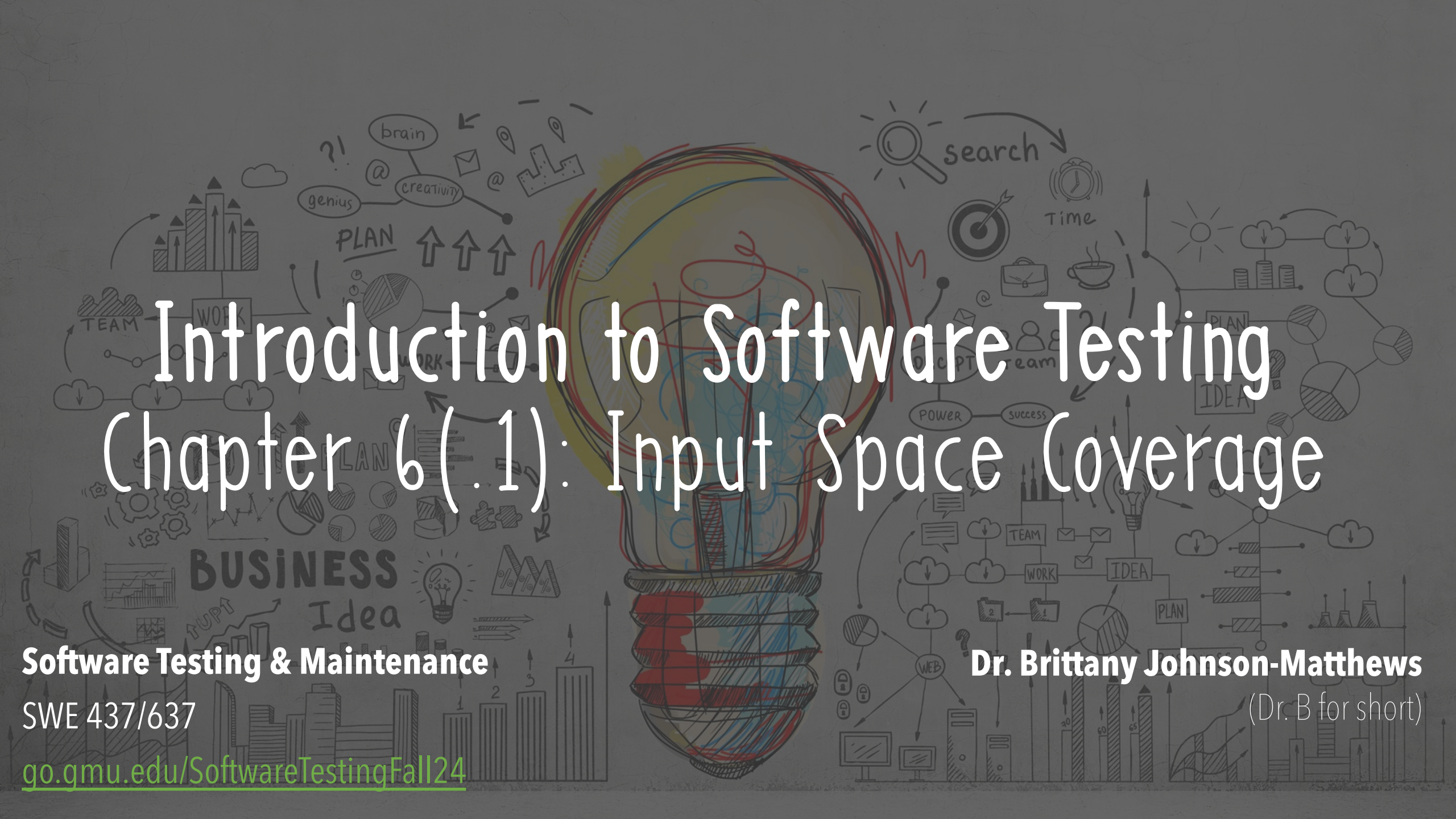


Introduction to Software Testing
Chapter 6(.1): Input Space Coverage

Software Testing & Maintenance

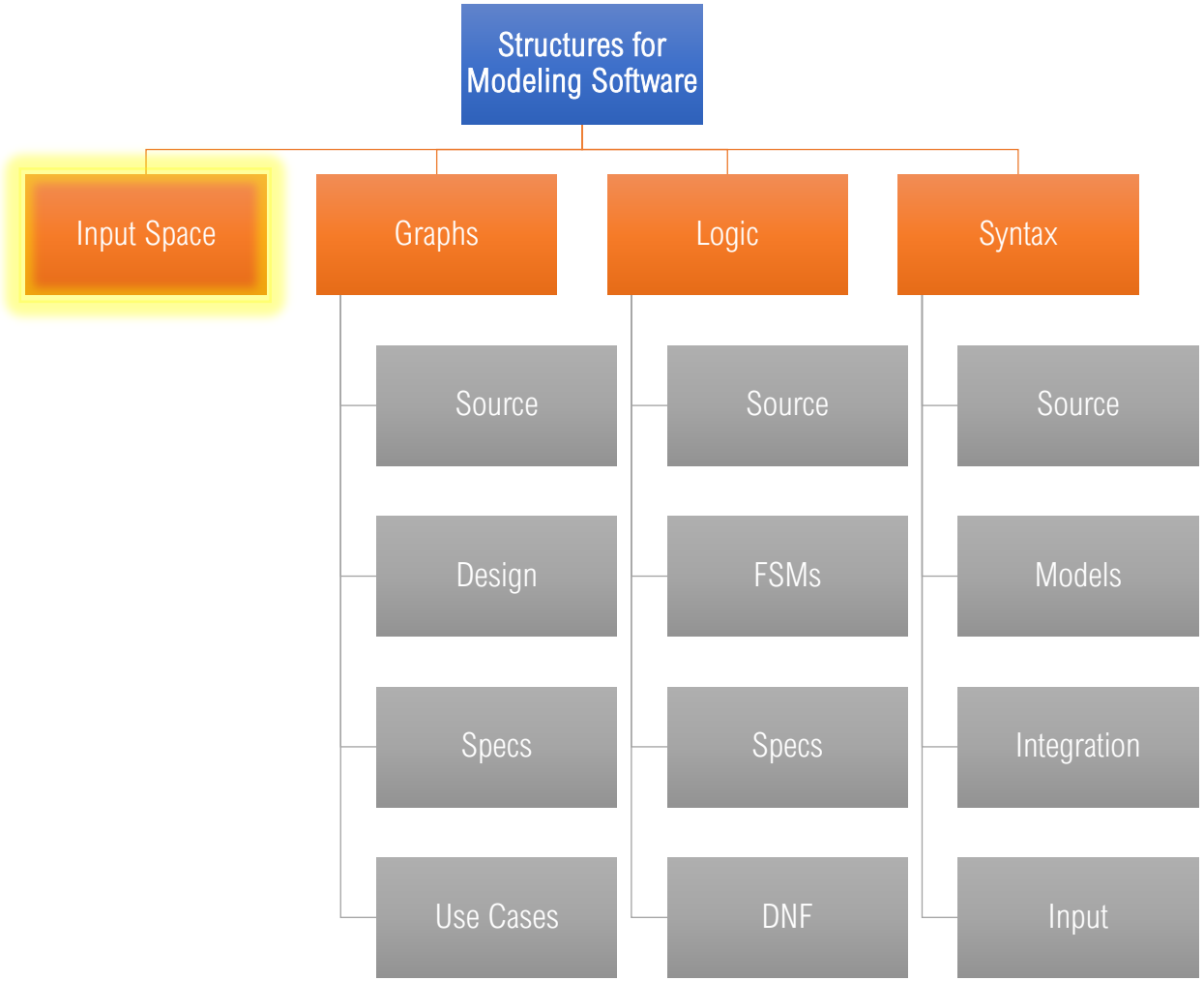
SWE 437/637

go.gmu.edu/SoftwareTestingFall24

Dr. Brittany Johnson-Matthews

(Dr. B for short)

Modeling Software



Benefits of Input Space Coverage

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 essentially **infinite**

Input parameters define 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 input domains

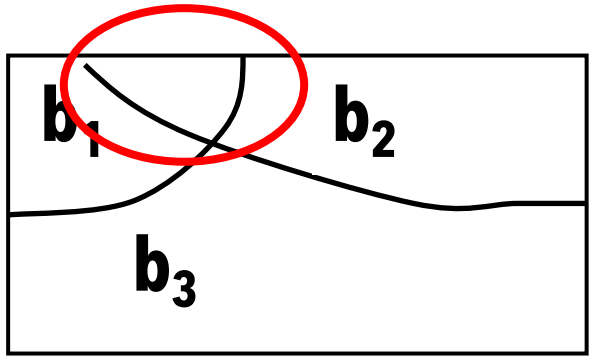
Domain ***D***

Partition scheme ***q*** *of* ***D***

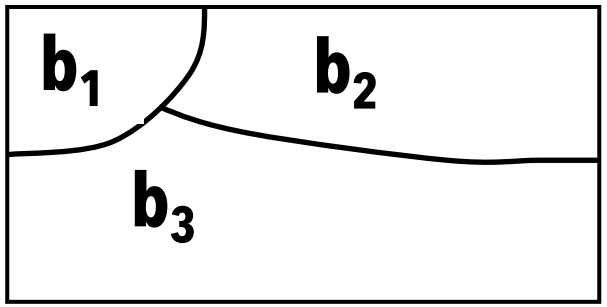
The partition ***q*** defines a set of blocks, $\boldsymbol{Bq = b_1, b_2, \ldots, b_q}$

The partition must satisfy two **properties**:

1. Blocks must be ***pairwise disjoint*** (no overlap)



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



Input Characteristics

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

Input: *people*

Concrete

Characteristics: hair color, major
Blocks:
A = (1) red, (2) black, (3) brown, (4) blonde, (5) other
B = (1) cs, (2) swe, (3) ce, (4) math, (5) ist, (6) other

Abstraction

A = [a1, a2, a3, a4, a5]
B = [b1, b2, b3, b4, b5, b6]

Characteristics & Partitions

More 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

- Each value in a block should be **equally useful** for testing

Choose a **value** from each block

Form tests by combining one value from each characteristic

Choosing partitions

Defining **partitions** is not hard, but is easy to get wrong.

Consider the characteristic "***order of elements in list F***"

Design blocks for that characteristic

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

but ... something's fishy ...

Length 1 : [14]

Can you spot the problem?

This list is in all three blocks

That is, disjointness is not satisfied

Can you think of a solution?

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

Creating an Input Domain Model (IDM)

Step 1: Identify testable functions

Step 2: Find all **inputs, parameters, & characteristics**

Move from imp level to design abstraction level

Step 3: Model the **input domain**

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

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:

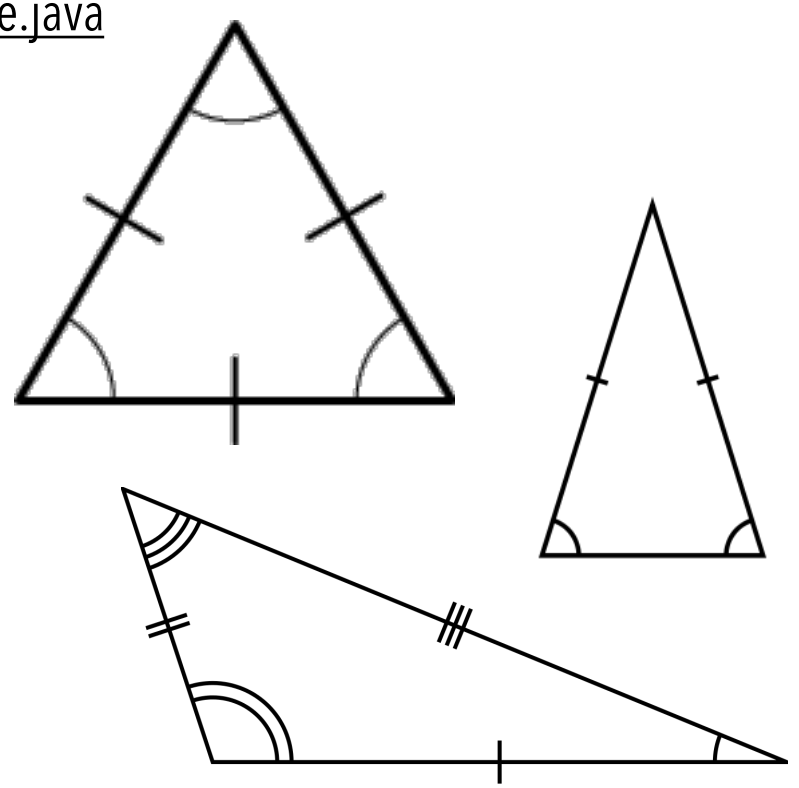
- https://www.cs.gmu.edu/~offutt/softwaretest/java/Triangle.java
- https://www.cs.gmu.edu/~offutt/softwaretest/java/TriangleType.java

Three parameters represent a *triangle*

The IDM can combine all parameters

Characteristic: *type of triangle*

Blocks: Scalene; Isosceles; Equilateral; Invalid



Steps 1 & 2

Identify testable functions

Find inputs, parameters, characteristics

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 like UML can be used to design characteristics

Systems of integrated hardware and software components can have many testable functions – devices, operating systems, hardware platforms, browsers, etc.

Step 2 – Parameters & Characteristics

Often straightforward or mechanical

- Preconditions and postconditions
- Relationships among variables
- Special values (zero, null, etc.)

Do not use program source code, **characteristics should be based on the *input domain***

Methods: parameters and state variables
Components: parameters to methods and state variables
Systems: all inputs, including files and databases

In-class Exercise

Functions, parameters, and characteristics

```
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
```

Identify functionalities, parameters, and characteristics for *findElement()*

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 <u>parameters</u> : list, element

<u>Characteristics</u> based on *syntax* :

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

<u>Characteristics</u> 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 the input domain

The domain is scoped by the *parameters*

The structure is defined by *characteristics*

Each characteristic is partitioned into *sets of blocks*

Each block represents a *set of values*

This is the most creative design step in ISP

- Better to have **more characteristics and fewer blocks**; leads to fewer tests
- Strategies include valid/invalid/special values, boundary values, "normal" values

`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

```
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
```

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

Characteristic	b_1	b_2	b_3	b_4
Side1	5	1	0	-5

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

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 can also be isosceles!

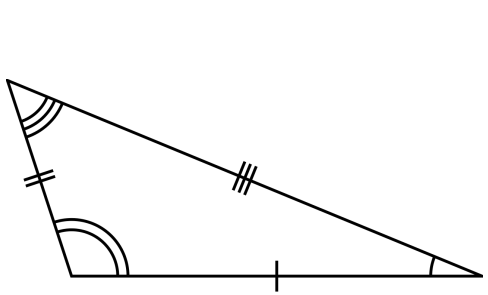
We need to **refine** the example to make partitioning valid

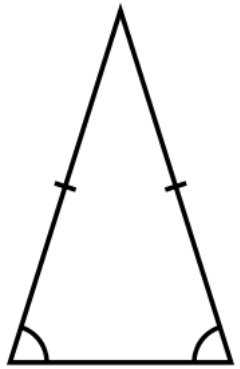
Correct geometric characterizations of *triang()'s inputs*

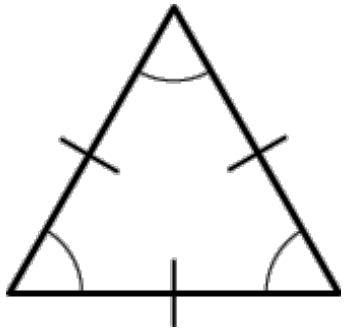
Characteristic	B_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	Isosceles, not equilateral	equilateral	invalid

Values for `triang()`

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 = "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**

Advice for creating IDMs

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

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

In-class Exercise

*Creating an **Input Domain Model (IDM)***

Pick one of the programs from Chapter 1 (findLast, numZero, etc).

Create an IDM for the program you chose.

Modeling the input domain

Step 1: Identify testable functions

Step 2: Find all **inputs, parameters, & characteristics**

Move from imp level to design abstraction level

Step 3: Model the **input domain**

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

Modeling the input domain

Step 1: Identify testable functions

Step 2: Find all **inputs, parameters, & characteristics**

Move from imp level to design abstraction level

Step 3: Model the **input domain**

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

After partitioning characteristics into blocks, testers design tests by combining blocks from different characteristics

- 3 characteristics: A, B, C
- Three blocks each: A = a1, a2, a3; B = b1, b2, b3; 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.

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

a1 b1 c1	a2 b1 c1	a3 b1 c1
a1 b1 c2	a2 b1 c2	a3 b1 c2
a1 b1 c3	a2 b1 c3	a3 b1 c3
a1 b2 c1	a2 b2 c1	a3 b2 c1
a1 b2 c2	a2 b2 c2	a3 b2 c2
a1 b2 c3	a2 b2 c3	a3 b2 c3
a1 b3 c1	a2 b3 c1	a3 b3 c1
a1 b3 c2	a2 b3 c2	a3 b3 c2
a1 b3 c3	a2 b3 c3	a3 b3 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

In-class Exercise

All Combinations Criterion *(ACoC)*

Consider our previous example.

3 characteristics: A, B, C

Three blocks each: A = a1, a2, a3; B = b1, b2, b3; C = c1, c2, c3

How many tests do we need to satisfy ACoC?

In-class Exercise

All Combinations Criterion *(ACoC)*

Consider our previous example.

3 characteristics: A, B, C

Three blocks each: A = a1, a2, a3; B = b1, b2, b3; C = c1, c2, c3

How many tests do we need to satisfy ACoC?

3 * 3 * 3 = **27 tests**

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?

ISP Criteria – Each Choice (ECC)

We should try **at least one** value from each block

> Each Choice Coverage(ECC) : 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:

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

In-class Exercise

Each Choice Criterion *(ECC)*

3 characteristics: A, B, C

Three blocks each: A = a1, a2, a3; B = b1, b2, b3; C = c1, c2, c3

1. *How many tests do we need (with ECC)?*
2. *Write the (abstract) tests.*

In-class Exercise

Each Choice Criterion *(ECC)*

3 characteristics: A, B, C

Three blocks each: A = a1, a2, a3; B = b1, b2, b3; C = c1, c2, c3

1. *How many tests do we need (with ECC)?*
   Max # of blocks is 3 → **minimum of 3 tests**
2. *Write the (abstract) tests.*

In-class Exercise

Each Choice Criterion *(ECC)*

3 characteristics: A, B, C

Three blocks each: A = a1, a2, a3; B = b1, b2, b3; C = c1, c2, c3

1. *How many tests do we need (with ECC)?*
   Max # of blocks is 3 → **minimum of 3 tests**
2. *Write the (abstract) tests.*
   (a1, b1, c1); (a2, b2, c2); (a3, b3, c3)

ISP Criteria – Base Choice (BCC)

ECC is **simple**, but very few tests

The **base choice criterion** recognizes that

-Some blocks are more **important** than others

-Using **diverse combinations** can strengthen testing

Let testers bring in **domain knowledge** of the program

> Base Choice Coverage(BCC) : 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 "non-base" other block:

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

Base choice considerations

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)*

Write BCC tests for our previous example.

3 characteristics: A, B, C

Three blocks each: A = a1, a2, a3; B = b1, b2, b3; C = c1, c2, c3

1. *How many tests do we need?*
2. *Pick base values and write one base test*
3. *Write remaining tests*

ISP Criteria – Multiple Base Choice (MBCC)

We sometimes have **more than one** logical base choice

> Multiple Base Choice Coverage (MBCC) 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:

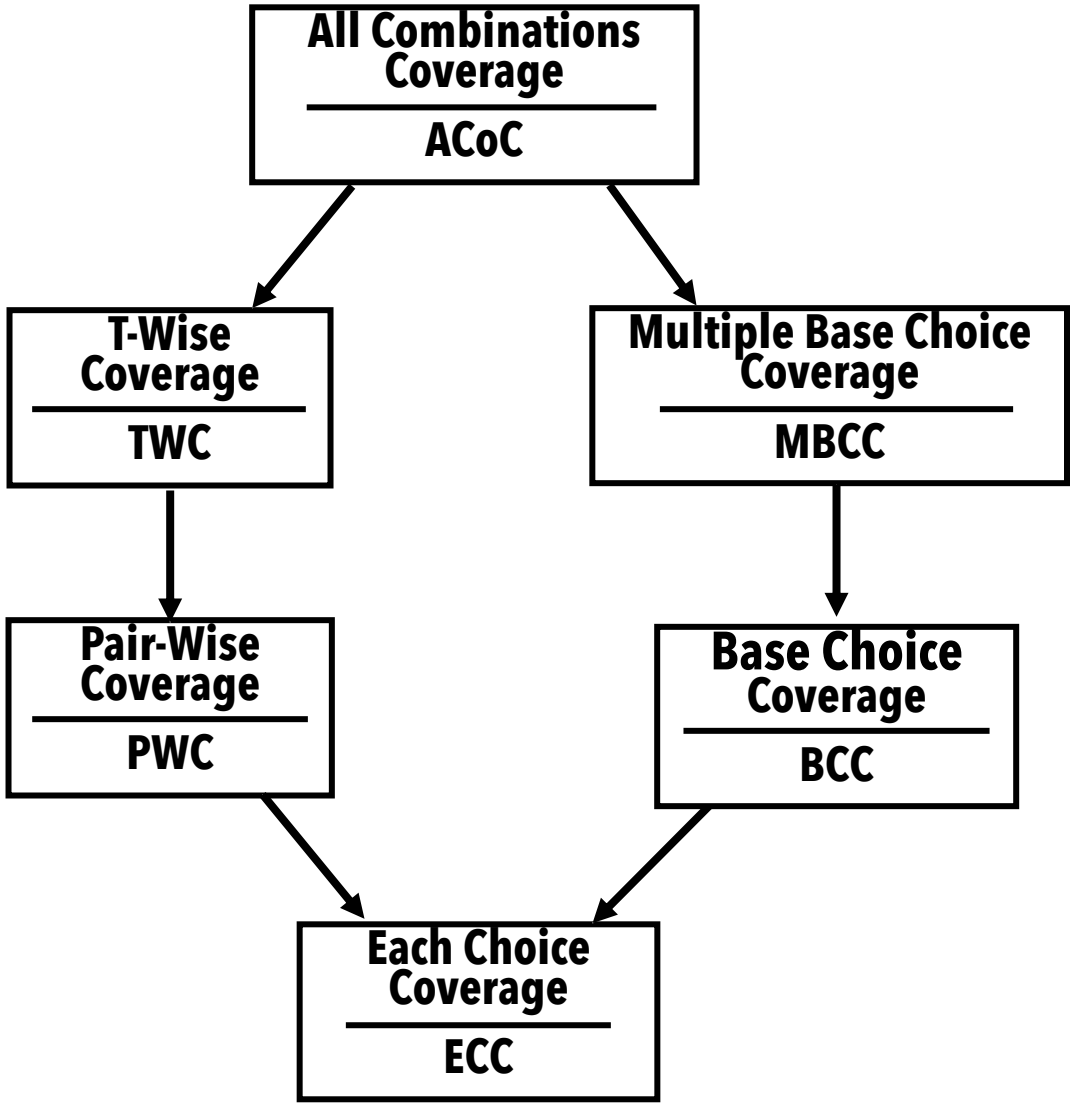
$$M + \sum_{i=1}^{Q} (M * (B_i - m_i))$$

For our example… two base tests: **a1, b1, c1** **a2, b2, c2**

Tests from **a1, b1, c1**: a1, b1, **c3**; a1, **b3**, c1; **a3**, b1, c1

Tests from **a2, b2, c2**: a2, b2, **c3**; a2, **b3**, c2; **a3**, b2, c2

ISP Coverage Criteria Subsumption



Input Space Partitioning Summary

Fairly easy to apply, even with **no automation**

Convenient ways to **add more or less** testing

Equally applicable to **all levels** of testing – unit, class, integration, system, etc.

Based only on the **input space** of the program, not the implementation

Simple, straight-forward, effective, and widely used.