

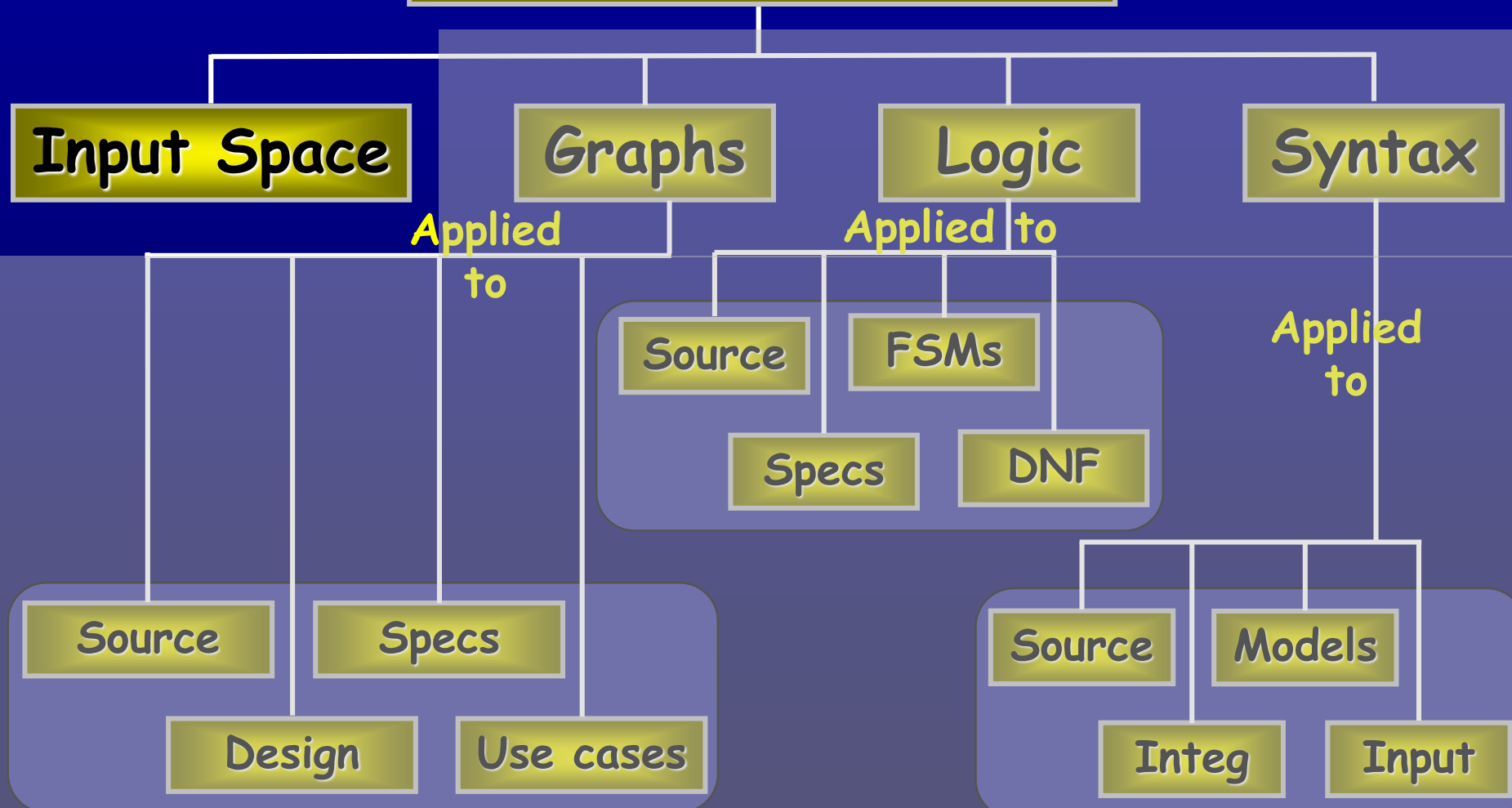
# **Introduction to Software Testing Chapter 6 Input Space Partition Testing**

Paul Ammann & Jeff Offutt

<https://www.cs.gmu.edu/~offutt/softwaretest/>

# Ch. 6 : Input Space Coverage

## Four Structures for Modeling Software



# Overview: Recall from Chapter 5

- Describe the **input domain** of the software
  - Identify **input** parameters
  - Partition each input into **finite sets** of representative values
  - Choose **combinations** of values
- **Example**
  - Parameters  $F(\text{int } X, \text{int } Y)$
  - Possible values  $X: \{ <0, 0, 1, 2, >2 \}, Y: \{ 10, 20, 30 \}$
  - Tests
    - $F(-5, 10), F(0, 20), F(1, 30), F(2, 10), F(5, 20)$

# Benefits of ISP

- Equally **applicable** at several levels of testing
  - Unit
  - Integration
  - System
- Easy to apply with **no automation**
- Can **adjust** the procedure to get more or fewer tests
- No **implementation knowledge** is needed
  - Just the input space

# Input Domains

- **Input domain**: all possible inputs to a program
  - Most input domains are so large that they are effectively **infinite**
- **Input parameters** define the scope of the input domain
  - Parameter values to a method
  - Data from a file
  - Global variables
  - User inputs
- We **partition** input domains into **regions** (called *blocks*)
- Choose at least **one value** from each block

Input domain: Alphabetic letters

Partitioning characteristic: Case of letter

- Block 1: upper case
- Block 2: lower case

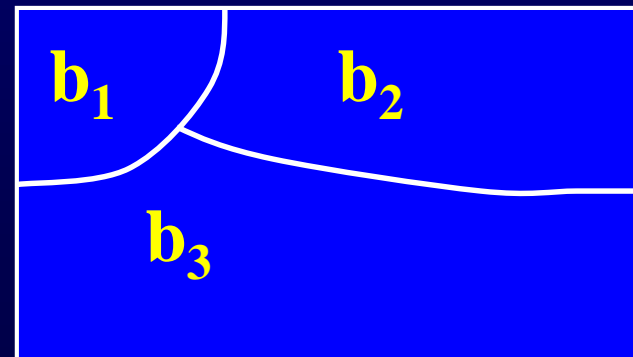
# Partitioning Domains

- Domain  $D$
- Partition scheme  $q$  of  $D$
- The partition  $q$  defines a set of blocks,  $B_q = b_1, b_2, \dots, b_Q$
- The partition must satisfy two **properties** :
  1. Blocks must be **pairwise disjoint** (no overlap)

$$b_i \cap b_j = \Phi, \forall i \neq j, b_i, b_j \in B_q$$

2. Together the blocks **cover** the domain  $D$  (complete)

$$\bigcup_{b \in B_q} b = D$$



# What is a characteristic?

“A feature or quality belonging typically to a person, place, or thing and serving to identify it.”

Input: people

Characteristics: hair color, major

Blocks:

A=(red, black, brown, blonde, other)

B=(cs, swe, ce, math, ist, other)

Abstraction:

A = [ a1, a2, a3, a4, a5 ]

B = [ b1, b2, b3, b4, b5, b6 ]

concrete  
level

abstract  
level

# Examples

- Example **characteristics**
  - Whether X is null
  - Order of the list F (sorted, inverse sorted, arbitrary, ...)
  - Input device (DVD, CD, VCR, computer, ...)
  - Hair color, height, major, age
- **Partition** characteristic into blocks
  - Blocks may be single-value or a set of values
  - Each value in a block should be **equally useful** for testing
- Each abstract test has one block from each characteristic



# Choosing Partitions

- Defining **partitions** is not hard, but is easy to get wrong
- Consider the “*order of elements in list F*”

$b_1$  = sorted in ascending order  
 $b_2$  = sorted in descending order  
 $b_3$  = arbitrary order

but ... something's fishy ...

Length 1 : [ 14 ]

The list will be in all three blocks ...  
That is, disjointness is not satisfied

Solution:

Two characteristics that address just one property

C1: List F sorted ascending

- c1.b1 = true
- c1.b2 = false

C2: List F sorted descending

- c2.b1 = true
- c2.b2 = false

# Modeling the input domain

- Step 1 : Identify testable functions
  - Step 2 : Find all inputs, parameters, & characteristics
  - Step 3 : Model the input domain
  - Step 4 : Apply a test criterion to choose combinations of values (6.2)
  - Step 5 : Refine combinations of blocks into test inputs
- 
- Concrete level
- Move from imp level to design abstraction level
- Entirely at the design abstraction level
- Back to the implementation level

# Steps 1 & 2

Identify testable functions

Find inputs, parameters, characteristics

# Two Approaches to Input Domain Modeling (IDM)

## 1. Interface-based approach

- Develops characteristics directly from individual input parameters
- Mechanically consider each parameter in isolation
- This is an easy modeling technique and relies mostly on syntax
- Ignores semantic relationships among parameters

## 2. Functionality-based approach

- Develops characteristics from a behavioral view (or functionality) of the program under test
- Incorporates semantic knowledge (i.e., relationships among parameters)
- Harder to develop—requires more design effort
- May result in better tests, or fewer tests that are as effective

# Example IDM (syntax)

- Method *triang()* from class *TriangleType* on the book website :
  - <https://www.cs.gmu.edu/~offutt/softwaretest/java/Triangle.java>
  - <https://www.cs.gmu.edu/~offutt/softwaretest/java/TriangleType.java>

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }  
public static Triangle triang (int Side1, int Side2, int Side3)  
// Side1, Side2, and Side3 represent the lengths of the sides of a triangle  
// Returns the appropriate enum value
```

IDM for each parameter is identical

Characteristic : *Relation of side with zero*

Blocks: negative; positive; zero

three characteristic each has three blocks9 requirement

**Relationship of variables  
with special values  
(zero, null, blank, ...)**

# Example IDM (behavior)

- Method *triang()* again :

The three parameters represent a *triangle*

The IDM can combine all parameters

Characteristic : *Type of triangle*

Blocks: Scalene; Isosceles; Equilateral; Invalid

# Steps 1 & 2—IDM

```
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
```

## Parameters and Characteristics

Two parameters : **list**, **element**

Characteristics based on syntax :

**list** is null (block1 = true, block2 = false)

**list** is empty (block1 = true, block2 = false)

Characteristics based on behavior :

A feasible requirement:

number of occurrences of **element** in list

(0, 1, >1)

**element** occurs **first** in list

(true, false)

**element** occurs **last** in list

(true, false)

num reqs: 485 characteristicseach charactrist

first, middle, last: overlapping criteriadisjointness

# Step 3

Model input domain

Partition characteristics into blocks

Choose values for blocks



# triang(): Relation of side with zero

- 3 inputs, each has the same partitioning

Characteristic	$b_1$	$b_2$	$b_3$
$q_1$ = "Relation of Side 1 to 0"	positive	equal to 0	negative
$q_2$ = "Relation of Side 2 to 0"	positive	equal to 0	negative
$q_3$ = "Relation of Side 3 to 0"	positive	equal to 0	negative

- Maximum of  $3*3*3 = 27$  tests
- Some triangles are **valid**, some are **invalid**
- **Refining** the characterization can lead to more tests ...

# Refining triang()'s IDM

## Second Characterization of triang()'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	negative
$q_2$ = "Refinement of $q_2$ "	greater than 1	equal to 1	equal to 0	negative
$q_3$ = "Refinement of $q_3$ "	greater than 1	equal to 1	equal to 0	negative

- Maximum of  $4*4*4 = 64$  tests = num requirements
- **Complete** only because the inputs are integers (0 .. 1)

### Values for partition $q_1$

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
Side 1	2	1	0	-1

Test boundary conditions

# triang() : Type of triangle

## Geometric Characterization of *triang()*'s Inputs

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

What's wrong with this partitioning?

disjointness is violated

- Equilateral is also isosceles !
- We need to **refine** the example to make characteristics valid

## Correct Geometric Characterization of *triang()*'s Inputs

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
$q_1$ = "Geometric Classification"	scalene	isosceles, <u>not equilateral</u>	equilateral	invalid

# Values for triang()

Possible values for geometric partition  $q_i$

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

# Yet another triang() IDM

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

## Four Characteristics for *triang()*

Characteristic	$b_1$	$b_2$
$q_1 = \text{"Scalene"}$	True	False
$q_2 = \text{"Isosceles"}$	True	False
$q_3 = \text{"Equilateral"}$	True	False
$q_4 = \text{"Valid"}$	True	False

- Use **constraints** to ensure that

num requirements =  $2^4 = 16$

- **Equilateral = True** implies **Isosceles = True** otherwise the test requirement is infeasible.
- **Valid = False** implies **Scalene = Isosceles = Equilateral = False**

# Constraints For triang() IDM

- Complete list of **constraints** in this example:
  - **Scalene = True** implies **Isosceles = Equilateral = False** and **Valid = True**
  - **Scalene = False** does not imply anything
  - **Isosceles = True** implies **Scalene = False** and **Valid = True**
  - **Isosceles = False** implies **Equilateral = False**
  - **Equilateral = True** implies **Scalene = False** and **Isosceles = Valid = True**
  - **Equilateral = False** does not imply anything
  - **Valid = False** implies **Scalene = Isosceles = Equilateral = False**
  - **Valid = True** does not imply anything

Characteristic	$b_1$	$b_2$
$q_1 = \text{"Scalene"}$	True	False
$q_2 = \text{"Isosceles"}$	True	False
$q_3 = \text{"Equilateral"}$	True	False
$q_4 = \text{"Valid"}$	True	False

# IDM hints (1)

- More characteristics → more tests
- More blocks → more tests
- Do **not** use program source prevent syntax, use semantics (behaviour)
  - Use **specifications** or other documentation instead of program code to develop characteristics
  - The tester should apply ISP by using **domain knowledge** about the problem, not the implementation
  - In practice, the code may be all that is available
  - Overall, the **more semantic information** the test engineer can incorporate into characteristics, the better the resulting test set is likely to be

# IDM hints (2)

- Design **more characteristics** with **fewer blocks**
  - Fewer mistakes
  - Fewer tests
  - More likely to satisfy the disjointness and completeness
- **Partition** and choose **values** strategically
  - Valid, invalid, special values (e.g., null, zero, empty, ...)
  - Explore boundaries
  - Balance the number of blocks in the characteristics



# Strategies for partitioning and choosing values (1)

- Valid, invalid and special values
  - Every partition must allow all values, whether valid or invalid. Special values should be used
- Sub-partition
  - A range of valid values can often be partitioned into sub-partitions (each sub-partition represents a different part of the functionality)
- Boundaries
  - Values at or close to boundaries often cause problems (stress test)
- Normal use (happy path)
  - Include values that represent normal use. If the operation focuses heavily on “normal use,” the failure rate depends on values that are not boundary conditions

# Strategies for partitioning and choosing values (2)

- **Enumerated types**
  - A partition where blocks are a discrete, enumerated set often makes sense. The triangle example uses this approach
- **Balance**
  - Try to balance the number of blocks in each characteristic
  - It may be cheap or even free to add more blocks to characteristics that have fewer blocks
  - The number of tests sometimes depends on the characteristic with the maximum number of blocks
- **Missing blocks and Overlapping blocks**
  - Check for completeness and disjointness

# Using More than One IDM

- Some programs may have dozens or even hundreds of parameters
- Create **several** small IDMs than one large
  - A divide-and-conquer approach
- Different parts of the software can be tested with different amounts of **rigor** (varying levels of **coverage**)
  - For instance, one IDM may contain only valid values and another IDM may contain invalid values to focus on error handling
  - The valid value IDM may be covered using a higher level of coverage. The invalid value IDM may use a lower level of coverage
- It is okay if the different IDMs **overlap**
  - The same variable may appear in more than one IDM

disjointness in between characteristicsintr

# Modeling the input domain

- Step 1 : Identify testable functions
  - Step 2 : Find all inputs, parameters, & characteristics
  - Step 3 : Model the input domain
- Move from imp level to design abstraction level
- Step 4 : Apply a test criterion to choose combinations of values (6.2)
- Entirely at the design abstraction level
- Step 5 : Refine combinations of blocks into test inputs
- Back to the implementation level

# Step 4 – Choosing combinations of values (6.2)

- After partitioning characteristics into blocks, testers design tests by **combining** blocks from different characteristics
  - 3 Characteristics (abstract): A, B, C
  - Abstract blocks: A = [a1, a2, a3, a4]; B = [b1, b2]; C = [c1, c2, c3] 24 test requirements
- A test starts by combining one block from each characteristic
  - Then values are chosen to satisfy the combinations
- We use **criteria** to choose **effective** combinations

# All combinations criterion (ACoC)

The most obvious criterion is to choose all combinations

**All Combinations (ACoC) : Test with all combinations of blocks from all characteristics.**

a1 b1 c1	a2 b1 c1	a3 b1 c1	a4 b1 c1
a1 b1 c2	a2 b1 c2	a3 b1 c2	a4 b1 c2
a1 b1 c3	a2 b1 c3	a3 b1 c3	a4 b1 c3
a1 b2 c1	a2 b2 c1	a3 b2 c1	a4 b2 c1
a1 b2 c2	a2 b2 c2	a3 b2 c2	a4 b2 c2
a1 b2 c3	a2 b2 c3	a3 b2 c3	a4 b2 c3

# All combinations criterion (ACoC)

- Number of tests is the product of the number of blocks in each characteristic :  
$$\prod_{i=1}^Q (B_i)$$
- The syntax characterization of triang()
  - Each side: >1, 1, 0, <1 (four blocks)
  - Results in  $4*4*4 = 64$  tests
- Most form invalid triangles
  - Only 8 are valid (all sides greater than zero)

How can we get fewer tests ?

# Example

Input: students

Characteristics: Level, Mode, Major, Classification

Blocks:

num requirements for all combinations approach: 24

Level: ( grad, undergrad )

Mode: ( full-time, part-time )

Major: ( cs, swe, other )

Classification: ( in-state, out-of-state )

Abstract IDM:

$A = [ a1, a2 ]$      $C = [ c1, c2, c3 ]$

$B = [ b1, b2 ]$      $D = [ d1, d2 ]$



# In-class exercise

All combinations criterion (ACoC)

Consider this abstract IDM

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];

C = [c1, c2, c3]; D = [d1, d2]

How many tests are needed to satisfy ACoC?

# In-class exercise (*answer*)

All combinations criterion (ACoC)

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];

C = [c1, c2, c3]; D = [d1, d2]

Number of tests:  $2 \times 2 \times 3 \times 2 = 24$

a1 b1 c1 d1	a1 b2 c1 d1	a2 b1 c1 d1	a2 b2 c1 d1
a1 b1 c1 d2	a1 b2 c1 d2	a2 b1 c1 d2	a2 b2 c1 d2
a1 b1 c2 d1	a1 b2 c2 d1	a2 b1 c2 d1	a2 b2 c2 d1
a1 b1 c2 d2	a1 b2 c2 d2	a2 b1 c2 d2	a2 b2 c2 d2
a1 b1 c3 d1	a1 b2 c3 d1	a2 b1 c3 d1	a2 b2 c3 d1
a1 b1 c3 d2	a1 b2 c3 d2	a2 b1 c3 d2	a2 b2 c3 d2

# ISP criteria – each choice

- We should try at least one value from each block

**Each Choice Coverage (ECC)**: Use at least one value from each block for each characteristic in at least one test case.

- Number of tests is the number of blocks in the largest characteristic :  $\text{Max}_{i=1}^Q (B_i)$

\*\*\* All selected reqs must be feasible in this approach. \*\*\*

# In-class exercise

Each choice criterion (ECC)

Apply ECC to our previous example

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];

C = [c1, c2, c3]; D = [d1, d2]

1. How many tests are needed for ECC?
2. Design the (abstract) tests

3

# In-class exercise (*answer*)

Each choice criterion (ECC)

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];

C = [c1, c2, c3]; D = [d1, d2]

Number of tests:  $\max(2, 2, 3, 2) = 3$

a1 b1 c1 d1

a2 b2 c2 d2

a1 b1 c3 d1

# Shortcomings of ACoC and ECC

- ECC does not yield very effective tests
- It can be called a relatively “**weak**” criterion
  - The weakness of ECC can be expressed as not requiring values to be combined with other values
- A natural approach is to require explicit combinations of values
  - ACoC combine values “**blindly**”, without regard for which values are being combined
- The next criterion strengthens ECC in a different
  - bringing in a small but crucial piece of **domain knowledge** of the program; asking what is the most “**important**” block for each partition
  - This block is called the **base choice**

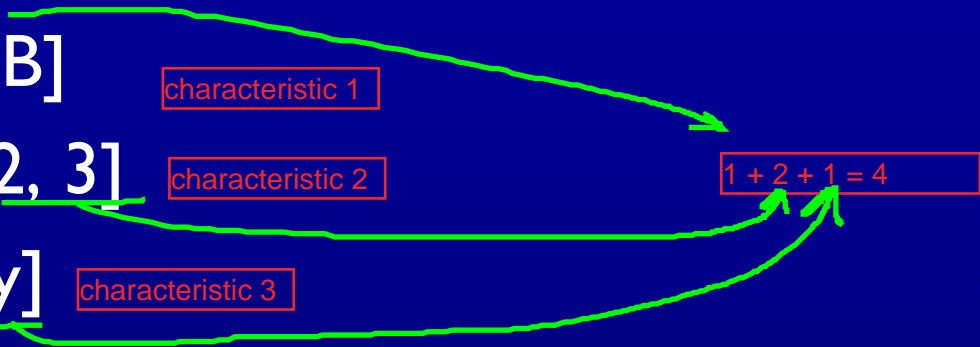
# ISP criteria – base choice (BCC)

- ECC is simple, but very few tests
- The base choice criterion recognizes :
  - Some blocks are more important than others
  - Using diverse combinations can strengthen testing
- Lets testers bring in domain knowledge of the program

**Base Choice Coverage (BCC) :** Choose a base choice block for each characteristic. Form a base test by using the base choice for each characteristic. Choose subsequent tests by holding all but one base choice constant and using each non-base choice in each other characteristic. base case must be feasible

- Number of tests is one base test + one test for each other block  $1 + \sum_{i=1}^Q (B_i - 1)$

# Base Choice Example

- Suppose we have **three partitions** with blocks:
  - [A, B] characteristic 1
  - [I, 2, 3] characteristic 2
  - [x, y] characteristic 3

The diagram illustrates the combination of three partitions. Each partition is underlined in green. Red boxes labeled 'characteristic 1', 'characteristic 2', and 'characteristic 3' are placed next to [A, B], [I, 2, 3], and [x, y] respectively. Green arrows originate from the underlined partitions and point to a red box containing the equation  $1 + 2 + 1 = 4$ .
- Suppose base choice blocks are 'A', 'I' and 'x'
- Then the base choice test is (A, I, x)
- And the following additional tests would be needed:
  - ❖ (B, I, x)
  - ❖ (A, 2, x)(A, 3, x)
  - ❖ (A, I, y)



# Base choice notes

- The base test must be **feasible**
  - That is, all base choices must be **compatible**
- **Base choices** can be
  - Most likely from an end-use point of view ex: > 1 for triangle
  - Simplest
  - Smallest block
  - First block in some ordering
- **Happy path** tests often make good base choices
- The base choice is a **crucial design** decision
  - Test designers should **document** why the choices were made

# In-class exercise

Base choice criterion (BCC)

Apply BCC to our previous example

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];  
C = [c1, c2, c3]; D = [d1, d2]

1. How many tests are needed for BCC?
2. Pick base values and write one base test
3. Design the remaining (abstract) tests

a1, b1, c2, d1

5 + 1 (bcc)

a1, b1, c3, d1

# In-class exercise (*answer*)

## Base choice criterion (BCC)

4 Characteristics: A, B, C, D

Abstract blocks: A = [a1, a2]; B = [b1, b2];

C = [c1, c2, c3]; D = [d1, d2]

Number of tests:  $1(\text{base}) + 1 + 1 + 2 + 1 = 6$

ex: in case of mbc:a1b1c1 c2d1 result: 2 test requirements for

Base	a1 b1 c1 d1
A	<b>a2</b> b1 c1 d1
B	a1 <b>b2</b> c1 d1
C	a1 b1 <b>c2</b> d1
C	a1 b1 <b>c3</b> d1
D	a1 b1 c1 <b>d2</b>

# ISP criteria – multiple base choice

- We sometimes have more than one logical base choice

Multiple Base Choice Coverage (MBCC) : Choose at least one, and possibly more, base choice blocks for each characteristic. Form base tests by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.

- If  $M$  base tests and  $m_i$  base choices for each characteristic:

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

For our example: Two base tests: a1, b1, c1, d1    a2, b2, c2, d2

Tests from a1, b1, c1, d1: a1, b1, c3, d1

Tests from a2, b2, c2, d2: a2, b2, c3, d2

# Triang Example

- Consider the “second characterization” of Triang as given before:

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

- For convenience, we relabel the blocks:

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
A	A1	A2	A3	A4
B	B1	B2	B3	B4
C	C1	C2	C3	C4

# triang() – ACoC Tests

A1 B1 C1	A2 B1 C1	A3 B1 C1	A4 B1 C1
A1 B1 C2	A2 B1 C2	A3 B1 C2	A4 B1 C2
A1 B1 C3	A2 B1 C3	A3 B1 C3	A4 B1 C3
A1 B1 C4	A2 B1 C4	A3 B1 C4	A4 B1 C4
A1 B2 C1	A2 B2 C1	A3 B2 C1	A4 B2 C1
A1 B2 C2	A2 B2 C2	A3 B2 C2	A4 B2 C2
A1 B2 C3	A2 B2 C3	A3 B2 C3	A4 B2 C3
A1 B2 C4	A2 B2 C4	A3 B2 C4	A4 B2 C4
A1 B3 C1	A2 B3 C1	A3 B3 C1	A4 B3 C1
A1 B3 C2	A2 B3 C2	A3 B3 C2	A4 B3 C2
A1 B3 C3	A2 B3 C3	A3 B3 C3	A4 B3 C3
A1 B3 C4	A2 B3 C4	A3 B3 C4	A4 B3 C4
A1 B4 C1	A2 B4 C1	A3 B4 C1	A4 B4 C1
A1 B4 C2	A2 B4 C2	A3 B4 C2	A4 B4 C2
A1 B4 C3	A2 B4 C3	A3 B4 C3	A4 B4 C3
A1 B4 C4	A2 B4 C4	A3 B4 C4	A4 B4 C4

ACoC yields  
 $4*4*4 = 64$  tests  
for Triang!

This is almost  
certainly more  
than we need

Only 8 are valid  
(all sides greater  
than zero)

# triang() – ECC Tests

- Number of tests is the number of blocks in the largest characteristic :  $\text{Max}_{i=1}^Q(B_i)$

For *triang()* : A1, B1, C1

A2, B2, C2

A3, B3, C3

A4, B4, C4

Substituting values: 2, 2, 2

1, 1, 1

0, 0, 0

-1, -1, -1

# triang() – BCC Tests

- Number of tests is one base test + one test for each other block  $1 + \sum_{i=1}^Q (B_i - 1)$

For <i>triang()</i> : <u>Base</u>	A1, B1, C1	A1, B1, C2	A1, B2, C1	A2, B1, C1
		A1, B1, C3	A1, B3, C1	A3, B1, C1
		A1, B1, C4	A1, B4, C1	A4, B1, C1

- Each parameter for triang() has three characteristics with four blocks, thus BCC requires  $1 + 3 + 3 + 3$  tests



# triang() – MBCC Tests

- If **M** base tests and **m<sub>i</sub>** base choices for each characteristic:

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

For *triang()* : Bases

A1, B1, <u>C1</u>	A1, B1, C3	A1, B3, C1	A3, B1, C1
	A1, B1, C4	A1, B4, C1	A4, B1, C1
A2, B2, <u>C2</u>	A2, B2, C3	A2, B3, C2	A3, B2, C2
	A2, B2, C4	A2, B4, C2	A4, B2, C2

- In this example:
- With  $M = 2$ ,  $B_i = 4$ , and  $m_i = 2 \forall i, 1 \leq i \leq 3$ :
  - Number of tests:  $2 + (2*(4-2)) + (2*(4-2)) + (2*(4-2)) = 14$

# triang() – MBCC Tests

- Another example:
- For triang(), we may choose to include two base choices for side 1, “greater than 1” (A1) and “equal to 1” (A2)
- With  $M = 2$ ,  $m1 = 2$ , and  $m_i = 1 \forall i, 2 \leq i \leq 3$ :
  - Number of tests:  $2 + (2*(4-2)) + (2*(4-1)) + (2*(4-1)) = 18$

## For triang() : Bases

A1, B1, C1	A1, B1, C2	A1, B1, C3	A1, B1, C4
	A1, B2, C1	A1, B3, C1	A1, B4, C1
	A3, B1, C1	A4, B1, C1	
duplicate	A2, B1, C1	A2, B1, C2	A2, B1, C3
	A2, B1, C4		
	A2, B2, C1	A2, B3, C1	A2, B4, C1
	A3, B1, C1	A4, B1, C1	

# triang() – MBCC Tests

- The MBCC criterion sometimes results in duplicate tests
- For example, (A3, B1, C1) and (A4, B1, C1) both appear twice for triang ()
- Duplicate test cases should be eliminated (which makes the formula for the number of tests an upper bound)

## For triang() : Bases

**A1, B1, C1   A1, B1, C2   A1, B1, C3   A1, B1, C4**

**A1, B2, C1   A1, B3, C1   A1, B4, C1**

This doesn't subsume pair-wise because some pairs

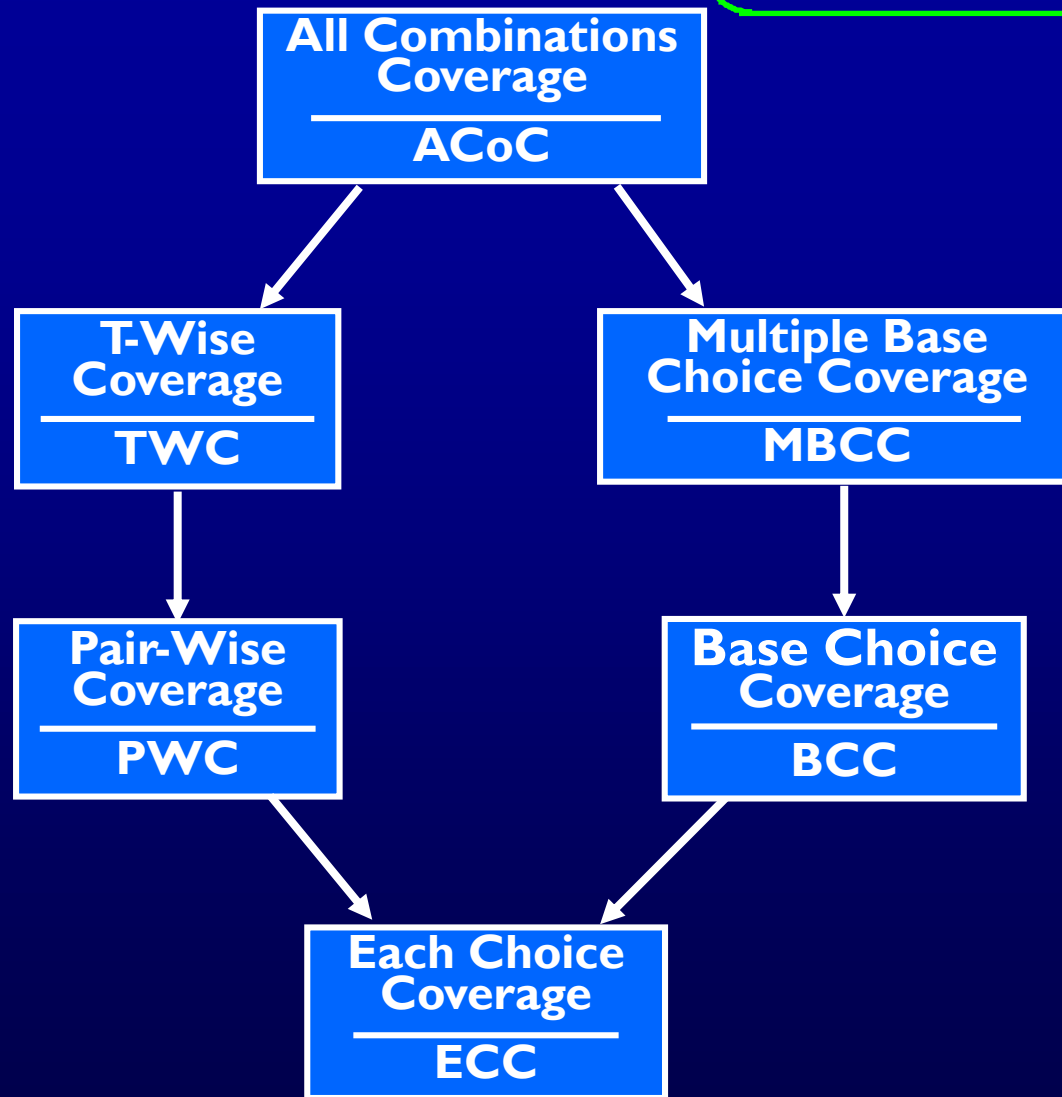
**A3, B1, C1   A4, B1, C1**

**A2, B1, C1   A2, B1, C2   A2, B1, C3   A2, B1, C4**

**A2, B2, C1   A2, B3, C1   A2, B4, C1**

**A3, B1, C1   A4, B1, C1**

# ISP Coverage Criteria Subsumption



# Constraints Among Characteristics (6.3)

- 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 from another characteristic
- Handling constraints depends on the criterion used
  - **ACC, PWC, TWC** : Drop the infeasible pairs
  - **BCC, MBCC** : Change a value to another non-base choice to find a feasible combination

# Example of Constraints

```
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
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	One element	More than one, unsorted	More than one, sorted <u>all identical</u>	More than one, all identical
B : match	element not found	element found once	element found more than once	
Invalid combinations : (A1, B3), (A4, B2)				

element **cannot** be in a one-element list more than once

If the list only has one element, but it appears multiple times, we **cannot** find it just once

# Example Handling Constraints

- For BCC and MBCC:
- If a particular variation (for example, “less than zero” for “Relation of Side 1 to zero”) conflicts with the base case (for example, “scalene” for “Geometric Classification”)
  - We can change the choice for the base case to make the test requirement feasible
  - In this case, “Geometric Classification” can be changed to “invalid”

base test

ex: a4 b1 c1 d1 is infeasible to make it feasible

a	q <sub>1</sub> = “Refinement of q <sub>1</sub> ”	greater than 1	equal to 1	equal to 0	less than 0
b	q <sub>2</sub> = “Refinement of q <sub>2</sub> ”	greater than 1	equal to 1	equal to 0	less than 0
c	q <sub>3</sub> = “Refinement of q <sub>3</sub> ”	greater than	equal to 1	equal to 0	less than 0
d	q <sub>4</sub> = “Geometric Classification”	scalene	isosceles, not equilateral	equilateral	invalid

# 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**



# **FURTHER READING PAIR-WISE & T-WISE**

# ISP Criteria – Pair-Wise

**Pair-Wise Coverage (PWC) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.**

# Pair-Wise (An Example)

- Given the example of three partitions with blocks [A, B], [1, 2, 3], and [x, y]:
- PWC needs tests to cover the following 16 combinations:

(A, 1)	(B, 1)	(1, x)
(A, 2)	(B, 2)	(1, y)
(A, 3)	(B, 3)	(2, x)
(A, x)	(B, x)	(2, y)
(A, y)	(B, y)	(3, x)
		(3, y)

16 test requirements

- PWC allows the same test case to cover more than one unique pair of values. So the above combinations can be combined in several ways, including:

(A, 1, x)	(B, 1, y)
(A, 2, x)	(B, 2, y)
(A, 3, x)	(B, 3, y)
(A, -, y)	(B, -, x)

We can also do it by only 6 requirements, not 16

- '-' means that any block can be used

# ISP Criteria –T-Wise

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

**t-Wise Coverage (TWC) : A value from each block for each group of  $t$  characteristics must be combined.**

- If  $t$  is the number of characteristics  $Q$  (i.e.,  $t = Q$ , where  $Q$  is the number of characteristics), then all combinations
  - That is ...  $Q$ -wise = AC
- $t$ -wise is **expensive** and benefits are not clear
  - Experience suggests going beyond pair-wise (that is,  $t = 2$ ) does not help much