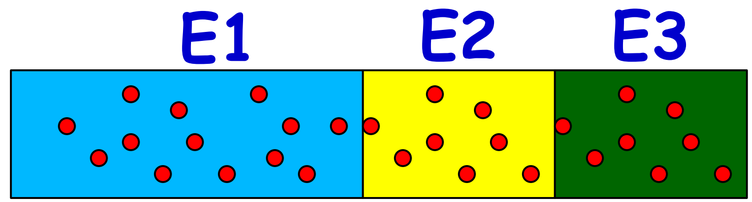
Equivalence Class and Boundary Value Testing

Equivalence Class Partitioning

- The input values to a program:
  - Partitioned into equivalence classes.
- Partitioning is done such that:
  - **Program behaves in similar ways to every input value belonging to an equivalence class.**
  - **At the least there should be as many equivalence classes as scenarios.**

Why Define Equivalence Classes?



- Premise:
  - Testing code with any one representative value from a equivalence class:
  - As good as testing using any other values from the equivalence class.

Equivalence Class Partitioning

- How do you identify equivalence classes?
  - Identify scenarios
  - Examine the input data.
  - Examine output
- Few guidelines for determining the equivalence classes can be given…

Guidelines to Identify Equivalence Classes

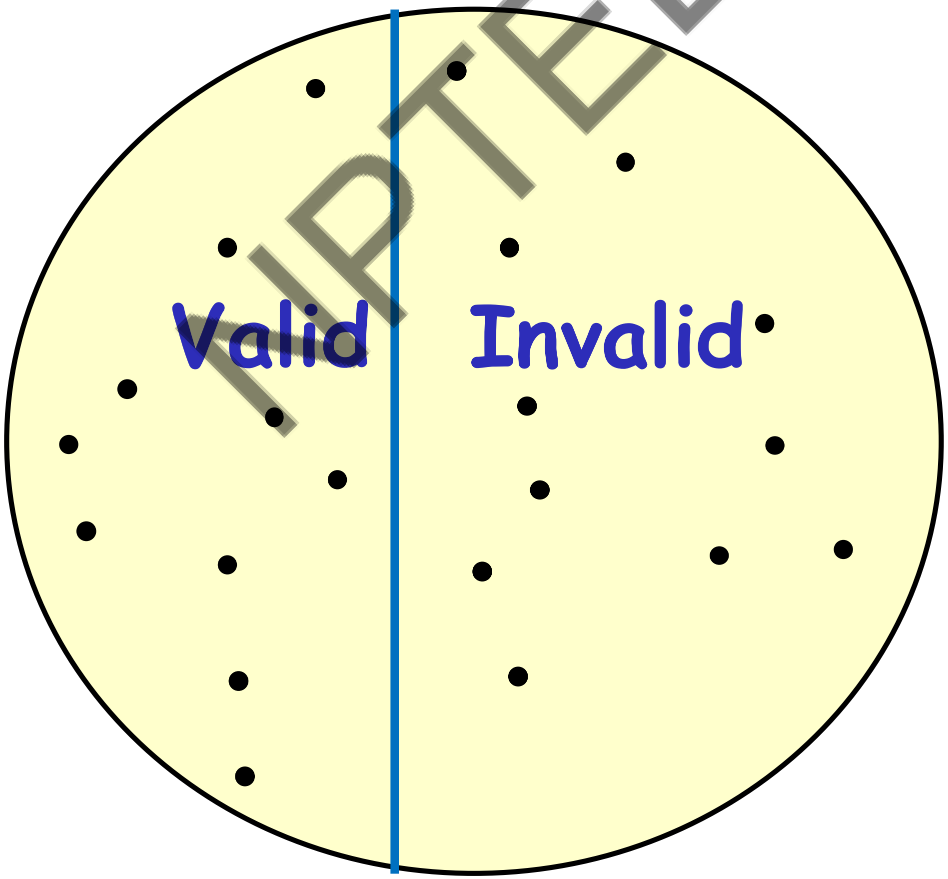
- If an input condition specifies a range, one valid and two invalid equivalence class are defined.
- If an input condition specifies a member of a set, one valid and one invalid equivalence classes are defined.
- If an input condition is Boolean, one valid and one invalid classes are defined.
- Example:
- Area code: range --- value defined between 10000 and 90000
- Password: value - six character string.

Equivalent class partition: Example

- Given three sides, determine the type of the triangle:
  - Isosceles
  - Scalene
  - Equilateral, etc.
- Hint: scenarios expressed in output in this case.

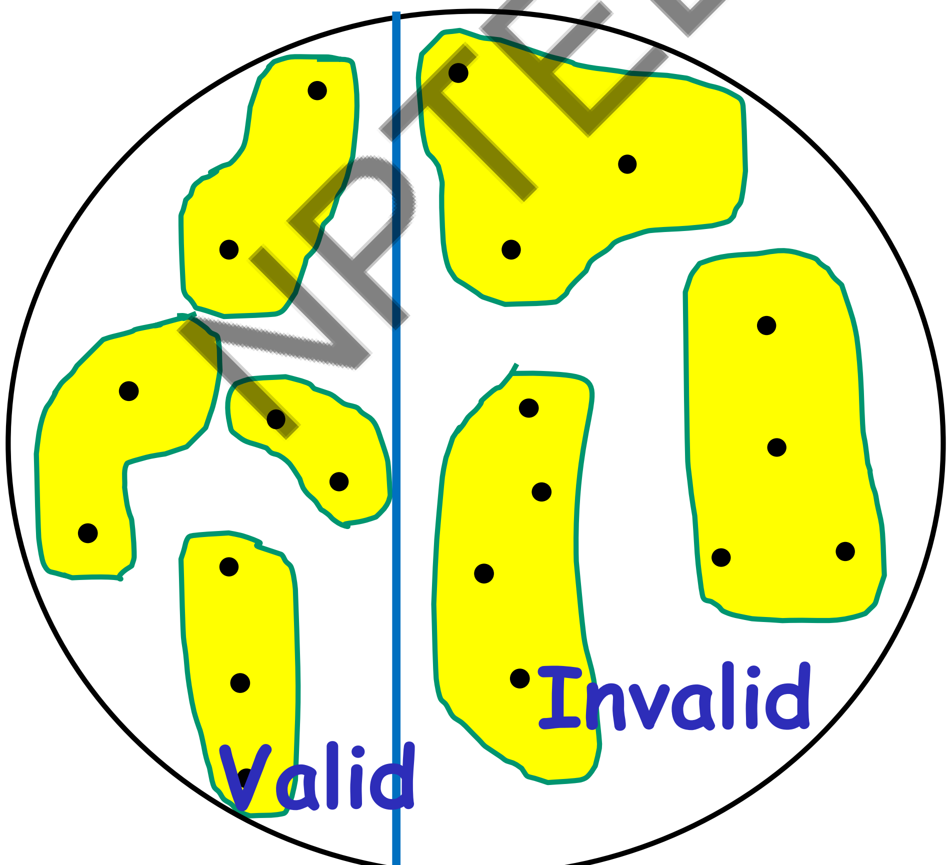
Equivalence Partitioning

- First-level partitioning:
  - Valid vs. Invalid test cases



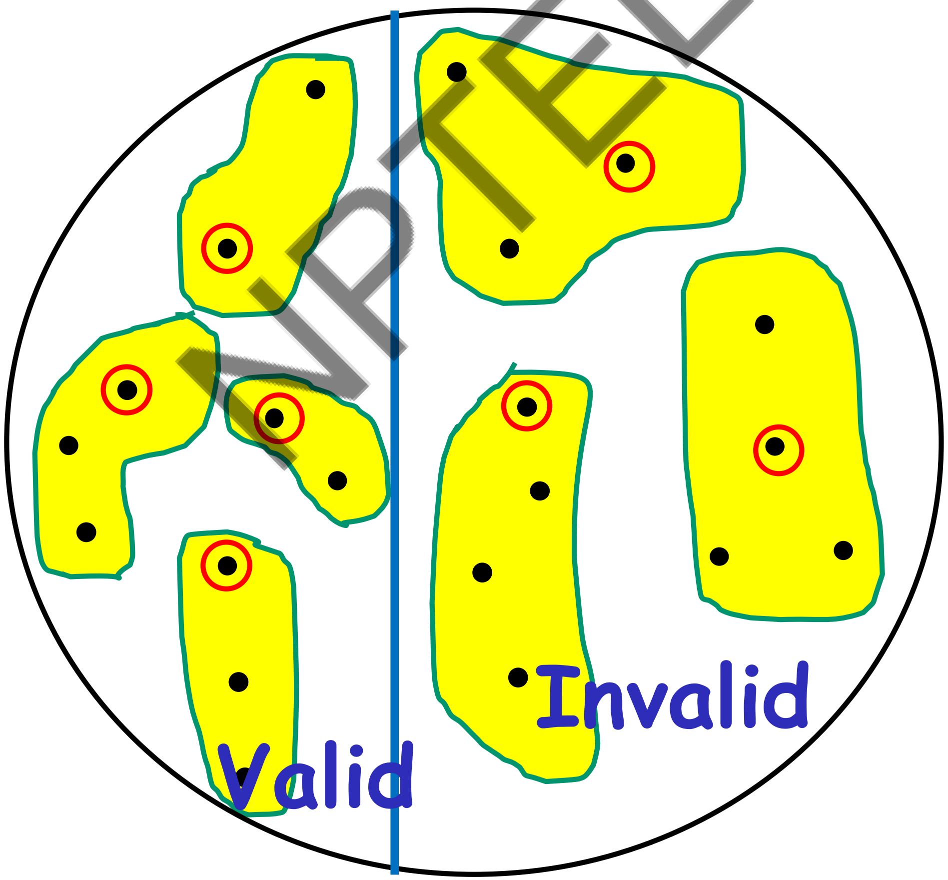
Equivalence Partitioning

- Further partition valid and invalid test cases into equivalence classes



Equivalence Partitioning

- Create a test case for at least one value from each equivalence class



Equivalence Class Partitioning

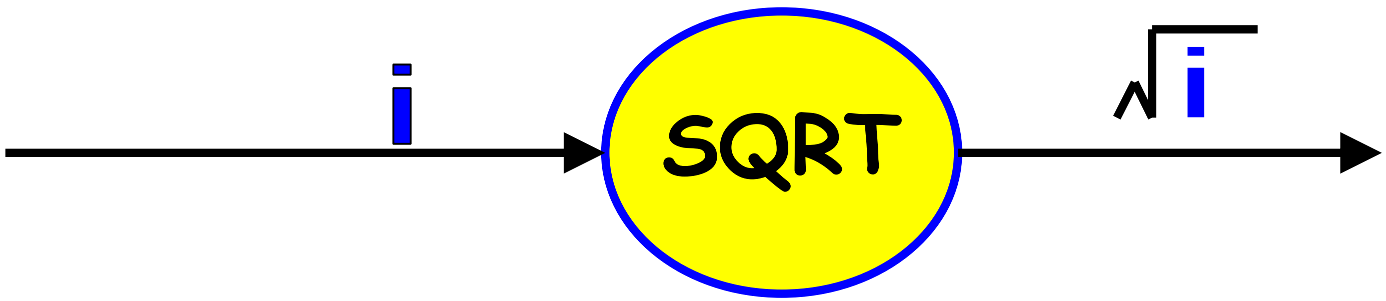
- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - One valid and two invalid equivalence classes are defined.

Equivalence Class Partitioning

- If input is an enumerated set of values, e.g. :
  - {a,b,c}
- Define:
  - One equivalence class for valid input values.
  - Another equivalence class for invalid input values should be defined.

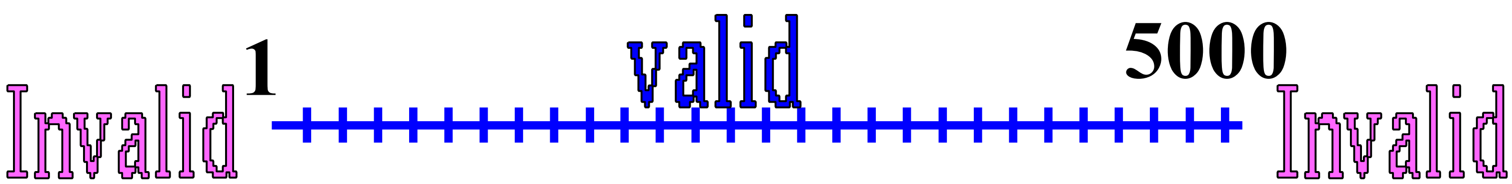
Example

- A program reads an input value in the range of 1 and 5000:
  - Computes the square root of the input number



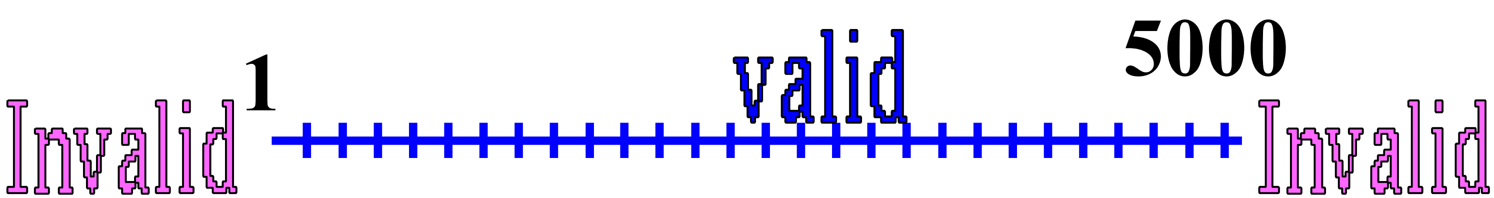
Example (cont.)

- Three equivalence classes:
  - The set of negative integers,
  - Set of integers in the range of 1 and 5000,
  - Integers larger than 5000.



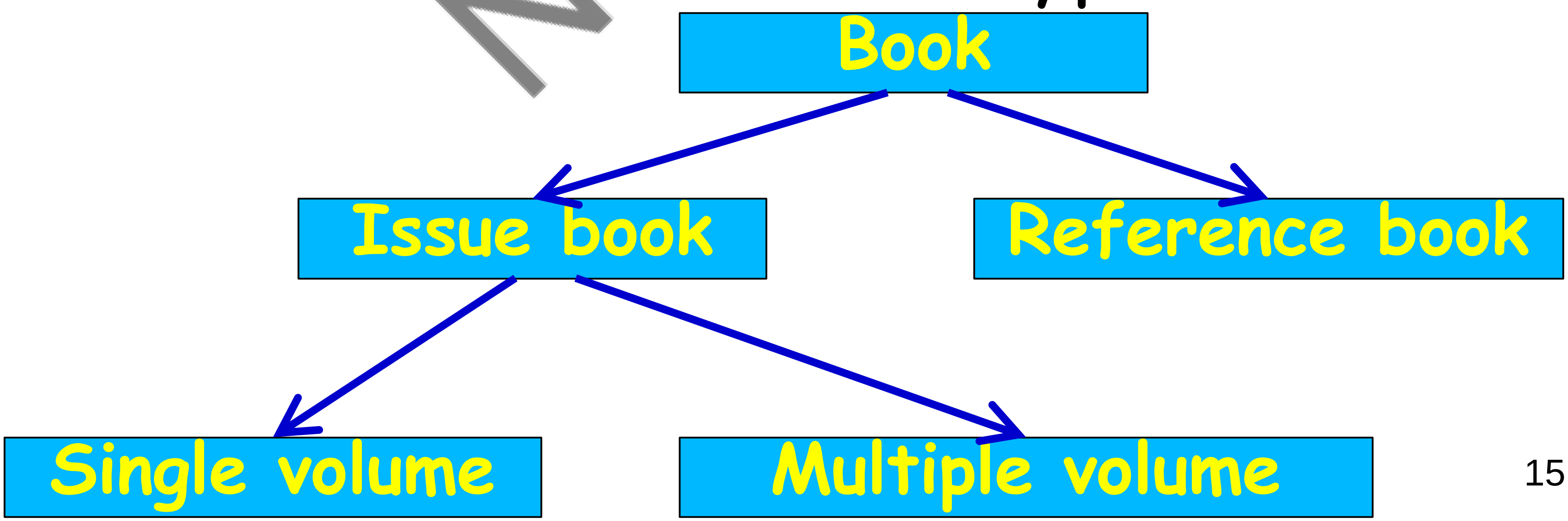
Example (cont.)

- The test suite must include:
  - Representatives from each of the three equivalence classes:
  - A possible test suite can be: {-5,500,6000}.



Equivalence Partitioning

- A set of input values constitute an equivalence class if the tester believes these are processed identically:
  - Example : **`issue book(book id);`**
  - Different set or sequence of instructions may be executed based on book type.



Equivalence Partitioning: Example 1

- Example: **Image Fetch-image(URL)**
  - **Equivalence Definition 1:** Partition based on URL protocol ("http", "https", "ftp", "file", etc.)
  - **Equivalence Definition 2:** Partition based on type of file being retrieved (HTML, GIF, JPEG, Plain Text, etc.)

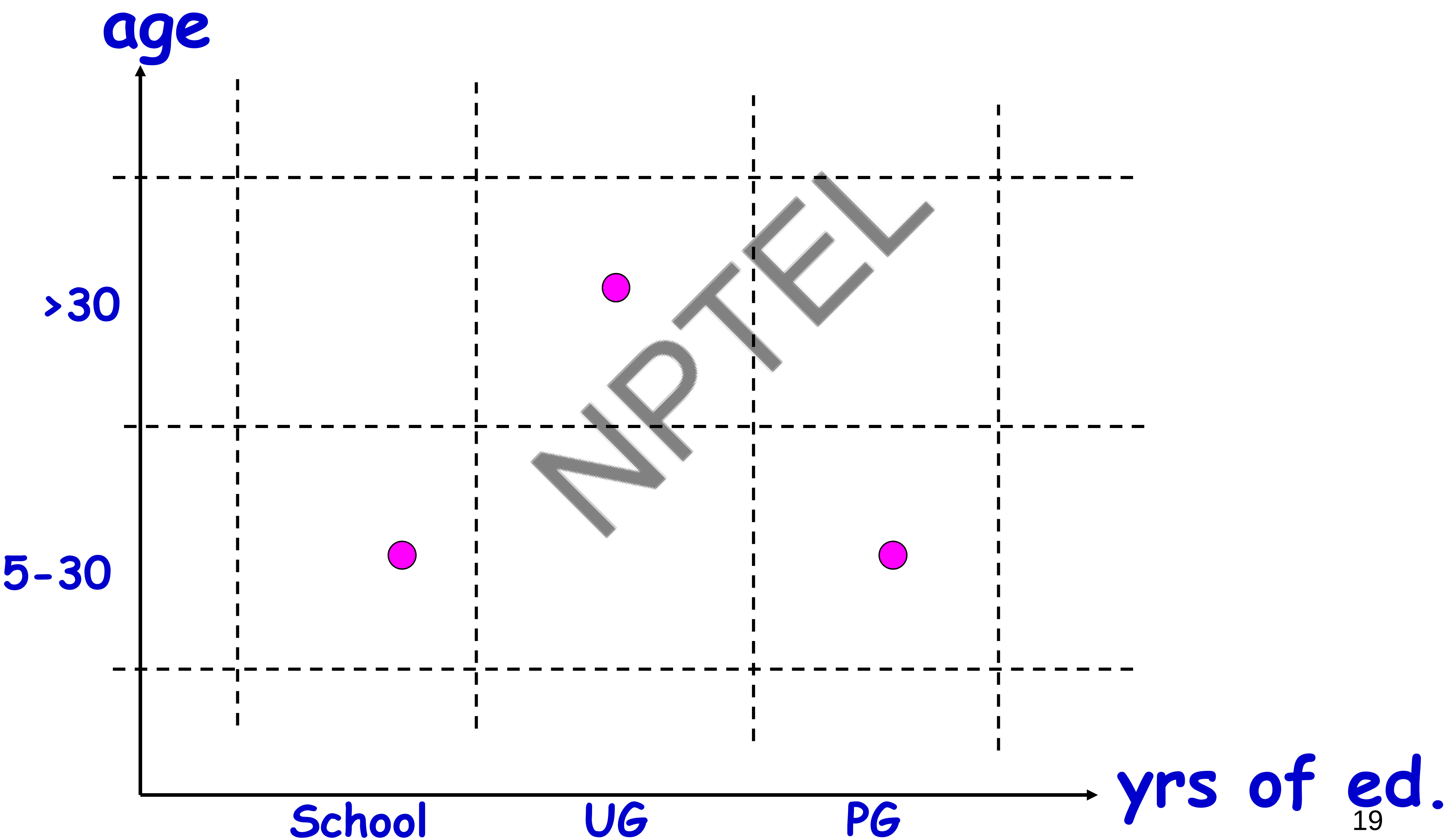
Equivalence Partitioning: Example 2

Input	Valid Equivalence Classes	Invalid Equivalence Classes
An integer N such that: -99 <= N <= 99	?	?
Phone Number Area code: [11,…, 999] Suffix: Any 6 digits	?	?

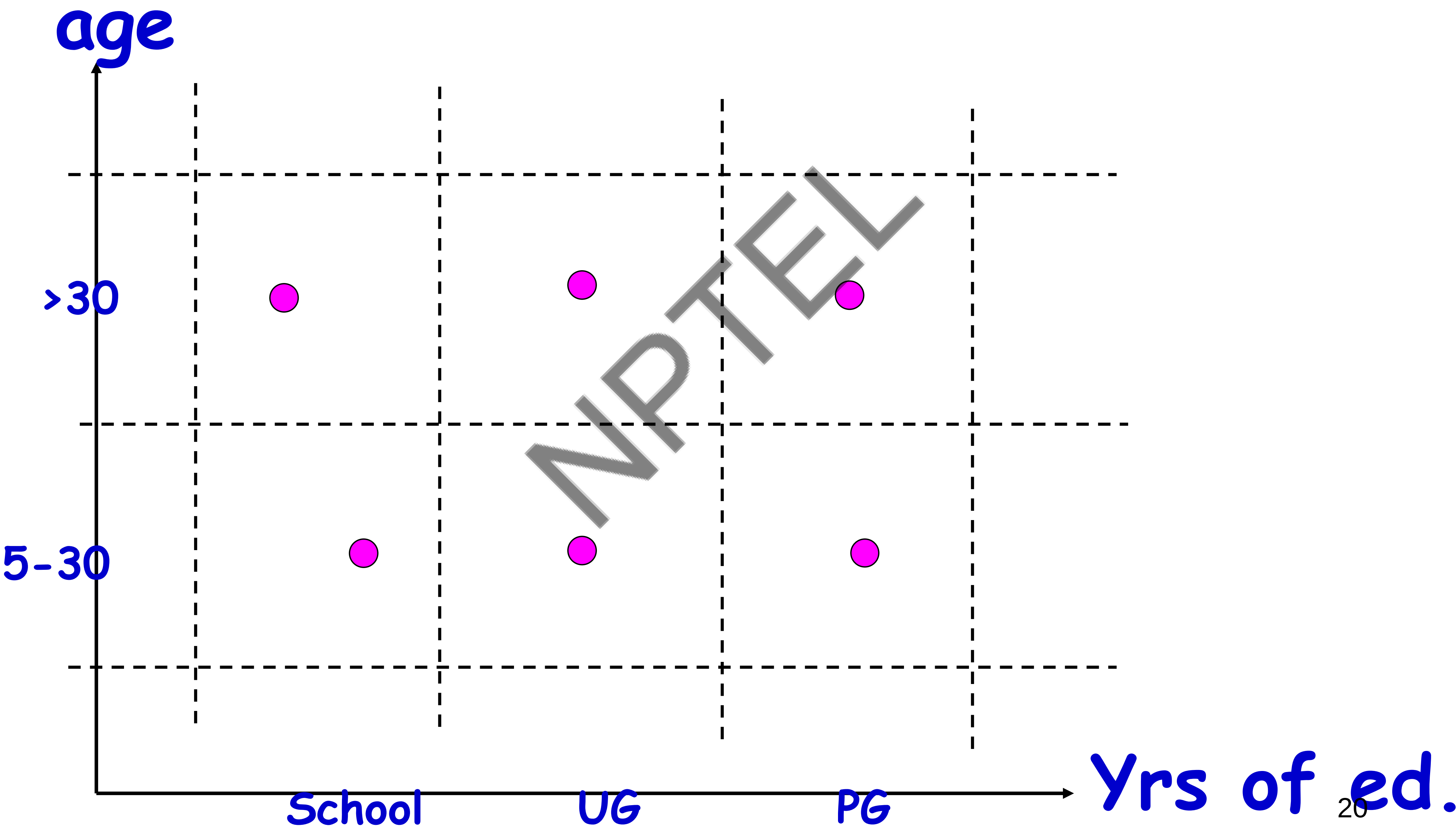
Equivalence Partitioning: Example 2

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Prefix: [11, 999] Suffix: Any 6 digits	[11,999][000000,999999]	Invalid format 5555555, (555)(555)555566, etc. Area code < 11 or > 999 Area code with non-numeric characters *Similar for Prefix and Suffix*

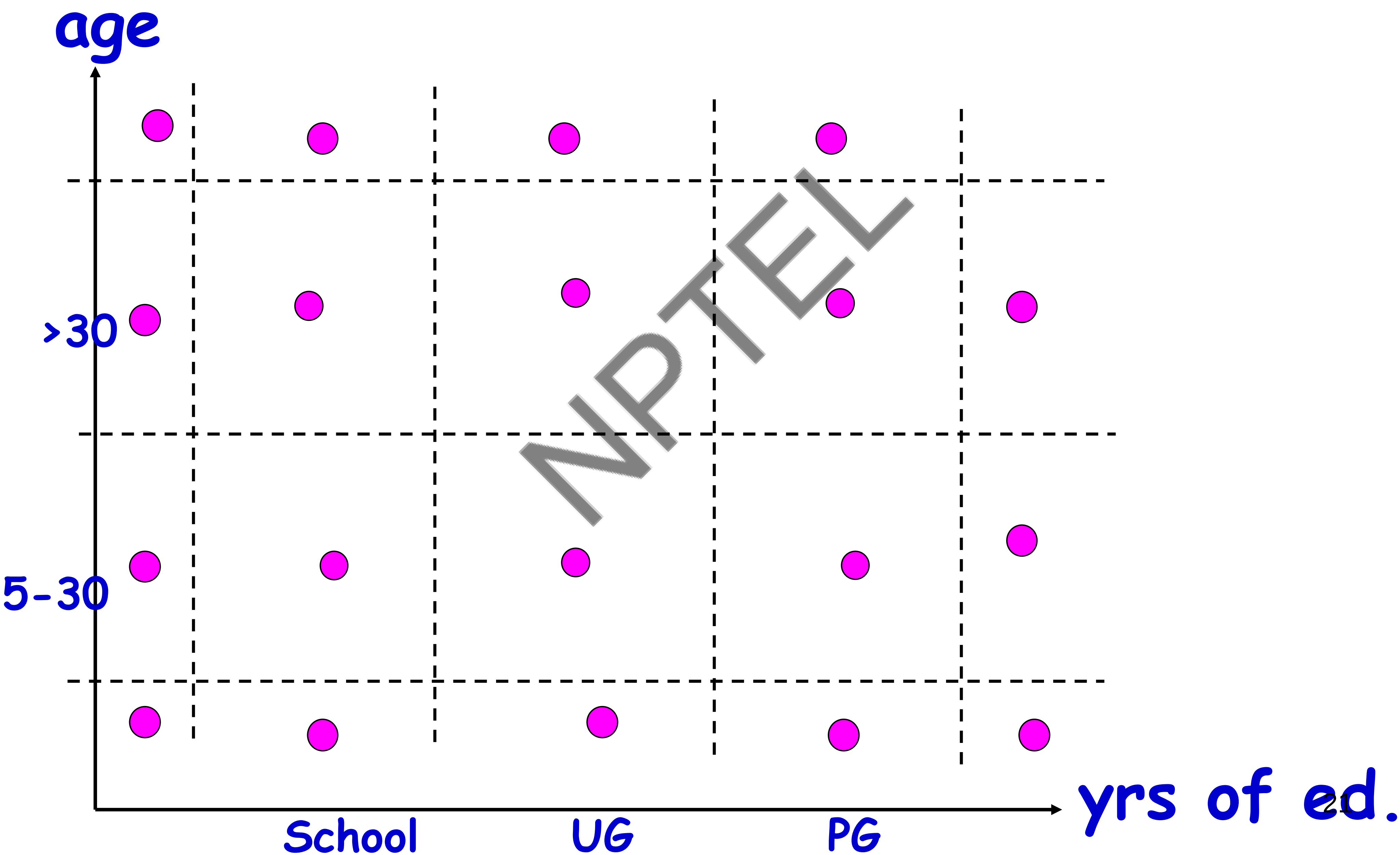
Weak Equivalence Class Testing



Strong Equivalence Class Testing



Strong Robust Equivalence Class Testing



Quiz 1

- Design Equivalence class test cases:
- A bank pays different rates of interest on a deposit depending on the deposit period.
  - 3% for deposit up to 15 days
  - 4% for deposit over 15days and up to 180 days
  - 6% for deposit over 180 days upto 1 year
  - 7% for deposit over 1 year but less than 3 years
  - 8% for deposit 3 years and above

Quiz 2

- Design Equivalence class test cases:
- For deposits of less than Rs. 1 Lakh, rate of interest:
  - 6% for deposit upto 1 year
  - 7% for deposit over 1 year but less than 3 years
  - 8% for deposit 3 years and above
- For deposits of more than Rs. 1 Lakh, rate of interest:
  - 7% for deposit upto 1 year
  - 8% for deposit over 1 year but less than 3 years
  - 9% for deposit 3 years and above

Quiz 3

- Design equivalence class test cases.
  - Consider a program that takes 2 strings of maximum length 20 and 5 characters
  - Checks if the second is a substring of the first
  - **substr(s1,s2);**

Special Value Testing

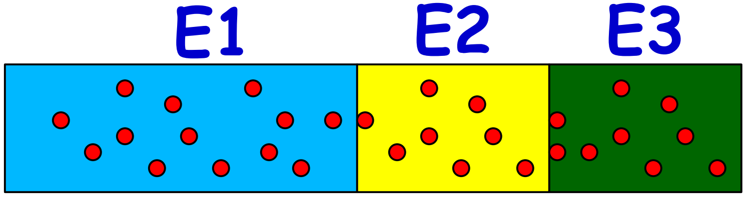
25

Special Value Testing

- What are special values?
  - The tester has reasons to believe these values would execute statements having the risk of containing bugs:
  - **General risk: Example-- Boundary value testing**
  - **Special risk: Example-- Leap year not considered**

Boundary Value Analysis

- Some typical programming errors occur:
  - At boundaries of equivalence classes
  - **Might be purely due to psychological factors.**



- Programmers often fail to see:
  - Special processing required at the boundaries of equivalence classes.

Boundary Value Analysis

- Programmers may mistakenly use < instead of <=
- Boundary value analysis:
  - Select test cases at the boundaries of different equivalence classes.

Boundary Value Analysis: Guidelines

- If an input condition specifies a range, bounded by values a and b:
  - **Test cases should be designed with value a and b, and just above a just below b.**
- **Example:** Integer D with input range [-3, 10],
  - test values: -3, 10, 9, -2, 0
- If an input condition specifies a number values:
  - **Test cases should exercise minimum and maximum numbers.**
  - **Values just above minimum and below maximum are also to be tested.**
- **Example:** Enumerate data E with input condition: {3, 5, 100, 102}
  - test values: 3, 102, -1, 200, 5

Boundary Value Testing: HR Application Example

- Process employment applications based on a person's age.

0-12	Do not hire
12-18	May hire as intern
18-65	May hire full time
65-100	Do not hire

- Notice the problem at the boundaries.
  - Age "12" is included in two different equivalence classes (as are 18 and 65).

Boundary Value Testing Example (cont)

- If (applicantAge >= 0 && applicantAge **<=12**)
  hireStatus="NO";
- If (applicantAge **>= 12** && applicantAge <=18)
  hireStatus="Intern";
- If (applicantAge >= 18 && applicantAge <=55)
  hireStatus="FULL";
- If (applicantAge >= 65 && applicantAge <=99)
  hireStatus="NO";

Boundary Value Testing Example (cont)

- Corrected boundaries:

0–11 Don't hire

12–17 Can hire as intern

18–64 Can hire as full-time employees

65–99 Don't hire

- What about ages -3 and 101?

- The requirements do not specify how these values should be treated.

Boundary Value Testing Example (cont)

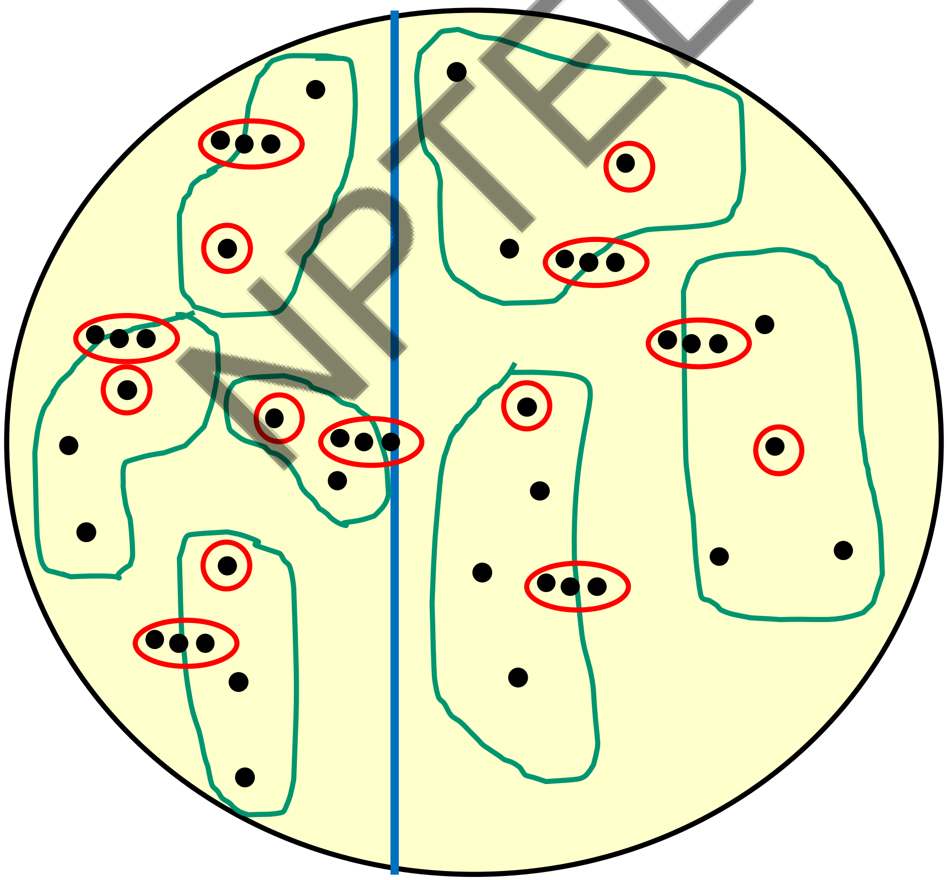
- The code to implement the corrected rules is:

```
If (applicantAge >= 0 && applicantAge <=11)
   hireStatus="NO";

If (applicantAge >= 12 && applicantAge <=17)
   hireStatus="Intern";

If (applicantAge >= 18 && applicantAge <=64)
   hireStatus="FULL";

If (applicantAge >= 65 && applicantAge <=99)
   hireStatus="NO";
```

- Special values on or near the boundaries in this example are {-1, 0, 1}, {11, 12, 13}, {17, 18, 19}, {64, 65, 66}, and {98, 99, 100}.

Boundary Value Analysis

- Create test cases to test boundaries of equivalence classes



Example 1

- For a function that computes the square root of an integer in the range of 1 and 5000:
  - Test cases must include the values: {0,1,2,4999,5000,5001}.

Example 2

- Consider a program that reads the “age” of employees and computes the average age.

ages → $\boxed{\text{Program}}$ → average age

Assume valid age is 1 to 100

- How would you test this?
  - How many test cases would you generate?
  - What would be test data?

Boundaries of the inputs

1 <= age <= 100

1 ⟵ **age** ⟶ 100

The "basic" boundary value testing would include 5 situations:

1. - at minimum boundary
2. - immediately above minimum
3. - between minimum and maximum (nominal)
4. - immediately below maximum
5. - at maximum boundary

Test Cases for this Example

- How many test cases for this example ?
  - answer : 5
- Test input values? :

1	at the minimum
2	at one above minimum
45	at middle
99	at one below maximum
100	at maximum

Independent Data

- Suppose there are 2 "distinct" inputs that are assumed to be independent of each other.
  - Input field 1: **years of education** (say 1 to 23)
  - Input field 2: **age** (1 to 100)
- If they are independent of each other, then we can start with 5 + 5 = 10 sets.

input data: yrs of ed

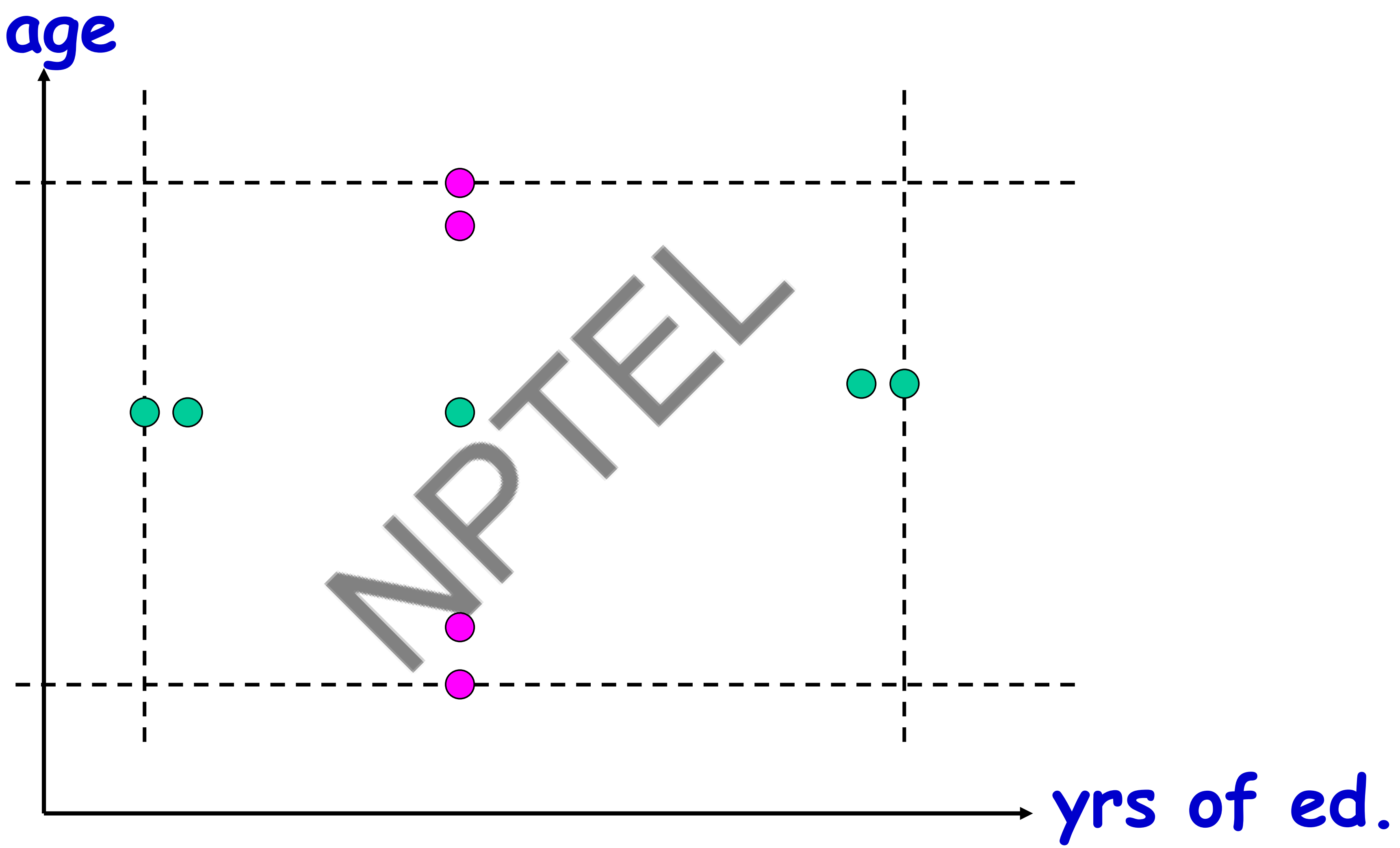
1. n= 1 ; age = whatever(37)
2. n =2; age = whatever
3. n = 12; age = whatever
4. n = 22; age = whatever
5. n = 23; age = whatever

←→

input data: age

1. n= 12; age = 1
2. n =12; age = 2
3. n = 12; age = 37
4. n = 12; age = 99
5. n = 12; age = 100

2 – Independent inputs



- Note that there needs to be only 9 test cases for 2 independent inputs.
- In general, need (4z + 1) test cases for z independent inputs.

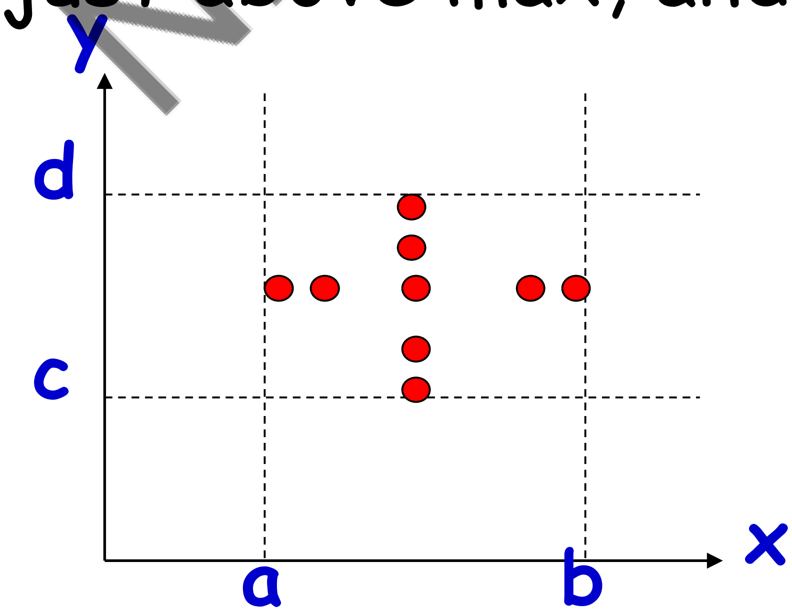
Boundary Value Test

Given F(x,y) with constraints $a \le x \le b$

$$c \le y \le d$$

Boundary Value analysis focuses on the boundary of the input space to identify test cases.

Use input variable value at min, just above min, a nominal value, just above max, and at max.



Single Fault Assumption

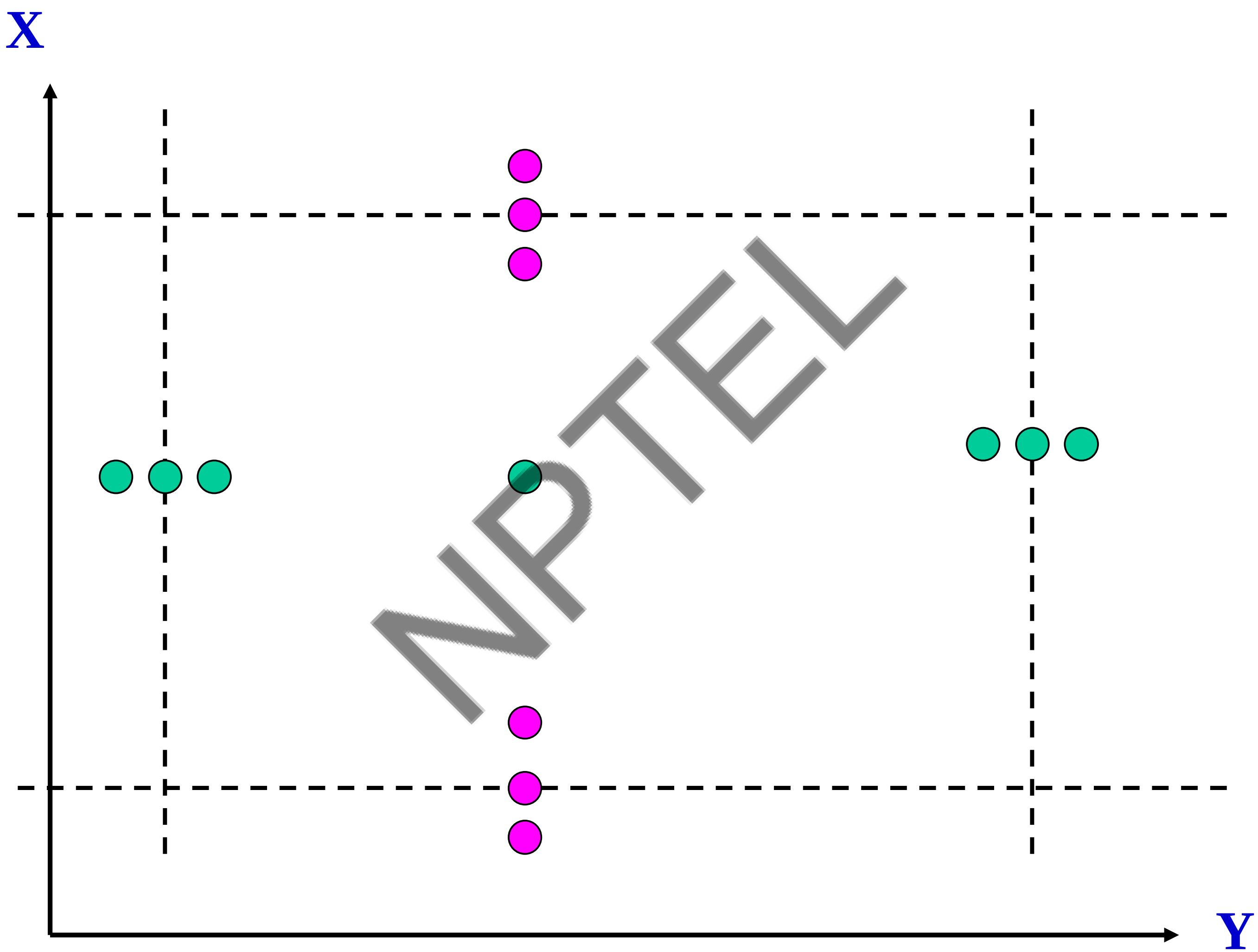
- **Premise:** "Failures rarely occur as the result of the simultaneous occurrence of two (or more) faults"
- Under this:
  - **Hold the values of all but one variable at their nominal values, and let that one variable assume its extreme values.**

Robustness testing

- This is just an extension of the Boundary Values to include:
  - Less than minimum
  - Greater than maximum
- There are 7 cases or values to worry about for each independent variable input
- The testing of robustness is really a test of "error" handling.

1. *Did we anticipate the error situations?*
2. *Did we issue informative error messages?*
3. *Did we allow some kind of recovery from the error?*

2 - independent inputs for robustness test



- Note that there needs to be only 13 test cases for 2 independent variables or inputs.
- In general, there will be *(6n+ 1)* test cases for n <u>independent inputs</u>.

Some Limitations of Boundary Value Testing

- How to handle boolean variables?
  - True
  - False

  (these may be radio buttons)

- What about non-numerical variable where the values may be text?

Quiz: BB Test Design

- Design black box test suite for a function that solves a quadratic equation of the form $ax^2+bx+c=0$.
- Equivalence classes
  - Invalid Equation
  - Valid Equation: Roots?

Complex

Real

Coincident

Unique

Combinatorial Testing

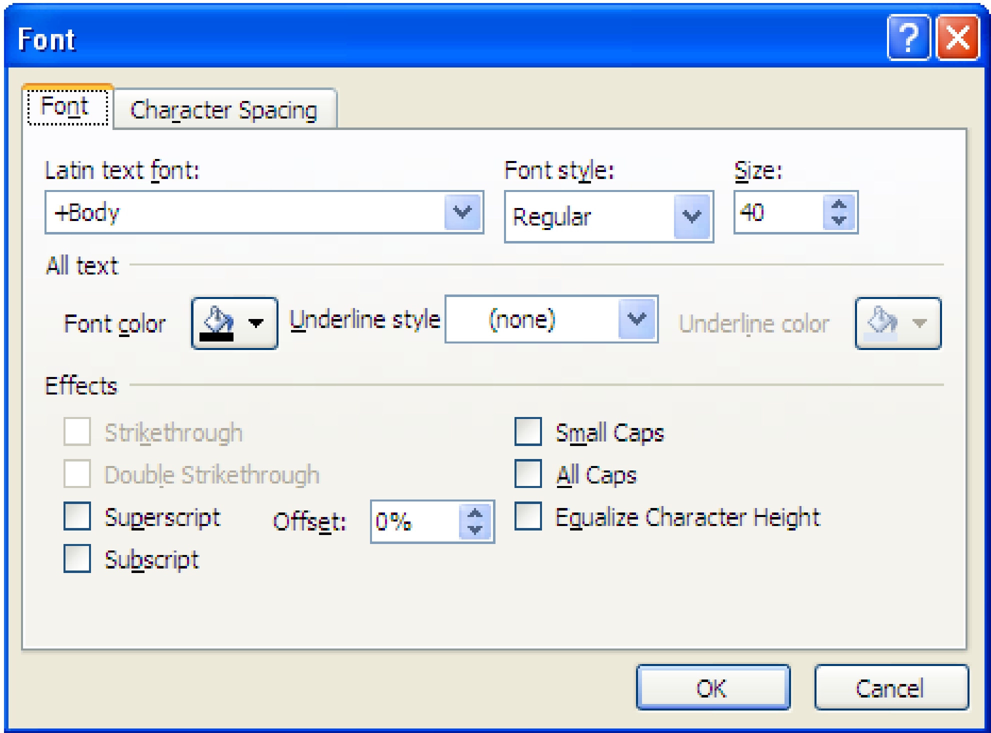
47

Combinatorial Testing: Motivation

- The behavior of a program may be affected by many factors:
  - Input parameters,
  - Environment configurations,
  - State variables. ..
- Equivalence partitioning of an input variable:
  - Identify the possible types of input values requiring different processing.
- If the factors are more than 2 or 3:
  - It is impractical to test all possible combinations of values of all factors.

Combinatorial Testing: Motivation

- Many times, the specific action to be performed depends on the value of a set of Boolean variable:
  - Controller applications
  - User interfaces



Combinatorial Testing

- Several types of combinatorial testing strategies:
  - Decision table-based testing
  - Cause-effect graphing
  - Pair-wise testing

Decision table-based Testing (DTT)

- Applicable to requirements involving conditional actions.
- Can be automatically translated into code
  - Conditions = inputs
  - Actions = outputs
  - Rules =test cases

	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

- Assume the independence of inputs
- Example
  - If c1 AND c2 OR c3 then A1

Combinations

	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

Conditi ons

Actions

Sample Decision table

- A decision table consists of a number of columns (rules) that comprise all test situations
- Action a_i will take place if c1 and c2 are true
- Example: the triangle problem
  - C1: a, b,c form a triangle
  - C2: a=b
  - C3: a= c
  - C4: b= c
  - A1: Not a triangle
  - A2:scalene
  - A3: Isosceles
  - A4:equilateral
  - A5: impossible

	r1	r2	…				rn
C1	0	1					0
c2	-	1					0
C3	-	1					1
C4	-	1					0
a1	1	0					0
a2	0	1					1
a3	0	0					0
a4	0	1					1
a5	0	0					

Test cases from Decision Tables

Test Case ID	a	b	c	Expected output
TC1	4	1	2	Not a Triangle
TC2	2888	2888	2888	Equilateral
TC3	?	|	)	Impossible
TC4				
…				
TC11				

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 – Example 2

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	Check 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

Quiz: Develop BB Test Cases

- Policy for charging customers for certain in-flight services:

If the flight is more than half-full and ticket cost is more than Rs. 3000, free meals are served unless it is a domestic flight. The meals are charged on all domestic flights.

Fill all combinations in the table.

		POSSIBLE COMBINATIONS							
CONDITONS	*more than half-full*	N	N	N	N	Y	Y	Y	Y
	more than Rs.3000 per seat	N	N	Y	Y	N	N	Y	Y
	domestic flight	N	Y	N	Y	N	Y	N	Y
ACTIONS									

Analyze column by column to determine which actions are appropriate for each combination

		POSSIBLE COMBINATIONS							
CONDITONS	*more than half-full*	N	N	N	N	Y	Y	Y	Y
	more than Rs. 3000 per seat	N	N	Y	Y	N	N	Y	Y
	domestic flight	N	Y	N	Y	N	Y	N	Y
ACTIONS	*serve meals*					X	X	X	X
	free							X	

Reduce the table by eliminating redundant columns.

		POSSIBLE COMBINATIONS							
CONDITONS	more than half-full	N	N	N	N	Y	Y	Y	Y
	more than Rs. 3000 per seat	N	N	Y	Y	N	N	Y	Y
	domestic flight	N	Y	N	Y	N	Y	N	Y
ACTIONS	serve meals					X	X	X	X
	free							X	

Final solution

		Combinations			
CONDITONS	more than half-full	N	Y	Y	Y
	more than 3000 per seat	-	N	Y	Y
	domestic flight	-	-	N	Y
ACTIONS	serve meals		X	X	X
	free			X	

Assumptions regarding rules

- Rules need to be complete:
  - That is, every combination of decision table values including default combinations are present.
- Rules need to be consistent:
  - That is, there is no two different actions for the same combinations of conditions

Guidelines and Observations

- Decision Table testing is most appropriate for programs for which:
  - **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**
- **Decision tables do not scale up very well**

Quiz: Design test Cases

- Customers on a e-commerce site get following discount:
  - A member gets 10% discount for purchases lower than Rs. 2000, else 15% discount
  - Purchase using SBI card fetches 5% discount
  - If the purchase amount after all discounts exceeds Rs. 2000/- then shipping is free.

Cause-effect Graphs

- Overview:
  - Explores combinations of possible inputs
  - Specific combinations of inputs (causes) and outputs (effects)
  - **Avoids combinatorial explosion**
  - Represented as nodes of a cause effect graph
  - The graph also includes constraints and a number of intermediate nodes linking causes and effects

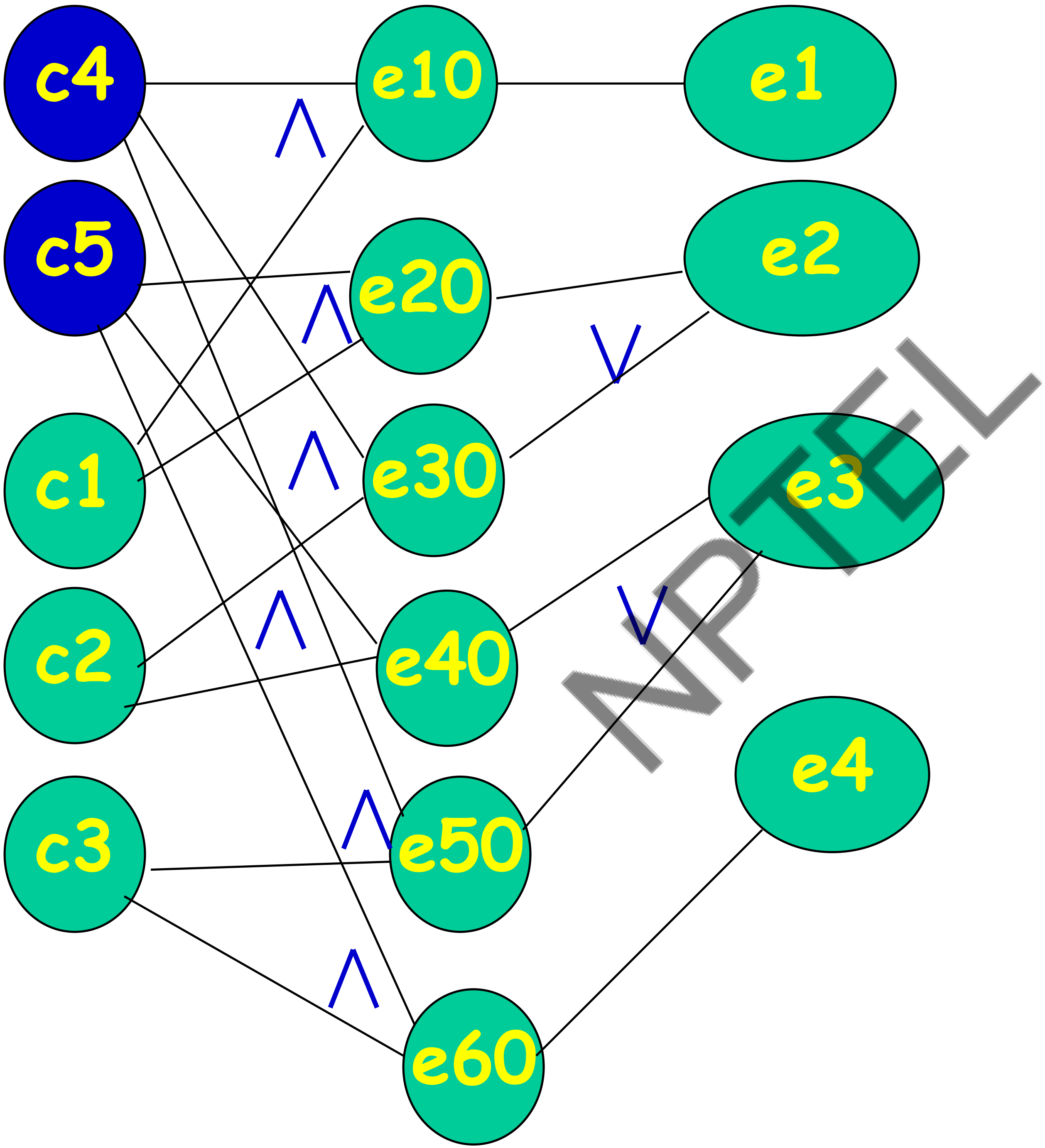
Cause-Effect Graph Example

- If depositing less than Rs. 1 Lakh, rate of interest:
  - 6% for deposit upto 1 year
  - 7% for deposit over 1 year but less than 3 yrs
  - 8% for deposit 3 years and above
- If depositing more than Rs. 1 Lakh, rate of interest:
  - 7% for deposit upto 1 year
  - 8% for deposit over 1 year but less than 3 yrs
  - 9% for deposit 3 years and above

Cause-Effect Graph Example

Causes	Effects
C1: Deposit<1yr	e1: Rate 6%
C2: 1yr<Deposit<3yrs	e2: Rate 7%
C3: Deposit>3yrs	e3: Rate 8%
C4:Deposit <1 Lakh	e4: Rate 9%
C5: Deposit >=1Lakh	

Cause-Effect Graphing



Develop a Decision Table

C1	C2	C3	C4	C5	e1	e2	e3	e4
1	0	0	1	0	1	0	0	0
1	0	0	0	1	0	1	0	0
0	1	0	1	0	0	1	0	0
0	1	0	0	1	1	0	1	0

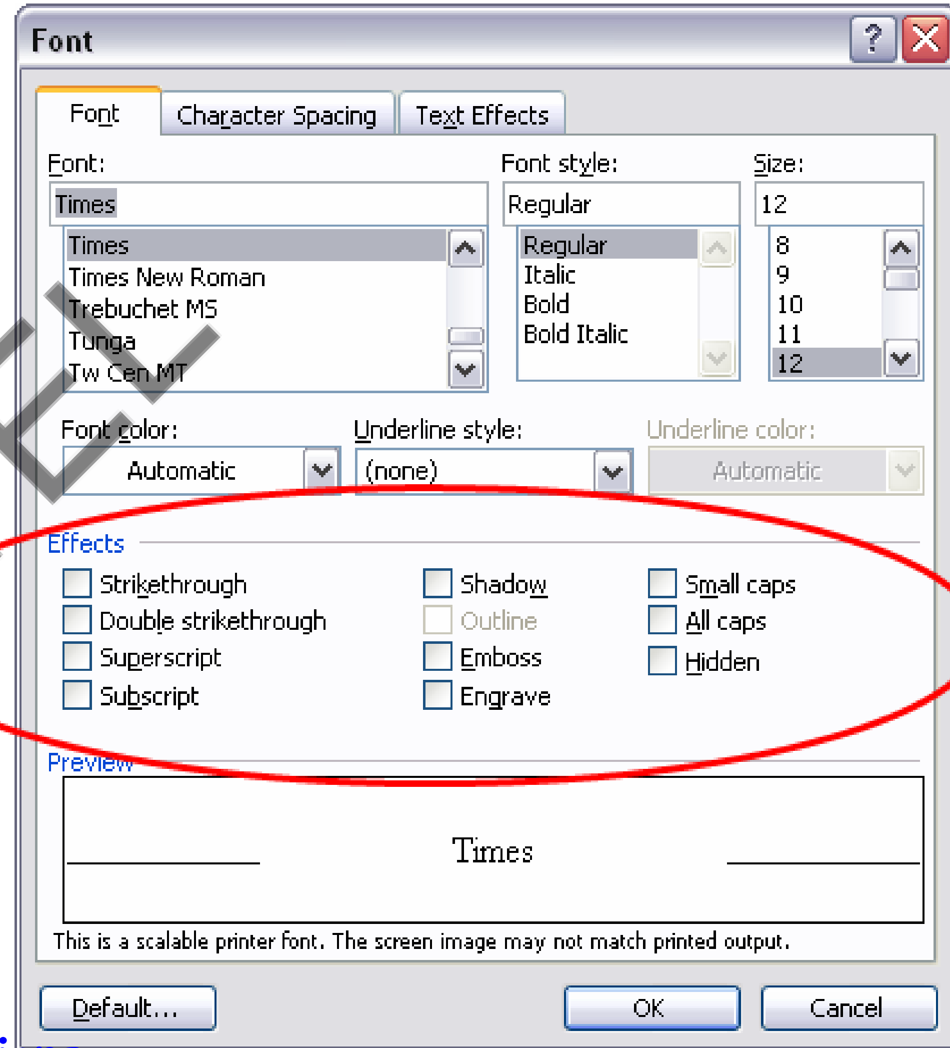
- Convert each row to a test case

Pair-wise Testing

71

Combinatorial Testing

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1



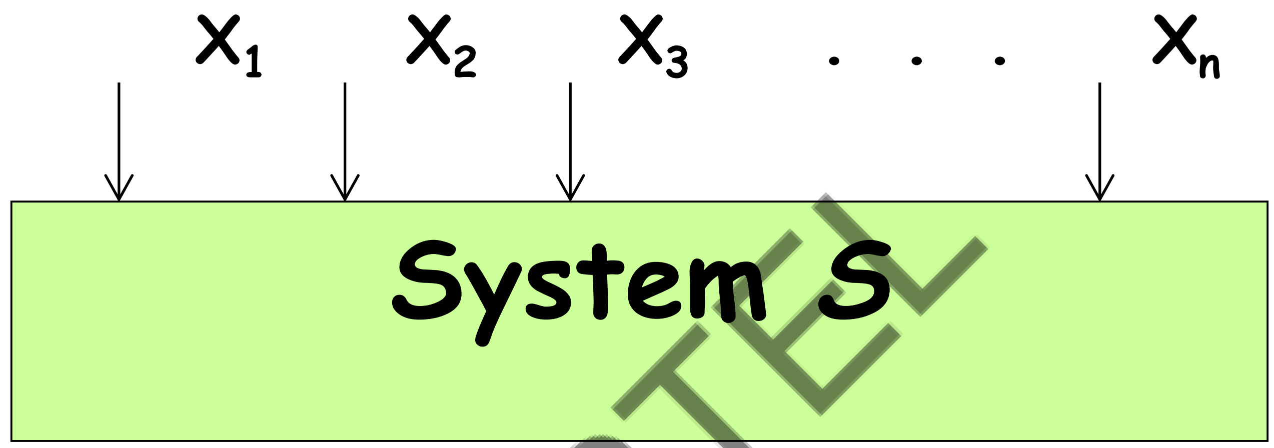
0 = effect off
1 = effect on

2^{10} = 1,024 tests for all combinations

* 10 ^{3} = 1024 * 1000 …. Just too many to test

72

Combinatorial Testing Problem



- Combinatorial testing problems generally follow a simple input-process-output model;
- The "state" of the system is not the focus of combinatorial testing.

Combinatorial Testing

- Instead of testing all possible combinations:
  - A subset of combinations is generated.
- Key observation:
  - **It is often the case that a fault is caused by interactions among a few factors.**
- Combinatorial testing can dramatically reduce the number of test cases:
  - but remains effective in terms of fault detection.

Interaction Testing

Interest Rate | Amount | Months | Down Pmt | Pmt Frequency

All values: every value of every parameters

All pairs: every value of each pair of parameters

etc. . . .

t-way interactions: every value of every t-way combination of parameters

Pairwise Reductions

Number of inputs	Number of selected test data values	Number of combinations	Size of pair wise test set
7	2	128	8
13	3	1.6×10^{6}	15
40	3	1.2×10^{19}	21

76

Fault-Model

- **A t-way interaction fault:**
  - Triggered by a certain combination of t input values.
  - A simple fault is a 1-way fault
  - Pairwise fault is a t-way fault where t = 2.

- **In practice, a majority of software faults consist of simple and pairwise faults.**

Single-mode Bugs

- The simplest bugs are single-mode faults:
  - **Occur when one option causes a problem regardless of the other settings**
  - **Example:** A printout is always gets smeared when you choose the duplex option in the print dialog box
    - Regardless of the printer or the other selected options

Double-mode Faults

- **Double-mode faults**
  - Occurs when two options are combined
  - **Example:** The printout is smeared only when duplex is selected and the printer selected is model 394

79

Multi-mode Faults

- **Multi-mode faults**
  - Occur when three or more settings produce the bug
  - This is the type of problems that make complete coverage seem necessary

Example of Pairwise Fault

- begin
  - int x, y, z;
  - input (x, y, z);
  - if (x == x1 and y == y2)
    - output (f(x, y, z));
  - else if (x == x2 and y == y1)
    - output (g(x, y));
  - Else // Missing (x == x2 and y == y1) f(x, y, z) – g(x, y);
    - output (f(x, y, z) + g(x, y))
- end
- Expected: x = x1 and y = y1 => f(x, y, z) – g(x, y); x = x2, y = y2 => f(x, y, z) + g(x, y)

Example: Android smart phone testing

- Apps should work on all combinations of platform options, but there are 3 x 3 x 4 x 3 x 5 x 4 x 4 x 5 x 4 = 172,800 configurations

HARDKEYBOARDHIDDEN_NO
HARDKEYBOARDHIDDEN_UNDEFINED
HARDKEYBOARDHIDDEN_YES

KEYBOARDHIDDEN_NO
KEYBOARDHIDDEN_UNDEFINED
KEYBOARDHIDDEN_YES

KEYBOARD_12KEY
KEYBOARD_NOKEYS
KEYBOARD_QWERTY
KEYBOARD_UNDEFINED

NAVIGATIONHIDDEN_NO
NAVIGATIONHIDDEN_UNDEFINED
NAVIGATIONHIDDEN_YES

NAVIGATION_DPAD
NAVIGATION_NONAV
NAVIGATION_TRACKBALL
NAVIGATION_UNDEFINED
NAVIGATION_WHEEL

ORIENTATION_LANDSCAPE
ORIENTATION_PORTRAIT
ORIENTATION_SQUARE
ORIENTATION_UNDEFINED

SCREENLAYOUT_LONG_MASK
SCREENLAYOUT_LONG_NO
SCREENLAYOUT_LONG_UNDEFINED
SCREENLAYOUT_LONG_YES

SCREENLAYOUT_SIZE_LARGE
SCREENLAYOUT_SIZE_MASK
SCREENLAYOUT_SIZE_NORMAL
SCREENLAYOUT_SIZE_SMALL
SCREENLAYOUT_SIZE_UNDEFINED

TOUCHSCREEN_FINGER
TOUCHSCREEN_NOTOUCH
TOUCHSCREEN_STYLUS
TOUCHSCREEN_UNDEFINED

Identifying Variables

- Before implementing all-pairs testing, we need to identify the variables
  - E.g., a sign-on component of a sales application might have the following variables:

Orientation	Screen	Keyboard
Portrait	Large	QUERTY
Landscape	Small	12Key
	Normal	

An exhaustive testing would have 12 combinations: (2x3x2)

Identifying Variables

- After identifying the variables,
  - Variables should be arranged by the number of values they contain from greatest to lowest

3 2

Screen	Orientation	Keyboard
Large	Portrait	QUERTY
Small	Landscape	12Key
Normal		

Creating the First Pair of Values (2)

- Match each value of the first factor with each value of the second one.

Screen	Orientation
Large	Portrait
Large	Landscape
Small	Portrait
Small	Landscape
Normal	Portrait
Normal	Landscape

Adding a Third Value

- Add a third variable
  - Start by entering the values in order in a third column, repeating as necessary

Screen	Orientation	Keyboard
Large	Portrait	QWERTY
Large	Landscape	12Key
Small	Portrait	QWERTY
Small	Landscape	12Key
Normal	Portrait	QWERTY
Normal	Landscape	12Key

What is White-box Testing?

- White-box test cases designed based on:
  - Code structure of program.
  - White-box testing is also called structural testing.

White-Box Testing Strategies

- **Coverage-based:**
  - Design test cases to cover certain program elements.
- **Fault-based:**
  - Design test cases to expose some category of faults

White-Box Testing

- Several white-box testing strategies have become very popular :
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - MC/DC coverage
  - Mutation testing
  - Data flow-based testing

Why Both BB and WB Testing?

Black-box

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- Does not tell if extra functionality has been implemented.

White-box

- Does not address the question of whether a program matches the specification
- Does not tell if all of the functionality has been implemented
- Does not uncover any missing program logic

Coverage-Based Testing Versus Fault-Based Testing

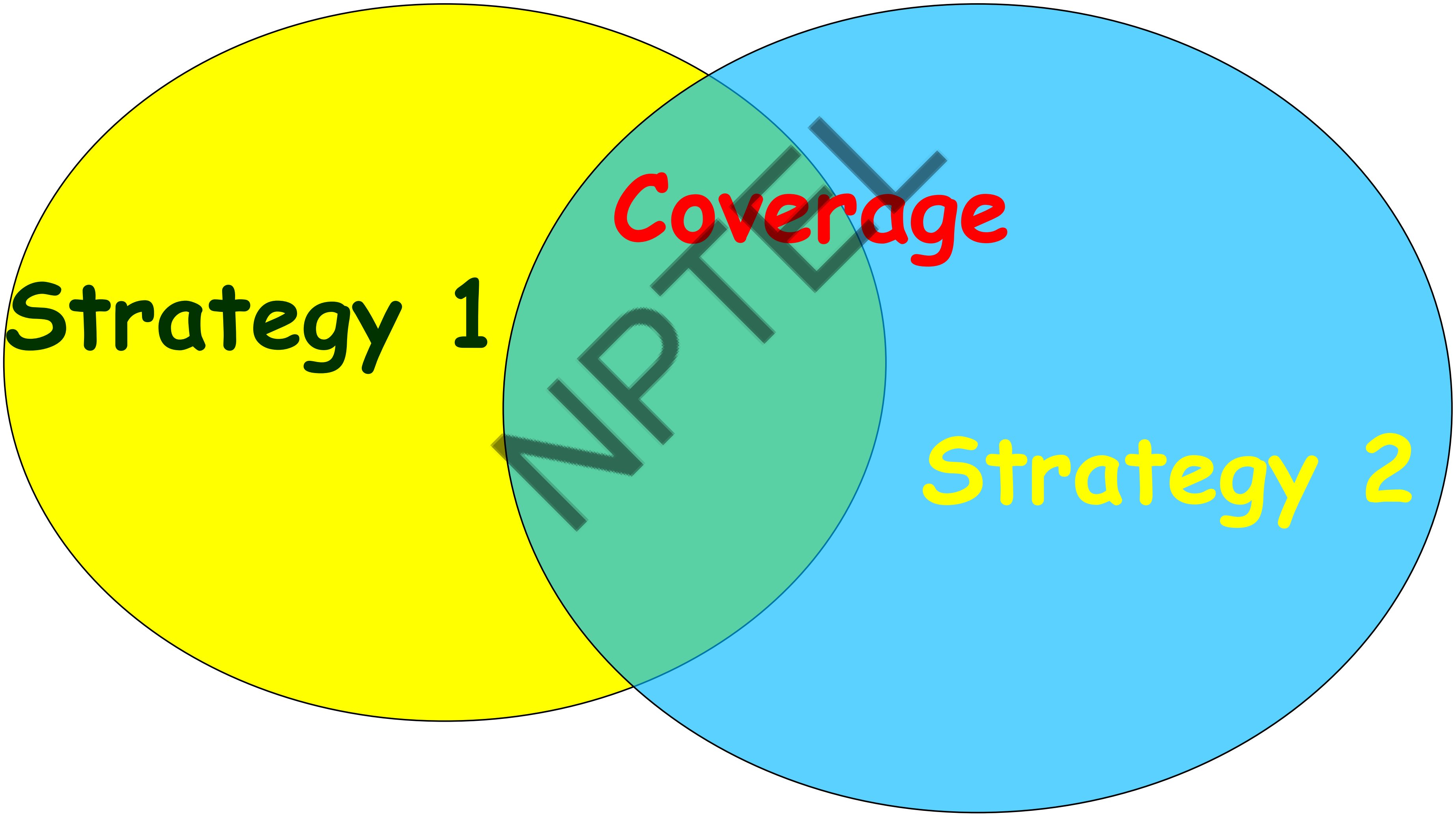
- Idea behind coverage-based testing:
  - Design test cases so that certain program elements are executed (or covered).
  - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing:
  - Design test cases that focus on discovering certain types of faults.
  - Example: Mutation testing.

Types of program element Coverage

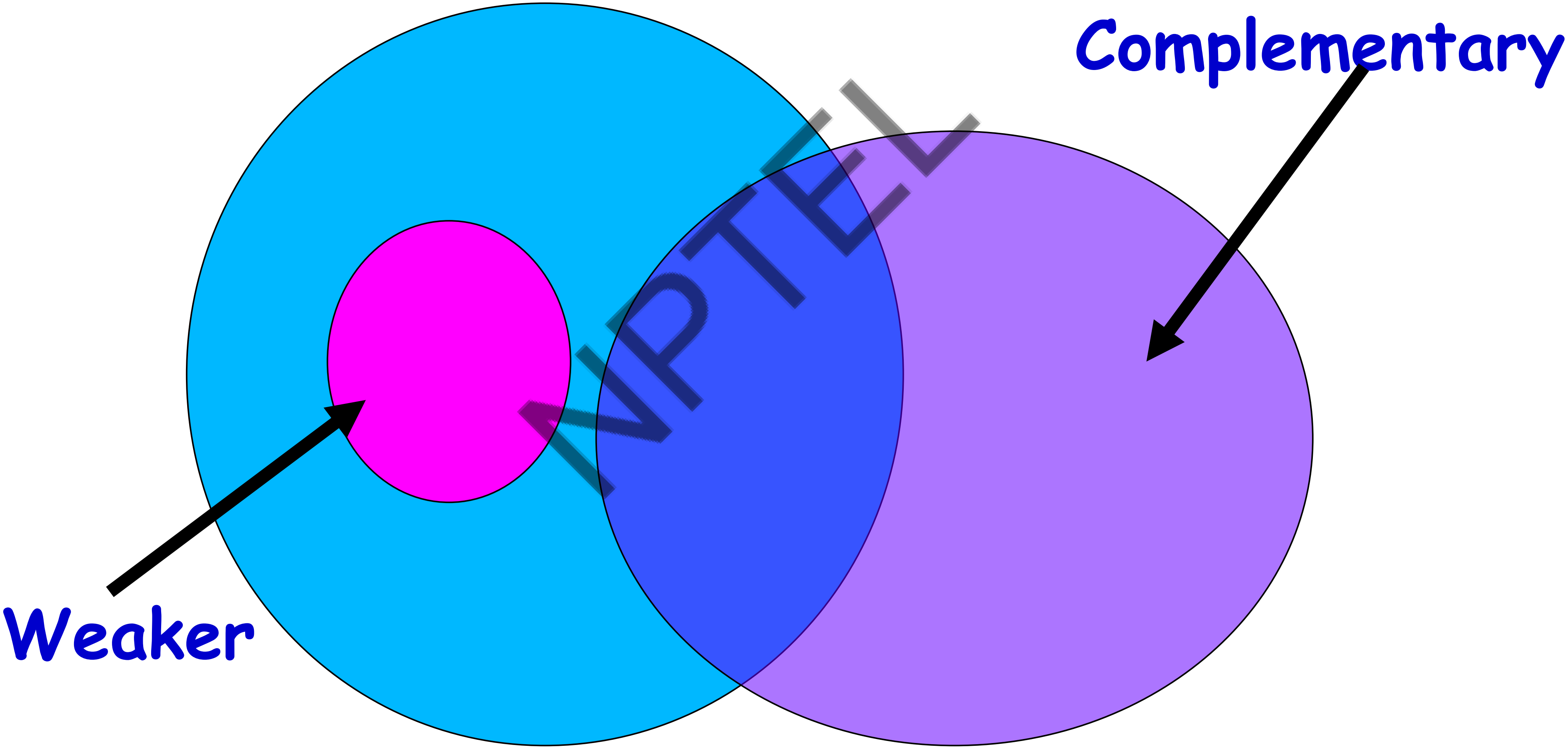
- **Statement:** each statement executed at least once
- **Branch:** each branch traversed (and every entry point taken) at least once
- **Condition:** each condition True at least once and False at least once
- **Multiple Condition:** All combination of Condition coverage achieved
- **Path:**
- **Dependency:**

Stronger and Weaker Testing

Complementary Testing



Stronger, Weaker, and Complementary Testing



Statement Coverage

- Statement coverage strategy:
  - Design test cases so that every statement in the program is executed at least once.

Statement Coverage

- The principal idea:
  - Unless a statement is executed,
  - We have no way of knowing if an error exists in that statement.

Statement Coverage Criterion

- However, observing that a statement behaves properly for one input value:
  - **No guarantee that it will behave correctly for all input values!**

Statement Coverage

- Coverage measurement:

$$\frac{\#\ \text{executed statements}}{\#\ \text{statements}}$$

- **Rationale:** a fault in a statement can only be revealed by executing the faulty statement

Example

- int f1(int x, int y){
- 1 while (x != y){
- 2   if (x>y) then
- 3   x=x-y;
- 4   else y=y-x;
- 5 }
- 6 return x;   }

Euclid's GCD Algorithm

Example

```
int f1(int x,int y){
 while (x != y){
     if (x>y) then
           x=x-y;
     else y=y-x;
 }
return x;            }
```

Euclid's GCD Algorithm

Euclid's GCD Algorithm

- By choosing the test set {(x=3,y=3),(x=4,y=3), (x=3,y=4)}
  - *All statements are executed at least once.*

Branch Coverage

- **Also called decision coverage.**
- Test cases are designed such that:
  - Each branch condition
    - Assumes true as well as false value.

Example

```
int f1(int x,int y){
 while (x != y){
     if (x>y) then
           x=x-y;
     else y=y-x;
  }
 return x;          }
```

Example

- Test cases for branch coverage can be:

- {(x=3,y=3),(x=3,y=2),
(x=4,y=3), (x=3,y=4)}

Branch Testing

- **Adequacy criterion:** Each branch (edge in the CFG) must be executed at least once
- Coverage:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

Quiz 1: Branch and Statement Coverage: Which is Stronger?

- Branch testing guarantees statement coverage:
  - A stronger testing compared to the statement coverage-based testing.

Stronger Testing

- Stronger testing:
  - Superset of weaker testing
  - A stronger testing covers all the elements covered by a weaker testing.
  - Covers some additional elements not covered by weaker testing

Coverage Report

Coverage Report

http://www.cafeconlech

Packages

All
org.jaxen
org.jaxen.dom
org.jaxen.dom.html

All Packages

Classes

BaseXPath *(77%)*
Context *(93%)*
ContextSupport *(91*
DefaultNavigator *(3*
DocumentNavigator
DOMXPath *(100%)*
DocumentNavigator
HTMLXPath *(0%)*
NamespaceNode *(*
DocumentNavigator
Dom4jXPath *(100%*

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	205	69%	80%	2.811
org.jaxen	24	77%	73%	1.38
org.jaxen.dom	3	55%	60%	1.907
org.jaxen.dom.html	2	0%	0%	1.364
org.jaxen.dom4j	2	78%	85%	2.395
org.jaxen.expr	73	73%	84%	1.566
org.jaxen.expr.iter	14	98%	100%	1.029
org.jaxen.function	27	64%	76%	5.373
org.jaxen.function.ext	6	63%	72%	4.235
org.jaxen.function.xslt	1	86%	100%	2.5
org.jaxen.javabean	4	44%	72%	1.87
org.jaxen.jdom	3	62%	63%	2.897
org.jaxen.pattern	13	49%	52%	2.135
org.jaxen.saxpath	8	51%	81%	1.887
org.jaxen.saxpath.base	6	95%	100%	10.723
org.jaxen.saxpath.helpers	2	28%	83%	1.34
org.jaxen.util	15	41%	50%	2.432
org.jaxen.xom	2	71%	66%	1.783

Reports generated by Cobertura.

Coverage Report

http://www.cafeconlech

```
110 128        else if ( nav.isElement( first ) )
111            {
112 100            return nav.getElementQName( first );
113            }
114  28        else if ( nav.isAttribute( first ) )
115            {
116   0            return nav.getAttributeQName( first );
117            }
118  28        else if ( nav.isProcessingInstruction( first ) )
119            {
120   0            return nav.getProcessingInstructionTarget( first );
121            }
122  28        else if ( nav.isNamespace( first ) )
123            {
124   0            return nav.getNamespacePrefix( first );
125            }
126  28        else if ( nav.isDocument( first ) )
127            {
128  28            return "";
129            }
130   0         else if ( nav.isComment( first ) )
131            {
132   0            return "";
133            }
134   0         else if ( nav.isText( first ) )
135            {
136   0            return "";
137            }
138            else {
139   0            throw new FunctionCallException("The argument to the name
140            }
141        }
142
143   8    return "";
144
```


Done

Disabled

Statements vs Branch Testing

- Traversing all edges of a graph causes all nodes to be visited
  - So a test suite that satisfies branch adequacy criterion also satisfies statement adequacy criterion for the same program.
- The converse is not true:
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate).

All Branches can still miss conditions

- Sample fault: missing operator (negation)

digit_high == 1 || digit_low == -1

- Branch adequacy criterion can be satisfied by varying only digit_low
  - **The faulty sub-expression might not be tested!**
  - Even though we test both outcomes of the branch

Condition Coverage

- Also called multiple condition (MC) coverage .
- Test case design:
  - Each component of a composite conditional expression
    - Made to assumes both true and false values.

Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once,
  - That is, given true and false values.

Basic condition testing

- Adequacy criterion: each basic condition must be executed at least once
- Coverage:

$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$

Branch Testing

- To think of it:
  - Branch testing is the simplest condition testing strategy:
  - Compound conditions determining different branches
    - Are given true and false values.

Branch Testing

- Condition testing:
  - Stronger testing than branch testing.
- Branch testing:
  - Stronger than statement coverage testing.

Multiple Condition
↓
Basic Condition
↓
Decision
↓
Statement