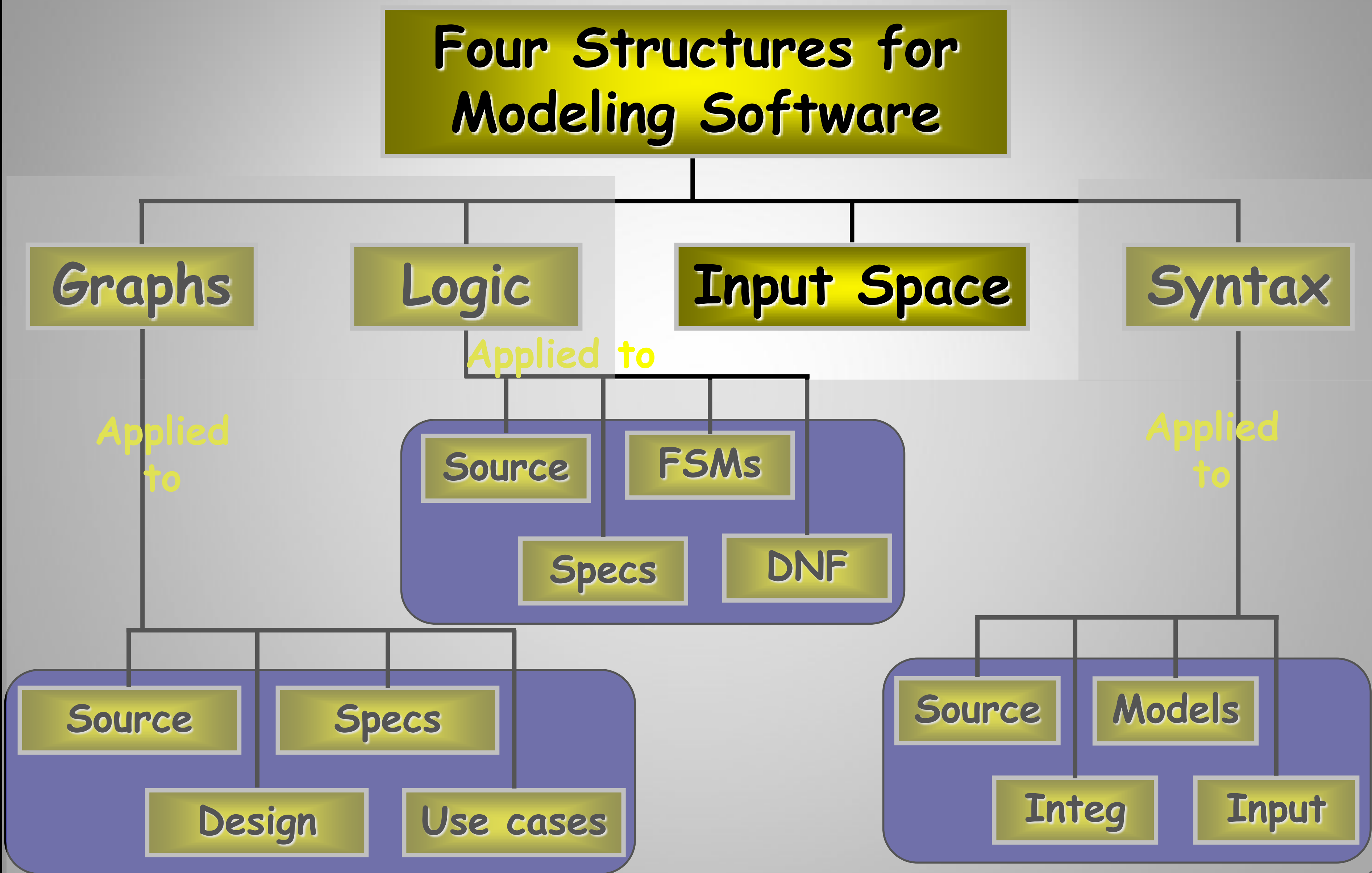
Introduction to Software Testing

Chapter 4
Input Space Partition Testing

Paul Ammann & Jeff Offutt

Updated by Sunae Shin

Ch. 4 : Input Space Coverage



Input Space Partitioning

- 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* can be
  - Parameters to a method
  - Data read from a file
  - Global variables
  - User level inputs
- Domain for each input parameter is partitioned into regions
- At least one value is chosen from each region

Benefits of Input Space Partitioning

- Can be equally applied at several levels of testing
  - Unit
  - Integration
  - System

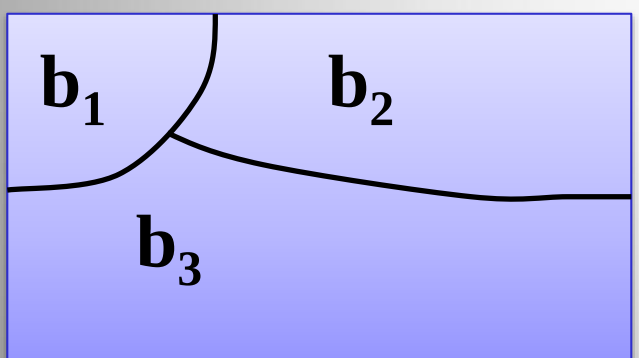
- Relatively easy to apply with no automation

- Easy to adjust the procedure to get more or fewer tests

- No implementation knowledge is needed
  - Everything is based on a description of the inputs

Partitioning Domains

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



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

$$\mathbf{\bigcup_{b \in Bq} b = D}$$

Input Space Partitioning – Assumptions

- Choose a value from each block
- Each value is assumed to be equally useful for testing
- Application to testing
  1. Find characteristics in the inputs : parameters, semantic descriptions, …
  2. Partition each characteristic
  3. Choose tests by combining values from characteristics

  – Example characteristics of partition
    - Input X is null
    - Order of the input file F (sorted, inverse sorted, arbitrary, …)
    - Input device (DVD, CD, VCR, computer, …)

Choosing Partitions

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

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

but … something's fishy …

What if the file is of length 1?

The file will be in all three blocks …

That is, disjointness is not satisfied

(disjointness – blocks must not overlap)

Solution:

Each characteristic should address just one property

File F sorted ascending

- b1 = true

- b2 = false

File F sorted descending

- b1 = true

- b2 = 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 attempt
- Different alternatives should be considered
- We model the input domain in five steps …

Modeling the Input Domain

- Step 1 : Identify testable functions
  - Individual methods have one testable function
  - In a class, each method often has the same characteristics
  - Programs have more complicated characteristics—modeling documents such as UML use cases 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**

Consider TriTyp program

Three *int* parameters

IDM for each parameter is identical

Reasonable characteristic : *Relation of side with zero*

TriTyp program

```java
// Jeff Offutt--Java version Feb 2003
// Classify triangles
import java.io.*;

class trityp
{
   private static String[] triTypes = { "", // Ignore 0.
           "scalene", "isosceles", "equilateral",
           "not a valid triangle"};
   private static String instructions = "This is the ancient
  TriTyp program.\nEnter three integers that represent the
  lengths of the sides of a triangle.\nThe triangle will be
  categorized as either scalene, isosceles, equilateral\n
  or invalid.\n";

 public static void main (String[] argv)
 {  // Driver program for trityp
    int A, B, C;
    int T;

    System.out.println (instructions);
    System.out.println ("Enter side 1: ");
    A = getN();
    System.out.println ("Enter side 2: ");
    B = getN();
    System.out.println ("Enter side 3: ");
    C = getN();
    T = Triang (A, B, C);

    System.out.println ("Result is: " + triTypes[T]);
 }

// ====================================
// The main triangle classification method
private static int Triang (int Side1, int Side2, int Side3)
{
   int triOut;

   // triOut is output from the routine:
   //    Triang = 1 if triangle is scalene
   //    Triang = 2 if triangle is isosceles
   //    Triang = 3 if triangle is equilateral
   //    Triang = 4 if not a triangle

   // After a quick confirmation that it's a valid
   // triangle, detect any sides of equal length
   if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
   {
      triOut = 4;
      return (triOut);
   }

   triOut = 0;
   if (Side1 == Side2)
      triOut = triOut + 1;
   if (Side1 == Side3)
      triOut = triOut + 2;
   if (Side2 == Side3)
      triOut = triOut + 3;
   if (triOut == 0)
   {  // Confirm it's a valid triangle before declaring
      // it to be scalene

```

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

Consider TriTyp again

The three parameters represent a *triangle*

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

Steps 1 & 2 : Interface vs 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

<u>Interface-Based Approach</u>
Two <u>parameters</u> : list, element
<u>Characteristics</u> :
- **list is null**
- **list is empty**

<u>Functionality-Based Approach</u>
Two <u>parameters</u> : list, element
<u>Characteristics</u> :
- **number of occurrences of element in list**
- **element occurs first in list**
- **element occurs last in list**

Step 3 : Modeling the Input Domain

- **Partitioning characteristics into blocks and values** is a very creative engineering step
  - **More blocks means more tests**

- The partitioning often flows directly from the definition of characteristics and both steps are sometimes 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
  - Explore boundaries of domains
  - Try to balance the number of blocks in each characteristic
  - Check for completeness and disjointness

Steps 3 : Interface vs 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)**

Interface-Based IDM – TriTyp

- TriTyp program had **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 – TriTyp (*cont*)

<u>Second</u> Characterization of TriTyp'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
- This is only complete because the inputs are integers (0 . . 1)

Possible values for partition q_1

Characteristic	b_1	b_2	b_3	b_4
Side1	2	1	0	-1

Test boundary conditions

Functionality-Based IDM – TriTyp

- 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

<u>Geometric</u> Characterization of TriTyp's Inputs

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

- Oops … something's fishy … equilateral is also isosceles !
- We need to refine the example to make characteristics valid

<u>Correct</u> Geometric Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

Functionality-Based IDM – TriTyp (*cont*)

- 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 – TriTyp (*cont*)

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

<u>Four</u> Characteristics for TriTyp

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**

Step 4 – Choosing Combinations of Values

- 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 …

<u>All Combinations (ACoC)</u> : 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 TriTyp results in 4*4*4 = 64 tests – too many ?

ISP Criteria – Each Choice

- 64 tests for TriTyp 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 (EC) : 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 TriTyp: 2, 2, 2
1, 1, 1
0, 0, 0
-1, -1, -1

ISP Criteria – Pair-Wise

- Each choice yields few tests – cheap but perhaps ineffective
- Another approach asks values to be combined with other values

<u>Pair-Wise (PW)</u> : 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

$$(\mathrm{Max}\,{}_{i=1}^{Q}(B_i)) * (\mathrm{Max}\,{}_{j=1,\, j!=i}^{Q}(B_j))$$

For TriTyp:	**2, 2, 2**	**2, 1, 1**	**2, 0, 0**	**2, -1, -1**
	1, 2, 1	**1, 1, 0**	**1, 0, -1**	**1, -1, 2**
	0, 2, 0	**0, 1, -1**	**0, 0, 2**	**0, -1, 1**
	-1, 2, -1	**-1, 1, 2**	**-1, 0, 1**	**-1, -1, 0**

ISP Criteria – Base Choice

- Testers sometimes recognize that certain values are important
- This uses domain knowledge of the program

<u>Base Choice (BC)</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 TriTyp: <u>Base</u>	**2, 2, 2**	**2, 2, 1**	**2, 1, 2**	**1, 2, 2**
		2, 2, 0	**2, 0, 2**	**0, 2, 2**
		2, 2, -1	**2, -1, 2**	**-1, 2, 2**

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
- The base choice is a crucial design decision
  - Test designers should document why the choices were made

ISP Criteria – Multiple Base Choice

- Testers sometimes have more than one logical base choice

<u>Multiple Base Choice (MBC)</u> : One or more base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choices in each other characteristic.

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

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

For TriTyp: <u>Base</u>

2, 2, 2	**2, 2, 0**	**2, 0, 2**	**0, 2, 2**
	2, 2, -1	**2, -1, 2**	**-1, 2, 2**
1, 1, 1	**1, 1, 0**	**1, 0, 1**	**0, 1, 1**
	1, 1, -1	**1, -1, 1**	**-1, 1, 1**

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 in practice

Example

```
public static int search (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//   else if element is in the list, return an index
//   of element in the list; else return -1
//   for example, search ([3,3,1], 3) = either 0 or 1
//       search ([1,7,5], 2) = -1
```

- **Characteristic partitioning:**

```
Characteristic: Location of element in list
       Block 1: element is first entry in list
       Block 2: element is last entry in list
       Block 3: element is in some position other than first or last
```

- **(a) “Location of element in list” fails the disjointness property. Give an example that illustrates this.**
- **(b) “Location of element in list” fails the completeness property. Give an example that illustrates this.**
- **(c) Supply one or more new partitions.**

Example - solution

- **(a)** Lots of examples can be found. One is: list = [3, 4, 3]; e = 3

  Another is: list = [3]; e = 3

- **(b)** The problem is that e may not be in the list: list = [5, 3]; e = 4

- **(c)** The easiest approach is to separate the characteristics into separate partitions:
  - Whether e is first in the list: true, false
  - Whether e is last in the list: true, false

  You might also consider:
  - Whether e is in list: true, false

Example

- **Derive input space partitioning tests for the GenericStack class with the following method signatures:**

```
• public GenericStack ();
• public void Push (Object X);
• public Object Pop ();
• public boolean IsEmt ();
```

Assume the usual semantics for the stack. Try to keep your partitioning simple, choose a small number of partitions and blocks.

(a) Define characteristics of inputs

(b) Partition the characteristics into blocks

(c) Define values for the blocks

Example - solution

- **There are 4 testable units.**

Typical characteristics for the implicit state are

- *Whether the stack is empty.*
  - *true (Value stack = [])*
  - *false (Values stack = ["cat"], ["cat", "hat"])*
- *The size of the stack.*
  - *0 (Value stack = [])*
  - *1 (Possible values stack = ["cat"], [null])*
  - *more than 1 (Possible values stack = ["cat", "hat"], ["cat", null], ["cat", "hat", "ox"])*
- *Whether the stack contains null entries*
  - *true (Possible values stack = [null], [null, "cat", null])*
  - *false (Possible values stack = ["cat", "hat"], ["cat", "hat", "ox"])*

A typical characteristic for `Object x` *is*

- *Whether x is null.*
  - *true (Value x = null)*
  - *false (Possible values x = "cat", "hat", "")*