

# CS409

# Software Testing

**TAN, Shin Hwei**

陈馨慧

Southern University of Science and Technology


Slides adapted from Introduction to Software Testing, Edition 2 (Ch  
6)

# Administrative Info

- MP1 due on 10 October, 11.59pm. **Late submission will get 0!**

**\*If you have question on MP1, post it or read the discussion on <https://github.com/orgs/cs409-software-testing2020/teams/allstudents>**

## Q&A for MP1: Post your questions here

 **stan6**  
15 hours ago

1. There is a example test in the Android app that you uploaded on the Github. Is this test a example for task vi?

task vi. Add one JUnit test for each arithmetic operation ("+", "-", "\*", "/") (4 points)

```
@Test
public void addition_isCorrect() {
    assertEquals(4, 2 + 2);
}
```

Answer: The test is not an example for task VI. It is a test added by the author of the tutorial. You should create your own test (your test should be different) for task VI.

2. Is the task VII "implement the logic of "." button" the replication of task IV,V?

iv. Add little code to make the test pass in Step iii. Commit this code to GitHub repository with the commit message "Initial code for passing TDD test". (2 points)

v. Refactor the code for adding the functionality to support entering decimal points. In this step, you should remove any code duplication. (2 points)

vii. Implement the logic for the decimal point button "." (4 points)

Answer: No. Task VII is not replication of task IV and V. Task IV should have simple code that makes test in Step III passing and this task requires a commit to the repository. Task V clearly states that there should be a refactoring that removes duplication. Task VII requires implementing the complete logic of "." button.



Reply...

# Continue from previous lecture

---

# Example : Jelly Bean Coverage

## Flavors :

1. Lemon
2. Pistachio
3. Cantaloupe
4. Pear
5. Tangerine
6. Apricot



## Colors :

1. Yellow (Lemon, Apricot)
2. Green (Pistachio)
3. Orange (Cantaloupe, Tangerine)
4. White (Pear)

## • Possible coverage criteria :

1. Taste one jelly bean of **each flavor**
  - Deciding if yellow jelly bean is Lemon or Apricot is a controllability problem
2. Taste one jelly bean of **each color**

# Question: Code Coverage

Give tests to reach 100% coverage

Is it possible? If not, why?

```
1: public static boolean m(a){  
2:   if (a || !a)  
3:   {  
4:     return a;  
5:   }  
6:   else  
7:   {  
8:     b++;  
9:     return b;  
10:  }  
11: }
```

# Coverage

Given a set of test requirements  $TR$  for coverage criterion  $C$ , a test set  $T$  satisfies  $C$  coverage if and only if for every test requirement  $tr$  in  $TR$ , there is at least one test  $t$  in  $T$  such that  $t$  satisfies  $tr$

**Infeasible test requirements** : test requirements that cannot be satisfied

- No test case values exist that meet the test requirements
- Example: Dead code (like in previous slide)
- Detection of infeasible test requirements is formally undecidable for most test criteria

Thus, 100% coverage is **impossible** in practice

# More Jelly Beans

T1 = { three Lemons, one Pistachio, two Cantaloupes,  
one Pear, one Tangerine, four Apricots }

Does test set T1 satisfy the flavor criterion ?

T2 = { One Lemon, two Pistachios, one Pear, three  
Tangerines }

- ✓ Does test set T2 satisfy the flavor criterion ?
- ✓ Does test set T2 satisfy the color criterion ?

# Coverage Level

The ratio of the number of test requirements satisfied by  $T$  to the size of  $TR$

T2 on the previous slide satisfies 4 of 6 test requirements



# Two Ways to Use Test Criteria

1. **Directly generate** test values **to satisfy** the criterion
  - Often assumed by the research community
  - Most obvious way to use criteria
  - Very hard without automated tools
2. Generate test values **externally** and **measure** against the criterion
  - Usually favored by industry
  - Sometimes misleading
  - If tests do not reach 100% coverage, what does that mean?

**Test criteria are sometimes called  
metrics**

# Generators and Recognizers

**Generator** : A procedure that automatically generates values to satisfy a criterion

**Recognizer** : A procedure that decides whether a given set of test values satisfies a criterion

Both problems are provably **undecidable** for most criteria

It is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion

**Coverage analysis tools** are quite plentiful

# Comparing Criteria with Subsumption (5.2)

**Criteria Subsumption** : A test criterion  $C1$  subsumes  $C2$  if and only if every set of test cases that satisfies criterion  $C1$  also satisfies  $C2$

Must be true for **every set** of test cases

*Examples :*

- The flavor criterion on jelly beans subsumes the color criterion ... if we taste every flavor we taste one of every color
- If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement

# Advantages of Criteria-Based Test Design (5.3)

Criteria maximize the “bang for the buck”

- Fewer tests that are more effective at finding faults

Comprehensive test set with minimal overlap

Traceability from software artifacts to tests

- The “why” for each test is answered
- Built-in support for regression testing

A “stopping rule” for testing—advance knowledge of how many tests are needed

Natural to automate

# Characteristics of a Good Coverage Criterion

1. It should be fairly easy to compute test requirements **automatically**
  2. It should be **efficient to generate** test values
  3. The resulting tests should reveal as many **faults** as possible
- Subsumption is only a **rough approximation** of fault revealing capability
  - Researchers still need to gives us more data on how to **compare** coverage criteria

# Test Coverage Criteria

Traditional software testing is **expensive** and **labor-intensive**

Formal coverage criteria are used to decide **which test inputs** to use

More likely that the tester will **find problems**

Greater assurance that the software is of **high quality** and **reliability**

A goal or **stopping rule** for testing

Criteria makes testing more **efficient** and **effective**

**How do we start applying these ideas in practice?**

# How to Improve Testing ?

Testers need more and better software tools

Testers need to adopt practices and techniques that lead to more efficient and effective testing

- More education
- Different management organizational strategies

Testing / QA teams need more technical expertise

- Developer expertise has been increasing dramatically

Testing / QA teams need to specialize more

- This same trend happened for development in the 1990s

# Four Roadblocks to Adoption

## 1. Lack of test education

Microsoft and Google say half their engineers are testers, programmers test half the time

Number of UG CS programs in US that require testing ?

0

Number of MS CS programs in US that require testing ?

0

Number of UG testing classes in the US ?

~50

## 2. Necessity to change process

Adoption of many test techniques and tools require changes in development process

This is expensive for most software companies

## 3. Usability of tools

Many testing tools require the user to know the underlying theory to use them

Do we need to know how an internal combustion engine works to drive ?

Do we need to understand parsing and code generation to use a compiler ?

## 4. Weak and ineffective tools

Most test tools don't do much – but most users do not realize they could be better

Few tools solve the key technical problem – **generating test values automatically**



# Needs From Researchers

1. **Isolate** : **Invent** processes and techniques that isolate the theory from most test practitioners
2. **Disguise** : **Discover** engineering techniques, standards and frameworks that disguise the theory
3. **Embed** : Theoretical ideas in **tools**
4. **Experiment** : Demonstrate **economic value** of criteria-based testing and ATDG (*ROI*)
  - **Which** criteria should be used and **when** ?
  - **When** does the extra effort pay off ?
5. **Integrate** high-end testing with **development**

# Needs From Educators

1. **Disguise** theory from engineers in classes
2. **Omit** theory when it is not needed
3. **Restructure** curricula to teach more than test design and theory
  - Test automation
  - Test evaluation
  - Human-based testing
  - Test-driven development

# Changes in Practice

1. **Reorganize** test and QA teams to make effective use of individual abilities
  - One math-head can support many testers
2. **Retrain** test and QA teams
  - Use a process like MDTD
  - Learn more testing concepts
3. **Encourage** researchers to embed and isolate
  - We are very responsive to research grants
4. **Get involved** in curricular design efforts through industrial advisory boards

# Criteria Summary

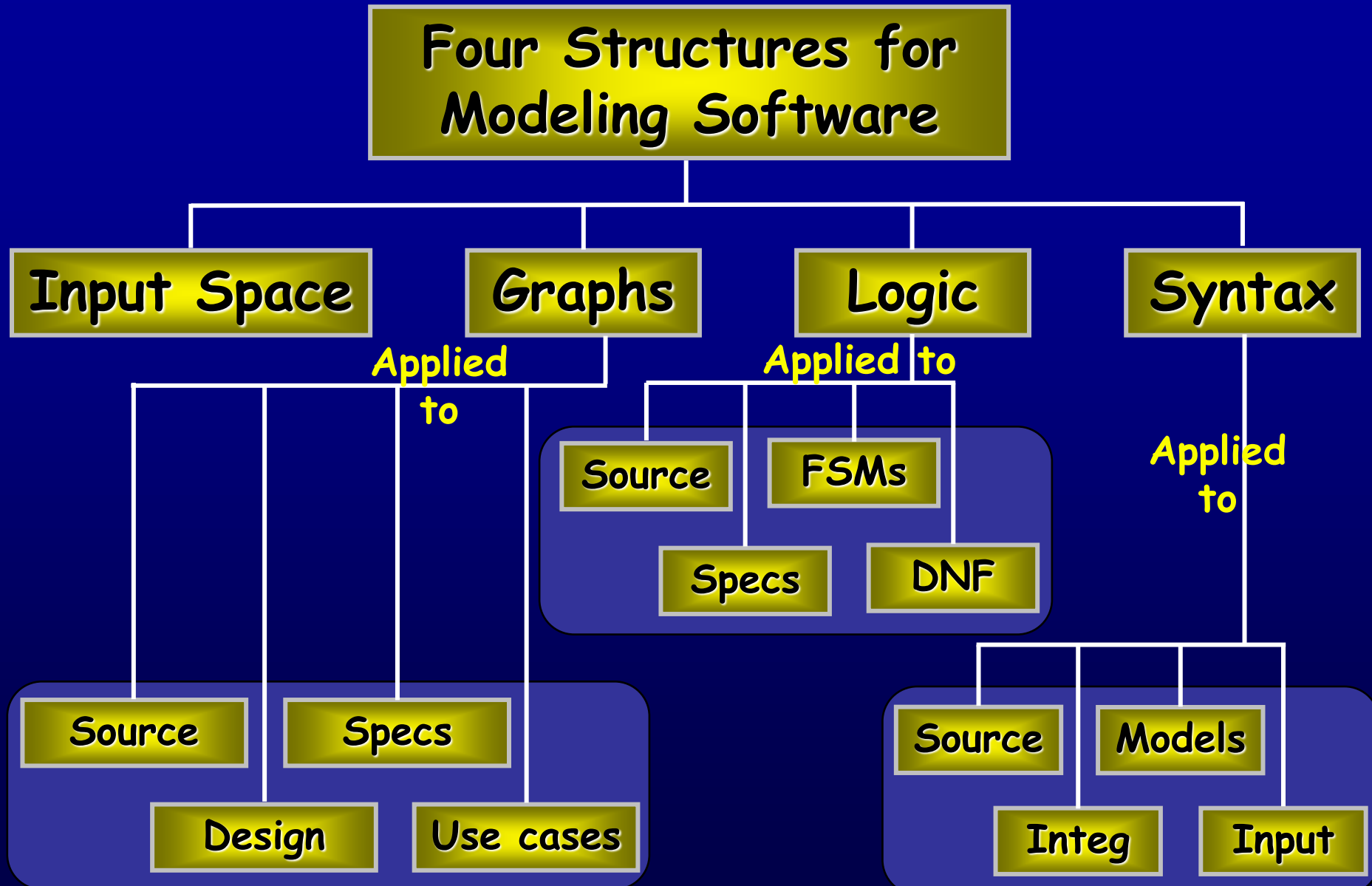
- Many companies still use “monkey testing”
  - A human sits at the keyboard, wiggles the mouse and bangs the keyboard
  - No automation
  - Minimal training required
- Some companies automate human-designed tests
- But companies that use both automation and criteria-based testing

**Save money**

**Find more faults**

**Build better software**

# Structures for Criteria-Based Testing



# **Introduction to Software Testing**

## **Chapter 6**

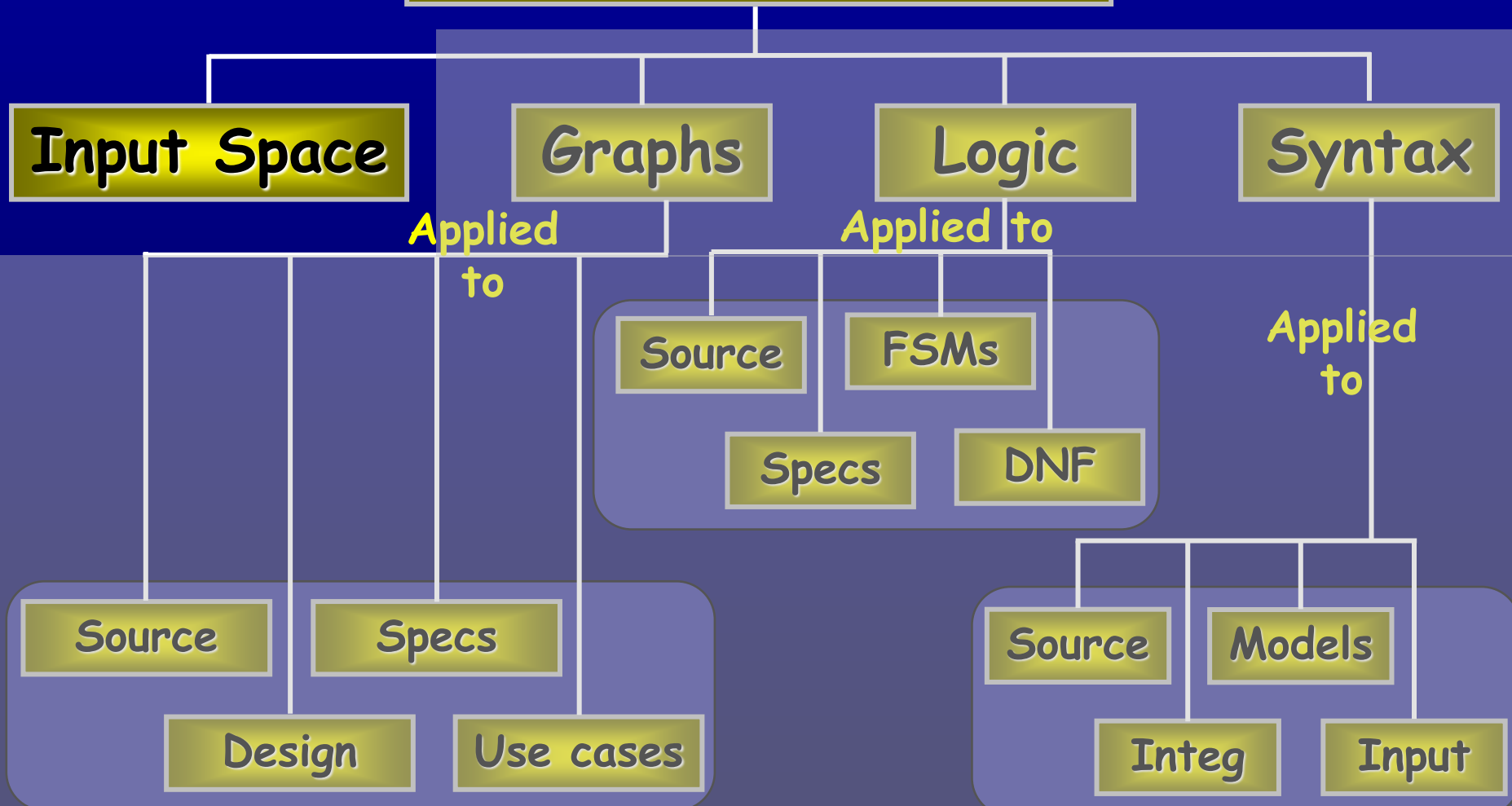
### **Input Space Partition Testing**

Paul Ammann & Jeff Offutt

<http://www.cs.gmu.edu/~offutt/softwaretest/>

# Ch. 6 : Input Space Coverage

## Four Structures for Modeling Software



# Benefits of ISP

- Can be **equally applied** at several levels of testing
  - Unit
  - Integration
  - System
- Relatively easy to apply with **no automation**
- Easy to **adjust** the procedure to get more or fewer tests
- No **implementation knowledge** is needed
  - Just the input space

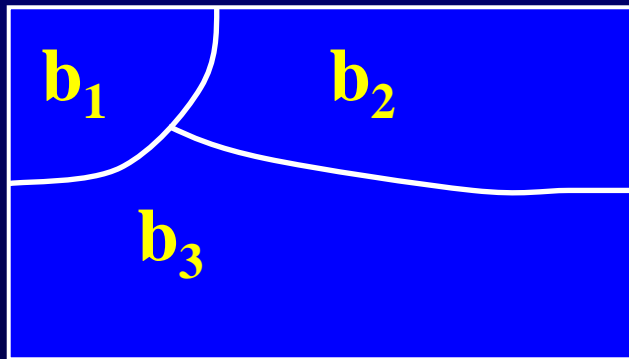


# Input Domains

- The **input domain** for a program contains all the possible inputs to that program
- For even small programs, the input domain is so large that it might as well be **infinite**
- Testing is fundamentally about **choosing finite sets** of values from the input domain
- **Input parameters** define the scope of the input domain
  - Parameters to a method
  - Data read from a file
  - Global variables
  - User level inputs
- Input domains are **partitioned into regions** (blocks)
- At least **one value** is chosen from each block

# Partitioning Domains

- Domain  $D$
- Partition scheme  $q$  of  $D$
- The partition  $q$  defines a set of blocks,  $B_q = b_1, b_2, \dots, b_Q$
- The partition must satisfy two properties :
  1. Blocks must be *pairwise disjoint* (no overlap)
  2. Together the blocks *cover* the domain  $D$  (complete)



$$b_i \cap b_j = \Phi, \forall i \neq j, b_i, b_j \in B_q$$

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

# 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*

# Using Partitions – Assumptions

- Choose a **value** from each block
- Each value is assumed to be **equally useful** for testing
- Application to testing
  - Find **characteristics** in the inputs : parameters, semantic descriptions, ...
  - **Partition** each characteristic
  - **Choose tests** by combining values from characteristics
- Example **Characteristics**
  - Input X is null
  - Order of the input file F (sorted, inverse sorted, arbitrary, ...)
  - Min separation of two aircraft
  - Input device (DVD, CD, VCR, computer, ...)

# Choosing Partitions

- Choosing (or defining) **partitions** seems easy, but is easy to get wrong
- Consider the characteristic “*order of file F*”

$b_1$  = sorted in ascending order  
 $b_2$  = sorted in descending order  
 $b_3$  = arbitrary order

Design blocks for  
that characteristic

but ... something's fishy ...

What if the file is of length 1?

Can you find the  
problem?

The file is partitioned into blocks

That is, disjointness is not satisfied

Solution:

Each characteristic should  
address just one property

Can you think of  
a solution?

C1: File F sorted ascending

- $c1.b1 = \text{true}$
- $c1.b2 = \text{false}$

C2: File F sorted descending

- $c2.b1 = \text{true}$
- $c2.b2 = \text{false}$

# Properties of 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
- We model the input domain in **five steps** ...
  - Steps 1 and 2 move us from the implementation abstraction level to the design abstraction level (from chapter 2)
  - Steps 3 & 4 are entirely at the design abstraction level
  - Step 5 brings us back down to the implementation abstraction level

# Modeling the Input Domain

- **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.
- **Step 2** : Find all the **parameters**
  - Often fairly **straightforward**, even mechanical
  - Important to be **complete**
  - **Methods** : Parameters and state (non-local) variables used
  - **Components** : Parameters to methods and state variables
  - **System** : All inputs, including files and databases

# Modeling the Input Domain (*cont*)

- **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
- **Step 4** : Apply a test 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 under test
- **Harder** to develop—requires more design effort
- May result in **better tests**, or fewer tests that are as effective

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

# 1. Interface-Based Approach

- Mechanically consider each parameter in isolation
- This is an easy modeling technique and relies mostly on syntax
- Some domain and semantic information won't be used
  - Could lead to an incomplete IDM
- Ignores relationships among parameters

# 1. Interface-Based Example

- Consider method *triang()* from class *TriangleType* on the book website :
  - <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**

**Reasonable characteristic : *Relation of side with zero***

## 2. Functionality-Based Approach

- Identify characteristics that correspond to the intended **functionality**
- Requires more **design effort** from tester
- Can incorporate **domain** and **semantic** knowledge
- Can use **relationships** among parameters
- Modeling 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

## 2. Functionality-Based Example

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

**The three parameters represent a *triangle***

**The IDM can combine all parameters**

**Reasonable characteristic : *Type of triangle***

# Steps 1 & 2—Identifying Functionalities, Parameters and Characteristics

- A creative engineering step
- More characteristics means more tests
- Interface-based : Translate parameters to characteristics
- Candidates for characteristics :
  - Preconditions and postconditions
  - Relationships among variables
  - Relationship of variables with special values (zero, null, blank, ...)
- Should 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

# 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
```

*Work with 2 or 3 classmates*

*Create two IDMs for findElement () :*

- 1) Interface-based*
- 2) Functionality-based*

# 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
  - Should **evaluate** them separately – sometimes fewer characteristics can be used with more blocks and vice versa
- **Strategies** for identifying values :
  - Include **valid**, **invalid** and **special** values
  - **Sub-partition** some blocks
  - Explore **boundaries** of domains
  - Include values that represent “**normal use**”
  - Try to **balance** the number of blocks in each characteristic
  - Check for **completeness** and **disjointness**

# Interface-Based –*triang()*

- *triang()* has one testable function and three integer inputs

## First Characterization of TriTyp's Inputs

Characteristic	$b_1$	$b_2$	$b_3$
$q_1$ = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
$q_2$ = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
$q_3$ = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

- A maximum of  $3*3*3 = 27$  tests
- Some triangles are **valid**, some are **invalid**
- **Refining** the characterization can lead to more tests ...

# Interface-Based IDM—*triang()*

## 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	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

- A maximum of  $4*4*4 = 64$  tests
- **Complete** because the inputs are integers (0 .. 1)

### Possible values for partition $q_1$

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
Side 1	2	1	0	-1

Test boundary conditions

# Functionality-Based *IDM*—*triang()*

- First two characterizations are based on **syntax**—parameters and their type
- A **semantic** level characterization could use the fact that the three integers represent a triangle

## Geometric Characterization of *triang()*'s Inputs

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
$q_1$ = "Geometric Classification"	scalene	isosceles	equilateral	invalid

- Oops ... something's **fishy** ... equilateral等边 is also isosceles等腰 !
- 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()*

- First two characterizations are based on **syntax**—parameters and their type
- A **semantic** level characterization could use the fact that the three integers represent a triangle

## Geometric Characterization of *triang()*'s Inputs

Characteristic	$b_1$	$b_2$	$b_3$	$b_4$
$q_1$ = "Geometric Classification"	scalene	isosceles	equilateral	invalid

- Equilateral is also isosceles
  - We need to **refine** the partitioning?
- What's wrong with this partitioning? 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()*

- A **different approach** would be to break the geometric characterization into four separate characteristics

## Four Characteristics for *triang()*

Characteristic	$b_1$	$b_2$
$q_1 = \text{"Scalene"}$	True	False
$q_2 = \text{"Isosceles"}$	True	False
$q_3 = \text{"Equilateral"}$	True	False
$q_4 = \text{"Valid"}$	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 **severity**
  - 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

*Work with 2 or 3 classmates*

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 :  $\prod_{i=1}^Q (B_i)$
- The second characterization of triang() results in  $4*4*4 =$  **64 tests**
  - Too many ?

# ISP Criteria – All Combinations

- Consider the “second characterization” of Triang as given before:

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:

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

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

For *triang()* : A1, B1, C1

A2, B2, C2

A3, B3, C3

A4, B4, C4

Substituting values: 2, 2, 2

1, 1, 1

0, 0, 0

-1, -1, -1

# 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  $(\text{Max}_{i=1}^Q (B_i)) * (\text{Max}_{j=1, j \neq i}^Q (B_j))$

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

# ISP Criteria –T-Wise

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

**t-Wise Coverage (TWC) : 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

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

- If  $t$  is the number of characteristics  $Q$ , then all combinations
- That is ...  $Q\text{-wise} = AC$
- $t\text{-wise}$  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

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

For <i>triang()</i> : <u>Base</u>	A1, B1, C1	A1, B1, C2	A1, B2, C1	A2, B1, C1
		A1, B1, C3	A1, B3, C1	A3, B1, C1
		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-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



# ISP Criteria – Multiple Base Choice

- 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.

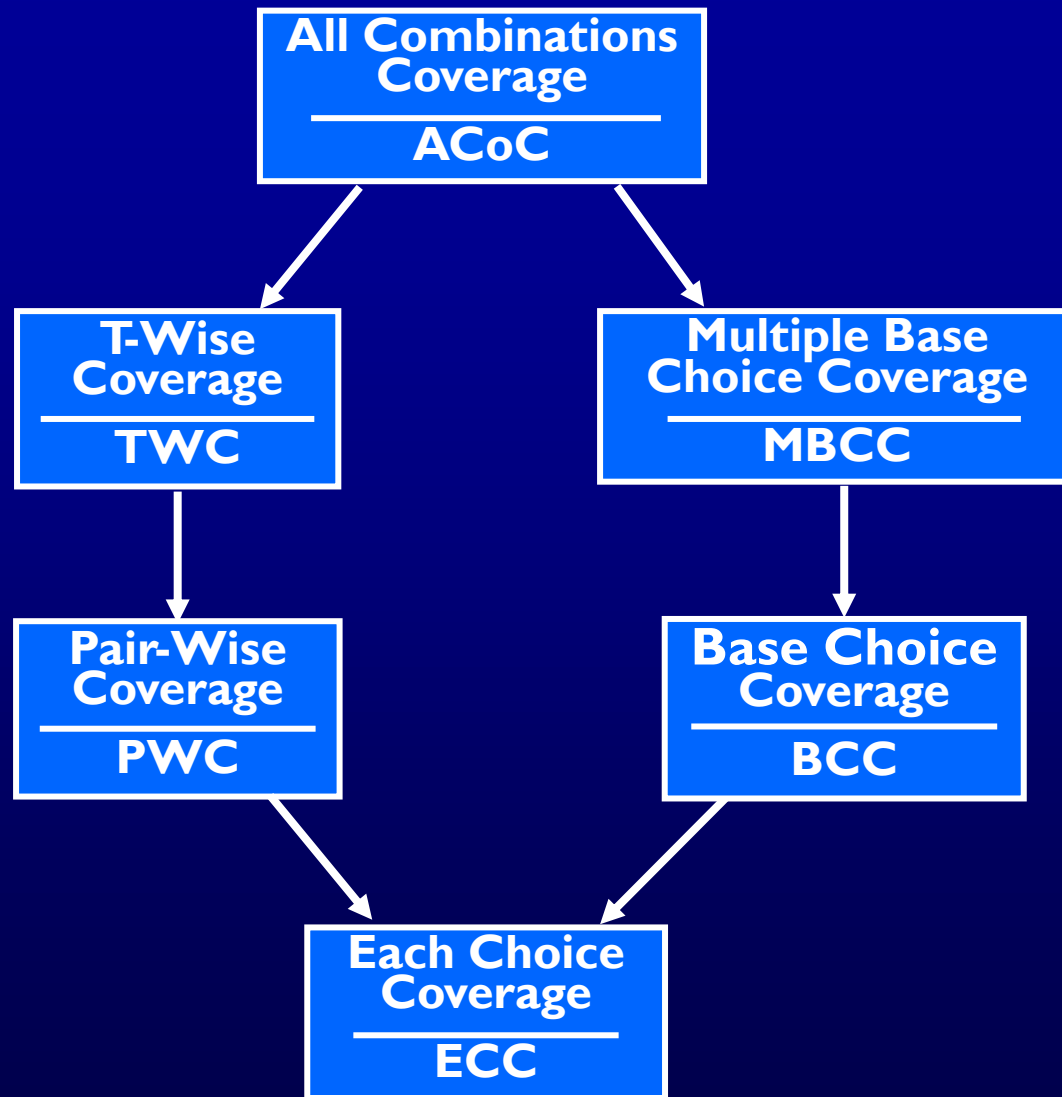
- If  $M$  base tests and  $m_i$  base choices for each characteristic:

$$\mathbf{M} + \sum_{i=1}^Q (\mathbf{M} * (\mathbf{B}_i - \mathbf{m}_i))$$

## For triang() : Bases

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 from 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**