# BLACK BOX TESTING (I)

General taxonomy

White and Black Box Testing

- **Software testing**
  - Black box testing (<u>validation</u>)
    - Tester has no access to code
  - White box testing (<u>verification</u>)
    - Tester focuses on the internal structure of the code
  - Writing test cases leads to more complete and specific requirements gathering process at the earlier design phase

"Pure" requirements from customers
OR
Requirements analysis/specification document

Black Box Testing
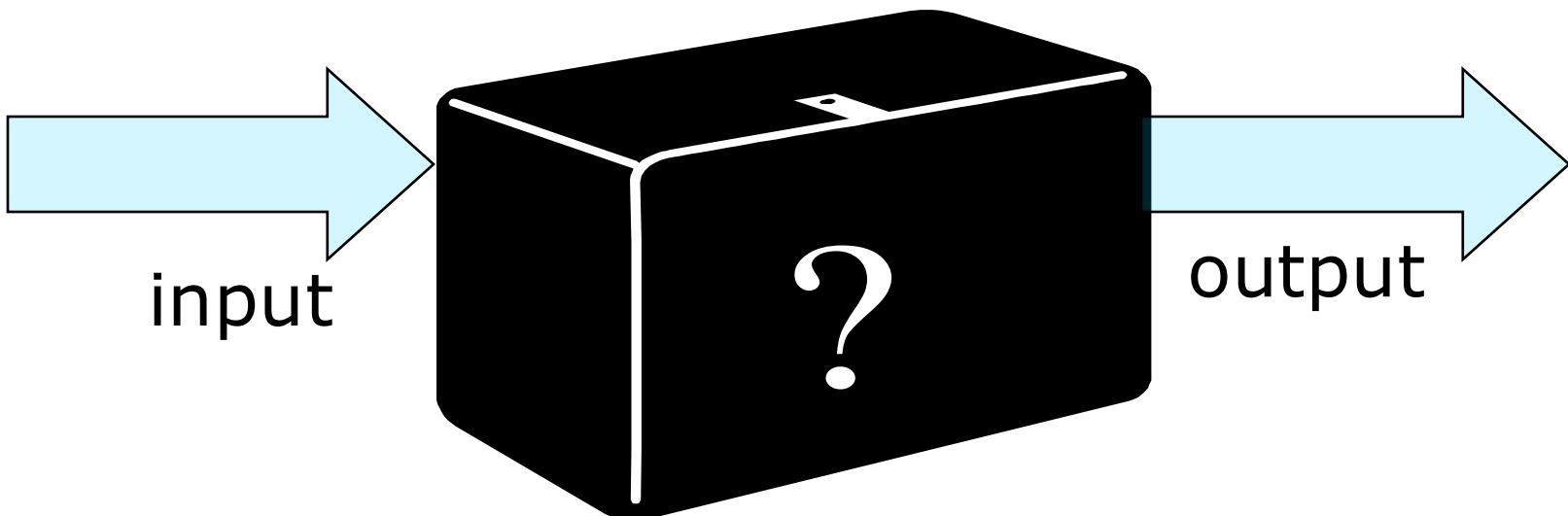
"Pure" requirements from customers
OR
Requirements analysis/specification document

Requirements not correctly specified, identified, reported, and understood

Unspecified functionalities (e.g., Easter egg) not identified

Coincidental correctness

Black Box testing

input → ? → output

Black Box testing – the paradigm

- a.k.a. <u>functional</u> testing and <u>behavioural</u> testing
- Attempts to find 5 general types of errors:
  - ☐ Incorrect or missing functions
  - ☐ Interface errors
  - ☐ Errors in data structures used by interfaces
  - ☐ Behaviour or performance errors
  - ☐ Initialization and termination errors
- Better not to be performed by the programmer
  - ☐ Third-party testers are more suited to test for what the customer specifies

Issues on Black Box testing

- How is validity tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operations?

Test case planning/design

- Test planning early in development cycle (cannot be emphasized enough)
- Example template for a test case:

Test ID	Description	Expected results	Actual results

Identifier

Set of steps and input conditions

- Be very specific in the field of description and expected result
  - Recreating the problem → to be repeated and fixed
- Test cases traceable to customer requirements
  - Success paths → no error conditions in test cases
  - Failure paths → error conditions added in test cases

Exercise

- Write a test case for the student information system that you use

- Explain the rationale

Collection of test scenarios (1)

ID	Description	Expected Results	Actual Results
1	Enrollment in exam – test whether a confirmation e-mail has been sent out	Receipt of email	
2	Average of exams 1. Enter details of a student 2. Compute manually the average - assume it is 3.7	3.7	
2	Average of exams 1. Log in 2. Compute the average	The average computed manually	

Collection of test scenarios(2)

ID	Description	Expected Results	Actual Results
3	Login and password testing Check the 4 cases with template users	Only for <correct user,correct password> the system is accessed	
4	Check of all the exams 1. Connect as a template student in a predefined department 2. Read in the student guide which exams should the student pass 3. Check if such exams are there	The exams in the student guide for the template student	

Collection of test scenarios(3)

ID	Description	Expected Results	Actual Results
5	**Timetable** Connect to the student information system and check if the details are the same as provided	?????	

Categories of Black Box Testing

- **Failure (dirty) test cases and error guessing**
- **Checklists**
- **Usage-based (operational profiles) testing**
- **Input domain testing**
  - Input equivalence classes (partitioning)
  - Boundary value analysis
- **Decision tables and cause-effect graphs**
- **Syntax-driven testing**
- **Finite state machines**

Categories of Black Box Testing

- **Failure (dirty) test cases and error guessing**
- **Checklists**
- **Usage-based (operational profiles) testing**
- **Input domain testing**
- **Decision tables and cause-effect graphs**
- **Syntax-driven testing**
- **Finite state machines**
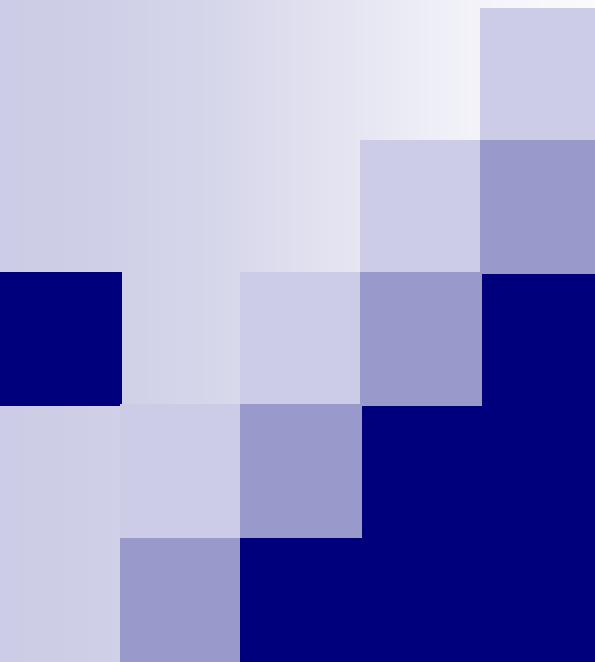
Level of formalism

Level of formalisms: from **semi-formal** to **formal** approaches

Black box test case automation
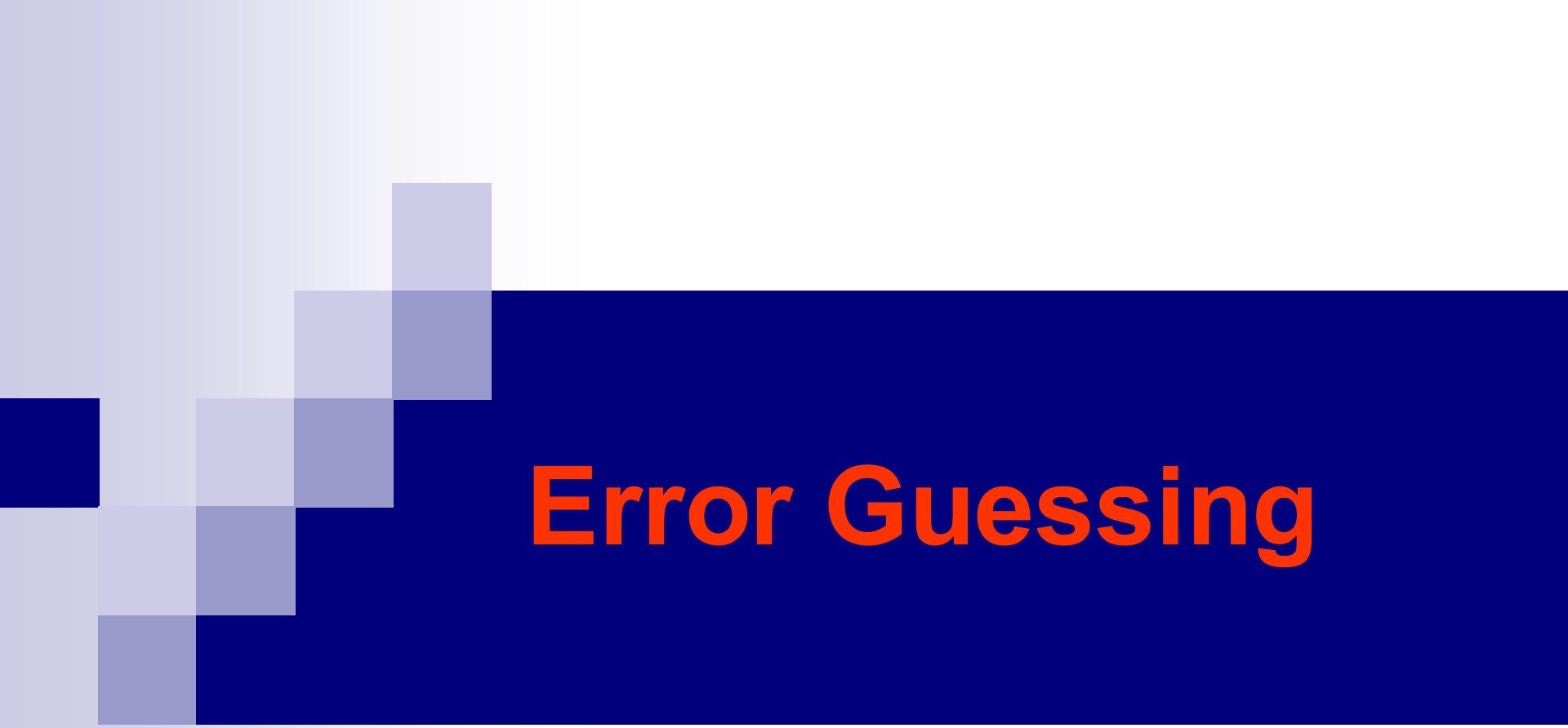
- Automated test cases to test the logic behind certain types of black box test cases

- More automated → run more frequently → faster defect-detection

- Black box test cases can then be run more smoothly through user interface (UI)

- Motivation to decouple program/business logic from user interface logic

- Once automated, these cases can be run many times to check new code

Failure (dirty) test cases

Failure ("Dirty") Test Cases

- Test _every_ possible thing a user could do with the system to demolish the software
- Think _diabolically_!
- Look at every input!
- Be creative → consider the user as someone from another planet
- Examples
  - Can any form of input to the program cause division by zero?
  - What if the input type is wrong?
  - What if mandatory fields are not entered?

Error Guessing

Error Guessing (1)

Intuition
Innovation
Experience

....

Sorting subroutine

Test cases

•Input list is empty
•Input list contains one entry
•All entries in the input have the same value
•The input is already sorted

Error Guessing (2)

Intuition
Innovation
Experience

….

Counting frequency of words

Test cases

- File empty
- File non-existent
- File only has blanks
- Contains a single word
- All words are the same
- Multiple consecutive blank lines
- Multiple blanks between words
- Blanks at the start
- Words in sorted order
- Blanks at the end of file
- …

Checklists

Checklists

Start running software and make observations about expected/unexpected behavior

Organize features to be tested (checklist)

Check off an item on the list once the corresponding Item (feature) has been tested

Checklists-example

Checklist No	Checkpoints	Checked Yes/No
1	Screen resolution for every page	
2	Browser compatibility	
3	Broken links	
4	Broken images	
5	Flash Animation	
6	Alt text	
7	Fonts – visual check	
8	Spell check	
9	Image alignments	

☐ Done ☐ Pending - **Navigation Buttons:** Ensure all navigation buttons function properly.

☐ Done ☐ Pending - **User Input:** Verify that user input fields accept and process data correctly.

☐ Done ☐ Pending - **Data Handling:** Test for correct data saving and loading.

☐ Done ☐ Pending - **Core Feature Functionality:** Confirm primary features work as intended.

☐ Done ☐ Pending - **Network Handling:** Assess app behavior under different network conditions.

☐ Done ☐ Pending - **Device Compatibility:** Ensure compatibility across various devices and OS versions.

☐ Done ☐ Pending - **Performance:** Check for any lag or crashes.

☐ Done ☐ Pending - **Notifications:** Verify correct functioning of push notifications and alerts.

☐ Done ☐ Pending - **Security Standards:** Ensure compliance with security and privacy regulations.

☐ Done ☐ Pending - **Accessibility Features:** Test for accessibility, including screen reader compatibility.

☐ Done ☐ Pending - **Multimedia Elements:** Check loading and display of images and videos.

High-level checklist of relational database (example)

- Abnormal termination
- Backup and restore
- Communication
- File I/O
- Installation
- Logging and recovery
- Migration

Checklists

Item	Test 1	Test 2	Test 3 …. Test P	
	●			
		●		●
			●	

Checklists

The simplest form of black box testing; no systematic method

Could be expanded and formalized

List of items to be checked (coding standard, specification items)

Levels of granularity (abstraction):
        high-level system functions down to specific
        low-level functions of individual components

Checklists-types of tests

1. **Unit Testing**
   - ☐ Test individual units or components of code.
   - ☐ Ensure each unit performs as expected.

2. **Integration Testing**
   - ☐ Test combined functionality of integrated units.
   - ☐ Check data flow and interaction between modules.

3. **System Testing**
   - ☐ Validate the complete and integrated software system.
   - ☐ Verify compliance with specified requirements.

4. **User Acceptance Testing (UAT)**
   - ☐ Conduct testing with actual end-users.
   - ☐ Validate software in real-world scenarios.

Hierarchical checklists

Item-1
        item-11
        Item-12
        Item-13

Item-2
        Item-21
   Item-22
        Item-23
        Item-24

Item-3

Types of checklists

Hierarchical checklists

High level checklist expandable to a full checklist at a lower level until stopped at a level of detail deemed enough by some criteria

Combination of checklists

	Component	Standard item
		s1 s2 …. sm

c1
c2
…….
cn

Example coding standard checklist

Limitations of checklists

High-level, lack of specificity
No detailed items to test

Difficult translation to test cases
Difficulties in re-running the test cases (additional info about environment)
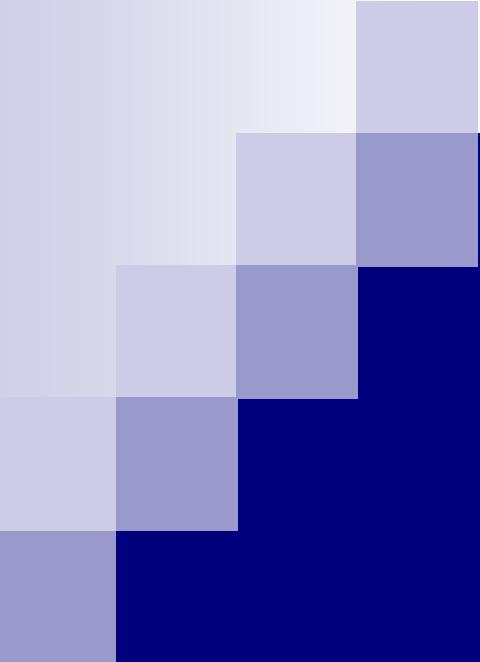
1.**It is difficult to come up with checklists that cover all functional (black box) or structural (white box) components → missed areas of coverage (holes in coverage)**

2.**To provide good coverage, we may end up with a lot of overlaps between Items on the checklists→ redundant testing**

3. **Complex interactions between components→ checklists are hard to formulate**
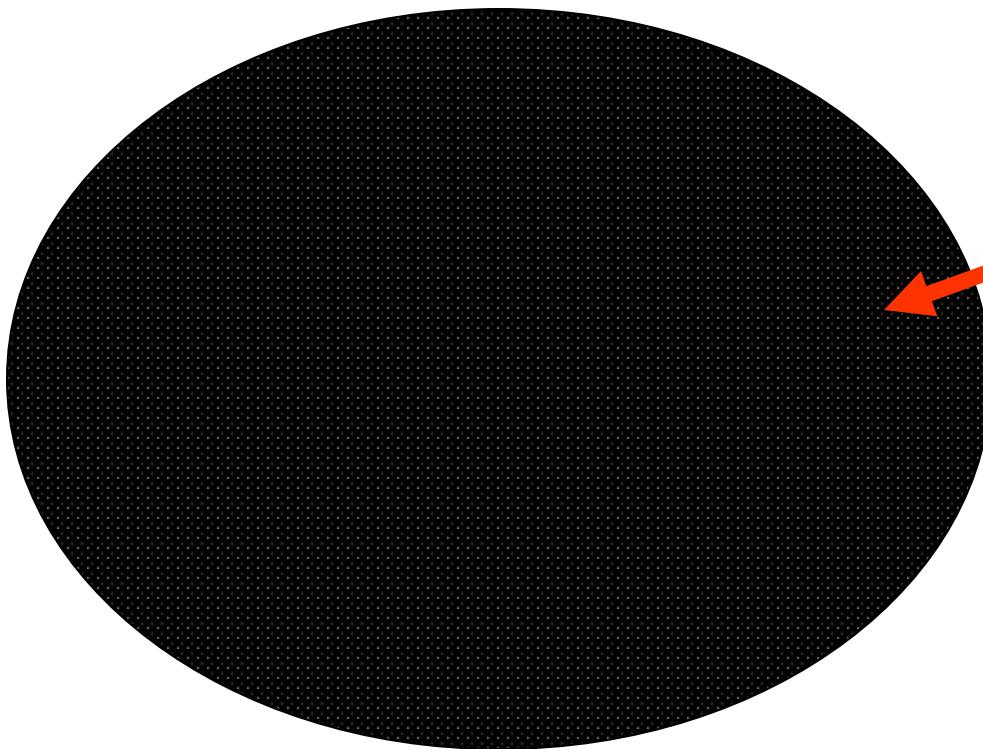
Solutions: 1-2: partition coverage   3: finite state machines

Partition – Based Testing

Partition coverage- example 1

$$ax^2 + bx + c = 0$$
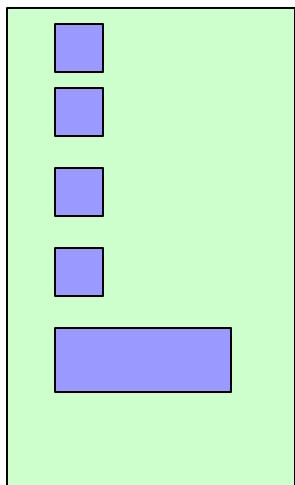
a, b, c - 32 bit numbers

Space of 32 bit numbers {a, b, c}

a, b, c - 32 bit numbers→ 2^{32} * 2^{32} * 2^{32} = 2^{96} test cases

Partition coverage- example 2

GUI : four check boxes, one two-position field (time…)

$2^4(16)$ *(00…99)   total 1,600 potential test cases
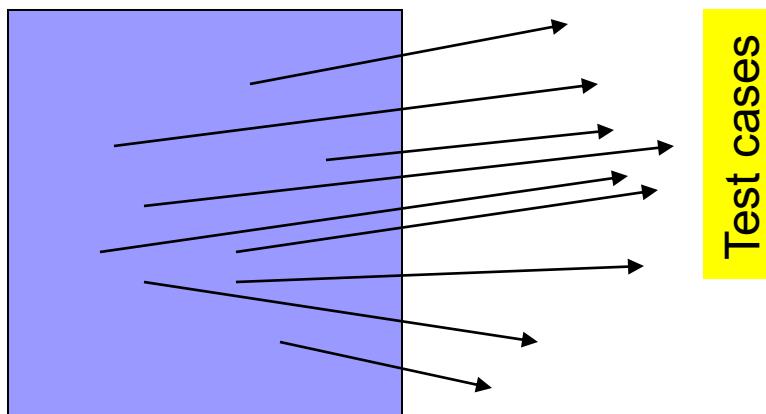
digit, space
letter, character, special symbol…

16 bit 2^{16} possibilities for the position
2 positions
16 (test cases for check boxes)

68,719,476,736 test cases
1 test /second 2,177.5 years

Equivalence partitioning

Input space

Problem: exhaustive-input testing is not feasible

Solution: need a small subset of all possible inputs
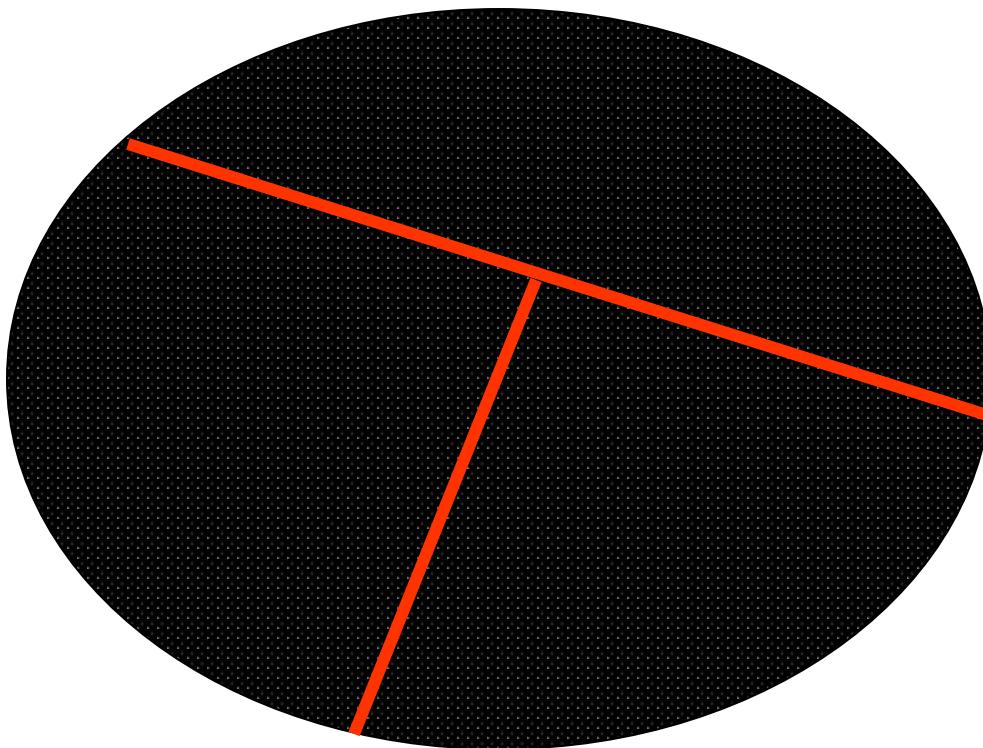
Subsets of high probability of finding errors

<u>test case</u>
reduces the number of other test cases that must be developed

Partition coverage

$$ax^2 + bx + c = 0$$
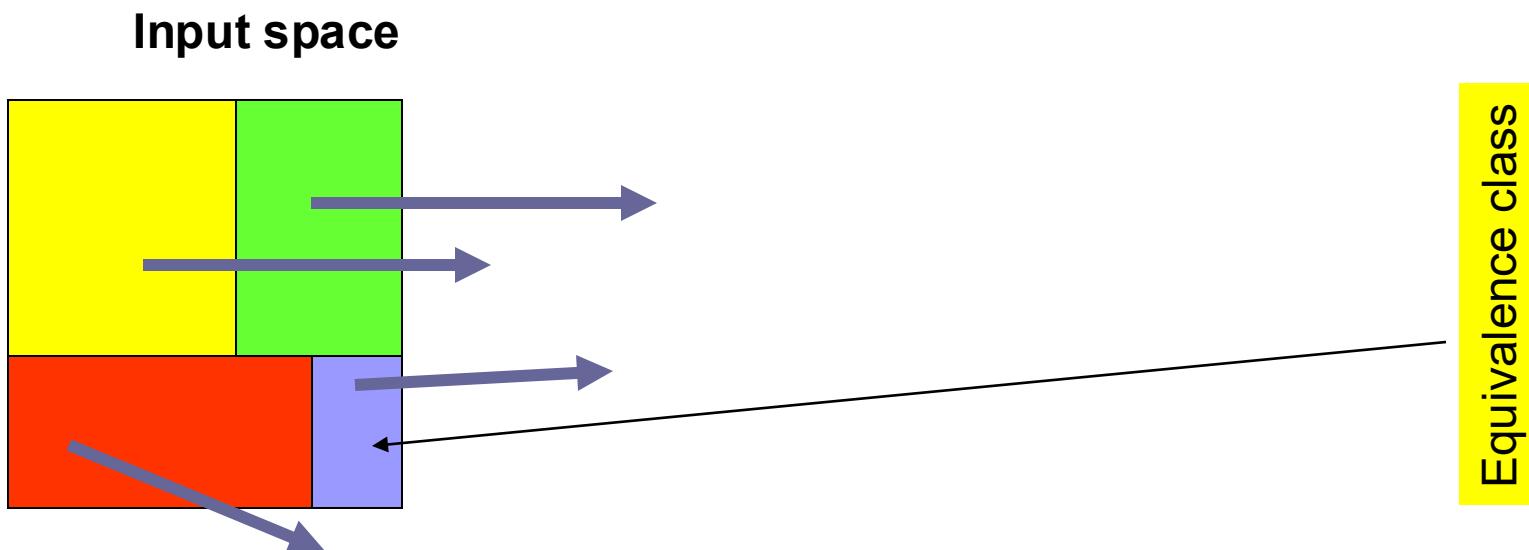
$d = b^2 - 4ac$

Partition – 3 test cases
d = 0
d > 0
d < 0

Equivalence class

Input space

Equivalence class

we assume that that a test of a representative value of each equivalence class is equivalent to any other value in the same equivalence class

Partition

Set S
Collection of subsets A_1, A_2, …, A_c
such that they satisfy the following conditions

- are mutually <u>exclusive</u>

- are collectively <u>exhaustive</u>

Partition and equivalence classes

Subsets A_i in the partition are called equivalence classes

Partitions based on:

Properties of elements

Relations

Logical conditions
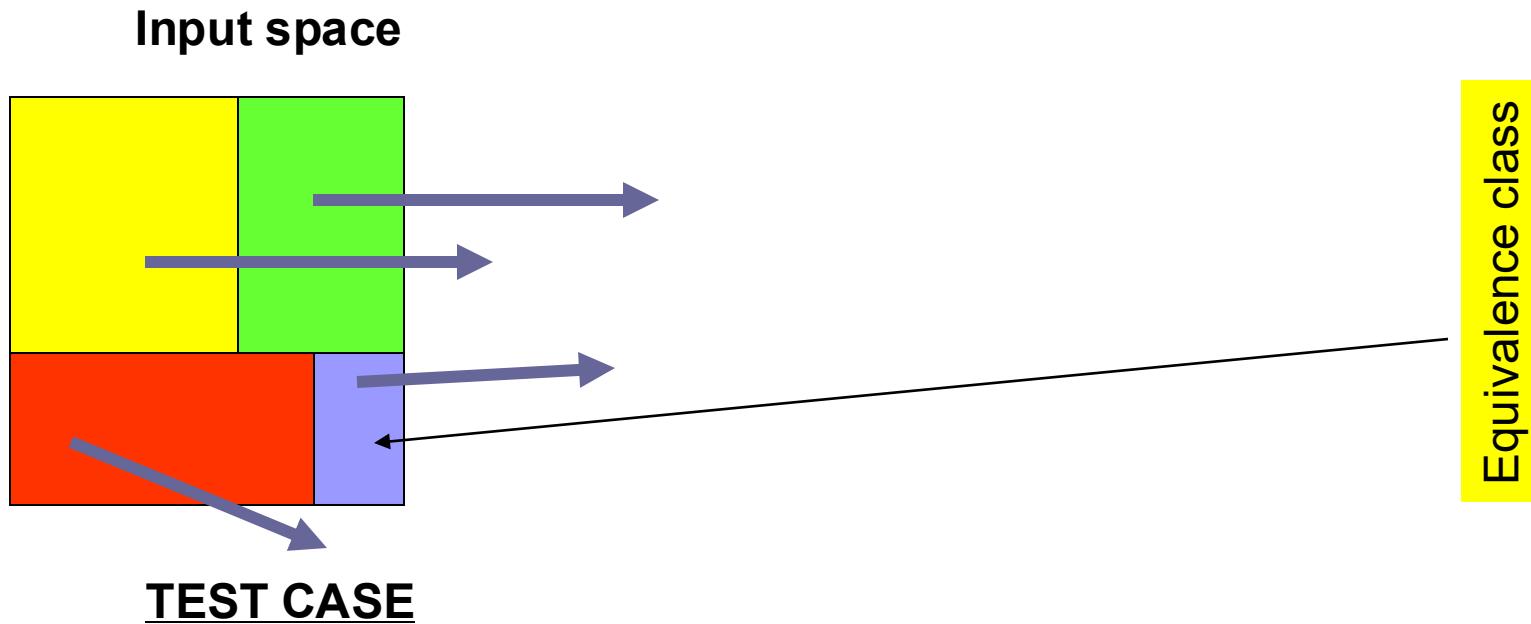
Partitions and equivalence classes

Equivalence class – relation R(a, b) that is

(a)Symmetric   R(a,b) = R(b,a)
(b) transitive   R(a,b) & R(b,c) → R(a,c)
(c)Reflexive   R(a,a)

Example   "==" implies some equivalence class

Equivalence partitioning: equivalence classes

Input space

TEST CASE

(a) reduces the number of other test cases that must be developed to achieve some predefined goal of "reasonable" testing

(b) It covers a large set of other possible test cases; tells us something about the presence of errors over and above the specific set of input values

Equivalence partitioning: Test case design

1. Identification of equivalence classes

2. Defining the test cases

EQUIVALENCE CLASSES

Two types of classes:
valid equivalence classes
invalid equivalence classes

Note: equivalence classes need to be homogeneous – otherwise split the equivalence class into smaller equivalence classes

Equivalence partitioning: Test case design

DEFINING 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 <u>as many of the uncovered valid equivalence classes as possible</u>

Until tests have covered all invalid equivalence classes, write a test case that covers <u>one and only one</u>, of the uncovered invalid equivalence classes

Equivalence classes: some examples

➡️ **Input: range of values, 1…999 (item count)**

Valid equivalence class 1…999
Invalid equivalence class "less than 1", "greater than 999"

➡️ **One through six owners (insurance policy)**

Valid equivalence class 1..6
Invalid equivalence class 0   >6

➡️ **First character of the identifier must be a letter**

Valid equivalence class: letter
Invalid equivalence class: not a letter

Equivalence classes: some examples

DIM statement **DIM name [, , , …]**
(a) Array name: up to six letters or digits, the first must be a letter
(b) Number of dimensions 1 , 2,…, 7

Input condition	Valid equivalence classes	Invalid equivalence classes
Length of array name	1-6 (1)	0 (2) >6 (3)
Array name starts with letter	yes (4)	no (5)
Number of dimensions	1-7 (6)	0 (7) > 7 (8)

Equivalence classes: some examples

Input condition	Valid equivalence classes	Invalid equivalence classes
Size of array name	1-6 (1)	0 (2) >6 (3)
Array name starts with letter	yes (4)	no (5)
Number of dim	1-7 (6)	0 (7) > 7 (8)

TEST CASES to cover all equivalence classes

DIM A [2]   covers (1) (4) (6)

DIM 1B[4]   covers (5)   DIM D[ ]   covers   (7)   DIM [10, 30, 11, 12, 13, 14, 45] covers (2)

DIM DATA [100, 110, 20, 50, 90, 200, 70, 100] (8)   DIM DATASENSOR1[100, 100] (3)

Example (1)

Specifications

Design a program that adds two integers which are entered

- **Each number should have one or two digits**
- **The program will print the sum**
- **Press** ENTER **after each number**
- **To start the program, type** ADDER

?
3
?
10
17

Example (2)
Equivalence analysis

Each variable: 1   to 99   99 values
                 0   1 value
                 -1 to -99   99 values

199 * 199 = **39, 601** combination tests

Boundaries: 100 and above
             -100 and below

Example (2) Myers' boundary table

Variable	Valid case equivalence classes	Invalid case equivalence classes	Boundary and special cases	Notes
1st no	-99 to 99	>99 <-99	99, 100 -99 -100	
2nd no	-99 to 99	>99 <-99	99, 100 -99 -100	

Example
Equivalence analysis (3)
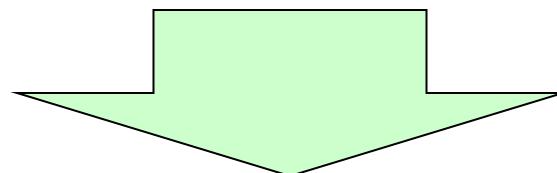
A program takes two inputs – a string *s* and an integer *n*

Program determines *n* most frequent characters

Example
Equivalence analysis (3)

A program takes two inputs – a string *s* and an integer *n*

Program determines *n* most frequent characters

A program takes two inputs – a string *s* of ASCII printable characters of limited length N and an integer *n*

Program determines *n* most frequent characters

Example
Equivalence analysis (3)

(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b

Example Equivalence analysis (3)

Input	Valid Equivalent Class	Invalid Equivalent class
s:string	1: Contains numbers 2: Lower case letters 3: upper case letters 4: special chars 5: string length between 0-N(max)	1: non-ascii char 2: string length > N
n:int	6: Int in valid range	3: Int out of range

Equivalence classes

$$\sqrt{1/x}$$

all positive numbers

all negative numbers

zero value (zero divide error)

Equivalence class testing: strategies

Weak normal equivalence class testing

[*single* fault assumption]

Design a test case so that it covers one and only one invalid equivalence class and as many as possible valid equivalence classes
(Note: weak robust equivalence class – only one <u>valid</u> equivalence class)

Strong normal equivalence class testing

[*multiple* fault assumption]

Select test cases from the Cartesian product of equivalence classes.
Covers all equivalence classes; one of each possible combinations of inputs

Equivalence class testing: example

Sensor-1:   [0,a]   (a, b) [b, 100] range [0, 100]   (a, b) - invalid

Sensor -2:   A, B, C   B -invalid

Sensor -3   a, b, c   a, b – invalid

Weak normal equivalence class testing:

Strong normal equivalence class testing:

Next Date program –equivalence classes

Takes 3 integers (month, day, year) in the Gregorian calendar and determines the date based on the Gregorian calendar

Next Date program –equivalence classes

Input/output	Valid class subsets	Invalid class subsets
Month	v1—30-day months	i1—>= 13
	v2—31-day months	i— <= 0
	v3—February	i3—any noninteger
		i4—empty
		i5—>= 3 integers

Next Date program –equivalence classes

Input/output	Valid class subsets	Invalid class subsets
Day	v4—1–30	i6—>=32
	v5—1–31	i7—<= 0
	v6—1–28	i8—any noninteger
	v7—1–29	i9—empty
		i10—>= 3 integers

Equivalence classes generate boundary conditions(1, 28, 29, 30, 31)

Next Date program –equivalence classes

Valid classes

Year

v8—1582–3000**
v9—non–leap year
v10—leap year
v11—century non–leap
     year
v12—century leap year

Invalid classes

i11—<= 1581
i12—>= 3001
i13—any noninteger
i14—empty
i15—>= 5 integers

Year 3000 – chosen arbitrarily

Next Date program –equivalence classes

	Valid classes	Invalid classes
Year	v8—1582–3000**	i11—<= 1581
	v9—non–leap year	i12—>= 3001
	v10—leap year	i13—any noninteger
	v11—century non–leap year	i14—empty
	v12—century leap year	i15—>= 5 integers

```
//Example algorithm used to calculate leap years
public static bool IsLeapYear(int year)
{
    return (year % 4 == 0 && year % 100 != 0 || year % 400 == 0);
}
```

Most years that can be divided evenly by 4 are leap years
Exception: Century years are NOT leap years UNLESS they can be evenly divided by 400.

Next Date program –equivalence classes

Calendar for year 1582 (Italy)

January
Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

1:1Q 8:F 17:3Q 24:N 31:1Q

February
Mo	Tu	We	Th	Fr	Sa	Su
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28				

7:F 15:3Q 22:N

March
Mo	Tu	We	Th	Fr	Sa	Su
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

1:1Q 9:F 17:3Q 24:N 31:1Q

April
Mo	Tu	We	Th	Fr	Sa	Su
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

May
Mo	Tu	We	Th	Fr	Sa	Su
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

June
Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

July
Mo	Tu	We	Th	Fr	Sa	Su
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

5:F 12:3Q 19:N 27:1Q

August
Mo	Tu	We	Th	Fr	Sa	Su
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

4:F 10:3Q 18:N 26:1Q

September
Mo	Tu	We	Th	Fr	Sa	Su
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

2:F 9:3Q 16:N 25:1Q

October
Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

November
Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

December
Mo	Tu	We	Th	Fr	Sa	Su
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Next Date program –equivalence classes

Valid classes

**Invalid classes**

Unique or special dates

v14—9/3/1752– 9/13/1752

v15—1/19/2038

i18—10/5/1582– 10/14/1582

Dates excluded from the Gregorian calendar
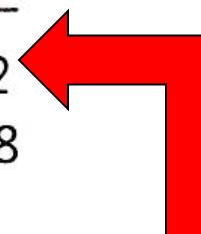
Next Date program –equivalence classes

Valid classes

Invalid classes

Unique or special dates

v14—9/3/1752– 9/13/1752

v15—1/19/2038

i18—10/5/1582– 10/14/1582

England and her colonies did not adopt the Gregorian calendar until 1752 and had to exclude 10 days to synchronize the calendar with the lunar cycle

Next Date program –equivalence classes

Next Date program –equivalence classes

Valid classes

Invalid classes

Unique or special dates

v14—9/3/1752–
        9/13/1752
v15—1/19/2038

i18—10/5/1582–
        10/14/1582

**BIOS clocks on PC – computing time 1 sec increment 32 from Jan 1, 1970**

Jan 19, 2038 3:14:08 Y 2038 the counter goes beyond max 2^{32};

Re-initialization of time to 00:00:00 date Dec 13, 1901
Store time using 64 bit representation

Valid and invalid equivalence classes for dates

January 19, 2038 UNIX Millennium Bug
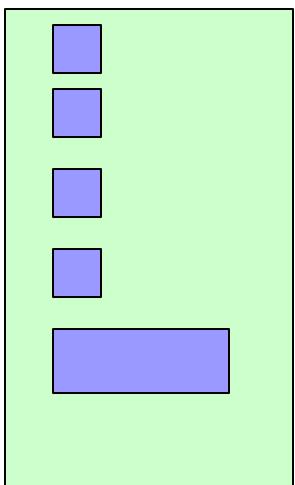
BIOS clocks on PC – computing time 1 sec increment 32 from Jan 1, 1970 [32 bit]

Jan 19, 2038 3:14:08 the counter goes beyond its maximum of 2^{32};

Re-initialization of time to 00:00:00 date Dec 13, 1901

Equivalence classes - GUI

GUI : four check boxes, one two-position field (time…)

Numeric values between 01-99 (leading zero present)

Numeric values between 1 and 9 (inclusive) with a leading and trailing space

Alphabetic symbol of one or two positions

Negative numeric values (-1… -9)

Special characters in one or two positions

Characters coded in two positions

Equivalence classes - guidelines

Valid equivalence classes

Invalid equivalence classes

Decomposition of data-equivalence classes

Four heuristics (valid and invalid classes):

Range a contiguous set of data in which any data point in the minima and maxima boundary values of the range is expected to produce the same result.

Number field where one can enter integer value from 1 to 999
valid equivalence class >=1 and <= 999
Invalid equivalence class integers <1 and >999

Group a group of items is permitted as long as these items are handled identically

Objects falling under the same taxation category

Decomposition of data-equivalence classes

Four heuristics (valid and invalid classes):

Unique data in a class or subset of a class that might be handled differently from other data in that class or a subset of a class.

Date January 19, 2038 at 3:14:07   (Friday 13)

Specific a condition must be or must not be present.

In Simple Mail Transfer Protocol (SMTP) the symbol @ is a specific character that cannot be used as part of the e-mail name or domain name.

Equivalence classes: further heuristics

Not limit to values – cover size of variables.

Test invalid equivalence classes (purchase of a negative number of items on the Internet?)

Understanding equivalence classes

What is equivalence?

Intuitive similarity
Two tests are equivalent if they are so similar to each other that it seems pointless to run both

Specified as equivalent
Two test values are equivalent if the specification says that the software handles them in the same way

Equivalence classes: the essence of testing

Systematic <u>sampling</u> approach to test design

• Cannot afford to run all tests
• Divide the population into subpopulations and test one representative of each equivalence class

Commonly used

Identification of unique defects, not combined defects

Input domain modeling

Interface-based input domain modeling

considering each input separately; almost mechanical to follow,
easy to translate into executable test cases

Functionality-based input domain modeling

Identify characteristics corresponding to the intended behavior (functionality)
of the system; more semantics involved

See textbook, Chapter 6.1

Equivalence class testing: strategies

Weak normal equivalence class testing

[*single* fault assumption]

Design a test case so that it covers one and only one invalid equivalence class and as many as possible valid equivalence classes

Strong normal equivalence class testing

[*multiple* fault assumption]

Select test cases from the Cartesian product of equivalence classes.
Covers all equivalence classes; one of each possible combinations of inputs

Combination strategies criteria (1)

terminology
block== equivalence class for each input
characteristics – inputs
Coverage criteria

All combinations coverage

All combinations of blocks from all characteristics must be used

Example.   {A, B} {1, 2, 3}   {x, y}

(A,1,x) (A,1,y) (A,2,x) (A,2,y) (A,3,x) (A,3,y)
(B,1,x) (B,1,y) (A,2,x) (B,2,y) (B,3,x) (B,3,y)

see textbook, Chapter 6.2

Combination strategies criteria (2)

terminology
block== equivalence class for each input
characteristics – inputs
Coverage criteria

Each choice coverage

One value from each block for each characteristic must be used in at least one test case

Example.   {A, B} {1, 2, 3} {x, y}
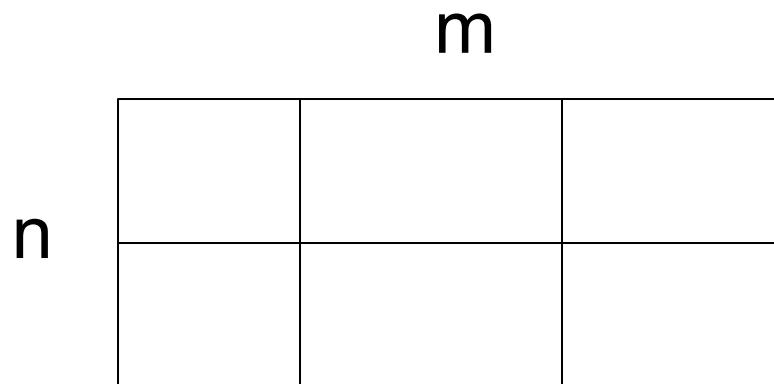
(A,1,x) (B, 2,y) (A,3, x)
many possible ways of satisfying this criterion

see textbook, Chapter 6.2

Combination strategies

Each choice coverage

m

n

Distribute tokens so each row and each column contains at least one token

Number of tokens: max (m, n)
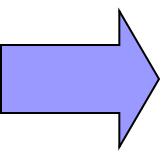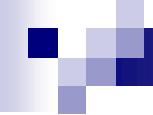
Partition?

Consider the property:

The *order of the files*
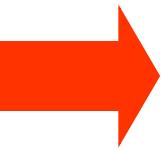
d1= files sorted in ascending order
*d2= files sorted in descending order*
d3= files in arbitrary order

Input equivalence classes (partitioning)
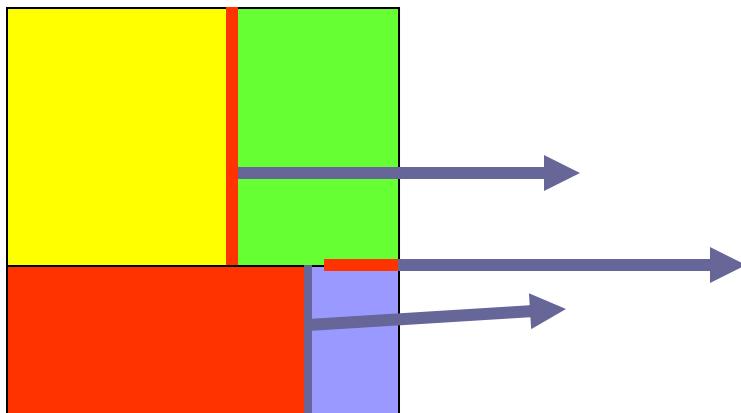
Boundary value analysis (BVA)

Boundary Value Analysis

Exploring boundary conditions (at the edges of input equivalence classes)

Input space

Input domain testing:
two categories of problems

Ambiguity **(under-defined processing) for some inputs**

Computational procedures not defined for some inputs

Contradiction **(over-defined processing) for some inputs**

Different outputs for the same input

It is observed that most of these problems occur at boundaries

Failing at a boundary?

```
INPUT < 10                          result: Error message
10 <= INPUT < 25                    result: Print "hello"
25 <=INPUT                          result: Error message
```

Faults :
(1) Program does not like numbers
(2) Inequality mis-specified (e.g., INPUT <=25 instead of < 25)
(3) Boundary value mistyped (say, transposition error, INPUT <52)
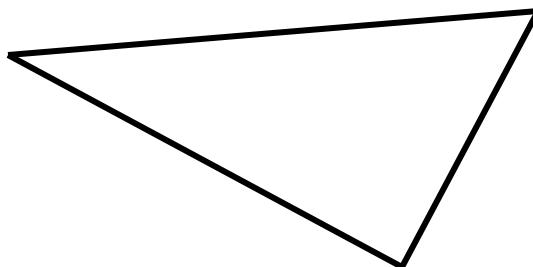
Boundary value test (25) will catch all faults
Non-boundary value (say, 53) may catch only 1 of 3 faults

Boundary Value Analysis: Examples

Triangle

Input represent triangle: integers > 0; sum of any two is greater than the third

Test cases 3-4-5   and 1-2-4

Expression improperly coded as A + B >=C instead of A + B > C

Boundary value test   1-2-3

Boundary Value Analysis: Example

`If (x >y) then S1 else S2`

Equivalence classes
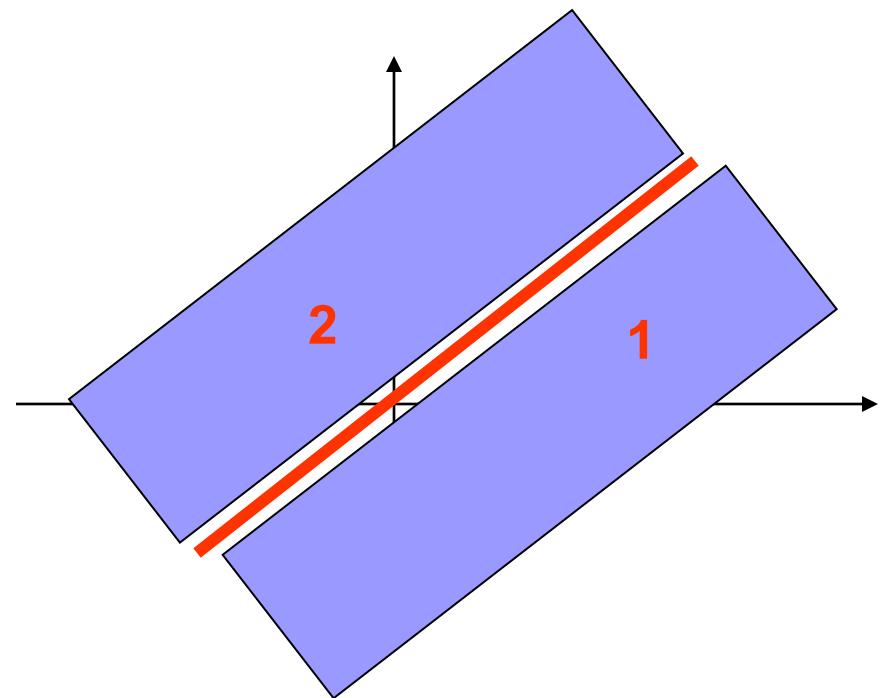
(x,y) such that x >y **1**

(x,y) such that x <=y **2**

Case x =y never selected for testing

Could be of interest:
•Implementation concerns of the equality condition
•Program realization does not meet specifications

Boundary Value Analysis: General Guidelines

If input conditions assume some range of values, write test cases for the ends of the range, invalid-input cases beyond the ends.

-1.0 --- 1.0
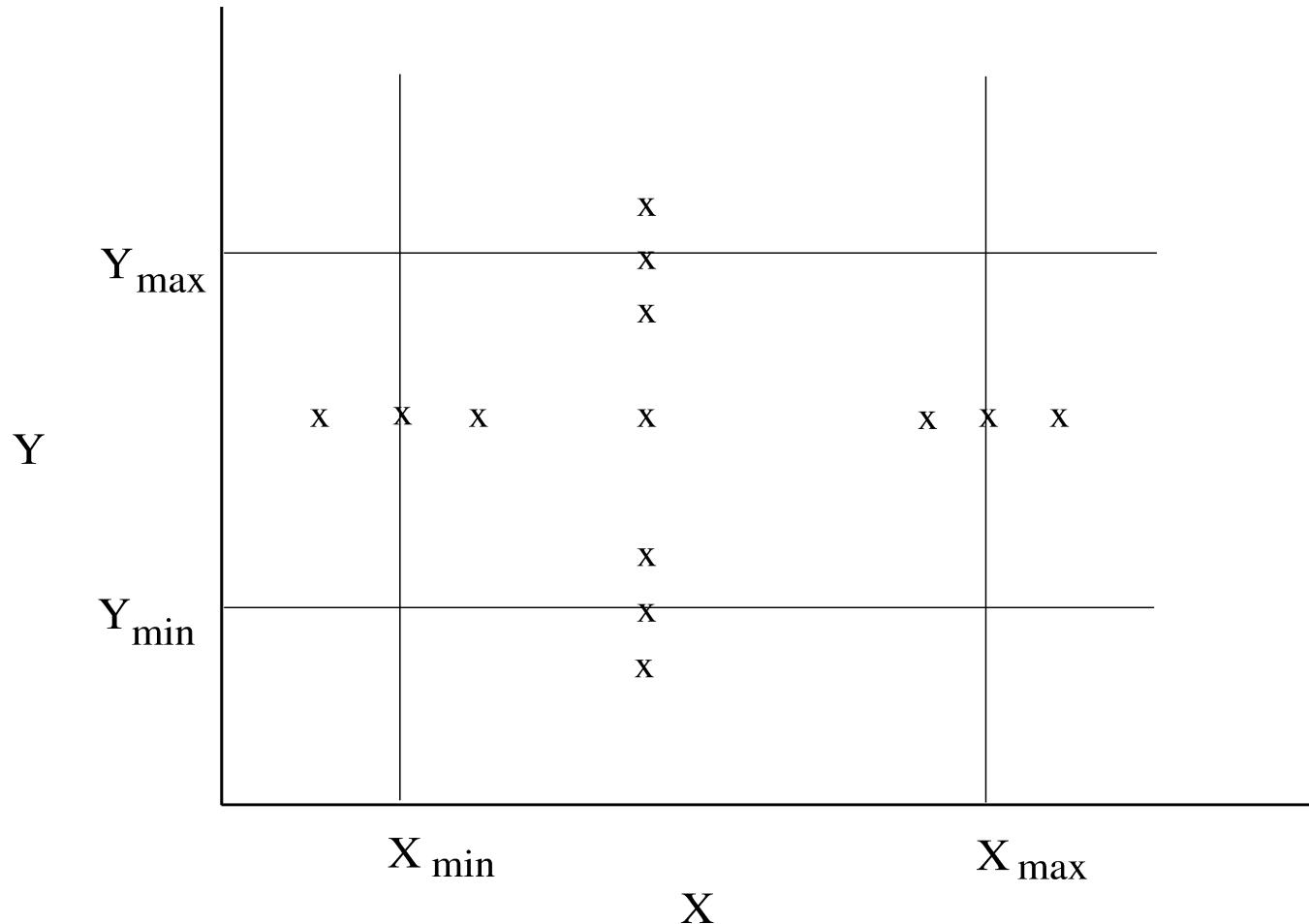
Test cases: -1.0, 1.0, 1.001, -1.001

If input conditions specify a number of values, write test cases for the min, max, and beyond these values

File 1-255 records

Test cases: 0, 1, 255, 256

Boundary Value Analysis test cases for two variables

Boundary Value Analysis (BVA)

Detection of faults:

- Incorrect artificial constraints of a data type

- Erroneously assigned relational operators

- Wrapping of data types

- Problems with looping structures

- Off-by-one errors

Equivalence partitioning versus Boundary Value Analysis (BVA)

Combinations of inputs are not explored

some high yield test could have been easily missed

Testing process