CSc 179 – Input Space Modeling

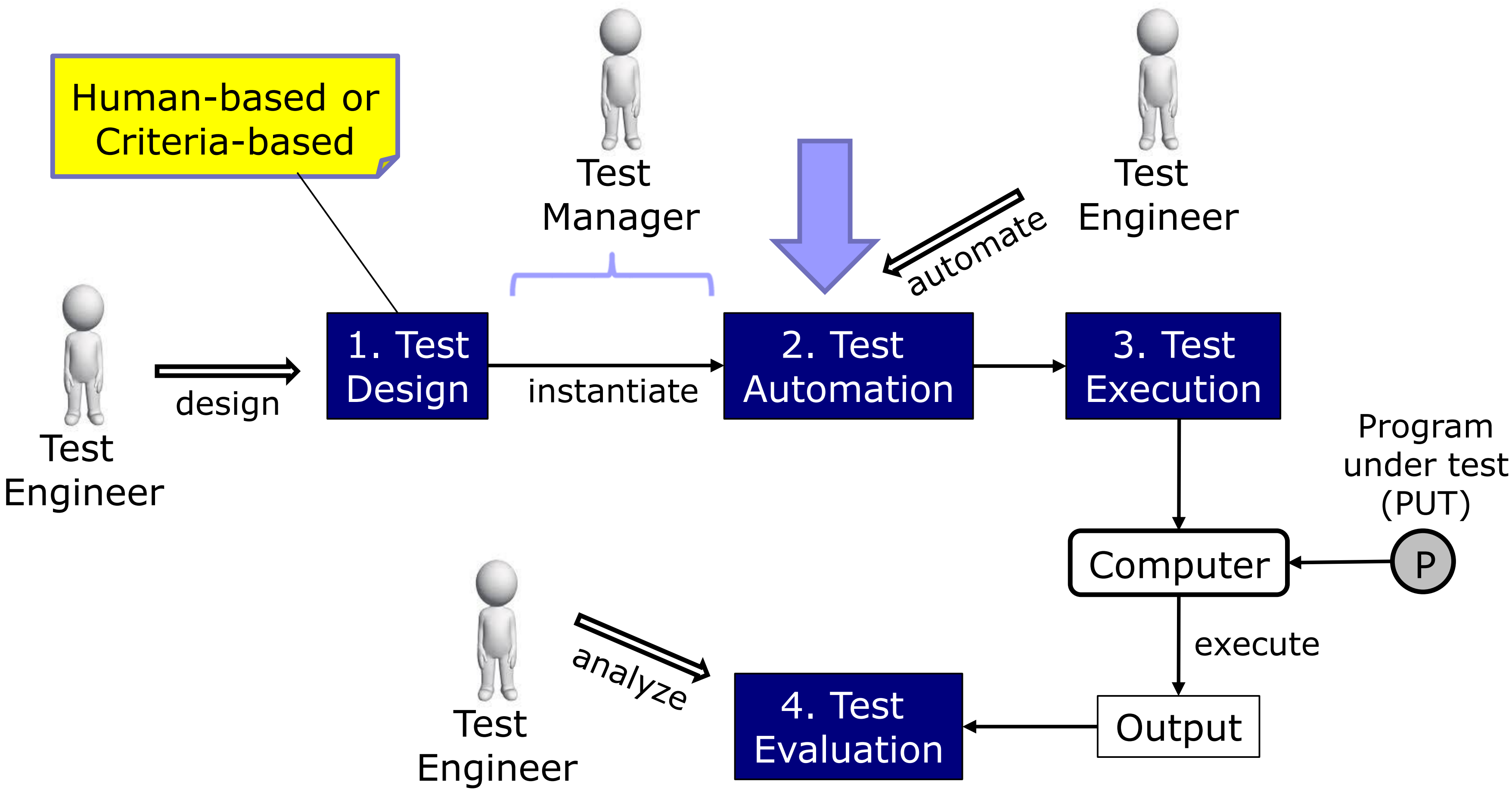
Credits:

Some earlier materials are from CSc 179/234 (Fall 2017)

AO – Ammann and Offutt, “Introduction to Software Testing,” Ch. 6

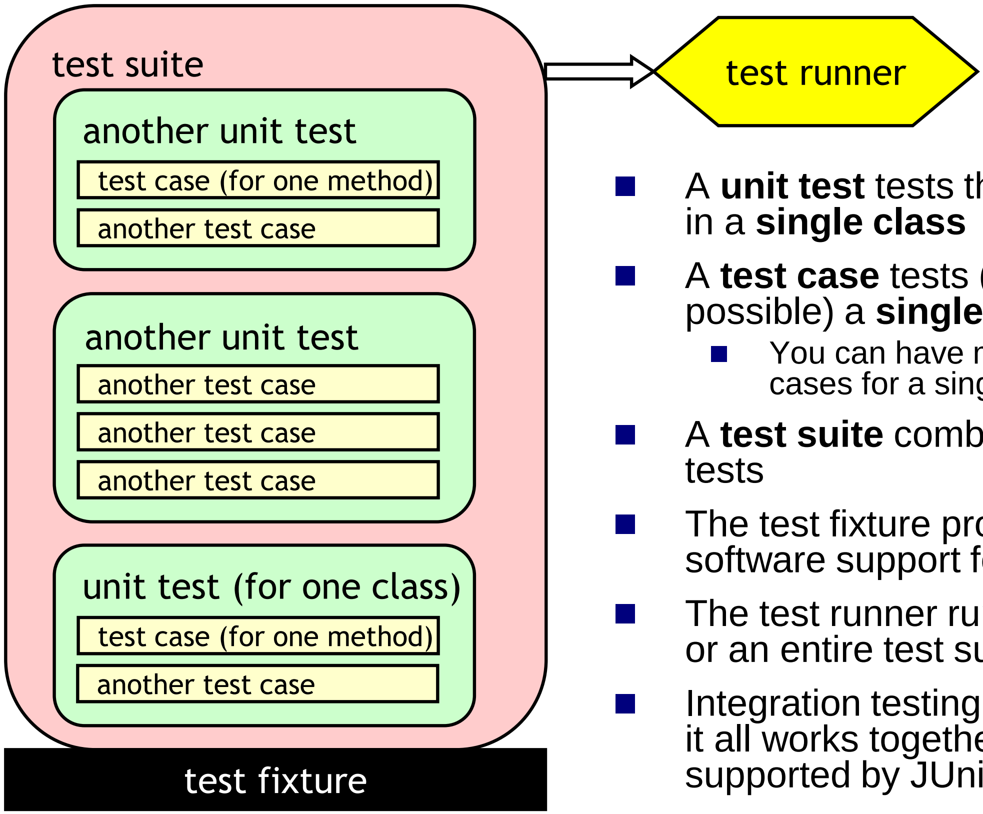
University of Virginia (CS 4501 / 6501)

Recap: Software Testing Activities



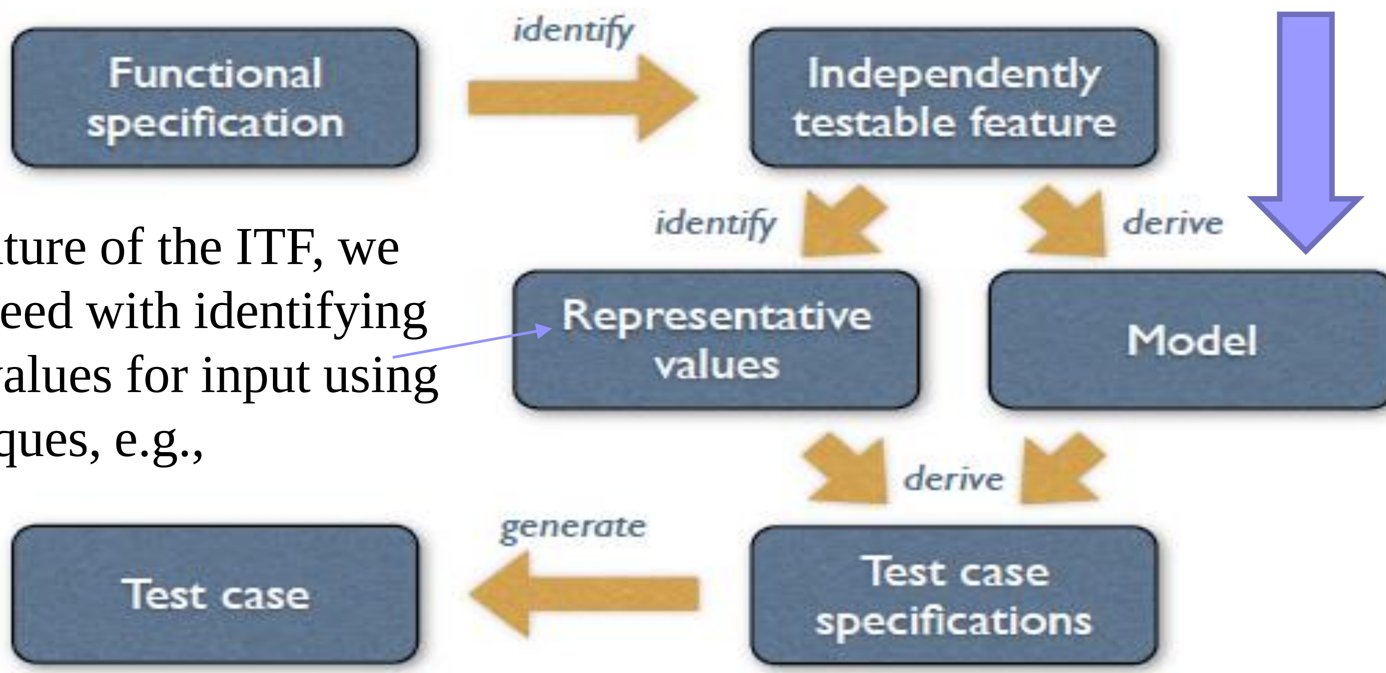
activity requires different skills, background knowledge, education, and training

Recap: In a picture



- A **unit test** tests the methods in a **single class**
- A **test case** tests (insofar as possible) a **single method**
  - You can have multiple test cases for a single method
- A **test suite** combines unit tests
- The test fixture provides software support for all this
- The test runner runs unit tests or an entire test suite
- Integration testing (testing that it all works together) is not well supported by JUnit

Systematic Functional Testing



Based on the nature of the ITF, we may either proceed with identifying representative values for input using different techniques, e.g.,

• **Equivalence partitioning** of the input space
• **Boundary value analysis** • Different types of random selection

Structures for Criteria-Based Testing

Four structures for modeling software

- **Input space**
- **Graph** — Applied to
  - Source
  - Design
  - Specs
  - Use cases
- **Logic** — Applied to
  - Source
  - Specs
  - FSMs
  - DNF
- **Syntax** — Applied to
  - Source
  - Models
  - Integration
  - Inputs

---R	R--R	RI-R	RIPR

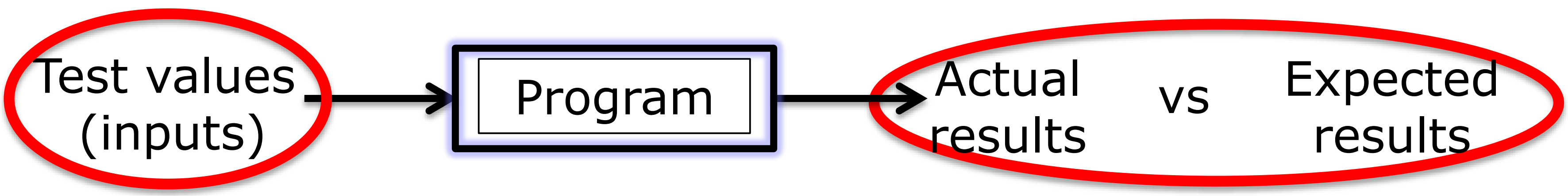
Today's Objectives

- Input domain (or input space)
- Fundamental of Input Space Partitioning (ISP)
  - Benefits of ISP
  - Partitioning input domain
  - Modeling input domain

What is Software Testing?

- Testing = process of finding input values to **check** against a software *(focus of this course)*

- Debugging = process of finding a fault given a failure



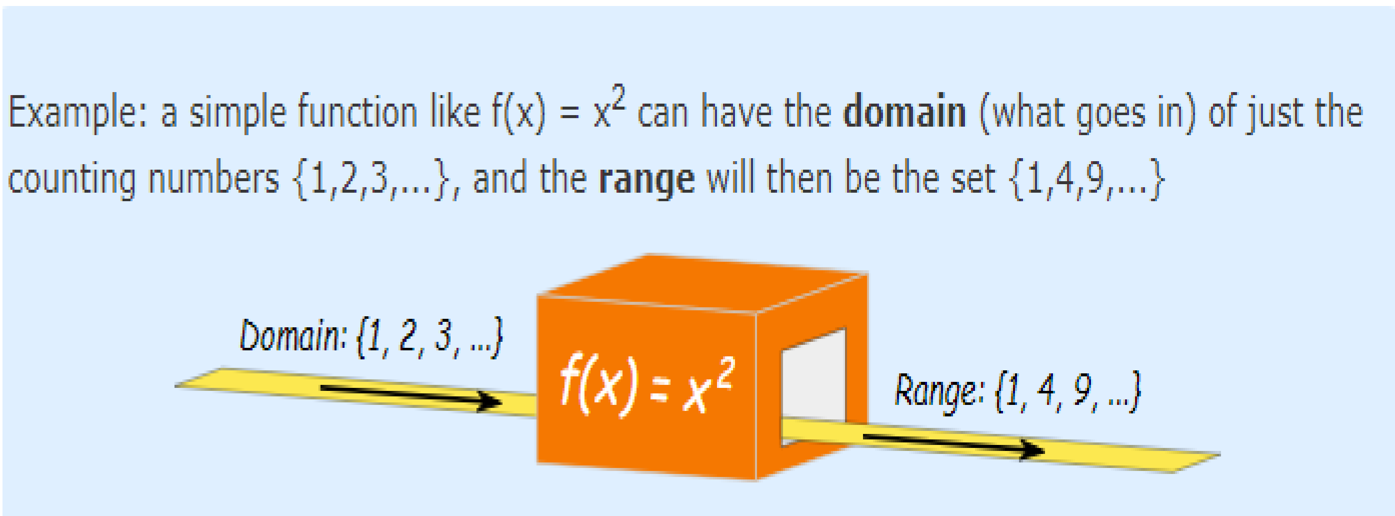
1. Testing is fundamentally about choosing finite sets of values from the input domain of the software being tested

2. Given the test inputs, compare the actual results with the expected results

Input Domains

- **All possible values** that the input parameters can have
- The input domain may be infinite even for a small program
- Testing is fundamentally about **choosing finite sets** of values from the input domain
- **Input parameters** can be
  - Parameters to a method (in unit testing)
  - Global variables (in unit testing)
  - Objects representing current state (in class or integration testing)
  - User level inputs (in system testing)
  - Data read from a file

Example: Input Domain



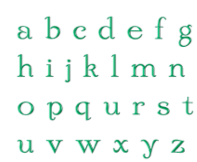
Source: https://www.mathsisfun.com/sets/domain-range-codomain.html

Example Input Domains

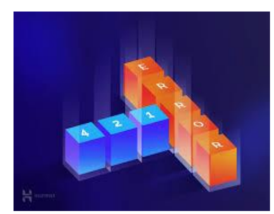
```
# Return index of the first occurrence of a letter in string,
# Otherwise, return -1 (Python Code)

def get_index_of(string, letter):
    index = -1
    for i in range(1, len(string)):
        if string[i] == letter:
            return i
    return index
```

What is the domain of `letter`?

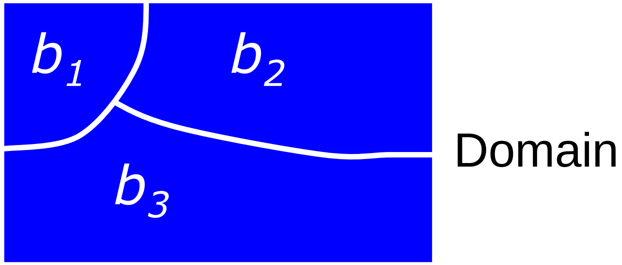


What is the domain of `string`?



Overview: ISP

- Input space partitioning **describes** the **input domain** of the software
- Domain (*D*) are **partitioned** into blocks (b_1, b_2, .., b_n)
- The partition (or block) must satisfy two properties
  - Blocks must not overlap (**disjointness**)
  - Blocks must cover the entire domain (**completeness**)



- At least **one value** is chosen from each block
  - Each value is assumed to be **equally useful** for testing

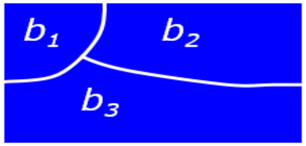
Input Space Modeling

- An Input Domain Model (IDM) represents the input space of the system under test in an **abstract way.**

- A test engineer describes the structure of the input domain in terms of input characteristics.

- The test engineer creates a partition for each characteristic.

- The partition is a set of blocks, each of which contains a set of values.



Domain

Benefits of ISP

- Easy to get started
  - Can be applied with no automation and very little training
- Easy to **adjust** to procedure to get more or fewer tests
- No **implementation knowledge** is needed
  - Just a description of the inputs
- Can be **equally applied** at several levels of testing
  - Unit (inputs from method parameters and non-local variables)
  - Integration (inputs from objects representing current state)
  - System (user-level inputs to a program)

Applying ISP

Identify testable functions

⇓

Identify parameters, return types, return values, exceptional behavior

⇓

Model the input domain

⇓ Input Domain Model (IDMs)

Apply a test criterion to choose combinations of blocks

⇓ Test requirements (TRs)

Derive test values

⇓ Test cases

Task I: Model input domain

The most creative design step in using ISP

Task II: Choose combinations of values

Modeling the Input Domain

- The domain is scoped by the parameters
- Characteristics define the structure of the input domain
  - Characteristics should be based on the input domain – not program source

Two Approaches

Interface-based (simpler)

Develop characteristics from individual parameters

Functionality-based (harder))

Develop characteristics from a behavior view

Design characteristics

⇩

Partition each characteristic into blocks

⇩

Identify values of each block

Design Characteristics

Interface-based

- Develop characteristics directly from **parameters**
  - Translate parameters to characteristics
- Consider each parameter separately
- Rely mostly on syntax
- Ignore some domain and semantic information
  - Can lead to an incomplete IDM
- Ignore relationships among parameters

Functionality-based

Develop characteristics that correspond to the intended **functionality**

Can use relationships among parameters, relationships of parameters with special values (null, blank, ...), preconditions, and postconditions

Incorporate domain and semantic knowledge

May lead to better tests

The same parameter may appear in multiple

Partition Characteristics

Strategies for both approaches

- Partition is a set of blocks, designed using knowledge of what the software is supposed to do
- Each block represents a set of values
- More blocks means more tests
- Partition must satisfy disjointness and completeness properties

Better to have more characteristics with few blocks

- Fewer mistakes and fewer tests

How partitions should be identified and
how representative value should be selected from each block

Identify Values

Strategies for both approaches:

- Include valid, invalid and special values
- Sub-partition some blocks
  - A range of valid values can often be partitioned into sub-partitions
- Explore boundaries of domains
- Enumerated types: A partition where blocks are a discrete, enumerated set often makes sense
- Include values that represent "normal use"
- Try to balance the number of blocks in each characteristic
  - Cut down effort
- Check for completeness and disjointness
  - To ensure good coverage

Interface-based Example1

```
# Return index of the first occurrence of a letter in string,
# Otherwise, return -1

def get_index_of(string, letter):
```

Task I: Model Input Domain

1. Identify testable functions
   - **`get_index_of()`**

2. Identify parameters, return types, return values, and exceptional behavior
   - Parameters: **`string`**, **`letter`**
   - Return type: **`int`**
   - Return value: index of the first occurrence, -1 if no occurrence
   - Exceptional behavior: **`??`**

Interface-based Example1 *(cont)*

3. Model the input domain
   - Develop characteristics
     - C1 = **`string`** is empty
     - C2 = **`letter`** is empty

What are other possible characteristics?

   - Partition characteristics

Complete? Disjoint?

Characteristic	b1	b2
C1 = **string** is empty	True	False
C2 = **letter** is empty	True	False

   - Identify (possible) values

Characteristic	b1	b2
C1 = **string** is empty	""	"testing"
C2 = **letter** is empty	""	"t"

Interface-based Example1 *(cont)*

Task II: Choose combinations of values

4. Combine partitions to define **test requirements (tr(s))**
   - Assumption: choose all possible combinations
   - Test requirements -- number of tests (upper bound) = 2 * 2 = 4

     (True, True) (False, True)

     (True, False) (False, False)
   - Eliminate redundant tests and infeasible tests

5. Derive test values

Test	string	letter	Expected result
T1 (True, True)	""	""	-1
T2 (True, False)	""	"t"	-1
T3 (False, True)	"testing"	""	-1
T4 (False, False)	"testing"	"t"	0

Functionality-based

```
# Return index of the first occurrence of a letter in string,
# Otherwise, return -1

def get_index_of(string, letter):
```

Task I: Model Input Domain

1. **Identify testable functions**
   - `get_index_of()`

2. **Identify parameters, return types, return values, and exceptional behavior**
   - Parameters: `string`, `letter`
   - Return type: `int`
   - Return value: index of the first occurrence, -1 if no occurrence
   - Exceptional behavior: `??`

Functionality-based Example1 *(cont)*

3. Model the input domain
   - Develop characteristics
     - C1 = number of occurrence of **`letter`** in **`string`**
     - C2 = **`letter`** occurs first in **`string`**
   - Partition characteristics

What are other possible characteristics?

Complete? Disjoint?

Characteristic	b1	b2	b3
C1 = number of occurrence of **letter** in **string**	0	1	> 1
C2 = **letter** occurs first in **string**	True	False	

   - Identify (possible) values

C	b1	b2	b3
C1	"software engineering", ""	"software engineering", "s"	"software engineering", "n"
C2	"software engineering", "s"	"software engineering", "t"	

Functionality-based Example1 *(cont)*

Task II: Choose combinations of values

4. Combine partitions into tests
   - Assumption: choose all possible combinations
   - Test requirements -- number of tests (upper bound) = 3 * 2 = 6

     ~~(0, True)~~ (1, True) (>1, True)
     (0, False) (1, False) (>1, False)
   - Eliminate redundant tests and infeasible tests
5. Derive test values

Test	string	letter	Expected result
T1 (0, False)	"software engineering"	""	-1
T2 (1, True)	"software engineering"	"s"	0
T3 (1, False)	"software engineering"	"t"	3
T4 (>1, True)	"software testing"	"s"	0
T5 (>1, False)	"software engineering"	"n"	10

Interface-based Example2

```
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.
# Return the appropriate enum value
```

Task I: Model Input Domain

1. **Identify testable functions**
   - **`triang()`**

2. **Identify parameters, return types, return values, and exceptional behavior**
   - Parameters: **`Side1`**, **`Side2`**, **`Side3`**
   - Return type: **`enum`**
   - Return value: enum describing type of a triangle
   - Exceptional behavior: **`??`**

Interface-based Example2 *(cont)*

3. Model the input domain
   - Develop characteristics
     - C1 = relation of **`Side1`** to 0
     - C2 = relation of **`Side2`** to 0
     - C3 = relation of **`Side3`** to 0

What are other possible characteristics?

   - Partition characteristics

Complete? Disjoint?

Characteristic	b1	b2	b3
C1 = relation of **Side1** to 0	greater than 0	equal to 0	less than 0
C2 = relation of **Side2** to 0	greater than 0	equal to 0	less than 0
C3 = relation of **Side3** to 0	greater than 0	equal to 0	less than 0

   - Identify (possible) values

Valid triangles?

Characteristic	b1	b2	b3
C1 = relation of **Side1** to 0	7	0	-3
C2 = relation of **Side2** to 0	3	0	-1
C3 = relation of **Side3** to 0	2	0	-2

Interface-based Example2 *(cont)*

- Refine characteristics (can lead to more tests)
  - C1 = length of **`Side1`**
  - C2 = length of **`Side2`**
  - C3 = length of **`Side3`**

Refining
get more fine-grained testing
(if the budget allows)

- Partition characteristics

Complete? Disjoint?

Characteristic	**b1**	**b2**	**b3**	**b4**
C1 = length of **Side1**	greater than 1	equal to 1	equal to 0	less than 0
C2 = length of **Side2**	greater than 0	equal to 1	equal to 0	less than 0
C3 = length of **Side3**	greater than 0	equal to 1	equal to 0	less than 0

- Identify (possible) values

Valid triangles?

Characteristic	**b1**	**b2**	**b3**	**b4**
C1 = length of **Side1**	2	1	0	-1
C2 = length of **Side2**	2	1	0	-1
C3 = length of **Side3**	2	1	0	-1

Boundary tests

Interface-based Example2 *(cont)*

Task II: Choose combinations of values

4. Combine partitions to define test requirements
   - Assumption: choose all possible combinations
   - Test requirements -- number of tests (upper bound) = 4*4*4 = 64

(C1b1, C2b1, C3b1)	(C1b1, C2b2, C3b1)	(C1b1, C2b3, C3b1)	(C1b1, C2b4, C3b1)
(C1b1, C2b1, C3b2)	(C1b1, C2b2, C3b2)	(C1b1, C2b3, C3b2)	(C1b1, C2b4, C3b2)
(C1b1, C2b1, C3b3)	(C1b1, C2b2, C3b3)	(C1b1, C2b3, C3b3)	(C1b1, C2b4, C3b3)
(C1b1, C2b1, C3b4)	(C1b1, C2b2, C3b4)	(C1b1, C2b3, C3b4)	(C1b1, C2b4, C3b4)
(C1b2, C2b1, C3b4)	(C1b2, C2b2, C3b4)	(C1b2, C2b3, C3b4)	(C1b2, C2b4, C3b4)

…

Do we really need these many tests?

   - Eliminate redundant tests and infeasible tests

5. Derive test values

(2, 2, 2)	(2, 1, 2)	(2, 0, 2)	(2, -1, 2)
(2, 2, 1)	(2, 1, 1)	(2, 0, 1)	(2, -1, 1)

…

Functionality-based Example2

```
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.
# Return the appropriate enum value
```

Task I: Model Input Domain

1. Identify testable functions
   - `triang()`

2. Identify parameters, return types, return values, and exceptional behavior
   - Parameters: `Side1`, `Side2`, `Side3`
   - Return type: `enum`
   - Return value: enum describing type of a triangle
   - Exceptional behavior: `??`

Functionality-based Example2 *(cont)*

3. Model the input domain
   - Develop characteristics
     - C1 =Geometric classification

What are other possible characteristics?

   - Partition characteristics

Complete? Disjoint?

Characteristic	b1	b2	b3	b4
C1 = Geometric classification	scalene	isosceles	equilateral	invalid

   - Refine characteristics

Complete? Disjoint?

Characteristic	b1	b2	b3	b4
C1 = Geometric classification	scalene	Isosceles, not equilateral	equilateral	invalid

   - Identify (possible) values

Characteristic	b1	b2	b3	b4
C1 = Geometric classification	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Functionality-based Example2 *(cont)*

Task II: Choose combinations of values

4. Combine partitions into tests
   - Assumption: choose all possible combinations
   - Test requirements -- number of tests (upper bound) = 4
     (C1b1) (C1b2) (C1b3) (C1b4)
   - Eliminate redundant tests and infeasible tests

5. Derive test values

Test	Side1	Side2	Side3	Expected result
T1 (scalene)	4	5	6	scalene
T2 (isosceles, not equilateral)	3	3	4	isosceles
T3 (equilateral)	3	3	3	equilateral
T4 (invalid)	3	4	8	invalid

This characteristic results in a simple set of test requirements.
Is this good enough?
If we define the characteristics differently? Multiple

ISP Task I Summary

- Easy to apply, even with no automation and little training
- Easy to add more or fewer tests
- Rely on the input space, not implementation knowledge
- Applicable to all levels of testing, effective and widely used

Interface-based approach

Strength

Easy to identify characteristics

Easy to translate abstract tests into executable test cases

Weakness

Some information will not be used – lead to incomplete IDM

Ignore relationships among parameters

Functionality-based approach

Strength

Incorporate semantic

Input domain modeling and test case generation in early development phases

Weakness

Difficult to design reasonable characteristics

Hard to generate tests

What's Next?

- How should we consider multiple partitions or IDMs at the same time?
- What combinations of blocks should we choose values from?
- How many tests should we expect?