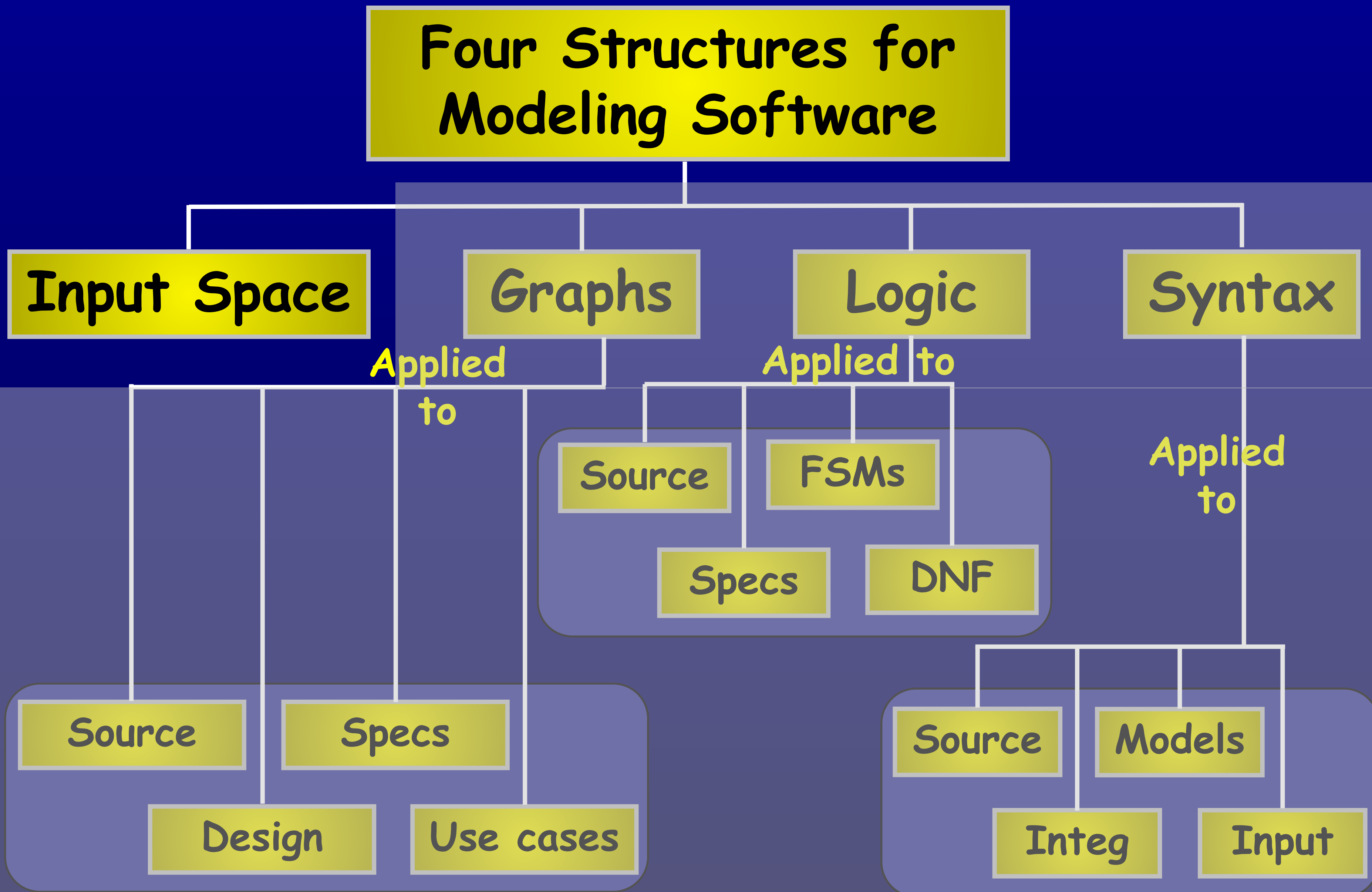
# Introduction to Software Testing
# Chapter 6
# Input Space Partition Testing

Paul Ammann & Jeff Offutt

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

# Ch. 6 : Input Space Coverage



# 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

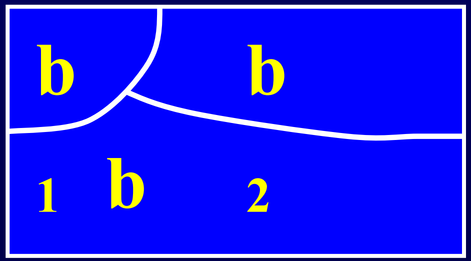
# Partitioning Domains

- *Domain D*
- *Partition scheme q* of *D*
- The partition *q* defines a *set of blocks*, $Bq = b_1, b_2, \ldots, b_Q$
- 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)

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



# What is a characteristic?

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

Input: people

concrete level

Characteristics: hair color, major

Blocks:

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

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

abstract level

Abstraction:

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

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

# Examples

- Example characteristics
  - Whether X is null
  - Order of the list F (sorted, inverse sorted, arbitrary, …)
  - Min separation of two aircraft
  - 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

Concrete level

- 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 abstraction level

# Steps 1 & 2

Identify testable functions

Find inputs, parameters, characteristics

# 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

# 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 :

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

# 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
- Complete only because the inputs are integers (0 . . 1)

| Values for partition $q_1$ | | | | |
|---|---|---|---|---|
| Characteristic | $b_1$ | $b_2$ | $b_3$ | $b_4$ |
| Side1 | 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?*

- 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, not equilateral | equilateral | invalid |

# Values for triang()

| Possible values for geometric partition $q_1$ | | | | |
|---|---|---|---|---|
| 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

| <u>Four</u> Characteristics for *triang*() | | |
|---|---|---|
| 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

# IDM hints

- More characteristics □ more tests
- More blocks □ more tests
- Do not use program source
- Design more characteristics with fewer blocks
  - Fewer mistakes
  - Fewer tests
- Choose values strategically
  - Valid, invalid, special values
  - Explore boundaries
  - Balance the number of blocks in the characteristics

# 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 abstraction 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]
- 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

**<u>All Combinations</u> (<u>ACoC</u>) : 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
  - Results in 4*4*4 = 64 tests
- Most form invalid triangles

How can we get fewer tests ?

# Example

Input: students

Characteristics: Level, Mode, Major, Classification

Blocks:

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*2*3*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 : $\mathrm{Max}_{i=1}^{Q}(B_i)$

# 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

# 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 |

# 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

**<u>Base Choice Coverage</u> (<u>BCC</u>) : 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.**

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

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

# 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
  - Simplest
  - Smallest
  - First 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

# 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(base)+1+1+2+1 = 6

| | |
|---|---|
| Base | a1 b1 c1 d1 |
| A | **a2** b1 c1 d1 |
| B | a1 **b2** c1 d1 |
| C | a1 b1 **c2** d1 |
| C | a1 b1 **c3** d1 |
| D | a1 b1 c1 **d2** |

# 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:

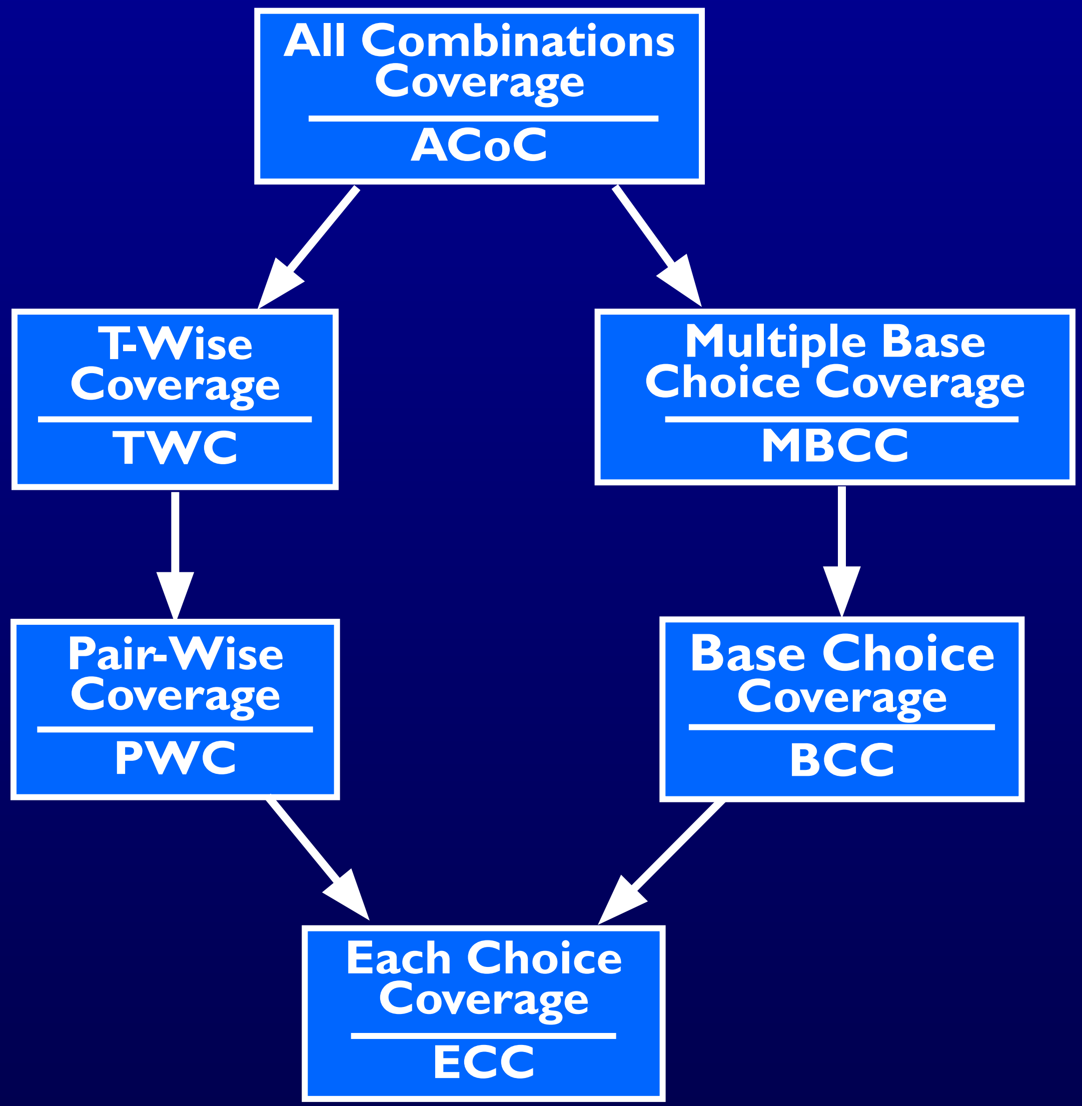
$$M + \sum_{i=1}^{Q} (M * (B_i - m_i))$$

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

# 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 form 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 Handling 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 | 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**

# 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**