Module 2

Functional Testing

Functional Testing

- Introduced by W.E Howden in late 1970s
- Also called Black box testing
- Involves testing a code / design without the knowledge of its internals (like the actual code structure, statements, design details etc.)
- Deals only with inputs and outputs applied to code / design / requirements
- Test Cases deal only with the inputs and outputs
- Program P - function that takes some inputs and produces outputs
  - Given inputs x_i, P computes outputs y_i

    such that $y_i = P(x_i)$
  - Eg: Sorting program
    - Input x_i - array of numbers
    - Output y_i - array in sorted order

Steps in Functional Testing

- Identify the domain of each i/p and each o/p variable
- Select values from domain of each variable having important properties
- Consider *combinations* of special values from different i/p domains to design testcases
- Consider input values such that the program under test produces special values from the domains of the output variables

Note: *Need to have minimal context information to obtain relevant values for inputs & outputs - knowing the context will help decide the right inputs to be given for the testcases*

Types of Functional Testing

- Boundary value analysis
- Equivalence class partitioning
- Decision tables
- Pair-wise testing
- Random testing

1st TECHNIQUE - BOUNDARY VALUE TESTING

- Also called Input domain testing
- Based on the fact that input values near the boundary have higher chances of errors
- Boundary values:
  - Values lying on the boundary
  - Values just above the boundary
  - Values just below the boundary
- Example: 1 to 10 [range]
- Consider 2 variables $a \le x \le b$ and another set $c \le y \le d$
- X varying from 1 to 100

- Boundary value Analysis is a testing technique which aims to detect errors related to boundary values

    - Range related errors
    - Syntactically there may be no errors
    - Logical operators are prone to generate such errors
    - Identify lower limit value and upper limit value
    - Hence, check for UL-1, UL, UL+1, LL-1, LL, LL+1 (UL – Upper limit and LL – lower limit)

- Areas of applications such as string, banking transactions etc.
- Saves time in testing

BOUNDARY VALUE ANALYSIS

- Based on experience / heuristics:
  - Testing boundary conditions of equivalence classes is more effective, i.e. values directly on, above, and beneath edges of classes
  - If a system behaves correctly at boundary values, than it probably will work correctly at "middle" values
- Choose input boundary values as tests in input classes instead of, or additional to arbitrary values
- Choose also inputs that invoke output boundary values (values on the boundary of output classes)
- Example strategy as extension of equivalence class partitioning:
  - choose one (or more) arbitrary value(s) in each eq. class
  - choose values exactly on lower and upper boundaries of eq. class
  - choose values immediately below and above each boundary (if applicable)

- Thus, BVA includes for testing, the test inputs to be values lying on the boundary
- Value just above the boundary value (lower limit range)
- Value just below the boundary value (upper limit range)

BOUNDARY VALUE ANALYSIS

- Boundary conditions are those situations at the edge of the planned operational limits of the software
- Test valid data just inside the boundary, test the last possible data and test the invalid data just outside the boundary

- Boundary types: Numeric, Character, Position, quantity, Speed, Location, Size….
- Think of First / Last, Min / Max, Start / Finish, Over / Under, Empty / Full, Slowest / Fastest, Soonest / Latest

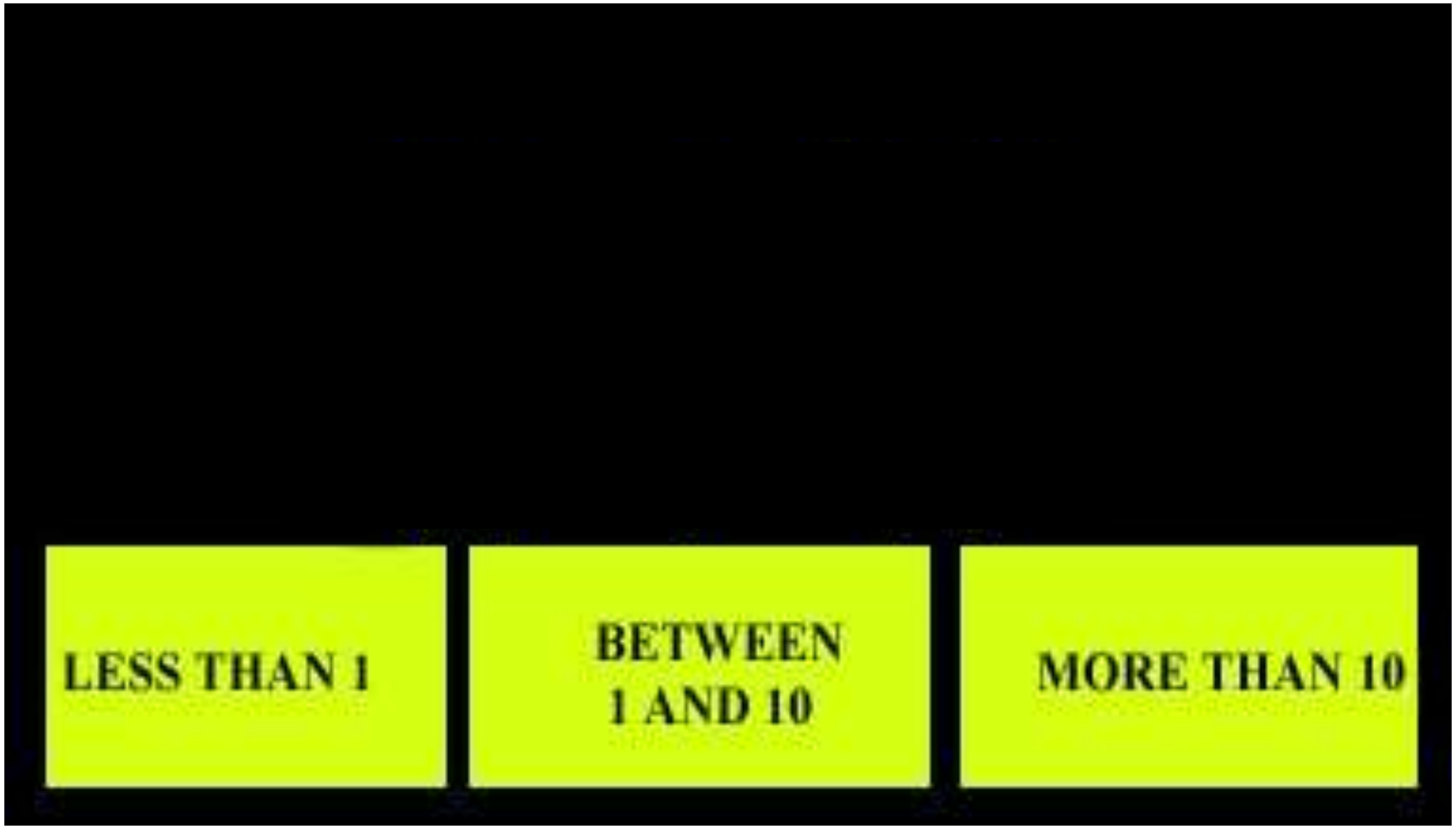
BOUNDARY VALUE BASIC EXAMPLE

- Let’s suppose that you wanted to test a program that only accepted integer values from 1 to 10
  - The possible test cases for such a program would be the range of all integers

- How can we narrow the number of test cases down from all integers to a few good test cases?

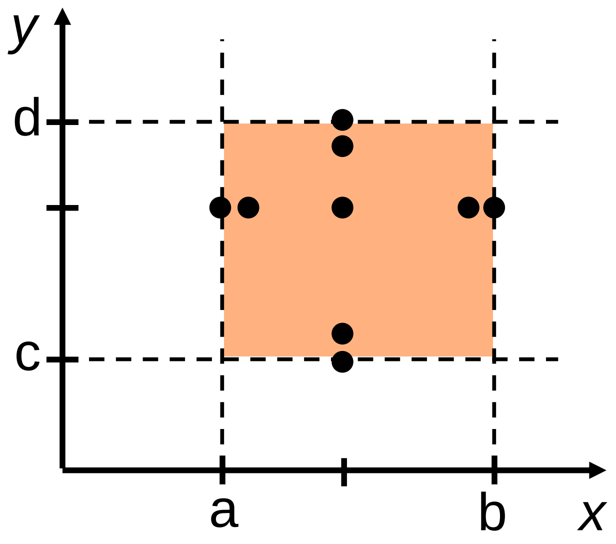
- What are boundary values?
  - Values lying on the boundary (upper and lower limit) -- 1 and 10)
  - Values just above the boundary (lower boundary -- 2, 2.5 1.5)
  - Values just below the boundary (Upper boundary , 9, 9.5)

SAMPLE BOUNDARY VALUE



BOUNDARY VALUE TESTING

- Detect errors to do with input domain bounds
- For integer input x with domain [a,b], test input values around a and b
- # tests = for *n* inputs, 4*n*+1 input combinations
  - Min, min+, nominal, max-, max values
- Assumes:
  - Independent quantity inputs
  - Single-fault assumption
  - At any point of time only one variable can be at fault and not all variables
  - Eg: if x is in boundary then y has to be a nominal value for 2 input variables



SINGLE VALUE ASSUMPTION

- Consider x values y values , (x, y) where x=100 and y =300,
- If x is in boundary state then y cannot be faulty hence x is boundary then y should be a nominal / middle value / assumed correct value
    - (100, 200)
    - (101, 200)
    - (50, 200)
    - (299, 200)
    - (300, 200)
  - For y, where y is in boundary then x should be a nominal value
    - (200, 100)
    - (200, 101)
    - (200, 50) (Repeated – Hence, written for in one case)
    - (200, 299)
    - (200, 300)
    - Hence test cases should have 4n+1 = 9 test cases

Example 2

- Consider a program for determining the previous (next date. Input: day, month, year, with valid ranges as
- 1≤ month ≥ 12
- 1≤ day ≥ 31
- 2000 ≤ year ≥ 2020

- Solution: Let us assume 1, 12, 1, 31 and 1900, 2000 as boundary values
- Since, there are 3 variables, we should have 4(3) + 1 = 13 Test Cases
- 1 variable in each case, to be at boundary and other 3 variables to be nominal values
- **Apply (4n+1) formula to generate the total number of test cases**

- Possible outputs – Previous date or Invalid date

Test case	Month	Day	Year	Expected output	Actual Output
001			1900 (min)		
002			1901(min+1)		
003			1960 (nom)		
004			1999 (max-1)		
005			2000(max+1)		
	6	15		14th June 1900	
	6	15		14th June 1901	
	6	15		14th June 1960	
	6	15		14th June 1999	
	6	15		14th June 2000	
006	6	1	1960	31st May 1960	
007	6	2	1960	1st may 1960	
008	6	30	1960	29th June 1960	
009	6	31	1960	30th June 1960	
010	1	15	1960	14th Jan 1960	
011	2	15	1960	1st Jan 1960	
012	11	15	1960	14th Nov 1960	
013	12	15	1960	14th Dec 1960	

- Possible outputs – Next date or Invalid date

Test case	Month	Day	Year	Expected output	Actual Output
001			2000 (min)		
002			2001(min+1)		
003			2010 (nom)		
004			2019 (max-1)		
005			2020(max)		
	6	15	2000 (min)	16/06/2000	
	6	15	2001(min+1)	16/06/2001	
	6	15	2010 (nom)	16/06/2010	
	6	15	2019 (max-1)	16/06/2019	
	6	15	2020(max)	16/06/2020	
006	6	1	2010	02/06/2010	
007	6	2	2010	03/06/2010	
008	6	30	2010	01/07/2010	
009	6	31	2010	Invalid input	
010	1	15	2010	16/01/2010	
011	2	15	2010	16/02/2010	
012	11	15	2010	16/11/2010	
013	12	15	2010	16/12/2010	

Example 3

- Consider an example of triangle where the condition is sum of 2 sides cannot be greater than the third side, write test cases which will consider BVA
- Note: If sum of any 2 sides is greater than the third side, then indicate as Not a Triangle
- Also, consider Single Fault Assumption

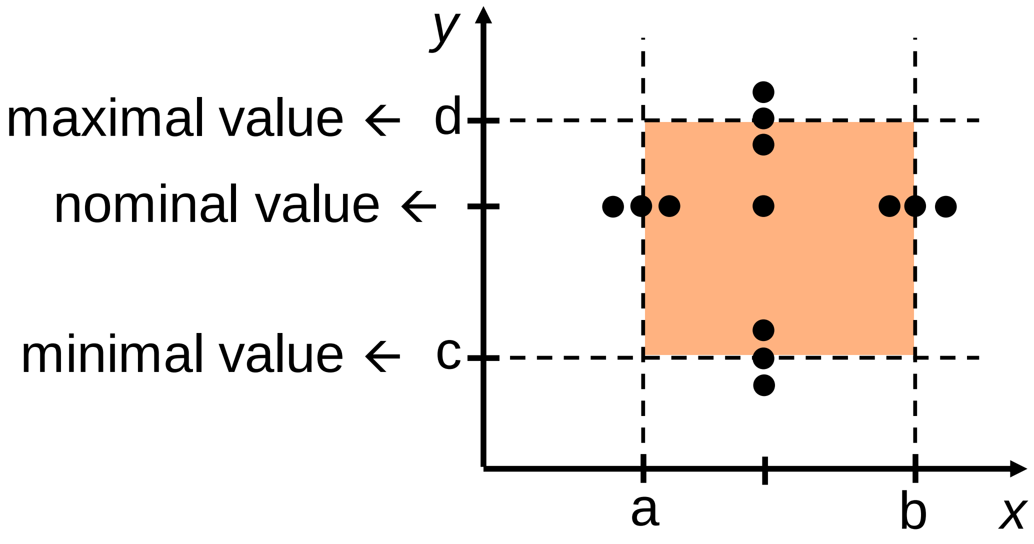
- Solution: Let us assume 1 and 100 as boundary values
- Since, there are 3 variables, we should have 4(3) + 1 = 13 Test Cases
- 1 variable in each case, to be at boundary and other 2 variables to be nominal values

- Sum of two sides should not be greater than third side

Test case	X	Y	Z	Expected output	Actual Output
001	50	50	1	Isosceles	
002	50	50	2	Isosceles	
003	50	50	50	Equilateral	
004	50	50	99	Isosceles	
005	50	50	100	Not a triangle	
006	50	1	50	Isosceles	
007	50	2	50	Isosceles	
008	50	99	50	Isosceles	
009	50	100	50	Not a triangle	
010	1	50	50	Isosceles	
011	2	50	50	Isosceles	
012	99	50	50	Isosceles	
013	100	50	50	Not a triangle	

ROBUSTNESS BOUNDARY VALUE

- Also test values outside the domain
- # tests =
- For *n* input variables, (**6n+1)** input combinations
- A test tuple: <*x*nom, *y*max+1, expected output>



6n +1 NUMBER OF TEST CASES

Test case	X	Y	Z	Expected output	Actual Output
001	50	50	0 (min-1)	Invalid	
002	50	50	1(min)	Isosceles	
003	50	50	2(min+1)	Isosceles	
004	50	50	50(Nom)	Equilateral	
005	50	50	99(max-1)	Isosceles	
006	50	50	100(max)	Not a triangle	
007	50	50	101(max+1)	Invalid	
008	50	0	50	Invalid	
009	50	1	50	Isosceles	
010	50	2	50	Isosceles	
011	50	99	50	Isosceles	
012	50	100	50	Not a triangle	
013	50	101	50	Invalid	
014	0	50	50	Invalid	
015	1	50	50	Isosceles	
016	2	50	50	Isosceles	
017	99	50	50	Isosceles	
018	100	50	50	Not a triangle	
019	101	50	50	Invalid	

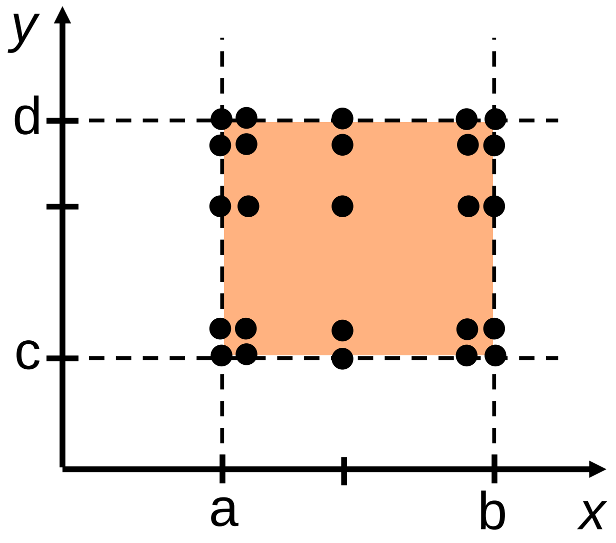
Next date Example – Single Fault Assumption (6n+1) number of Test cases

Test case	Month (1-12)	Day (1-31)	Year (2000-2020)	Expected output	Actual Output
001	6	15	1999 (min-1)	Invalid	
002	6	15	2000 (min)	14th June 2000	
003	6	15	2001(min+1)	14th June 2001	
004	6	15	2010 (nom)	14th June 2010	
005	6	15	2019 (max-1)	14th June 2019	
006	6	15	2020 (max)	14th June 2020	
007	6	15	2021(max+1)	Invalid	
008	6	0	2010	Invalid	
009	6	1	2010	May 31st 2010	
010	6	2	2010	1st May 2010	
011	6	30	2010	29th June 2010	
012	6	31	2010	30th June 2010	
013	6	32	2010	Invalid	
014	0	15	2010	Invalid	
015	1	15	2010	31st December 2009	
016	2	15	2010	1st Jan 2010	
017	11	15	2010	14th Nov 2010	

WORST-CASE BOUNDARY VALUE

- **Multiple-fault assumption**
- # tests =
- for *n* input variables, 5^n input combinations
- Worse Case Testing is an extension of BVA without Single Fault Assumption. Hence, it is also treated as BVA to be a proper subset of worse case testing

- Hence, total no of test cases will be 5^n combinations where n is the number of input variables



2 Examples

- Generate test cases for Triangle problem and next date function

- Solution: Both cases, has 3 variables hence, number of test cases will be 125 test cases

- Hence, consider,
    - x, y constant and vary z
    - Similarly, keep x, z constant and vary y
    - Keep y, z constant and vary x

- Thus, it is Cartesian product

- For case when x=1, then y = (1,2,50, 99, 100),
- for x= 2, y = (1,2,50, 99, 100),
- for x= 50, y = (1,2,50, 99, 100),
- for x= 99, y = (1,2,50, 99, 100),
- for x= 100, y = (1,2,50, 99, 100),

- Similarly, consider x with z
- Then y will take values having x and z
- Lastly, z will take values having x and y constant

- Do the similar exercise for next date example

Example for Worst Case BVA for variable x and y ranging from 100 to 300 - Number of Test cases is 5^n

Test case	X	Y	Expected Output	Actual Output
001	100	100		
002	100	101		
003	100	200		
004	100	299		
005	100	300		
006	101	100		
007	101	101		
008	101	200		
009	101	299		
010	101	300		
011	200	100		
012	200	101		
013	200	200		
014	200	299		
015	200	300		
016	299	100		
017	299	101		
018	299	200		
019	299	299		
020	299	300		
021	300	100		
022	300	101		
023	300	200		

- It is also an extended version of BVA.
- It ALSO conducts tests for cases when extreme values are exceeded with values slightly greater than the maximum and a value slightly less than minimum
- Hence, it is a stronger version of BV testing.
- Hence, move beyond the valid input domain range
- Thus, robustness testing will contain (6n+1) test cases.

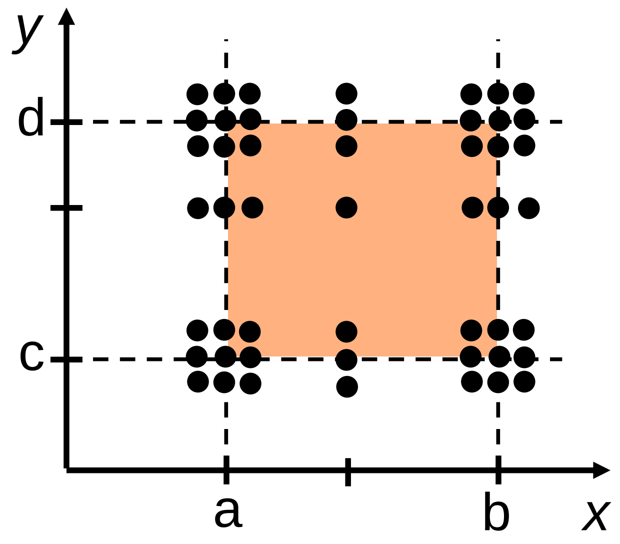
Examples

- Generate test cases with robustness testing for triangle problem and next date problem

- Solution: Both cases will have 6(3) + 1 test cases = 19 Test cases

WORST-CASE ROBUSTNESS BV

- Multiple-fault assumption, tests also outside the domain
- # tests =
- for n input variables, 7^n input combinations



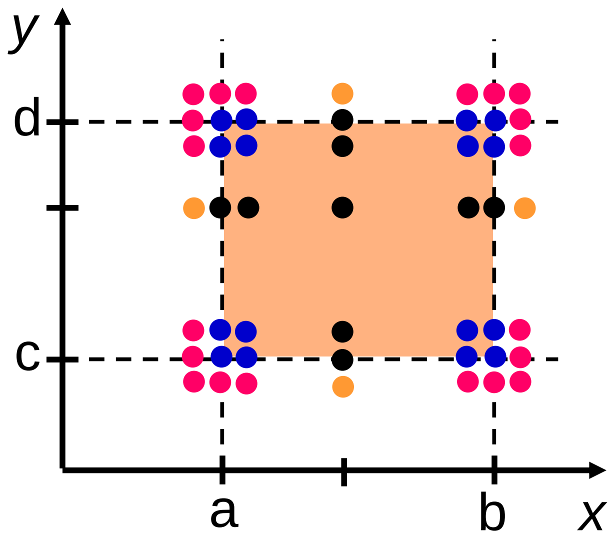
SUBSUME RELATIONS

$A \rightarrow B$ Technique A subsumes technique B if A tests at least what B tests (possibly more)

Which subsume relations for boundary value variants?

(Assume one fixed nominal value for each input)

$$\begin{tikzcd}
 & \text{boundary value} & \\
\text{worst-case} \arrow[ur] & & \text{robustness} \arrow[ul] \\
 & \text{worst-case robustness} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$



BOUNDARY VALUE TESTING SUMMARY

- Coverage: not good
- #tests: moderate to very many
- Usage: straightforward, easy to implement

- When to use:
  - independent inputs
  - enumerable quantities, e.g. age
  - (obviously) when suspecting boundary errors

- See literature:
  - Patton (chapter 5, pages 70-74)
  - Jorgensen (chapter 5)
  - Zhu et al. (section 4.3)

Decision table

Decision Table Based Testing

- Decision table is based on logical relationships just as the truth table.

- Decision Table is a tool that helps us look at the combination of conditions
  - Completeness of conditions
  - Inconsistency of conditions

Decision Table Based Testing

- Originally known as Cause and Effect Graphing
  - Done with a graphical technique that expressed AND-OR-NOT logic.
  - Causes and Effects were graphed like circuit components
  - Inputs to a circuit “caused” outputs (effects)
- Equivalent to forming a decision table in which:
  - inputs are conditions
  - outputs are actions
- Test every (possible) rule in the decision table.
- Recommended for logically complex situations.
- Excellent example of Model-Based Testing (MBT)

Content of a Decision Table

- Conditions
  - Binary in a Limited Entry Decision Table (LEDT)
  - Finite set in an Extended Entry Decision Table
  - Condition stub
  - Condition entries
- Actions
  - Also binary, either do or skip
  - The “impossible” action
- Rules
  - A rule consists of condition entries and action entries
  - A complete, non-redundant LEDT with n conditions has 2^n rules
  - Logically impossible combinations of conditions are “impossible rules”, denoted by an entry in the impossible action

Decision Table

- **Decision tables** are a precise yet compact way to model complicated logic. Decision tables, like *if-then-else* and *switch-case* statements, associate conditions with actions to perform. But, unlike the control structures found in traditional programming languages, decision tables can associate many independent conditions with several actions in an elegant way.
- Decision tables make it easy to observe that all possible conditions are accounted for.

- Decision tables can be used for:
  - Specifying complex program logic
  - Generating test cases (Also known as *logic-based testing)*

- *Logic-based testing* is considered as:
  - Structural testing when applied to structure (i.e. control flow graph of an implementation).
  - Functional testing when applied to a specification.
- **Decision** tree is a two dimensional matrix. It is divided into four **parts**, condition stub, action stub, condition entry, and action entry

Steps

- **Step** 1: Identify the conditions and their values. ...
- **Step** 2: Identify Actions. ...
- **Step** 3: **Draw** the Condition and Actions. ...
- **Step** 4: Calculate Number of Rules. ...
- **Step** 5: Populate the Condition Alternatives. ...
- **Step** 6: Fill in the Action Entries. ...
- **Step** 7: Reduce the **Table** if Necessary

Decision Tables - Structure

Conditions - *(Condition stub)*	Condition Alternatives – *(Condition Entry)*
Actions – *(Action Stub)*	Action Entries

- Each condition corresponds to a variable, relation or predicate
- Possible values for conditions are listed among the condition alternatives
    - Boolean values (True / False) – Limited Entry Decision Tables
    - Several values – Extended Entry Decision Tables
    - Don’t care value

- Each action is a procedure or operation to perform
- The entries specify whether (or in what order) the action is to be performed

- To express the program logic we can use a limited-entry decision table consisting of 4 areas called the *condition stub*, *condition entry*, *action stub* and the *action entry*:

Condition entry

Condition stub

Action stub

	Rule1	Rule2	Rule3	Rule4
Condition1	Yes	Yes	No	No
Condition2	Yes	X	No	X
Condition3	No	Yes	No	X
Condition4	No	Yes	No	Yes
Action1	Yes	Yes	No	No
Action2	No	No	Yes	No
Action3	No	No	No	Yes

Action Entry

- We can specify *default rules* to indicate the action to be taken when none of the other rules apply.
- When using decision tables as a test tool, default rules and their associated predicates must be explicitly provided.

	Rule5	Rule6	Rule7	Rule8
Condition1	X	No	Yes	Yes
Condition2	X	Yes	X	No
Condition3	Yes	X	No	No
Condition4	No	No	Yes	X
Default action	Yes	Yes	Yes	Yes

Decision Table - Example

Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized	Y	N	Y	N	Y	N	Y	N
Actions	Heck the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Printer Troubleshooting

Decision Tables - Usage

- The use of the decision-table model is applicable when:
  - The specification is given or can be converted to a decision table
  - The order in which the predicates are evaluated does not affect the interpretation of the rules or resulting action
  - The order of rule evaluation has no effect on resulting action
  - Once a rule is satisfied and the action selected, no other rule need be examined
  - The order of executing actions in a satisfied rule is of no consequence

- The restrictions do not in reality eliminate many potential applications.
  - In most applications, the order in which the predicates are evaluated is immaterial.
  - Some specific ordering may be more efficient than some other but in general the ordering is not inherent in the program's logic.

Decision Tables - Issues

- Before using the tables, ensure:
  - rules must be complete
    - every combination of predicate truth values plus default cases are explicit in the decision table
  - rules must be consistent
    - every combination of predicate truth values results in only one action or set of actions

Test Case Design

- To identify test cases with decision tables, we interpret conditions as inputs, and actions as outputs.
- Sometimes conditions end up referring to equivalence classes of inputs, and actions refer to major functional processing portions of the item being tested.
- The rules are then interpreted as test cases.

Definition

Condition entry

	Rule1	Rule2	Rule3	Rule4
Condition1	Yes	Yes	No	No
Condition2	Yes	X	No	X
Condition3	No	Yes	No	X
Condition4	No	Yes	No	Yes
Action1	Yes	Yes	No	No
Action2	No	No	Yes	No
Action3	No	No	No	Yes

Condition stub

Action stub

Action Entry

Example 1- Email

Email	False	True	False	True
Password	False	False	True	True
Expected Result	Error: Please enter email	Error: Please enter password	Error: Enter Email	Login processed

2*2=4

email	Blank	Blank	Blank	Invalid	Invalid	Invalid	Valid	Valid	Valid
psw	Blank	Invalid	Valid	Blank	Invalid	Valid	Blank	Invalid	Valid
Exp o/p	Error: Enter email	Error: Enter email	Error: Enter email	Error: Enter valid email	Error: Login Failed	Error: Enter valid email	Error; Enter psw	Error: Login Failed	------
Show o/p	Login Page	Login Page	Login Page	Login Page	Login Page	Login Page	Login Page	Login Page	Home Page

No of combinations:
3 power 2=9

Example 2- Library

User registered	False	False	False	False	True	True	True	True
No dues	False	False	True	True	False	False	True	True
Under borrow limit	False	True	False	True	False	True	False	True
Borrow book	No	No	No	No	No	No	No	Yes

No of test cases = 2*2*2 = 8

Simplified Library Example by following simplification step

User registered	False	False	True	True	True
No dues	False	True	False	True	True
Under borrow limit	- (Don't Care)	- (Don't Care)	- (Don't Care)	False	True
Borrow book	No	No	No	No	Yes

$\Longrightarrow$

User registered	False	True	True	True
No dues	False	False	True	True
Under borrow limit	-	- (Don't Care)	False	True
Borrow book	-	No	No	Yes

Decision Table for the Triangle Problem

Conditions											
C1: a < b+c?	F	T	T	T	T	T	T	T	T	T	T
C2: b < a+c?	-	F	T	T	T	T	T	T	T	T	T
C3: c < a+b?	-	-	F	T	T	T	T	T	T	T	T
C4: a=b?	-	-	-	T	T	T	T	F	F	F	F
C5: a=c?	-	-	-	T	T	F	F	T	T	F	F
C6: b=c?	-	-	-	T	F	T	F	T	F	T	F
Actions											
A1: Not a Triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: Impossible					X	X		X			

Test Cases for the Triangle Problem

Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

Decision Table for NextDate (First Attempt)

- Let us consider the following equivalence classes:

M1= {month | month has 30 days}
M2= {month | month has 31 days}
M3= {month | month is February}
D1= {day | 1 ≤ day ≤ 28}
D2= {day | day = 29}
D3= {day | day = 30}
D4= {day | day=31}
Y1= {year | year = 2000}
Y2= {year | year is a common year)
Y3= {year | year is a non century leap year}

Note year Y2 is a set of years between 1812 and 2012 evenly divisible by 4 excluding the year 2000

Decision Table for NextDate (1)

Conditions	1	2	3	4	5	6	7	8
C1: month in	M1	M1	M1	M1	M2	M2	M2	M2
C2: day in	D1	D2	D3	D4	D1	D2	D3	D4
C3: year in	-	-	-	-	-	-	-	-
Rule count	3	3	3	3	3	3	3	3
Actions								
A1: Impossible				X				
A2: Increment day	X	X			X	X	X	
A3: Reset day			X					X
A4: Increment month			X					?
A5: reset month								?
A6: Increment year								?

Decision Table for NextDate (2)

Conditions	9	10	11	12	13	14	15	16
C1: month in	M3	M3	M3	M3	M3	M3	M3	M3
C2: day in	D1	D1	D1	D2	D2	D2	D3	D3
C3: year in	Y1	Y2	Y3	Y1	Y2	Y3	-	-
Rule count	1	1	1	1	1	1	3	3
Actions								
A1: Impossible				X		X	X	X
A2: Increment day		X						
A3: Reset day	X		X		X			
A4: Increment month	X		X		X			
A5: reset month								
A6: Increment year								

Decision Table for NextDate (2nd Attempt)

- Let us consider the following equivalence classes:

M1= {month | month has 30 days}
M2= {month | month has 31 days}
M3= {month | month is December}
M4= {month | month is February}
D1= {day | $1 \leq \text{day} \leq 27$}
D2= {day | day = 28}
D3= {day | day = 29}
D4= {day | day = 30}
D5= {day | day=31}
Y1= {year | year is a leap year}
Y2= {year | year is a common year}

Decision Table for NextDate (1)

Conditions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
C1: month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2
C2: day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5
C3: year in	-	-	-	-	-	-	-	-	-	-
Actions										
A1: Impossible					X					
A2: Increment day	X	X	X			X	X	X	X	
A3: Reset day				X						X
A4: Increment month				X						X
A5: reset month										
A6: Increment year										

Decision Table for NextDate (2)

Conditions	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
C1: month in	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
C2: day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
C3: year in	-	-	-	-	-	-	Y1	Y2	Y1	Y2	-	-
Actions												
A1: Impossible										X	X	X
A2: Increment day	X	X	X	X		X	X					
A3: Reset day					X			X	X			
A4: Increment month								X	X			
A5: reset month					X							
A6: Increment year					X							

Guidelines and Observations

- Decision Table testing is most appropriate for programs where
  - there is a lot of decision making
  - there are important logical relationships among input variables
  - There are calculations involving subsets of input variables
  - There are cause and effect relationships between input and output
  - There is complex computation logic (high cyclomatic complexity)

- Decision tables do not scale up very well

- Decision tables can be iteratively refined

Components of a Decision Table

rules

	R1	R2	R3	R4	R5	R6	R7	R8
C1	T	T	T	T	F	F	F	F
C2	T	T	F	F	T	T	F	F
C3	T	F	T	F	T	F	T	F
a1	x			x	x			x
a2	x							x
a3		x				x		
a4			x	x			x	x
a5	x			x				

“binary” conditions

values of conditions

actions

actions taken

Read a Decision Table by columns of rules : (e.g. R1 from reqs. or design says when all conditions are T, then actions a1, a2, and a5 should occur)

Example (continued)

Stub	*Rule 1*	*Rule 2*	*Rules 3, 4*	*Rule 5*	*Rule 6*	*Rules 7, 8*
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	—	T	F	T
a1	X	X		X		—
a2	X				X	
a3		X				
a4			X	X		X

- The condition entries in rules 3 and 4, and rules 7 and 8 have the same actions. The "—" means ...
  - "Don't Care" (as in circuit analysis),
  - Irrelevant, or
  - not applicable, n/a

Example (continued)

Stub	*Rule 1*	*Rule 2*	*Rules 3, 4, 7, 8*	*Rule 5*	*Rule 6*
c1	T	T	—	F	F
c2	T	T	F	T	T
c3	T	F	—	T	F
a1	X	X		X	
a2	X				X
a3		X			
a4			X	X	
count	1	1	4	1	1

- Rule counting
  - A rule with no don’t care entries counts as 1
  - Each don’t care entry in a rule doubles the rule count
  - For a table with **n** limited entry conditions, the sum of the rule counts should be 2^n.

Problematic Decision Tables

- For LEDTs, simple rule counting helps identify decision tables that are ...
  - incomplete (rule count < 2^n) ,
  - redundant (rule count > 2^n) , or
  - inconsistent
    - (rule count > 2^n) AND
    - At least two rules have identical condition entries but different action entries.
- Redundancy and inconsistency are more likely with algebraically simplified tables that have been “maintained”.

A Redundant DT

Conditions	***1 – 4***	***5***	***6***	***7***	***8***	***9***
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	X
a2	—	X	X	X	—	—
a3	X	—	X	X	X	X

- Rule 9 is redundant with Rules 1 – 4 (technically, with what was rule 4)
- But the action entries are identical (No harm, no foul?)

Month Equivalence Classes

(to be used in the next example)

- M1 ={x : x is a 30-day month}
- M2 ={x : x is a 31-day month}
- M3 ={x : x is February}

Last Day of Month Decision Table

Conditions	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***
c1. M1?	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
c2. M2?	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
c3. M3?	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
c4. leap year?	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
a1. last day = 30							x	x								
a2. last day = 31											x	x				
a3. last day = 28														x		
a4. last day = 29													x			
a5. impossible	x	x	x	x	x	x			x	x					x	x

- Rule pairs 1 and 2, 3 and 4, 5 and 6, 9 and 10 don’t need c4, so they could be combined, BUT
- Impossible because c1, c2, and c3 are mutually exclusive.

Extended Entry Decision Tables

- When conditions are mutually exclusive, exactly one must be true.
- Extended entry decision tables typically (but not necessarily) have mutually exclusive conditions.
- The “extended” part is because a condition stub is an incomplete statement that is completed by the condition entry.
- (See the revised Last Day of Month EEDT)

Revised Last Day of Month DT

c1. 30-day month?	M1	M1	—	—	—	—
c2. 31-day month?	—	—	M2	M2	—	—
c3. February?	—	—	—	—	M3	M3
c4. leap year?	T	F	T	F	T	F
a1. last day = 30	x	x				
a2. last day = 31			x	x		
a3. last day = 28						x
a4. last day = 29					x	

- When conditions are mutually exclusive, exactly one must be true.
- This can be further simplified.

Triangle Program Decision Table

c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

Expanding c1...

c1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2: b<a+c?	—	F	T	T	T	T	T	T	T	T	T
c3: c<a+b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a1: Not a triangle	x	x	x								
a2: Scalene											x
a3: Isosceles						x			x	x	
a4: Equilateral				x							
a5: Impossible					x	x		x			

Triangle Problem Example ("short" form)

Pick input set, <a, b, c>, for each of the columns, or rules, below

Assume a, b and c are all between 1 and 200

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
1. a < b + c	F	T	T	T	T	T	T	T	T	T	T
2. b < a + c	-	F	T	T	T	T	T	T	T	T	T
3. c < a + b	-	-	F	T	T	T	T	T	T	T	T
4. a = b	-	-	-	T	T	T	T	F	F	F	F
5. a = c	-	-	-	T	T	F	F	T	T	F	F
6. b = c	-	-	-	T	F	T	F	T	F	T	F
1. Not triangle	X	X	X								
1. Scalene											X
2. Isosceles							X		X	X	
3. Equilateral				X							
4. "impossible"					X	X		X			

equivalent or all necessary

Note the Impossible cases

Req or Design should NOT have these cases

Concept of Rule Count

Rule Counting

c1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2: b<a+c?	—	F	T	T	T	T	T	T	T	T	T
c3: c<a+b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
Rule count	32	16	8	1	1	1	1	1	1	1	1
a1: Not a triangle	x	x	x								
a2: Scalene											x
a3: Isosceles						x			x	x	
a4: Equilateral				x							
a5: Impossible					x	x		x			

Corresponding Test Cases

Case ID	*a*	*b*	*c*	*Expected Output*
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

Advantages/Disadvantages of Decision Table

- Advantages: (check completeness & consistency)
  1. Allow us to start with a “complete” view, with no consideration of dependence
  2. Allow us to look at and consider “dependence,” “impossible,” and “not relevant” situations and eliminate some test cases.
  3. Allow us to detect potential *error in our Specifications*
- Disadvantages:
  1. Need to decide (or know) what conditions are relevant for testing - - - this may require Domain knowledge
     - e.g. need to know leap year for “next date” problem in the book
  2. Scaling up can be massive: *2^n rules for n conditions - - - that is if the conditions are binary and gets worse if the values are more than binary*

Decision table for triangle problem

- Conditions
  - A=B?
  - A=C?
  - B=C?
- Actions
  - Scalene
  - Isosceles
  - Equilateral

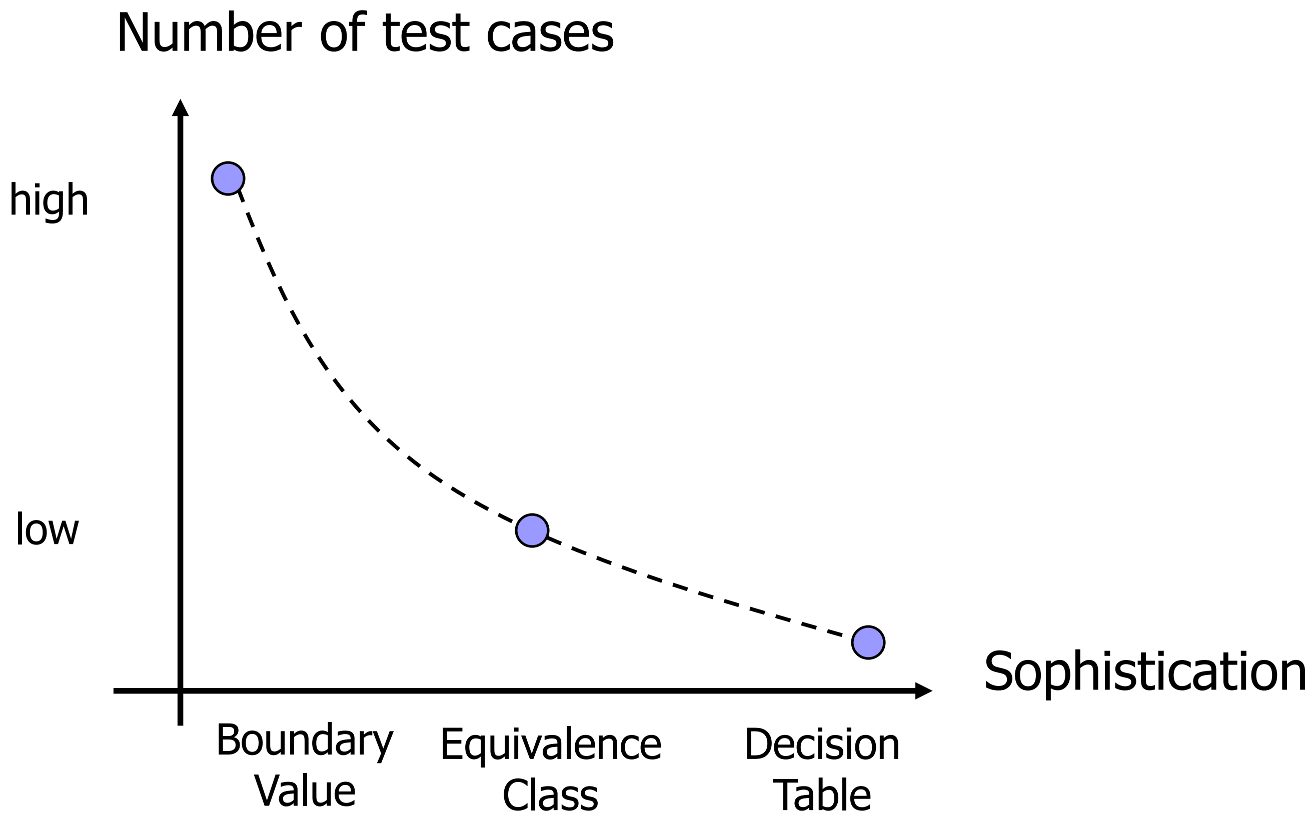
CASE STUDY- AIRLINE RESERVATION SYSTEM

- On-line flight reservation system (OFRS) is a software solution provided by "Always-We-Connect" travel agent for booking air tickets globally by Air India. The customer has to specify intended date of journey, from & to city names, max number of flight changes acceptable, class (business/economy), max waiting period acceptable and number of seats required. System shall not accept reservation less than 3 days from date of journey and more than 90 days gap from date of journey. Design adequate test cases to test this system. System will display whether reservations are available are not to customer based on conditions given by customer. If the seats are available in different class the same must be highlighted in the display

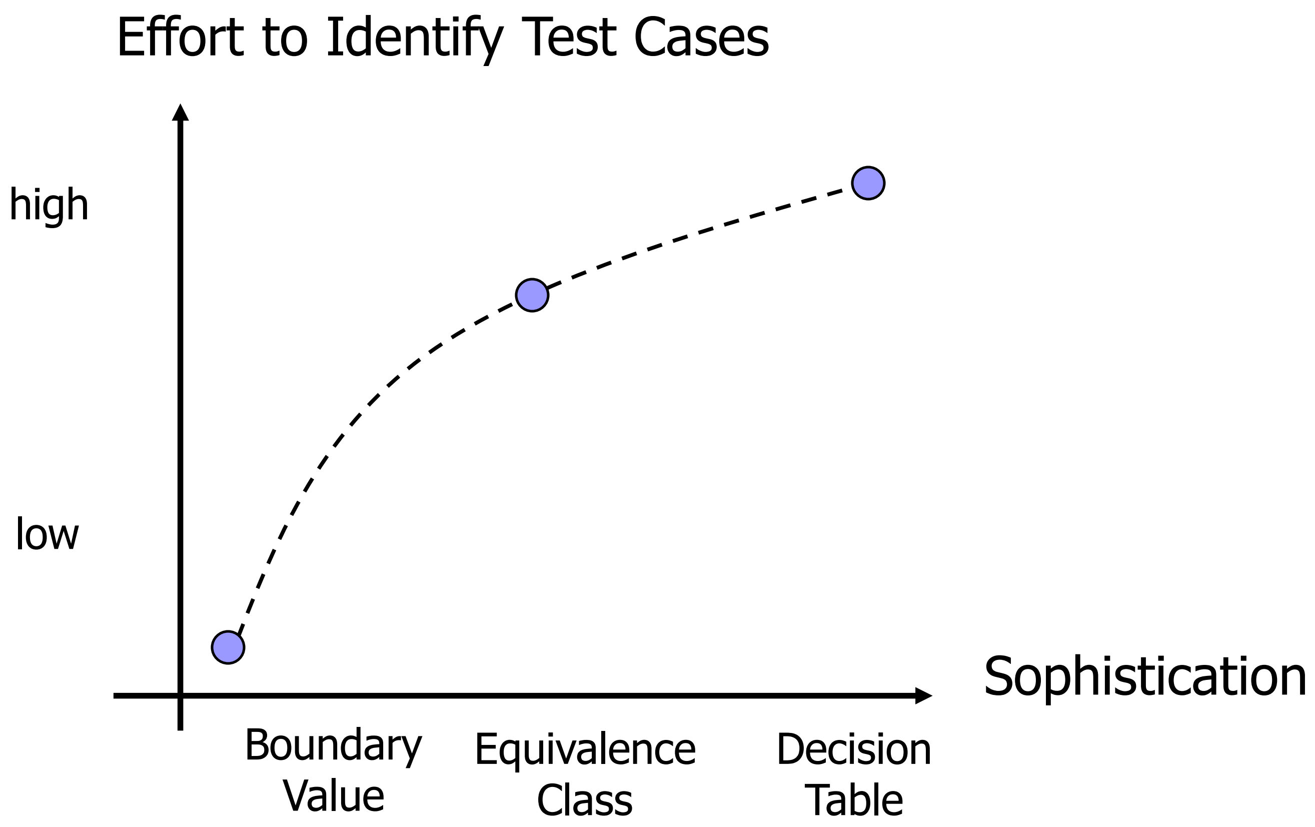
Example: Enquiry of tickets availability

- (Assumption: From and to city are entered correctly by user, the system will get all possible connections from database with relevant details)
  - Number of flight changes required must be less than max number of flight changes acceptable to customer
  - Total waiting period not to exceed max acceptable waiting period specified
  - Number of seats available must be greater or equal to number of seats requested
  - Reservations shall be allowed min 4-days in advance to a max of 90 days in advance from today
  - The class must be same as requested by customer if the seats are available in different class the same must be highlighted in the display

Summary


Number of test cases
high
low
Sophistication
Boundary Value
Equivalence Class
Decision Table

Summary-continued



Equivalence class partitioning

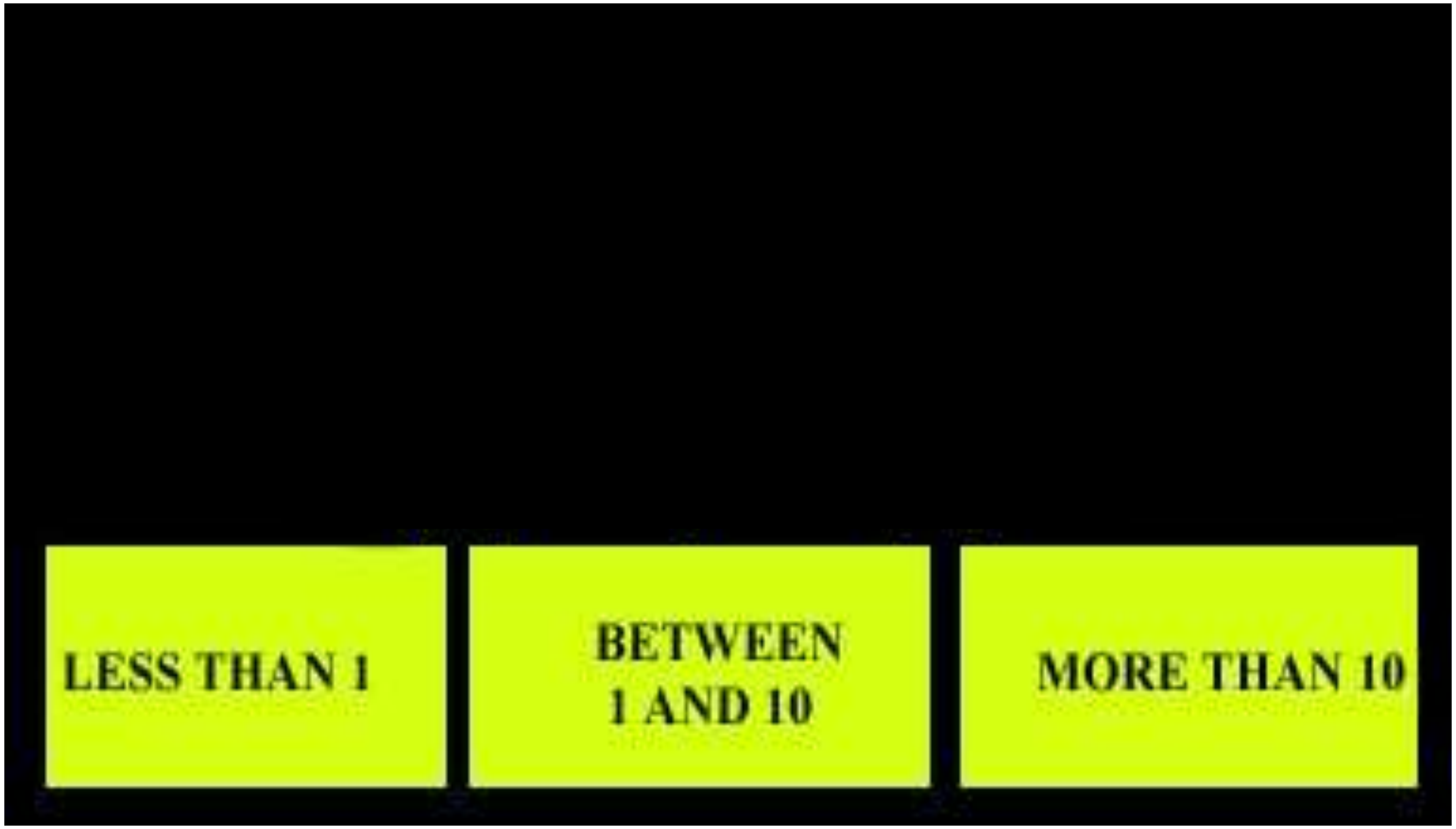
Classifying data-based black box testing

- All techniques
  - Requirements: data-based (some logical/mathematical input/output relation)
  - Purpose: unit testing, functional coding mistakes
  - Technique: black box, data-based
  - Assumption: both single and multiple-fault assumption
- Boundary value testing
  - Purpose: typical errors (boundaries of domains)
- Equivalence partitioning/decision tables
  - Purpose: input domain coverage, efficiency in # of tests

INTRODUCTORY EXAMPLE

- Let’s suppose that you wanted to test a program that only accepted integer values from 1 to 10.
- The possible test cases for such a program would be the range of all integers.
- How can we narrow the number of test cases down from all integers to a few good test cases?

ILLUSTRATION PROVIDED



GENERAL IDEA

- Equivalence partitioning is the process of taking all of the possible test values and placing them into classes (groups).
- Each element of a class should be “equivalent” in its likeliness to expose a bug.
- Then, we only need to use a few test cases per class to represent all test values.

GOOD TEST CASES

- A good test case has a reasonable probability of finding an error.
- In the previous program, all integers up to 0 and beyond 10 will elicit an error.
- There must be a way of reducing all possible test cases into a small subset.
- This small subset should have the highest probability of finding the most errors.

BEST TEST CASE SUBSET 1

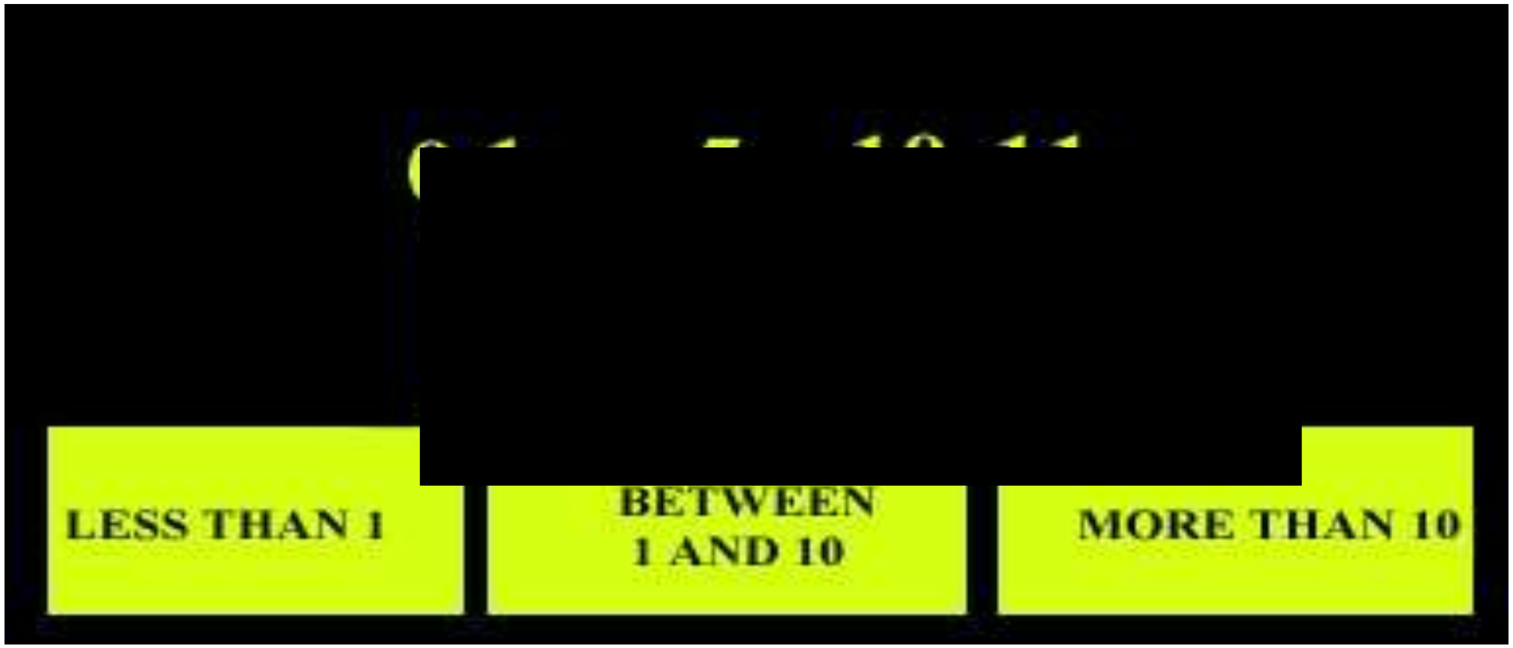
- A well-selected test case has two other properties:
- It reduces, by more than a count of one, the number of other test cases that must be developed to achieve “reasonable testing.”
- It does this by invoking as many different input conditions as possible.

BEST TEST CASE SUBSET 2

- The second property of a well-selected test case is that it covers a large set of other possible test cases.
- If such a test case detects an error, it is reasonable to assume that all other values in that subset of test cases will detect an error as well. If such a test case does not detect an error, assume that none will.

EXAMPLE PROVIDED

- It is reasonable to assume that if 11 fails, any value greater than 11 will fail.



SYNTHESIS OF IDEAS

- These two other properties of a well-selected test case form the black-box methodology of equivalence partitioning.
- First, develop a set of “interesting” conditions to be tested. In other words, identify the equivalence classes.
- Second, develop a minimal set of test cases that will cover those classes.

IDENTIFYING EQUIV. CLASSES

- Identify each input condition, it’s usually a sentence or phrase in the specification.
- Partition each condition into two or more groups.
- Valid equivalence classes represent valid inputs to the program.
- Invalid equivalence classes represent other states of the condition (erroneous input).

Few more tips

- Equivalence class for invalid inputs
  - Looks for Range in numbers
  - Look for membership in a group
  - Analyze responses to lists and menus
  - Looks for variables that must be equal
  - Create time-determined equivalence classes
  - Look for equivalent output events
  - Look for variable groups that must calculate to a certain value or range
  - Look for equivalent operating environments

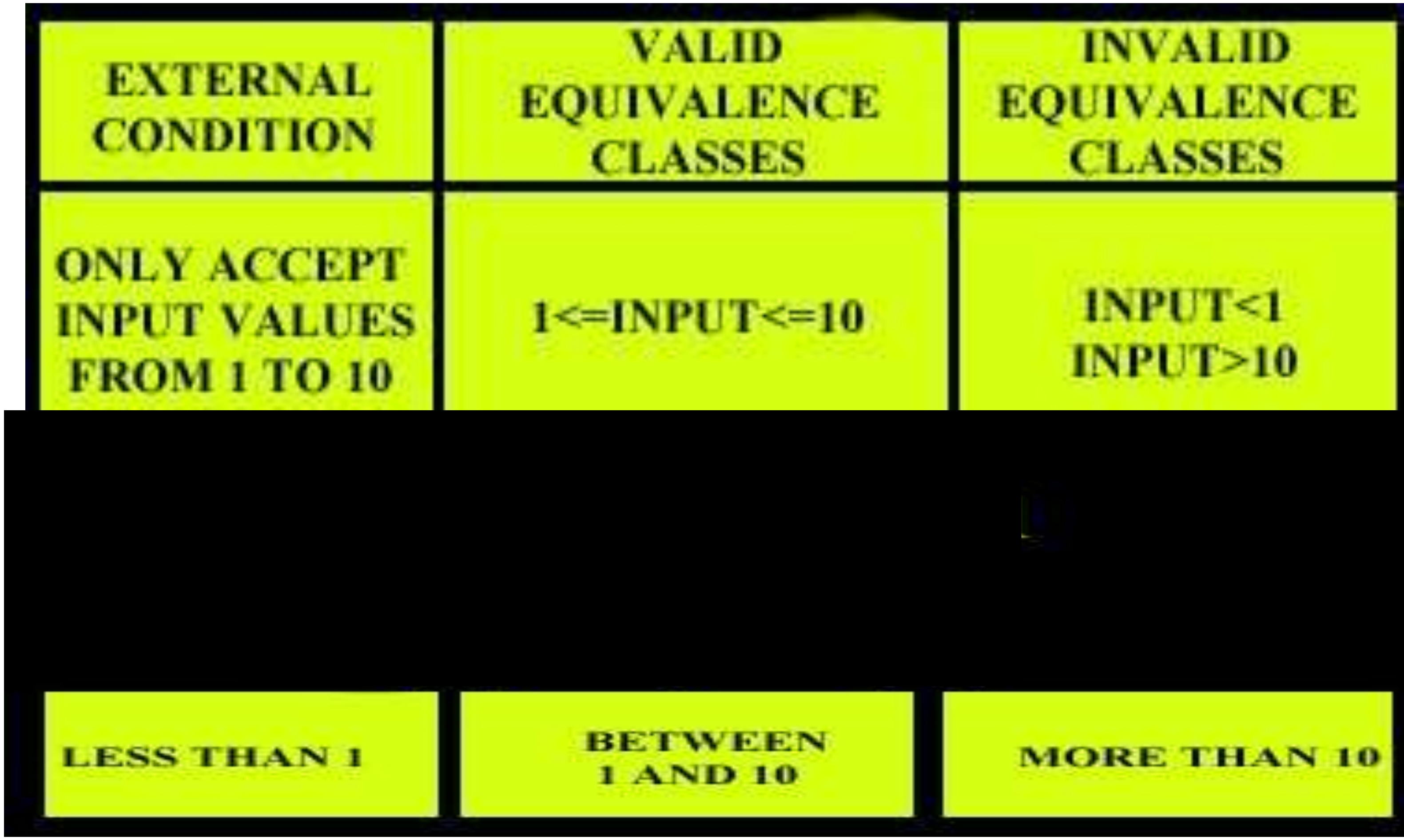
Equivalence Class

- It is one of the black box testing technique
- The input and output domain is partitioned into mutually exclusive parts i.e. disjoint sets (i.e. No overlap with input or output values does not overlap)
- The partition should be made in such a way that any one sample from a class is representative of the entire class
- Analyze before generation of test cases both output conditions and input conditions for the case considered

Equivalence Class Testing

- It is a technique of testing where input values are divided into valid and invalid inputs.
- Applied when
  - It is used in situations where exhaustive testing is desired
  - When there is a need to avoid redundancy
  -
  - Types of Equivalence Class Testing
    - Weak Normal Equivalence Class testing
    - Strong Normal Equivalence Class testing
    - Weak Robust Equivalence Class testing
    - Strong Robust Equivalence Class testing

EXAMPLE PROVIDED

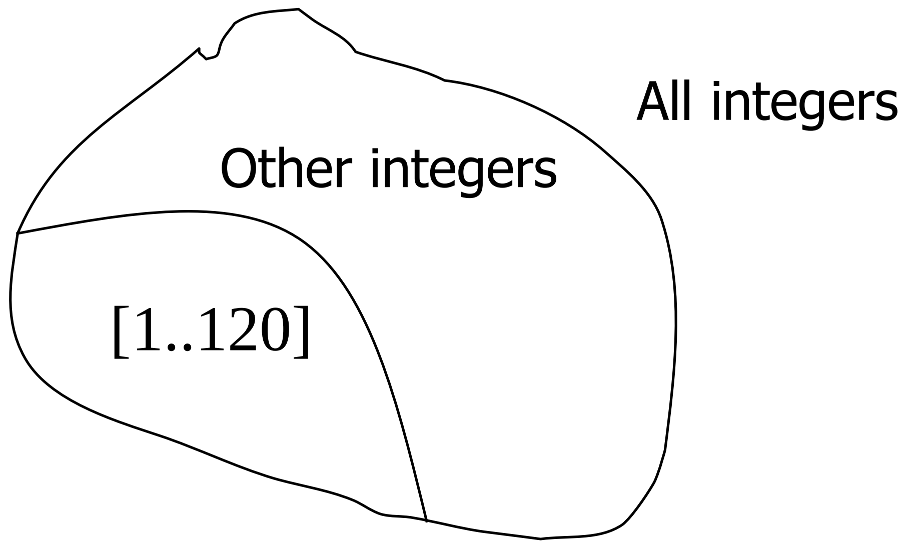


EXTERNAL CONDITION	VALID EQUIVALENCE CLASSES	INVALID EQUIVALENCE CLASSES
ONLY ACCEPT INPUT VALUES FROM 1 TO 10	1<=INPUT<=10	INPUT<1 INPUT>10

LESS THAN 1	BETWEEN 1 AND 10	MORE THAN 10

Sub domain classification

Consider an application A that takes an integer denoted by age as input. Let us suppose that the only legal values of age are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers.

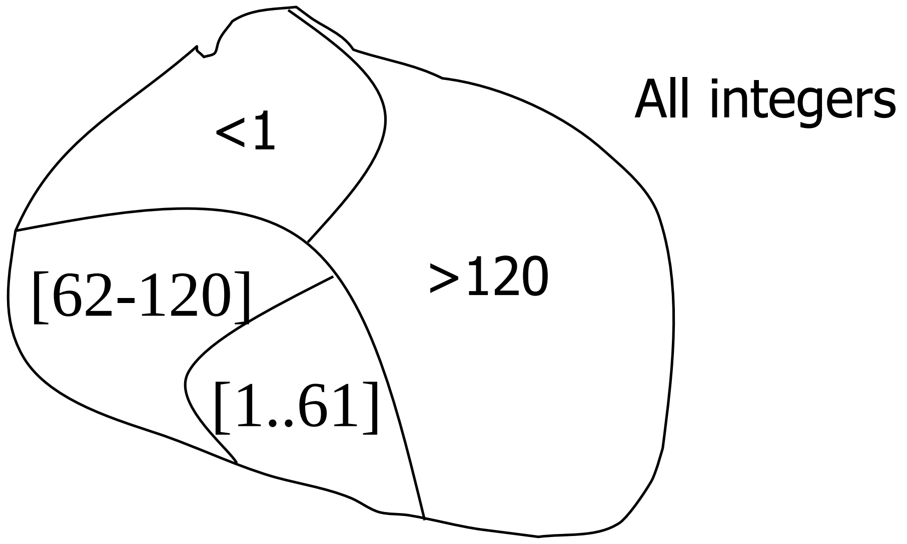


Sub domain classification(contd.)

Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2.
Thus E is further subdivided into two regions depending on the expected behavior.

Similarly, it is expected that all invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.

Sub domain classification(contd.)



GUIDELINES 1

- Identifying equivalence classes is mainly an intuitive process. However, guidelines exist to assist with such identifications.
- As in the previous example, if an input condition is a particular range of values, let one valid equivalence class be the range.
- Let the values below and above the range be two respective invalid equivalence classes.

GUIDELINES 2

- If an input condition specifies a number of values:
- Identify one valid equivalence class:
  - The range from 1 to the number of values.
- Identify two invalid equivalence classes:
  - Zero values
  - More than the number of values

EXAMPLE PROVIDED

- Requirement: “The names of 1 to 3 references must be entered on the form.”
- One valid equivalence class would be 1<=names<=3
- One invalid equivalence class would be zero names.
- Another invalid equivalence class would be names >3.

GUIDELINES 3

- For input conditions requiring the use of enumeration values, do the following:
- Identify a valid equivalence class for each enumeration value.
- Identify one invalid equivalence class representing values that are not defined in the enumeration type.

EXAMPLE PROVIDED

- Suppose an input condition states, “The printer color must be black, magenta, cyan, or yellow.”
- Valid equivalence classes will be:
  - Black
  - Magenta
  - Cyan
  - Yellow
- The invalid equivalence class represents all other values, i.e. Pink

GUIDELINES 4

- If the input condition specifies a mandatory (“must be”) condition:
- Identify one valid equivalence class representing the condition as being met.
- Identify one invalid equivalence class representing values that do not meet the condition.

EXAMPLE PROVIDED

- Suppose an input condition states “The first character of a code must be a digit.”
- The one valid equivalence class may contain the value 24432L.
- The one invalid equivalence class may contain a value such as G90125.

FINAL GUIDELINE

- After creating an equivalence class, ensure that the members will be handled in an identical manner by the program.
- If this does not happen to be the case, split the equivalence class into smaller equivalence classes.

IDENTIFYING TEST CASES

- After identifying the equivalence classes, identify the test cases.
- Assign a unique number to each equivalence class.
- Until all valid equivalence classes have been covered by test cases, write a new test case covering as many of the uncovered valid equivalence classes as possible.

IDENTIFYING TEST CASES

- Until all invalid equivalence classes have been covered by test cases, write a new test case that covers one, and only one, of the uncovered invalid equivalence classes as possible.
- The reason for this is that certain erroneous-input checks may supersede other erroneous-input checks.

ANALOGY NEEDED

- In programming languages, in an IF statement such as the following:
- if (var1=value1 AND var2=value2)
- If the var1=value1 condition evaluates to FALSE, the program will ignore, and therefore not test, the var2=value2 condition.

ANALOGY NEEDED

- In the same way, a test case that covers more than one invalid equivalence class may not evaluate the second condition if the first condition has a value of false.
- Likewise, a test case with two inputs may not check the second input for correctness if the first input is found to be invalid.

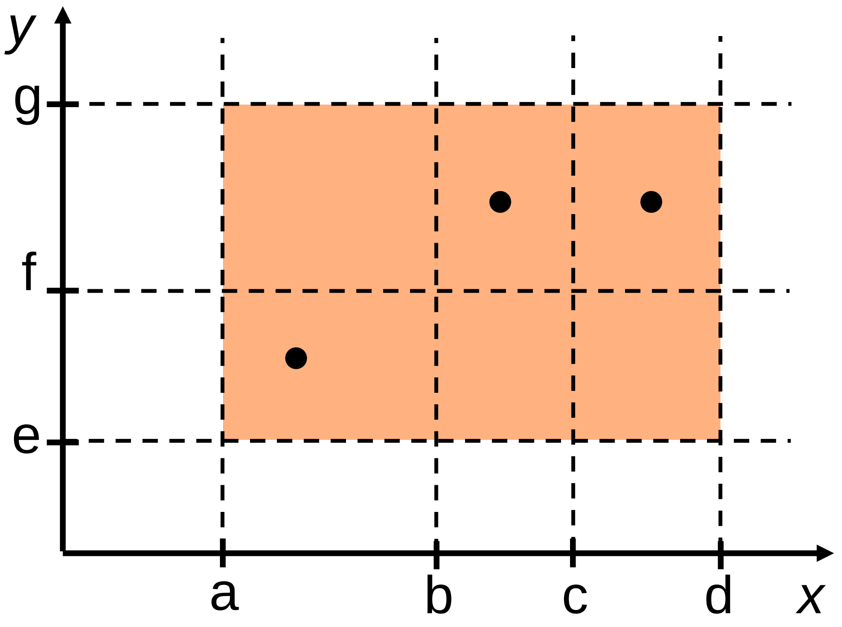
Equivalence partitioning

- detect errors to do with computational mistakes
- for integer input *x* with domain [*a*,*d*] partitioned in s_x subdomains, test input values from each subdomain
- assumes:
  - independent partitioning
  - redundancy in subdomain

weak normal variant:

- assumes:
  - independent variables
  - single-fault assumption
- #tests = $\mathbf{Max}_x\, s_x$

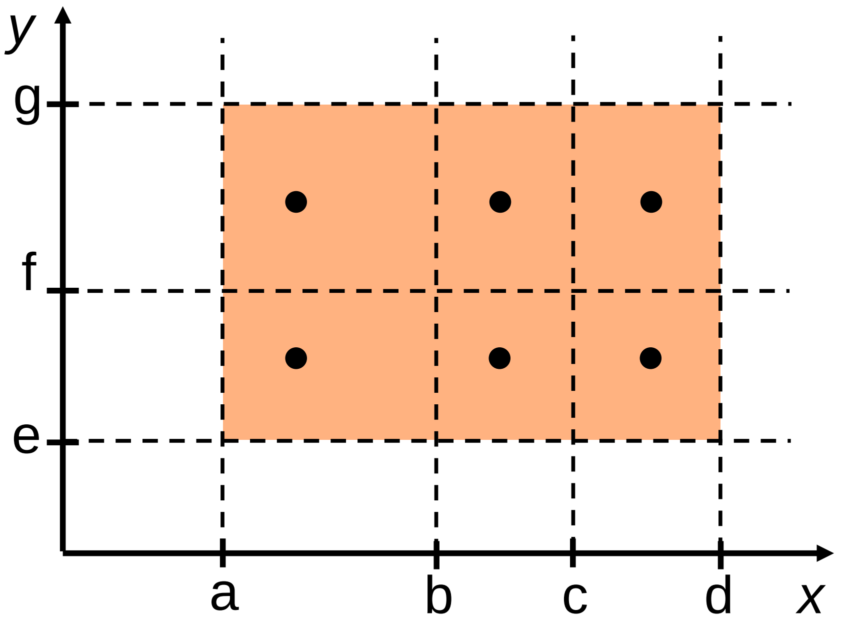
(maximal number of partions)



Equivalence partitioning

strong normal variant:

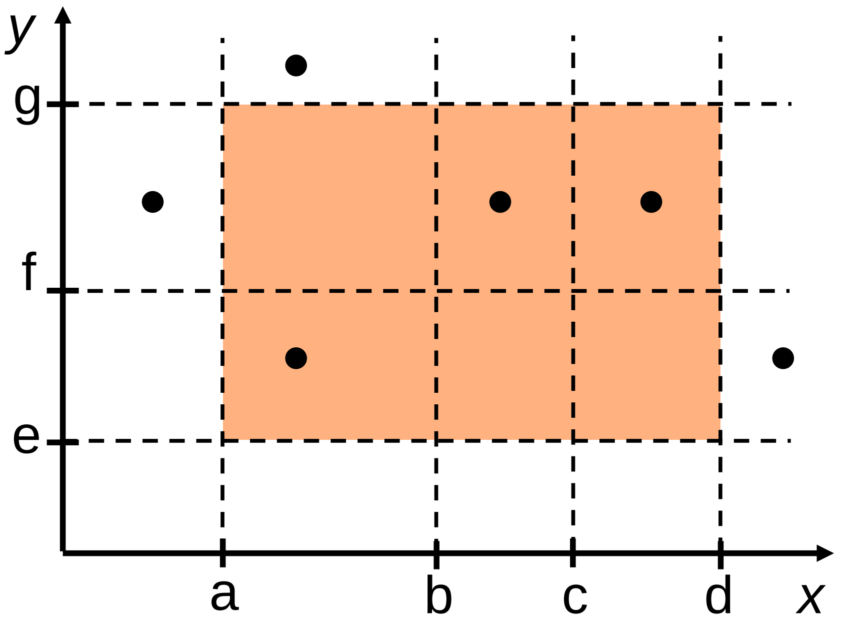
- assumes:
  - multiple-fault assumption
- #tests = $\Pi_x s_x$
  (product of number of partitions)



Equivalence partitioning

weak robust variant:

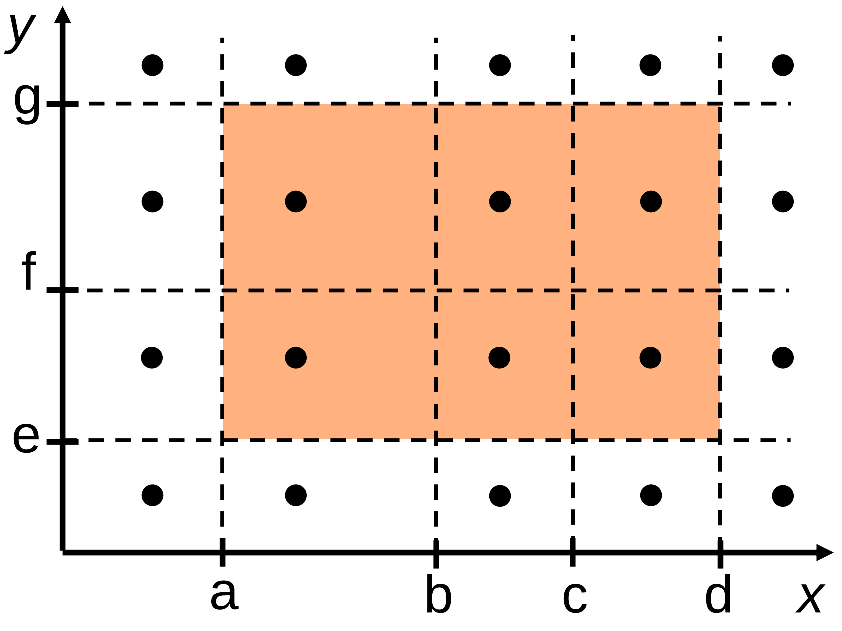
- tests outside domain
- assumes:
  - independent variables
  - single-fault assumption
- #tests = $\mathbf{Max}_{\boldsymbol{x}}\, s_x + \Sigma_x\, 2$
  (weak normal +2*(#inputs))



Equivalence partitioning

strong robust variant:

- tests outside domain
- assumes:
  - multiple-fault assumption
- #tests = $\Pi_x (s_x+2)$
- product of (X+2) *(y+2)

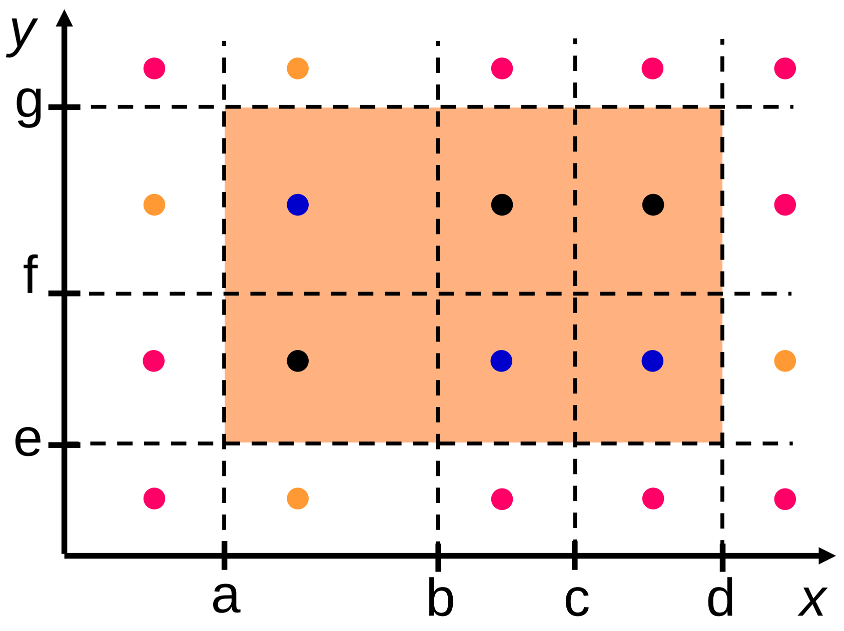


Subsume relations

Which subsume relations for equivalence partition variants?

(Assume same value selected for each input, for each subdomain)

$$\begin{tikzcd}
 & \text{weak normal} & \\
\text{strong normal} \arrow[ur] & & \text{weak robust} \arrow[ul] \\
 & \text{strong robust} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$



Equivalence partitioning summary (1-2)

- A group forms a EC if
  - They all test the same thing
  - If one test case catches a defect, the others probably will too
  - If one test case doesn’t catch a defect, the others probably won’t either
- What makes us consider them as equivalent
  - They involve the same input variable
  - They result in similar operations in the program
  - They affect the same output variable
  - None force the program to do error handling or all of them do

Equivalence partitioning summary (2-2)

Coverage: moderate
#tests: small to moderate
Usage: some study of requirements needed
When to use:

- independent inputs
- enumerable quantities
- when suspecting computational errors
- when redundancy can be assumed
- may easily be combined with boundary value

See literature:

- Patton (chapter 5, pages 67-69)
- Jorgensen (chapter 7)
- Zhu (section 4.4)

ADVANTAGES

Equivalence partitioning is vastly superior to a random selection of test cases. It uses the fewest test cases possible to cover the most input requirements.

DISADVANTAGES

- Although some guidelines are given to assist in the identification of equivalence classes, doing so is mostly an intuitive process.
- It overlooks certain types of high yield test cases that boundary-value analysis can determine.

Previous Date example

Previous Date problem

Steps

- Step 1 Identify both input equivalence class and output conditions by taking for input values both valid and invalid inputs
- Step 2 Generate Test cases using all equivalence classes

Solution: To first generate test cases based on input range

Then to generate test cases based on output range

Inputs:

- Valid input to get valid output
- Valid input but wrong output

Output conditions:

- Output 1 All valid outputs i.e. Previous Date
- Output 2 Invalid Date (eg 31 days in month of February)

Ie.

- Input1: 1 month 12
- Input 2: Month 1
- Input 3: Month 12
- Input 4: 1 Date 31
- Input 5: Date 1
- Input 6: Date 31
- Input 7: 1900 Year 2025
- Input 8: Year 2025
- Input 9: Year 1900

Test id	Month	Date	Year	Previous Date
001 (O/P conditions)	6	15	1962	14th June 1962
002	2 (3)	31 (1)	1962	Invalid Output
003 (I/P values)	6	15	1962	14th June 1962
004	0	15	1962	Invalid Input
005	13	15	1962	Invalid Input
006	6	15	1962	14th June 1962
007	6	-1	1962	Invalid Input
008	6	32	1962	Invalid Input
009	6	15	1962	14th June 1962
010	6	15	1989	Invalid Input
011	6	15	2026	Invalid Input

Triangle Problem

- Triangle problem

Sides lying between 1 Side 100

Output Conditions

- Output 1: Equilateral triangle (Say 50, 50, 50)
- Output 2: Isosceles triangle (Say 99, 50, 50)
- Output 3: Scalene triangle (Say 99, 50,100)
- Output 4: Not a triangle (50, 50,100)

Input Conditions

- Input 1: x 1
- Input 2: x 100
- Input 3: 1 x 100
- Input 4: y 1
- Input 6: y 100
- Input 7: 1 y 100
- Input 8: z 1
- Input 9: z 100
- Input 10: 1 z 100
- Input 10: x=y=z (Here we also look for existence of relationship between the variables (which was not found in previous date problem))
- Input 12: x=y , x z
- Input 13: x=z, x y
- Input 14: y=z, x
- Input 15: xy, yz
- Input 16: x=y + z,
- Input 17: y = x+ z
- Input 18: y x + z
- Input 19: z = x + y
- Input 20: z x + y

Test id	X	Y	Z	Expected Output
001	0	50	50	Input Invalid
002	101	50	50	Input Invalid
003	50	50	50	Equilateral
004	50	0	50	Invalid Input
005	50	101	50	Invalid Input
006	50	50	50	Equilateral
007	50	50	0	Input Invalid
008	50	50	101	Input Invalid
009	50	50	50	Equilateral
010	60	60	60	Equilateral
011	50	50	60	Isosceles
012	50	60	50	Isosceles
013	60	50	50	Isosceles
014	100	99	50	Scalene
015	100	50	50	Not a triangle (x= y+ z)
016	100	50	25	Not a triangle (x > y+ z)
017	50	100	50	Not a triangle Y = x + z
018	50	100	25	Not a triangle Y > x + z
019	50	50	100	Not a triangle (z = x+ Y)
020	25	50	100	Not a triangle (z > x + Y)

- Questions?