2, -1, -1**
	1, 2, 1	**1, 1, 0**	**1, 0, -1**	**1, -1, 2**
	0, 2, 0	**0, 1, -1**	**0, 0, 2**	**0, -1, 1**
	-1, 2, -1	**-1, 1, 2**	**-1, 0, 1**	**-1, -1, 0**

ISP Criteria –T-Wise

- A natural extension is to require combinations of *t* values instead of *2*

t-Wise (TW) : A value from each block for each group of t characteristics must be combined.

- Number of tests is at least the product of *t* largest characteristics
- If all characteristics are the same size, the formula is

$$\left(\mathrm{Max}_{i=1}^{Q}(B_i)\right)^t$$

- If *t* is the number of characteristics Q, then all combinations
- That is … *Q-wise = AC*
- *t*-wise is expensive and benefits are not clear

ISP Criteria – Base Choice

- Testers sometimes recognize that certain values are important
- This uses domain knowledge of the program

Base Choice (BC) : A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

- **Number of tests is one base test + one test for each other block**

$$1 + \sum_{i=1}^{Q} (B_i - 1)$$

For TriTyp: Base	2, 2, 2	2, 2, 1	2, 1, 2	1, 2, 2
		2, 2, 0	2, 0, 2	0, 2, 2
		2, 2, -1	2, -1, 2	-1, 2, 2

ISP Criteria – Multiple Base Choice

- Testers sometimes have more than one logical base choice

Multiple Base Choice (MBC) : One or more base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choices in each other characteristic.

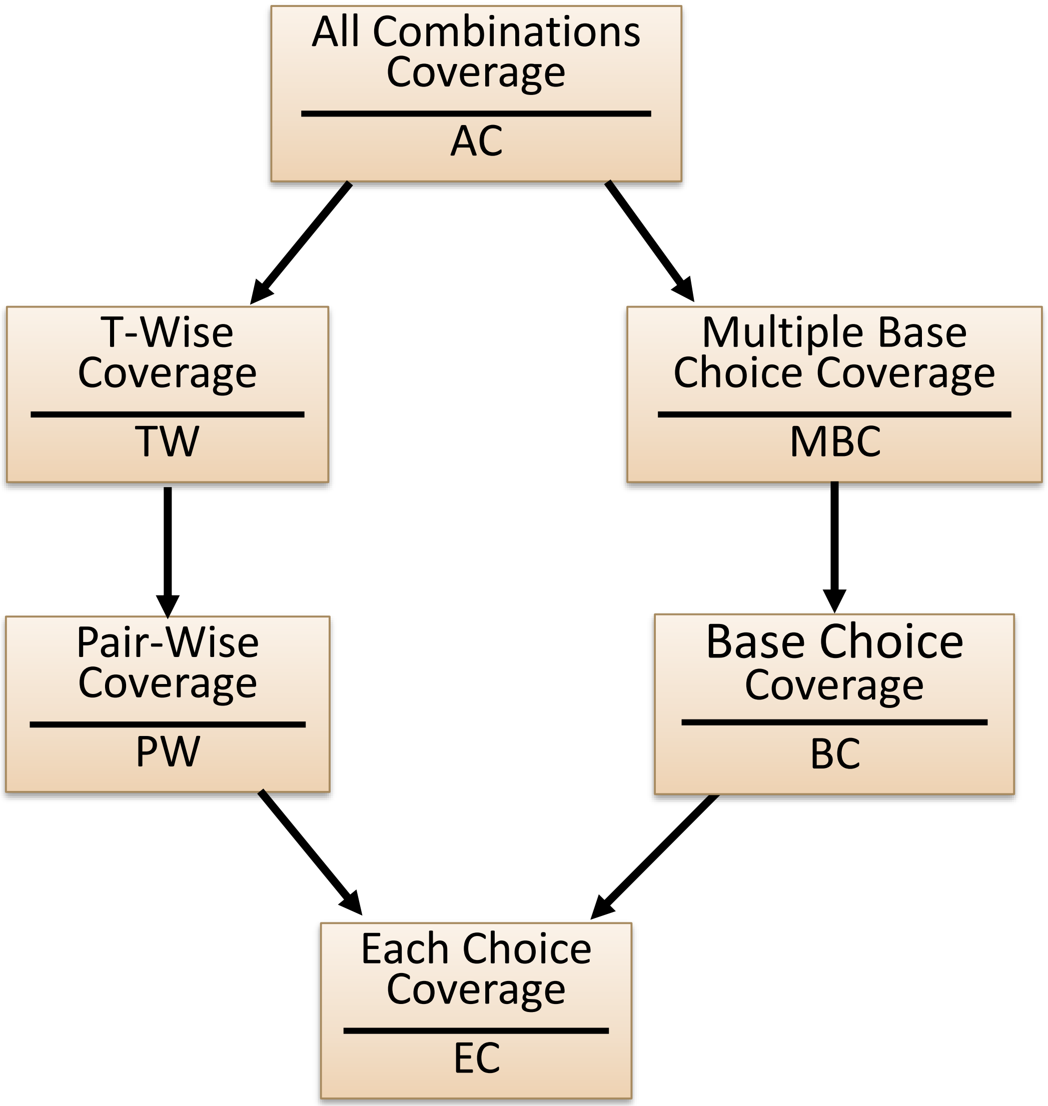
- If there are *M* base tests and m_i base choices for each characteristic:

$$\mathbf{M} + \sum_{i=1}^{Q} (\mathbf{M} * (\mathbf{B}_i - \mathbf{m}_i))$$

For TriTyp: Base

2, 2, 2	2, 2, 0	2, 0, 2	0, 2, 2
	2, 2, -1	2, -1, 2	-1, 2, 2
1, 1, 1	1, 1, 0	1, 0, 1	0, 1, 1
	1, 1, -1	1, -1, 1	-1, 1, 1

ISP Coverage Criteria Subsumption



Constraints Among Characteristics

- Some combinations of blocks are infeasible
  - “less than zero” and “scalene” … not possible at the same time
- These are represented as constraints among blocks
- Two general types of constraints
  - A block from one characteristic cannot be combined with a specific block from another
  - A block from one characteristic can ONLY BE combined with a specific block form another characteristic
- Handling constraints depends on the criterion used
  - AC, PW, TW : Drop the infeasible pairs
  - BC, MBC : Change a value to another non-base choice to find a feasible combination

Input Space Partitioning Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation

Simple, straightforward, effective, and widely used in practice