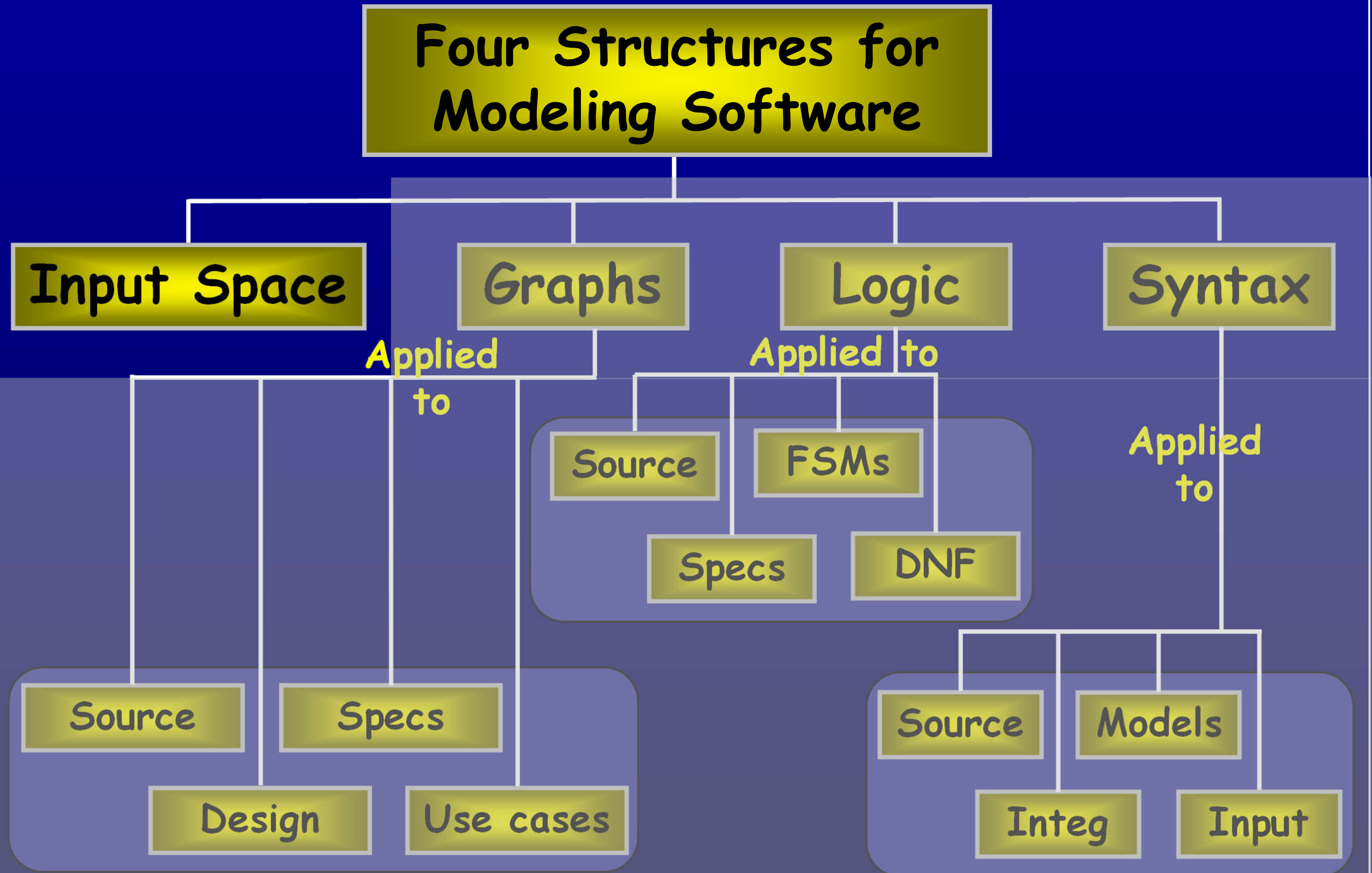
CS 5154

Input Space Partitioning

Owolabi Legunsen

The following are modified versions of the publicly-available slides for Chapter 6 in the Ammann and Offutt Book, “Introduction to Software Testing” (http://www.cs.gmu.edu/~offutt/softwaretest)

1st of four structures we'll cover



Why Input Space Partitioning?

- No implementation knowledge is needed
  - Just the input space
- Easy to apply without automation
- Can adjust the procedure to get more or fewer tests
- Equally applicable at several levels of testing
  - Unit, Integration, System, etc.

Recommended Reading

Empir Software Eng (2014) 19:558–581
DOI 10.1007/s10664-012-9229-5

An industrial study of applying input space partitioning to test financial calculation engines

Jeff Offutt · Chandra Alluri

Published online: 23 September 2012

Editor: James Miller

Input Domains and ISP

- 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
- ISP: First partition input domain into regions (called *blocks*)
  - values in each block are assumed equally useful for testing
- ISP: Then 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

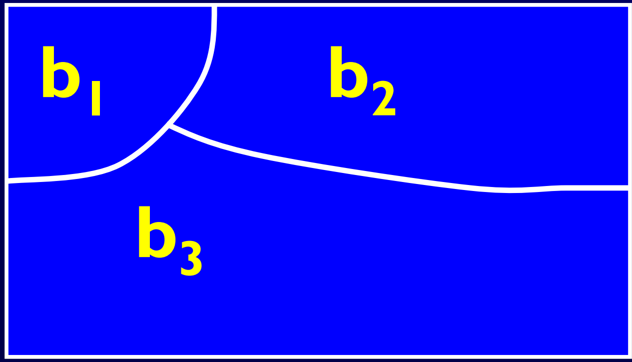
Partitioning Domains

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

$$\boxed{b_i \cap b_j = \emptyset, \forall\, i \neq j, b_i, b_j \in B_q}$$

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

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



In-Class Exercise

Design a partitioning for all integers

That is, partition integers into blocks such that each block seems to be equivalent in terms of testing

Make sure your partition is valid:

1) *Pairwise disjoint*
2) *Complete*

Example partition of all integers

Input Domain: all integers

Partitioning Characteristic: Sign

(1) Sign: (+ve, -ve, zero), (2) oddity [even or odd or neither]

(3) primes/non-primes

Block 1:

(4) Mod classes

Block 2:

(5) overflow/underflow

block 3:

Using Partitions – Assumptions

- Choose a value from each block
  - Each value is assumed to be equally useful for testing

- Forming partitions
  - Find characteristics of the inputs : case of letter, relationship to 0, parameters, semantic descriptions, …
  - Partition each characteristic into blocks
  - Choose tests by combining values from blocks

Using Partitions – Characteristics

- 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

Choosing Partitions

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

b_1 = sorted in ascending order
b_2 = sor … der
b_3 = arb … order

Design blocks for that characteristic

but … something's fishy …

What if the list is of length 0 or 1?

The list … blocks
That is, disjointness is not satisfied

Can you find the problem?

Solution:
Two characteristics that each address just one property

Can you think of a solution?

C1: … ng
- c1.b1 = true
- c1.b2 = false

C2: List F sorted descending
- c2.b1 = true
- c2.b2 = false

Thinking about Partitions

- If the partitions are not complete or disjoint, that means the partitions have not been considered carefully enough
- They should be reviewed carefully, like any design
- Different alternatives should be considered
- Input domain modeling happens in five steps …
  - Steps 1&2: move from implementation level to abstraction level
  - Steps 3&4: entirely at the abstraction level
  - Step 5: move back to the implementation level

Input domain modelling (step 1)

- Identify testable functions
  - Individual methods have one testable function
  - Methods in a class often have the same characteristics
  - Programs have more complicated characteristics—modeling documents such as UML can be used to design characteristics
  - Systems of integrated hardware and software components can use devices, operating systems, hardware platforms, browsers, etc.

Input domain modelling (step 2)

- Find all the parameters
  - Often straightforward, even mechanical
  - Important to be complete → thorough
  - Methods : Parameters and state (non-local) variables used
  - Components : Parameters to methods and state variables
  - System : All inputs, including files and databases

Input domain modelling (step 3)

- Model the input domain
  - The domain is scoped by the parameters
  - The structure is defined in terms of characteristics
  - Each characteristic is partitioned into sets of blocks
  - Each block represents a set of values
  - This is the most creative design step in using ISP

Input domain modelling steps 4&5

- Step 4: Use a criterion to choose combinations of 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
- Step 5 : Refine combinations of blocks into test inputs
  - Choose appropriate values from each block

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 a behavioral view of the program
   - Harder to develop—requires more design effort
   - May result in better tests, or fewer tests that are as effective

***Input Domain Model* (IDM)**

Interface-Based IDM

- Mechanically consider each parameter in isolation
- An easy modeling technique, 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 IDM Example

- Consider method *triang()* from class *TriangleType* :
  - http://www.cs.gmu.edu/~offutt/softwaretest/java/Triangle.java
  - http://www.cs.gmu.edu/~offutt/softwaretest/java/TriangleType.java

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

The IDM for each parameter is identical

Characteristic : *Relation of side with zero*

Blocks: negative; positive; zero

Functionality-Based IDM

- Find characteristics corresponding to intended functionality
- Requires more design effort from tester
- Can incorporate domain and semantic knowledge
- Can use relationships among parameters
- Model can be based on requirements, not implementation
- The same parameter may appear in multiple characteristics, so it's harder to translate values to test cases

Functionality-Based IDM

- Again, consider method *triang()* from class *TriangleType* :

The three parameters represent a *triangle*

The IDM can combine all parameters

Characteristic : *Type of triangle*

Blocks: Scalene; Isosceles; Equilateral; Invalid

Steps 1&2—Identifying functionalities, parameters, characteristics

- A creative engineering step
- More characteristics means more tests
- Interface-based : Translate parameters to characteristics
- Candidates for characteristics :
  - Preconditions and postconditions (may be encoded as exceptions)
  - Relationships among variables (aliasing, equality, …)
  - Relationship of variables with special values (zero, null, blank, …)

Steps 1&2—Identifying functionalities, parameters, characteristics (contd)

- Do not use program source—characteristics should be based on the input domain
  - Program source should be used with graph or logic criteria
- Better to have more characteristics with few blocks
  - Fewer mistakes and fewer tests
- Better to have more semantic information in the IDM
  - Likely to produce better tests

In-Class Exercise

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

Create two IDMs for findElement () :
1) Interface-based
2) Functionality-based

– list size
– list nullity
element nullity
– elem in list?

Steps 1 & 2—Interface & 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

Two parameters : list, element

Characteristics :

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

Functionality-Based Approach

Two parameters : list, element

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)

Step 3: Modeling the input domain

- Partitioning characteristics into blocks and values is a very creative engineering step
- More blocks means more tests
- Partitioning often flows directly from the definition of characteristics and both steps are done together
  - Once should evaluate them separately
  - Sometimes fewer characteristics can be used with more blocks and vice versa

Modeling the input domain (2)

- Some strategies for identifying values :
  - Include valid, invalid and special values
  - Sub-partition some blocks
  - Explore boundaries of domains
  - Include values that represent "normal use" (happy path ☺)
  - Try to balance the number of blocks in each characteristic
  - Check for completeness and disjointness

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

<u>Second</u> 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

Functionality-Based IDM—*triang*()

- Values for this partitioning can be chosen as

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)

Functionality-Based IDM—*triang*()

- 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 = "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

Using More than One IDM

- Some programs may have dozens or even hundreds of parameters
- Create several small IDMs
  - A divide-and-conquer approach
- Different parts of the software can be tested with different amounts of rigor
  - For example, some IDMs may include a lot of invalid values
- It is okay if the different IDMs overlap
  - The same variable may appear in more than one IDM

In-Class Exercise

What two properties must be satisfied for an input domain to be properly partitioned?

Step 4 – Choosing Combinations of Values (6.2)

- Once characteristics and partitions are defined, the next step is to choose test values
- We use criteria – to choose effective subsets
- 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 : $\prod_{i=1}^{Q}(B_i)$
- Second characterization of triang() gives 4*4*4 = 64 tests
  - Too many ?

ISP Criteria – All Combinations

- Consider again "second characterization" of Triang:

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 using abstractions:

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

ISP Criteria – 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)

ISP Criteria – Each Choice

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

<u>Each Choice Coverage (ECC)</u> : 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 : $\mathbf{Max}_{i=1}^{Q}(B_i)$

For *triang*() : A1, B1, C1
A2, B2, C2
A3, B3, C3
A4, B4, C4

Write down ECC tests
Use the abstract labels (A1,A2, …)

Substituting values: 2, 2, 2
1, 1, 1
0, 0, 0
-1, -1, -1

Suggest values …

ISP Criteria – Pair-Wise

- Each choice yields few tests—cheap but maybe ineffective
- Another approach combines values with other values

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.

- Number of tests is at least the product of two largest characteristics $(\text{Max}_{i=1}^{Q}(B_i)) * (\text{Max}_{j=1, j!=i}^{Q}(B_j))$

For *triang*() :

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

Write down PWC tests
Use the abstract labels
(Hint: Should be 16 tests)

ISP Criteria –T-Wise

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

<u>t-Wise Coverage (TWC)</u> : 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

<u>Base Choice Coverage (BCC)</u> : 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 *triang*() : <u>Base</u>				
	A1, B1, C1	A1, B1, C2	A1, B2, C1	A2, B1, C1
		A1, B1, C3	A1, B3, C1	A3, B1, C1
Write down BCC tests		A1, B1, C4	A1, B4, C1	A4, B1, C1

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-user 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

ISP Criteria – Multiple Base Choice

- We sometimes have more than one logical base choice

<u>Multiple Base Choice Coverage (MBCC)</u> : At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed 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.

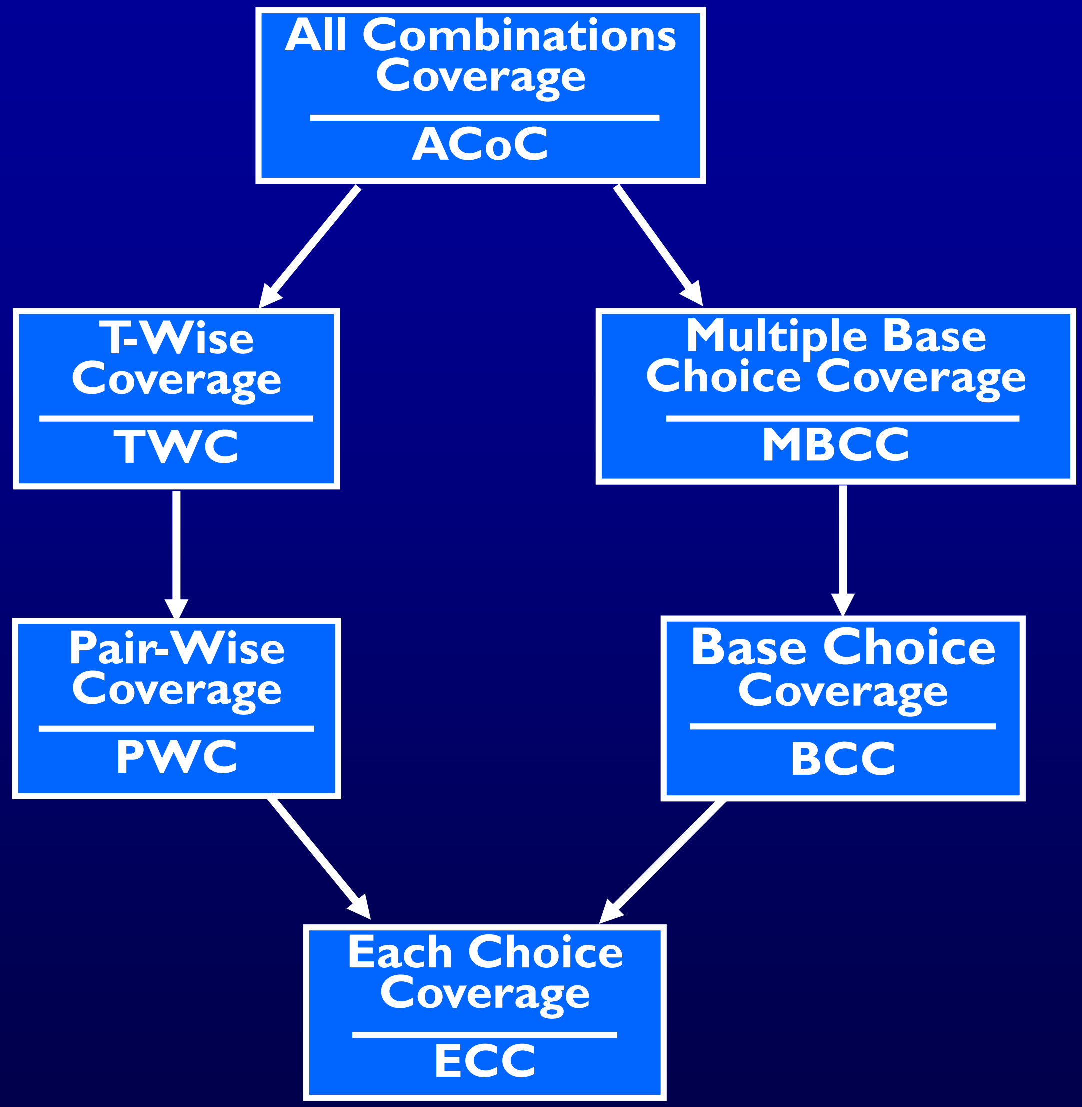
- **If *M* base tests and m_i base choices for each characteristic:**

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

For *triang*() : <u>Bases</u>

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

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