Introduction to Software Testing
Input Space Partition Testing (Ch. 6.1)

Software Testing & Maintenance

SWE 437

http://go.gmu.edu/swe437

Dr. Brittany Johnson-Matthews

(Dr. B for short)

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 effectively **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

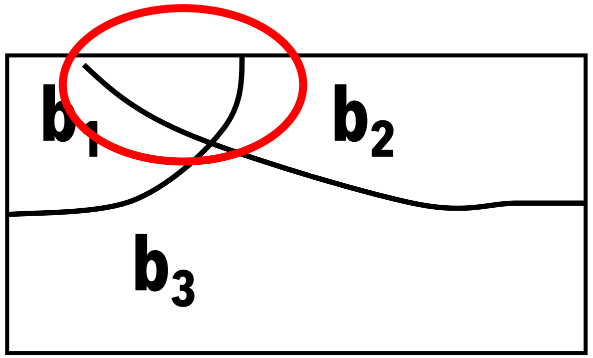
*Domain **D***

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

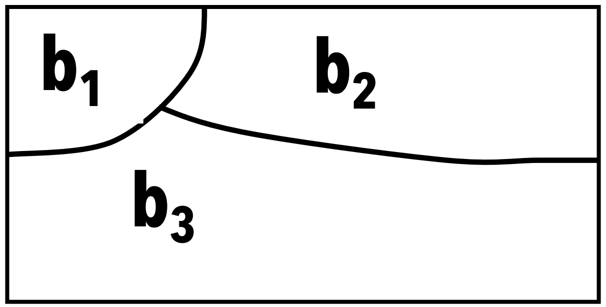
The partition ***q*** defines a set of blocks, ***Bq* = b_1, b_2, …, b_q**

The partition must satisfy two **properties**:

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



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



In-class Exercise

*Practice **partitioning** for integers*

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

Characteristics & Partitions

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

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

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

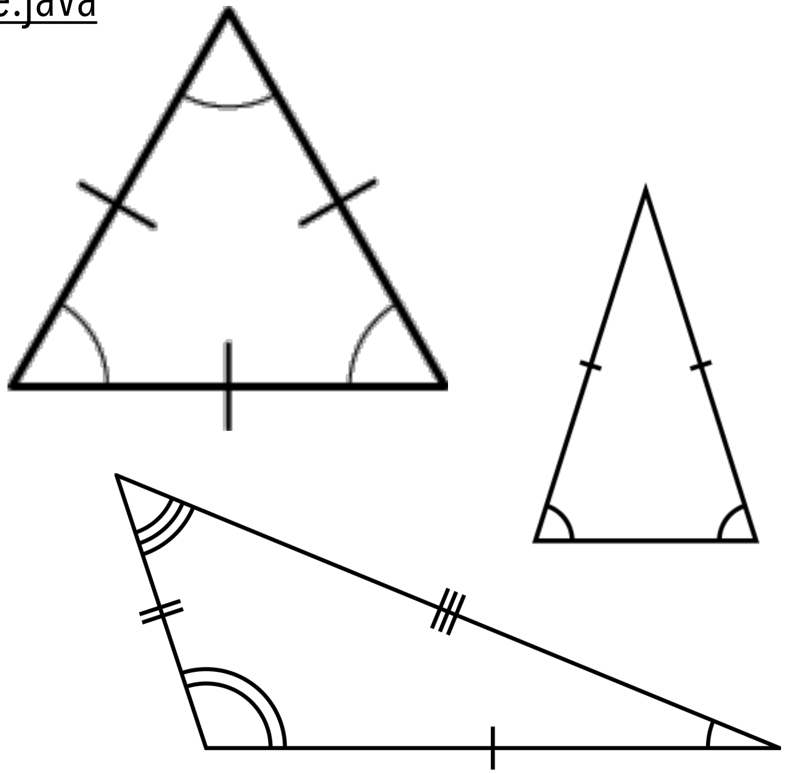
Characteristic: *Relation of side with zero*

Blocks: negative; positive; zero

Example IDM (behavior)

Method `triang()` again:

- 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

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 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

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

<u>Geometric</u> 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!

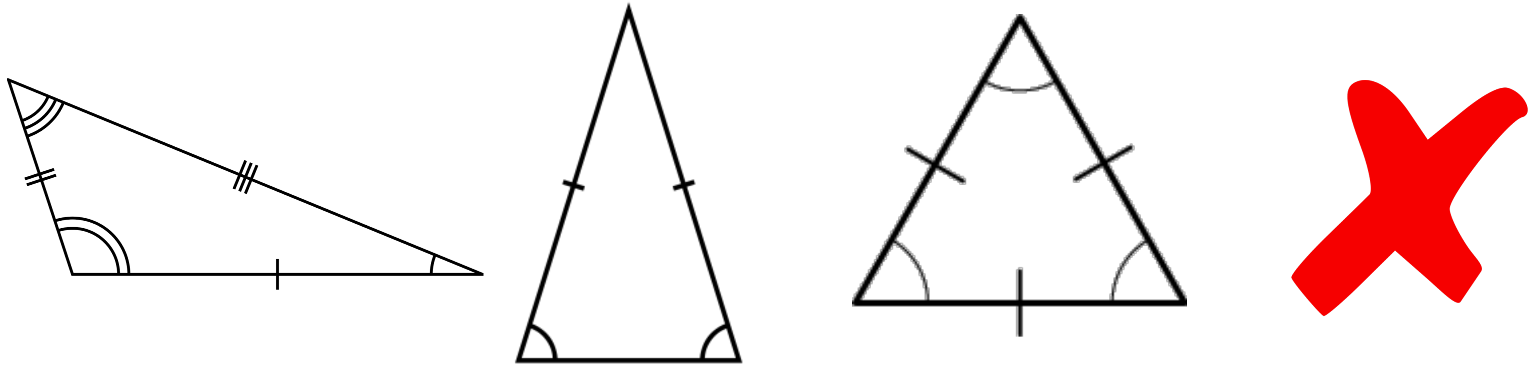
We need to **refine** the example to make characteristics valid

<u>Correct</u> 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

*Proper **partitioning**?*

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