Domain Testing Model

Domain testing models

- Domain analysis is a straightforward and effective way to select test values
- There are several domain testing models:
  - Meyer's equivalence class
  - …
  - Used in book: based on White and Cohen's paper in 1978

Domain analysis

- A domain is the set of all inputs accepted by the IUT
- Domain is defined by set of *boundary* conditions
  - Boolean expressions on the IUT input variables
    - Method parameters and instance variables
  - Example
    - Collection class that holds objects
    - Boundary condition: maximum number of items: 1000
- Fault model for domain testing?
  - IUT has incorrectly implemented a boundary
  - Example:
    - The IUT allows 999 items but reject 1000

Typical domain faults

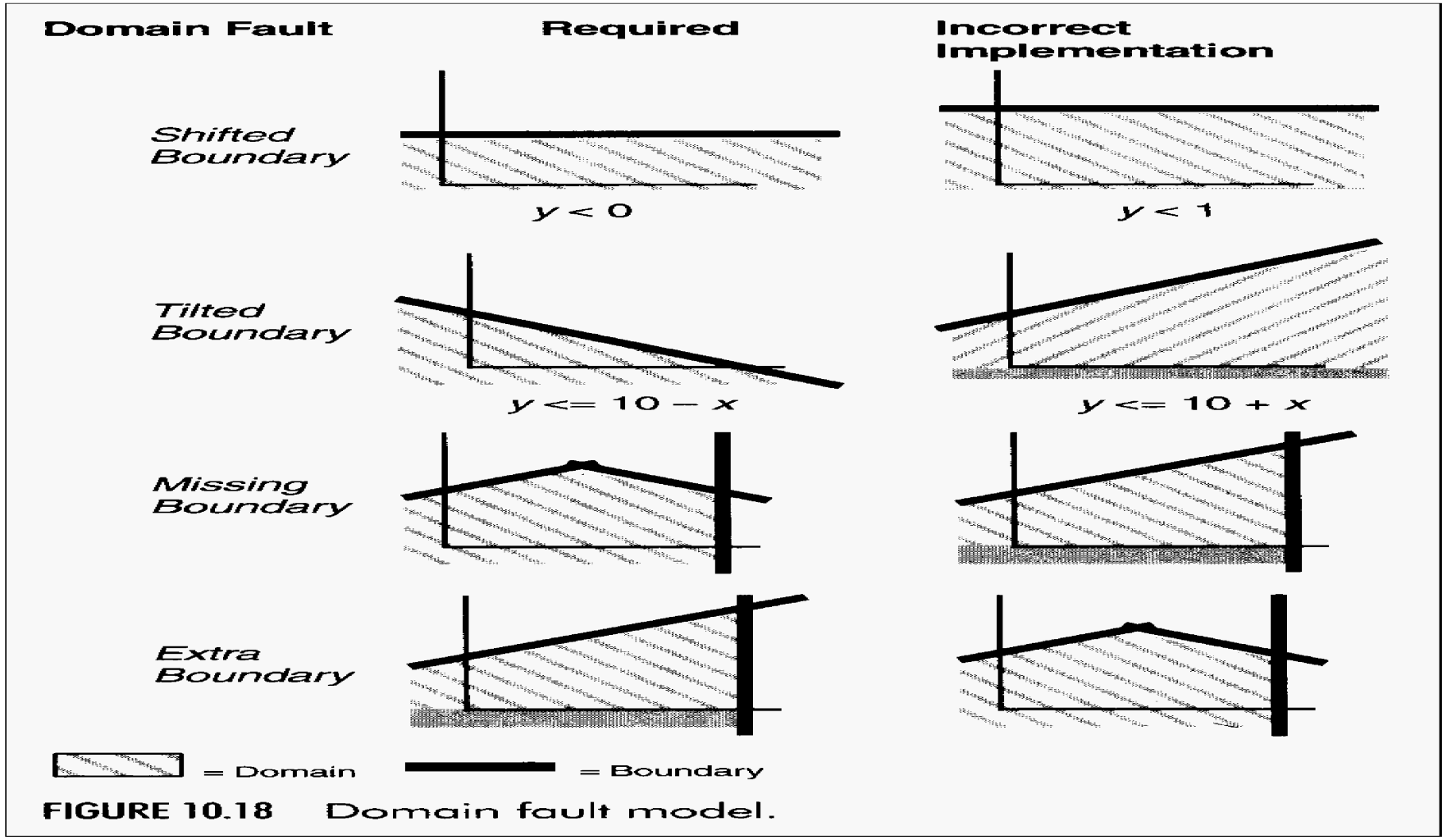


FIGURE 10.18 Domain fault model.

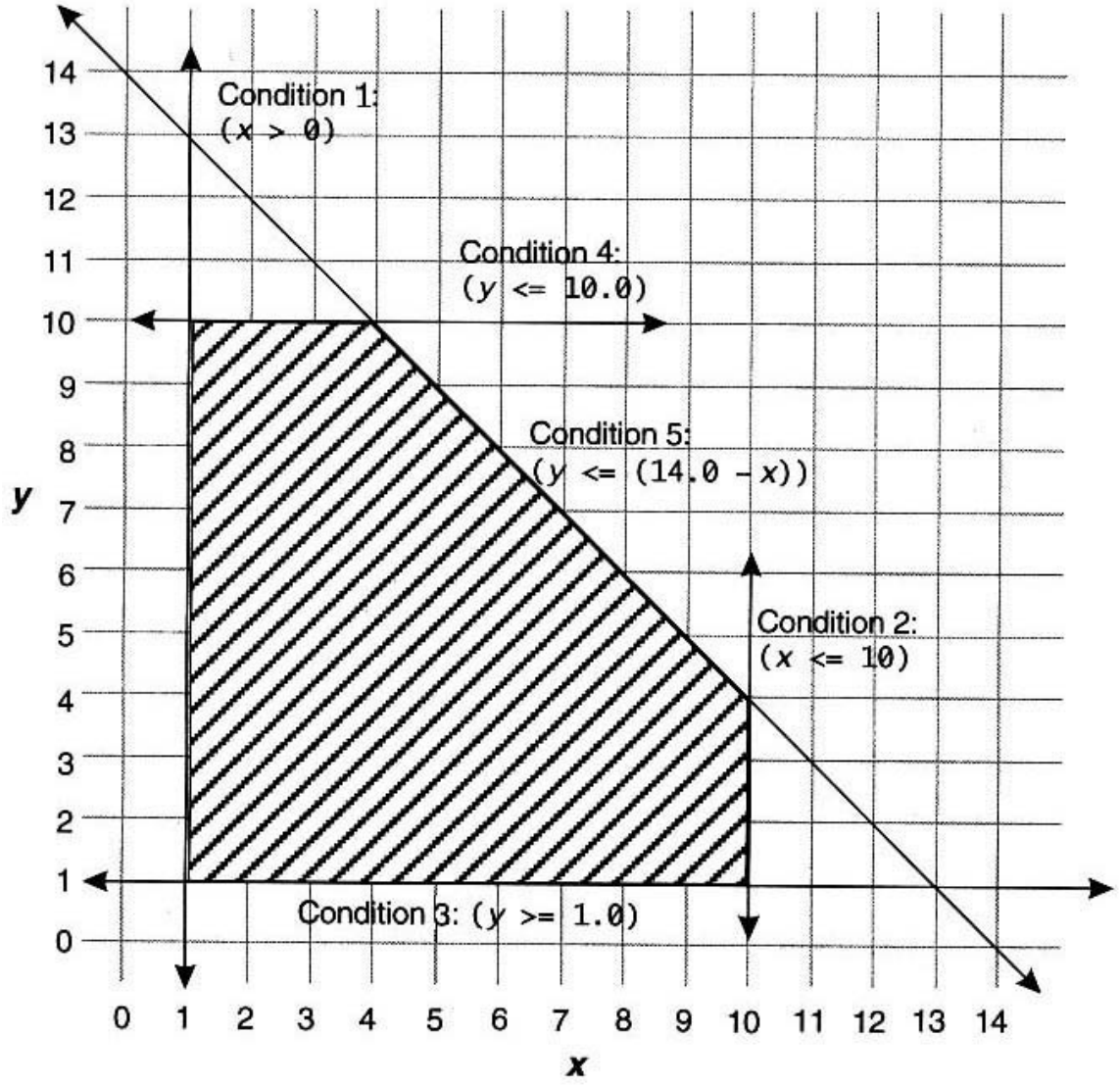
Development of a domain test suite

- A domain test suite is developed by domain analysis:
  1. Identify constraints for all input variables
  2. Select test values for each variable in each boundary
  3. Select test values for variables not given in the boundary
  4. Determine expected results for these input
- The results are represented in the domain matrix
- Example:
  - `void aFunction(int x, float y, Stack aStack)` with the following constraints:

```
assert (  (y >= 1.0 )       &&  // condition 1
          (x <= 10  )       &&  // condition 2
          (y <= 10.0)       &&  // condition 3
          (x > 0 )          && // condition 4
          (y <= 14.0 –x)  && // condition 5
          (! aStack.isFull)    // condition 6
```

Valid values for *aFunction*

- Just for x and y considering conditions 1 to 5
- For considering condition 6 need more dimensions



On, Off, In, Out points

- All domain testing selection criteria use **on** and **off** points
- With respect to a particular boundary
  - **On** points lie on the boundary
  - **Off** points lie off the boundary
    - But **as near as** possible
- With respect to all boundaries
  - **In** points satisfy all boundary conditions & are not on a boundary.
    - Normally used as typical values
  - **Out** points satisfy no boundary conditions & are not on a boundary.

Not useful

In points

- **In** points satisfy all boundary conditions & are not on a boundary
  - Meaning In points should not be On or Off points
- Consider example x >= 5 && x <= 10

- Is 5 an In point? No, it is on the boundary of for *x >= 5*

- Is 12 an In point? No, it is on the boundary for *x <= 10*

- Set of In points:] 5, 10 [

On and Off points - Open and closed boundaries

- Open Boundary - strict inequality (e.g., x > 0)
  - **On** point:
    - x = 0, makes boundary condition false
  - **Off** point
    - Two possible choices: -1 ✗ and 1 ✓
    - Which one?
    - Apply rule
- Closed Boundary - strict equality (e.g. y <= 10.0)
  - **On** point:
    - y = 10.0, makes boundary condition true
  - **Off** point:
    - Consider a 6-digit precision
    - Two possible choices: 9.999999 ✗ and 10.000001 ✓
    - Apply same rule

Rule: Off point must make the boundary condition true if On makes it false and vice-versa

What about OO systems?

- For primitive data types, applying domain analysis is straightforward
- Can we apply it to classes?
- Yes, if the boundary condition involves a single instance variable
  - e.g., aStack.size() <= 10
- Sometimes, boundary condition involves an abstract state of an object
- Abstract state
  - Represent a given behavior
  - Can involve several fields of the object
  - Can be represented by several boundary conditions
  - It may lead to a several test cases
- Solution: Define abstract On, Off and In points

An example

- Consider the following class

```
Class Account {
  AccountNumber number;
  Money balance;
  Date lastUpdate;
  …
}
```

- Account objects can be in one of three abstract state
  - **Open:** balance >= 0 and lastUpdate < 1825
  - **Overdrawn:** balance >= 0 and lastUpdate < 1825
  - **Inactive:** lastUpdate >= 1825

Three abstract states

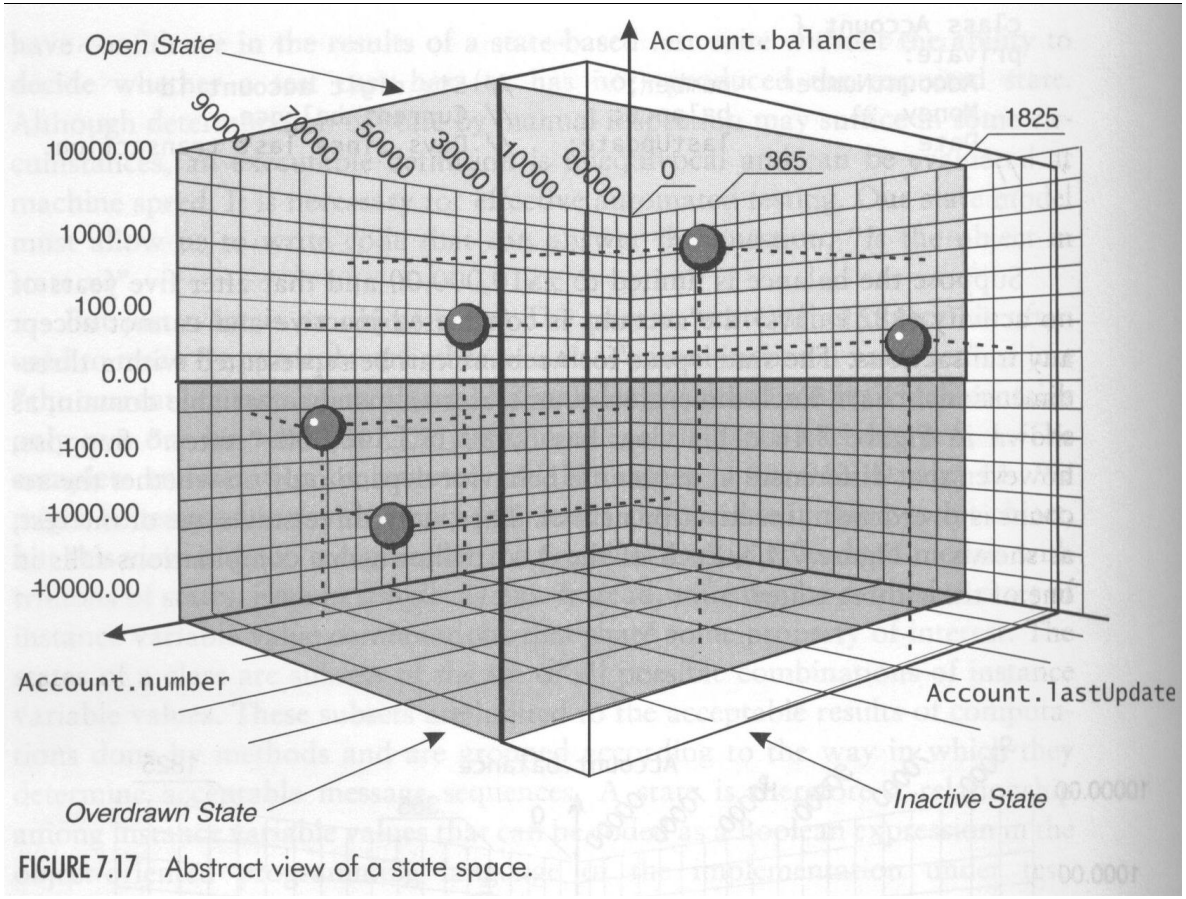


FIGURE 7.17 Abstract view of a state space.

Abstract state - State invariant

- Abstract state: A set of attribute value combinations that share some property of interest
- A valid state can be expressed with a state invariant
  - Boolean expression that can be checked
- Domain testing model to objects:
  - Treat each state invariant as a domain boundary

OO domain definitions

- For each abstract state, define abstract state on, off and in points
- An abstract state **on** point is a state such that the smallest possible change in some attribute will produce a state change
- An abstract state **off** point is a valid state that is not the focus state and differs from the focus state by the smallest possible change in some attribute
- An abstract state **in** point is neither an *on* or an *off* point (if possible)

Example: Stack class

- Abstract states of Stack:
  - *empty* → Stack.size == 0
  - *loaded* → Stack.size > 0 && Stack.size < MAXSTACK
  - *full* → Stack.size == MAXSTACK

Assuming MAXSTACK = 32767

Abstract state	Possible transitions	In Point	On Point	Off Point
empty	loaded	0	0	1
loaded	empty full	1<x<32766	1 32766	0 32767
full	loaded	32767	32767	32766

Example 2: Account class

- Abstract states of *Account*:
  - *Open* (balance >= 0 && lastUpdate < 1825)
  - *Overdrawn* (balance < 0 && lastUpdate < 1825)
  - *Inactive* (lastUpdate >= 1825)

State	Possible Transitions	In Point	On Point	Off Point
Open	Overdrawn Inactive	(x > 0, y < 1824)	(0, y < 1824) (x > 0, 1824)	(-0.01, y < 1824) (x > 0 , 1825)
Overdrawn	Open Inactive	(x < 0, y < 1824)	(-0.01, y < 1825) (x < 0, 1824)	(0, y < 1824) (x < 0 ,1825)
Inactive	Open Overdrawn	(x , y > 1825)	(x > 0, 1825) (x < 0, 1825)	(x > 0, 1824) (x < 0 ,1824)

One by one selection criteria

- Each test case exercises a single boundary condition
- 1 x 1 (“one by one”) domain testing strategy calls for one **on** point and one **off** (or more) point for each domain boundary
  - Number of off points depends on type of condition

- Using the abstract state model, 1 x 1 is applicable to the full range of data types used in object oriented programming.

One by one selection criteria - 2

- For relational conditions, e.g. `x <= 10`
  - One **on** point and one **off** point

- For strict equality conditions, e.g. `x == 10`
  - One **on** point and two **off** points

- For nonscalar types: one **on** point and one **off** point

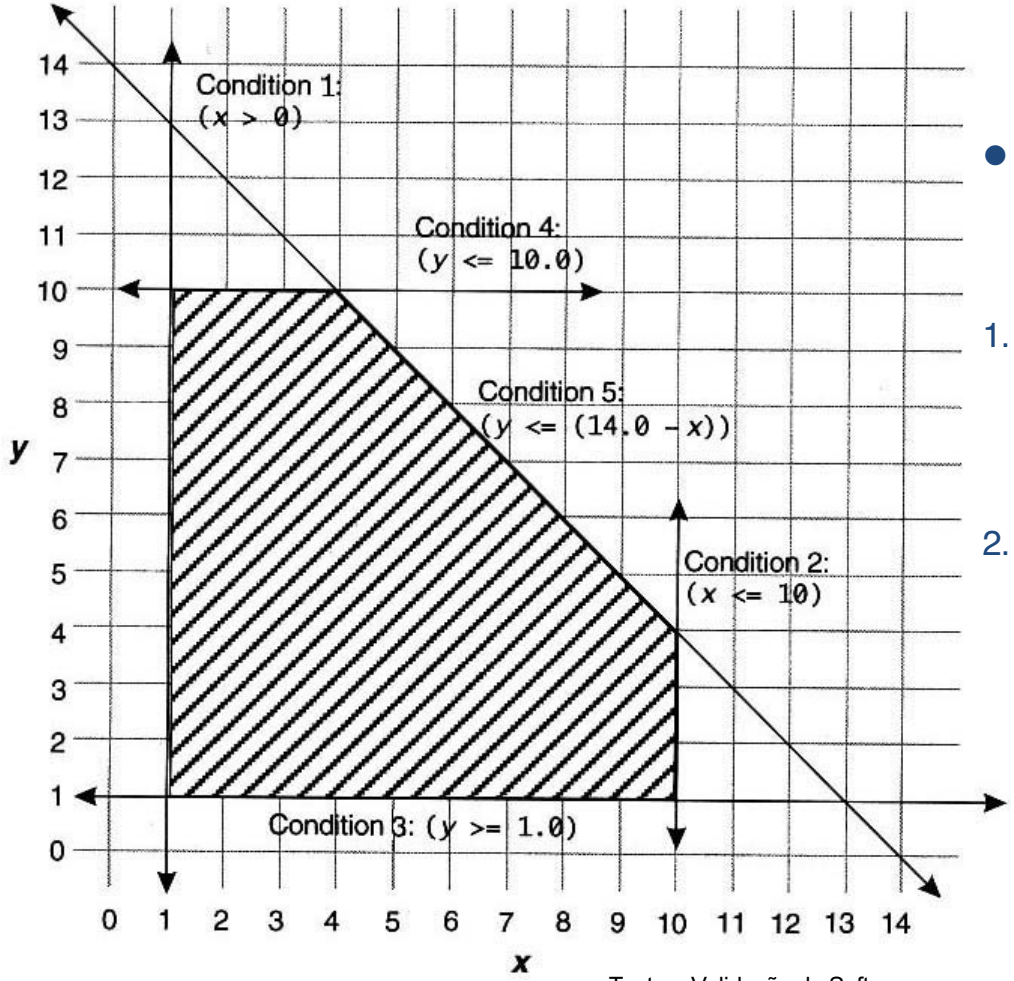
- For abstract state invariants: one **on** point and at least one **off** point

aFunction example

```
void aFunction(int x, float y, Stack aStack){
  assert( (y >= 1.0)       &&
          (x <= 10)        &&
          (y <= 10.0)      &&
          (x > 0)          &&
          (y <= 14.0 - x)  &&
          (!aStack.isFull());
  ......
}
```

	ON	OFF
(y >= 1.0)	1.0	0.999999
(x <= 10)	10	11
(y <= 10.0)	10.0	10.000001
(x > 0)	0	1
(y <= 14.0 - x)	?	?
(!aStack.isFull())	Full (32767)	Loaded (32766)

How to handle boundaries with 2 variables?



- On and Off for (y <= 14.0 – x)?

1. Fix a value for x
   - x in [4, 10]
   - Pick mean value
     - x = 7
2. Compute On and Off for y given x = 7
   - On – 7
   - Off – 7,000001

Domain Matrix design

- How to design the domain test suite?
- Define **on** and **off** points for each boundary condition
- Add expected results and **in** points for other values
  - Each test case only has one **off** or **on** point
  - Select **in** points for all other values in the test case
  - Avoid to repeat **in** points. Why?
- Result
  - Minimal test cases
  - Input variables to exercise boundary conditions
  - For any type of variable types
    - Including abstract complex types (objects)

Domain matrix

Boundary			Test Cases											
Variable	Condition	Type	1	2	3	4	5	6	7	8	9	10	11	12
x	>0	On	0											
		Off		1										
	<= 10	On			10									
		Off				11								
	Typical	In					2	3	4	5	6	7	8	9
y	> = 1.00000	On					1.00000							
		Off						0.99999						
	<= 10.00000	On							10.0000					
		Off								10.0001				
	y < = 14.0 –*x*	On									7.00000			
		Off										7.00001		
	Typical	In	7.97320	1.78386	8.11532	6.14728							3.33333	5.11205
aStack	!isFull( )	On											32766	
		Off												32767
	Typical	In	25	18432	4096	1	2	732	32718	9183	3718	20501		
Expected Result			✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗

Key: ✗ = IUT rejects this input, ✓ = IUT accepts this value and produces correct output; Stack values are size of stack.

FIGURE 10.21 Domain Matrix for `aFunction`.

Errors in the figure:

- The value of x for test case 9 must be 7, so that y is an *on* point.
- The in point for y test cases 3, 4 and 12 is also wrong (all invalidate y <=14-x)

What does this approach achieve?

- This is a systematic sampling approach to test case design

- Using *boundary values* for the tests offers a few benefits:
  - They will expose errors that mis-specify a boundary.
    - These can be coding errors (off-by-one errors such as saying “less than” instead of “less than or equal”) or typing mistakes (such as entering 57 instead of 75 as the constant that defines the boundary).
    - Mis-specification can also result from ambiguity or confusion about the decision rule that defines the boundary.
  - Non-boundary values are less likely to expose these errors

- It assumes that there is a single error