SCHULICH School of Engineering


# SENG 438
# Software Testing, Reliability & Quality

## Chapter 4: Black-box Testing

**Department of Electrical & Computer Engineering, University of Calgary**

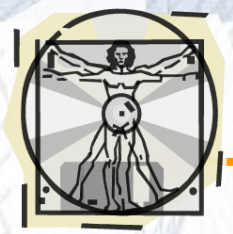
B.H. Far（far@ucalgary.ca）

http://people.ucalgary.ca/~far

# Last Week

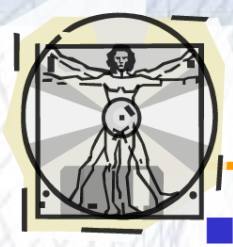
- Unit testing
  - JUnit can help with automation of test execution
  - We've learnt to write JUnit tests
- But how do we design those JUnit tests?
  - Black-box
  - White-box
- Goal is completeness of tests

**Our focus in this Chapter**



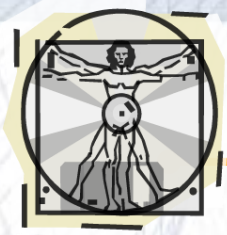
# Contents

- The need for systematic test case generation techniques
- Equivalence class partitioning techniques
- Boundary value analysis
- Decision tables
- Combinatorial testing
- Profile based testing



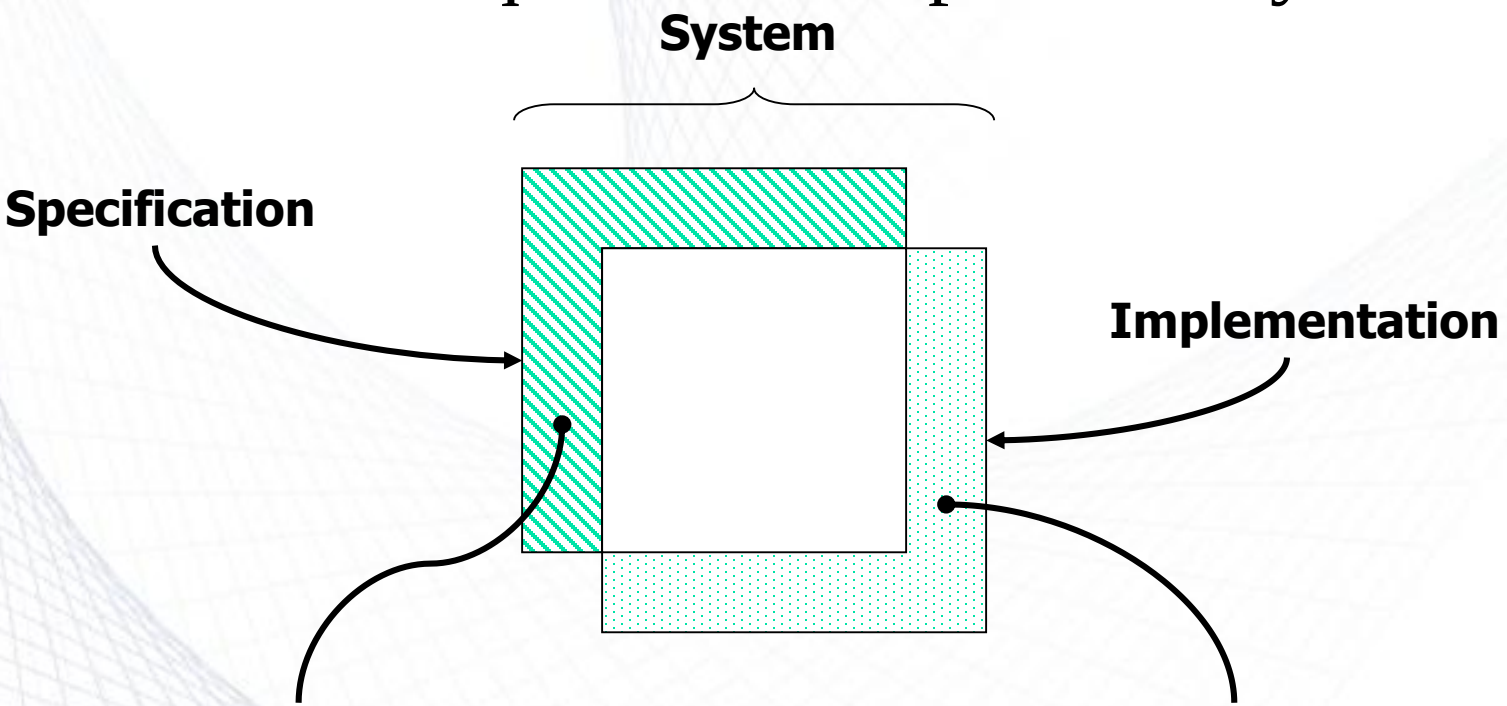
# Review: Black- vs. White-box

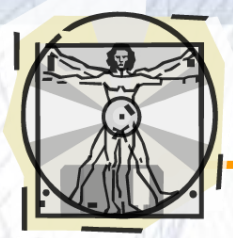
- **Black-box testing**
  - Testing, either functional or non-functional, without reference to the internal structure of the component or system
    - Synonyms: specification-based testing
- **White-box testing**
  - Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system
    - Synonyms: structural test design technique



# Review: Black- vs. White-box

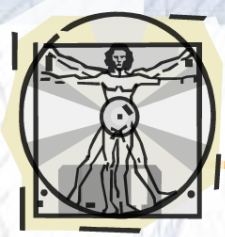
- The techniques are complementary





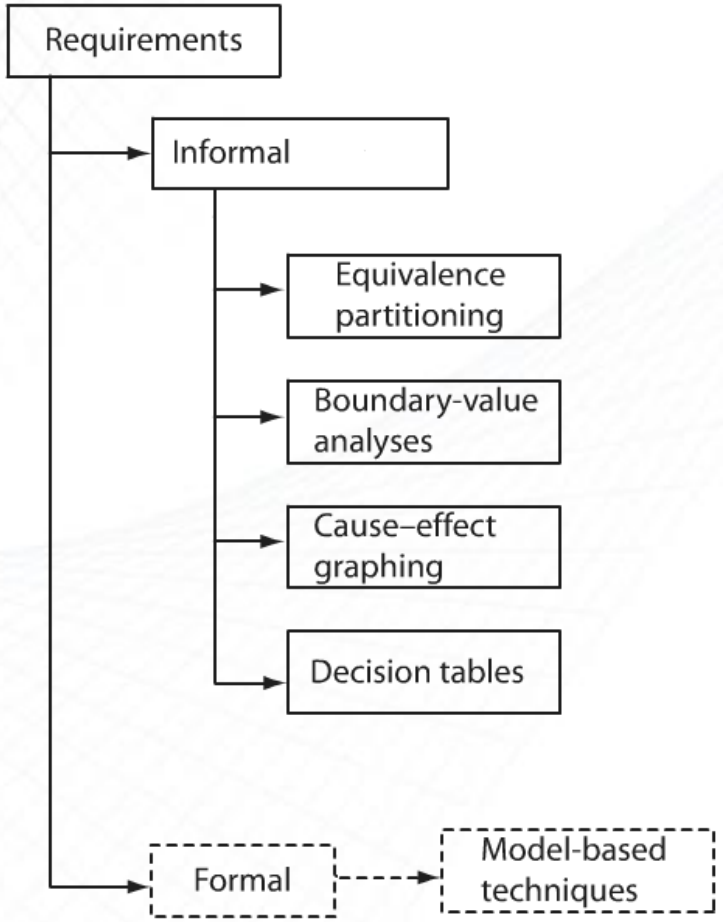
# Review: Black- vs. White-box

- BB testing applies at all granularity levels:
  - Unit (from module interface spec)
  - Integration (from API or subsystem spec)
  - System (from system requirements spec)
  - Regression (from system requirements + bug history)
- WB testing applies to relatively small parts of a system:
  - Unit
  - Integration

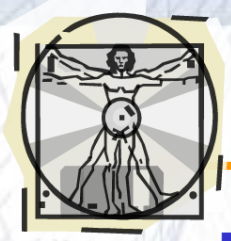


# BB Test Generation

- From system requirements
- Here our focus is on informal approaches

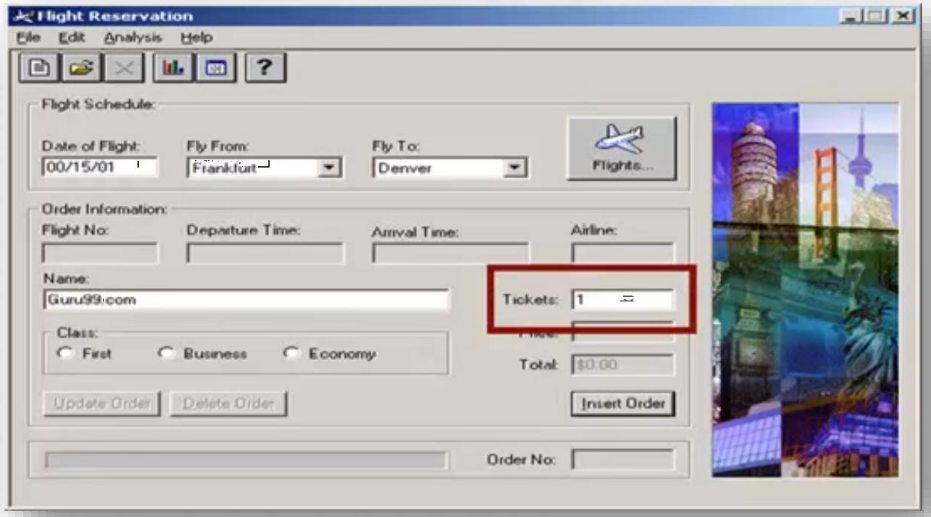



Combinatorial technique will be discussed later in this chapter



# Testing Flight Reservation App

- Between 1 and 10 tickets can be purchased (booked) in one transaction
- Value 11 to 99 are considered invalid for reservation and error message will appear, “Up to 10 tickets may be ordered at one time”
- How many test cases we need?



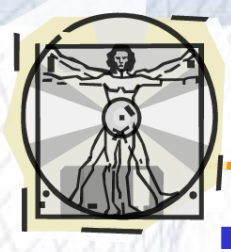
# Testing Flight Reservation App

- We would like to have a sense of complete testing based on specification of a unit
- We would like to avoid test redundancy (having several copies of test cases testing the same functionality)

- Our first solution is using **Equivalence Class Partitioning**

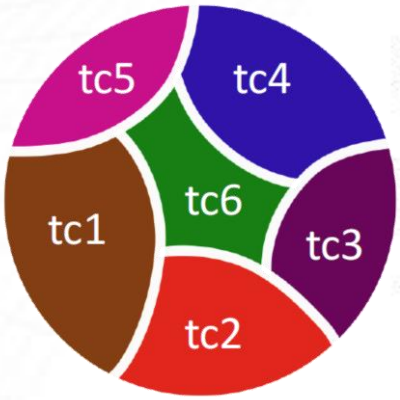
**Inputs in an "equivalence class" are assumed to be "treated the same" by the system …**

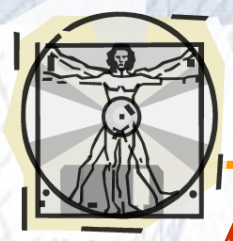
**This is the key to the definition of good equivalence classes**



# Equivalence Class Testing (ECT)

- You divide input set into partition that can be considered the same
- **Equivalence classes**: partitions of the input space in such a way that input data have the **same effect** on the SUT
  - Note the difference between "same behavior" and "same output"
- Entire input set is covered: **Completeness**
- Disjoint classes: to avoid redundancy
  - Test cases: one element of each equivalence class
  - But equivalence classes have to be chosen wisely
  - Guessing the likely system behavior is needed





# Equivalence Classes

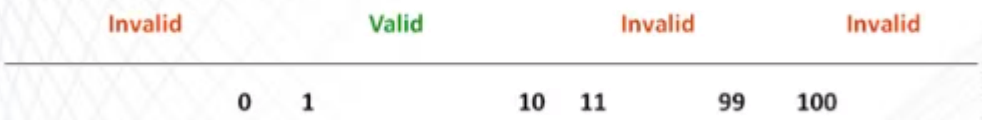
**A group of tests cases are "equivalent" if:**

- They all test the same *unit* (usually a method or a class)
- If one test case can catch a bug, the others will probably do
- If one test case does not catch a bug, the others probably will not do
- They involve the same *input variables*
- They all affect the same *output variables*

# Partitioning Example 1

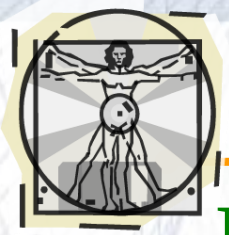
- Step 1

**Analyzing**



# Equivalence Class Testing

- **Two trivial partitions**
- The entire set of inputs to any application can be divided into at least two subsets:
  - One containing all the Expected or legal inputs (E)
  - The other containing all Unexpected (illegal) inputs (U)
- Each of the two subsets, can be further subdivided into subsets on which the application is required to behave differently (e.g., E1, E2, E3, and U1, U2)

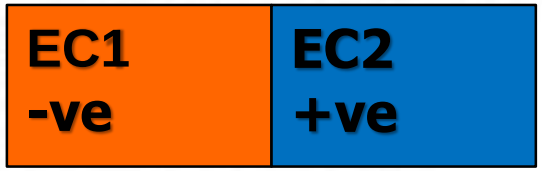


# Partitioning Example 2

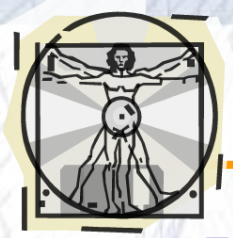
## Example: `abs(x)`

- Equivalence classes are partitions of the input set in which input data result in the same behavior of the program

```
x>=0  :   abs(x) = x
x<0   :   abs(x) = -x
```




Note: Unexpected (illegal) inputs (U) set may be much larger than the expected set, e.g. non-numbers, symbols, etc.

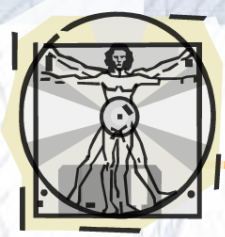


# Partitioning Example 3

- Consider an application that requires two integer inputs **x** and **y** (as its input)
- Each of these inputs is expected to lie in the following ranges: $3 \le x \le 7$ and $5 \le y \le 9$
- For **one-dimensional partitioning**, we apply the partitioning guidelines to **x** and **y** individually/independently
- This leads to the following six equivalence classes

| | | | |
|---|---|---|---|
| U1: $x<3$ | E1: $3 \le x \le 7$ | U2: $x>7$ | ⟸ y ignored |
| U3: $y<5$ | E2: $5 \le y \le 9$ | U4: $y>9$ | ⟸ x ignored |

**Nine possible combinations**



# Partitioning Example 3 (cont'd)

- **Multi-dimensional partitioning**
- Consider the input domain to be the Cartesian product **`x*y`**
- Multi-dimensional partitioning of **`3≤x≤7`** and **`5≤y≤9`** leads to 9 equivalence classes
- Which one is the legal input sub-space?

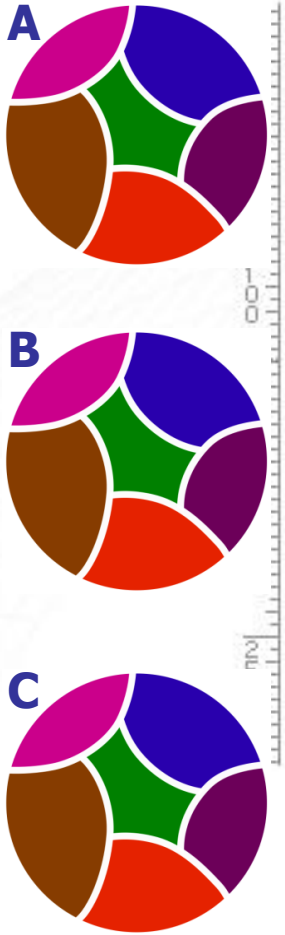
| | | |
|---|---|---|
| **U1:** x<3, y<5 | **U2:** x<3, 5≤y≤9 | **U3:** x<3, y>9 |
| **U4:** 3≤x≤7, y<5 | **E1:** 3≤x≤7, 5≤y≤9 | **U5:** 3≤x≤7, y>9 |
| **U6:** x>7, y<5 | **U7:** x>7, 5≤y≤9 | **U8:** x>7, y>9 |

**Expected or legal inputs (E)** **Unexpected (illegal) inputs (U)**

# Weak/Strong ECT

For a SUT, suppose there are three input variables from three domains: A, B, C

- $A = A_1 \cup A_2 \cup A_3 \cup \ldots \cup A_m$ where $a_i \in A_i$
- $B = B_1 \cup B_2 \cup B_3 \cup \ldots \cup B_n$ where $b_j \in B_j$
- $C = C_1 \cup C_2 \cup C_3 \cup \ldots \cup C_o$ where $c_k \in C_k$




- **Weak Equivalence Class Testing (one-dimensional):** Choosing one variable value form each equivalence class (one $a_i$, $b_j$, and $c_k$) such that all classes are covered

  **# of test cases?** max (|A|, |B|, |C|)

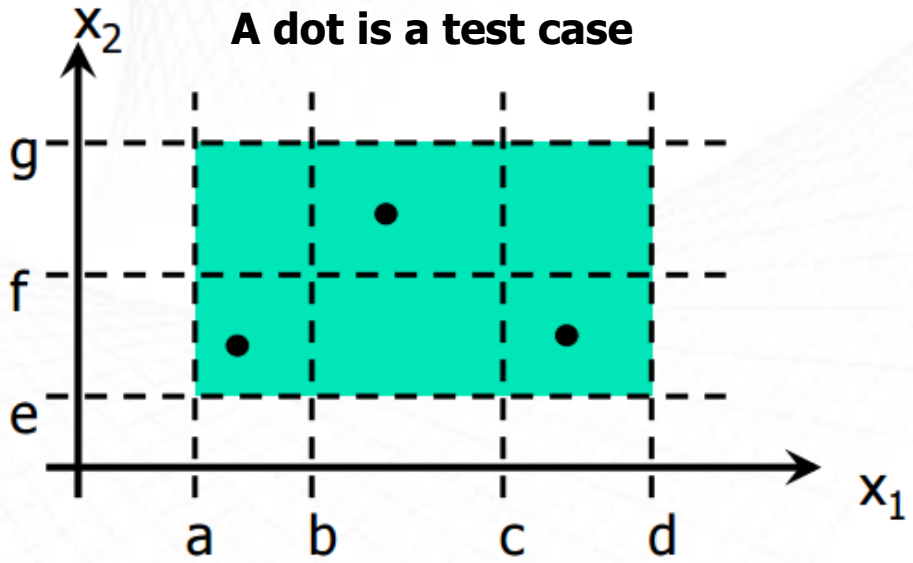
- **Strong Equivalence Class Testing (multi-dimensional):** based on the Cartesian product of the partition subsets (A×B×C), i.e., testing all interactions of all equivalence classes

  **# of test cases?** |A| x |B| x |C|

# Weak-Normal ECT

Number of test cases =

max / [[ v : 1 .. #variables • number_equivalence_classes (variable$_v$) ]]

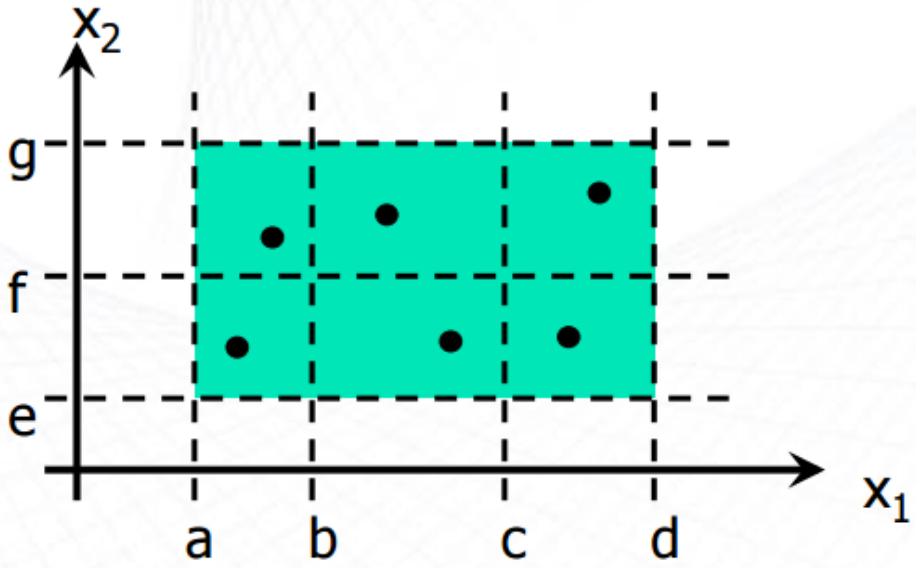



**max (3,2) = 3**

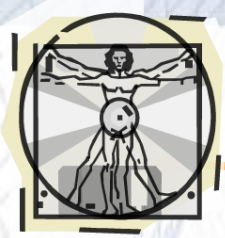
# Strong-Normal ECT

Number of test cases =

× / [[ v : 1 .. #variables • number_equivalence_classes (variable$_v$) ]]



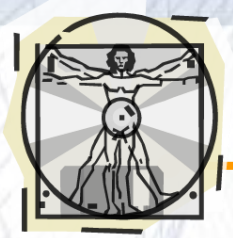

**(3x2) = 6**



# Example: Weak ECT

- Input variable A has 3 partitions
- Input variable B has 4 partitions
- Input variable C has 2 partitions
- max (3,4,2)=4 test cases are enough for WECTs

| Test Case | A | B | C |
|---|---|---|---|
| WECT1 | a1 | b1 | c1 |
| WECT2 | a2 | b2 | c2 |
| WECT3 | a3 | b3 | c1 |
| WECT4 | a1 | b4 | c2 |



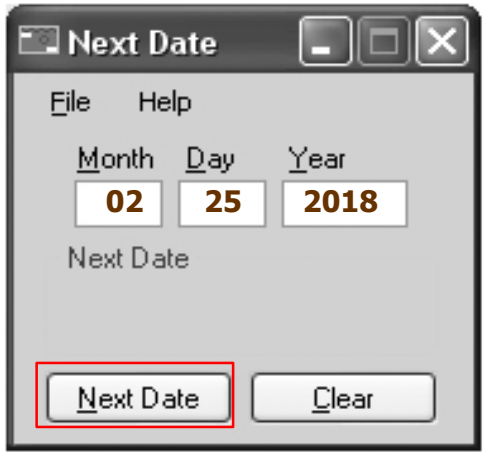
# Example: Strong ECT

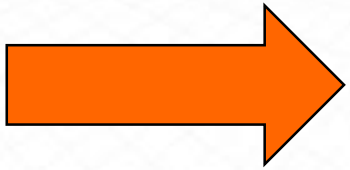
- |A| = 3
- |B| = 4
- |C| = 2
- # of test cases =3x4x2=24

| Test Case | A | B | C |
|---|---|---|---|
| SETC1 | a1 | b1 | c1 |
| SETC2 | a1 | b1 | c2 |
| SETC3 | a1 | b2 | c1 |
| SETC4 | a1 | b2 | c2 |
| SETC5 | a1 | b3 | c1 |
| SETC6 | a1 | b3 | c2 |
| SETC7 | a1 | b4 | c1 |
| SETC8 | a1 | b4 | c2 |
| SETC9 | a2 | b1 | c1 |
| SETC10 | a2 | b1 | c2 |
| SETC11 | a2 | b2 | c1 |
| SETC12 | a2 | b2 | c2 |
| SETC13 | a2 | b3 | c1 |
| SETC14 | a2 | b3 | c2 |
| SETC15 | a2 | b4 | c1 |
| SETC16 | a2 | b4 | c2 |
| SETC17 | a3 | b1 | c1 |
| SETC18 | a3 | b1 | c2 |
| SETC19 | a3 | b2 | c1 |
| SETC20 | a3 | b2 | c2 |
| SETC21 | a3 | b3 | c1 |
| SETC22 | a3 | b3 | c2 |
| SETC23 | a3 | b4 | c1 |
| SETC24 | a3 | b4 | c2 |

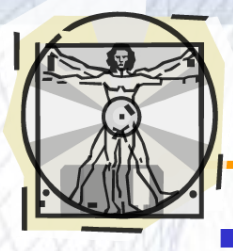
# ECT Example: `NextDate`

- **`NextDate`** is a function with three variables: **`month`**, **`day`**, **`year`**. It returns the date of the day after the input date
- e.g., if we give Feb. 25, 2018 as input, and push "Next Date" button, it will provide the output: Feb. 26, 2018
- Simplifying assumption: years between 1824-2024. The app just needs to support that range
- Note that proper treatment of leap years makes determining the last day of a month interesting!



**02 26 2018**



# ECT Example (cont'd)

- Definition of leap years: The Gregorian calendar adds a 29th day to February in all years evenly divisible by 4, except for centennial years (those ending in -00) which are not evenly divisible by 400

- Thus 1600, 2000 and 2400 are leap years, but 1700, 1800, 1900, 2100, 2200 and 2300 are not

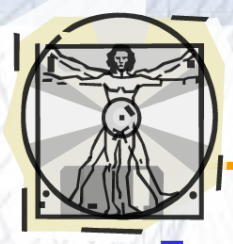
- Only the year 1900 falls in the range 1824-2024

divisible by 4, except for centennial years (those ending in -00)

Leap Years in this range:

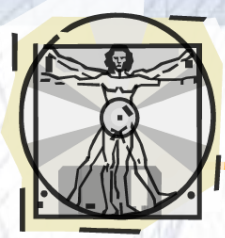
1824, 1828, 1832, 1836, 1840, 1844, 1848, 1852, 1856, 1860, 1864, 1868, 1872, 1876, 1880, 1884, 1888, 1892, 1896, 1900 **Leap Year?**
1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936, 1940, 1944, 1948, 1952, 1956, 1960, 1964, 1968, 1972, 1976, 1980, 1984, 1988, 1992, 1996, 2000, 2004, 2008, 2012, 2016, 2020, 2024 **Leap Year?**

UNIVERSITY OF CALGARY



# ECT Example (cont'd)

- NextDate Equivalence Classes
  - How many variables? 3 How may equivalent classes for each?
  - M1 = {month: month has 30 days} // Apr., Jun., Sept., Nov.
  - M2 = {month: month has 31 days} //Jan., Mar., May, Jul., Aug., Oct., Dec.
  - 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 = 1900}
  - Y2 = {year: (1824 <= year <= 2024) AND (year != 1900) AND (year mod 4 = 0)}
  - Y3 = {year: (1824 <= year <= 2024) AND (year mod 4 != 0 )}



# ECT Example (cont'd)

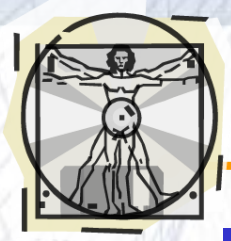
- NextDate Weak Equivalence Classes

  #WECT test cases=maximum partition size (D)

  = max(4,3,3)=4

- Suggested WECT test suite

| Test Case ID | Month | Day | Year | Output |
|---|---|---|---|---|
| WECT1 | 6 | 14 | 1900 | 6/15/1900 |
| WECT2 | 7 | 29 | 1912 | 7/30/1912 |
| WECT3 | 2 | 30 | 1913 | Invalid Input date (not possible) |
| WECT4 | 6 | 31 | 1900 | Invalid Input date |

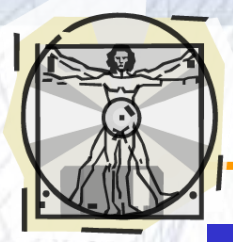


# ECT Example (cont'd)

- NextDate Strong Equivalence Classes

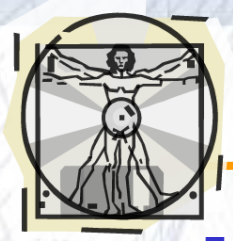
#SECT test cases= partition size (D) x partition size (M) x partition size (Y) = 3x4x3=36 test cases

| Test Case ID | Month | Day | Year | Expected Output |
|---|---|---|---|---|
| SE1 | 6 | 14 | 1900 | 6/15/1900 |
| SE2 | 6 | 14 | 1912 | 6/15/1912 |
| SE3 | 6 | 14 | 1913 | 6/15/1913 |
| SE4 | 6 | 29 | 1900 | 6/30/1900 |
| SE5 | 6 | 29 | 1912 | 6/30/1912 |
| SE6 | 6 | 29 | 1913 | 6/30/1913 |
| SE7 | 6 | 30 | 1900 | 7/1/1900 |
| SE8 | 6 | 30 | 1912 | 7/1/1912 |
| SE9 | 6 | 30 | 1913 | 7/1/1913 |
| SE10 | 6 | 31 | 1900 | ERROR |
| SE11 | 6 | 31 | 1912 | ERROR |
| SE12 | 6 | 31 | 1913 | ERROR |
| SE13 | 7 | 14 | 1900 | 7/15/1900 |
| SE14 | 7 | 14 | 1912 | 7/15/1912 |
| SE15 | 7 | 14 | 1913 | 7/15/1913 |
| SE16 | 7 | 29 | 1900 | 7/30/1900 |
| SE17 | 7 | 29 | 1912 | 7/30/1912 |
| SE18 | 7 | 29 | 1913 | 7/30/1913 |
| SE19 | 7 | 30 | 1900 | 7/31/1900 |
| SE20 | 7 | 30 | 1912 | 7/31/1912 |
| SE21 | 7 | 30 | 1913 | 7/31/1913 |
| SE22 | 7 | 31 | 1900 | 8/1/1900 |
| SE23 | 7 | 31 | 1912 | 8/1/1912 |
| SE24 | 7 | 31 | 1913 | 8/1/1913 |
| SE25 | 2 | 14 | 1900 | 2/15/1900 |
| SE26 | 2 | 14 | 1912 | 2/15/1912 |
| SE27 | 2 | 14 | 1913 | 2/15/1913 |
| SE28 | 2 | 29 | 1900 | ERROR |
| SE29 | 2 | 29 | 1912 | 3/1/1912 |
| SE30 | 2 | 29 | 1913 | ERROR |
| SE31 | 2 | 30 | 1900 | ERROR |
| SE32 | 2 | 30 | 1912 | ERROR |
| SE33 | 2 | 30 | 1913 | ERROR |
| SE34 | 2 | 31 | 1900 | ERROR |
| SE35 | 2 | 31 | 1912 | ERROR |
| SE36 | 2 | 31 | 1913 | ERROR |



# ECT Example (cont'd)

- **Strong ECT is not perfect**
- Potential issues:
  - May miss Feb 28 → Solution: make 5 classes for Days
  - Most "1900" test cases are not interesting
    - You can ignore that class → only 24 test cases rather than 36
    - And perhaps add one extra edge case (end of year for 1900) to the suite?
- Adding two invalid classes for each variable = 150 equivalence class test cases!



# Example 2

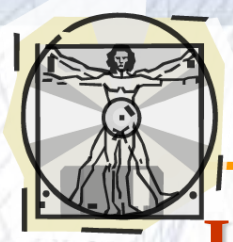
- We have developed a program that returns the two roots of a quadratic equation $ax^2+bx+c=0$
- Java code for this program is given
- The roots are calculated as:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Let's assume that a, b and c can take values between -10 and +10
- What are equivalent classes for a, b and c?
- What is the number of test cases for
  - Weak ECT
  - Strong ECT

```
class Roots {
    double root_one, root_two;
    int num_roots;
    public roots(double a, double b, double c) {
        double q;
        double r;
        q = b*b - 4*a*c;
          if (q>0) {      // If b^2 > 4ac, there are two distinct roots
                num_roots = 2;
                r= (double) Math.sqrt(q) ;
                root_one = ((0 - b) + r)/(2*a);
                root_two = ((0 - b) - r)/(2*a);
          } else if (q==0) {
                //The equation has one root only
                num_roots = 1;
                root_one = (0 - b)/(2*a);
                root_two = root_one;
          } else {   // The equation has no roots if b^2 <4ac
                num_roots = 0;
                root_one = -1;
                root_two = -1;
          }
        }
        public int num_roots() { return num_roots; }
        public double first_root() {return root_one; }
        public double second_root() { return root_two; }
}
```

Bug here?



# Equivalence Classes /1

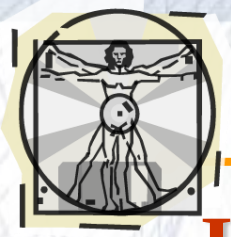
## Hint (1):

- Don't forget equivalence classes for *invalid inputs*

**Example:** For a program that is supposed to accept any number between 1 and 99

There are 4 equivalence classes:

- Any number between 1 and 99 (valid)
- Any number less than 1 (including zero and negative numbers) (invalid)
- Any number greater that 99 (invalid)
- Any non-number (invalid)



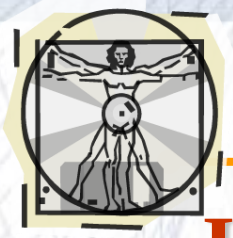
# Equivalence Classes /2

## Hint (2):

- Look for *extreme range* of variables

**Example:** For a program that is supposed to accept any integer between 1 and 99

- Try very large values for the number (check if both 16 bit or 32 bit integers work)
- Try very small negative values to check if sign can be handled properly



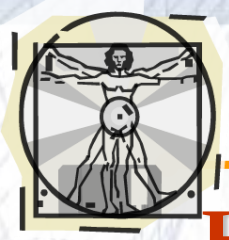
# Equivalence Classes /3

## Hint (3):

- Look for *maximum size of memory* (stackable) variables

**Example:** For a depth-first (breath-first) graph search operation

- Set the length of the path (number of the child nodes) to be very long (to be very high) and check for memory overflow



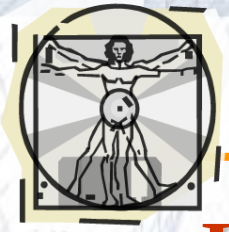
# Equivalence Classes /4

## Hint (4):

- If an input must belong to a group, one equivalence class must include all members of the group

**Example:** For a program that accepts users names as its input

- All strings of alphabet starting with a capital or lower case letters are acceptable
- Limit the size of the string to one character or set it to be very long and check if both work
- Check for non-ascii characters, if needed



# Equivalence Classes /5

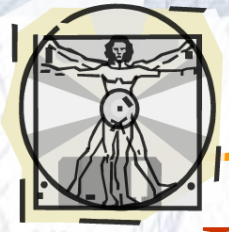
## Hint (5):

- Analyze *levels for binary value variables*

**Example:** For a program that asks

"Are you sure? (Y/N)

- One equivalence class contains "Y" (and may be "y" and "Yes" and "yes")
- The other class contains "N" (and may be "n" and "No" and "no") and anything else other than "Y"

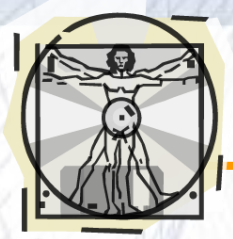


# Equivalence Classes /6

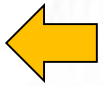
## Hint (6):

- Look for *dependent variables* and replace them by their equivalents

**Example:** For a program that accepts three angles of a triangle as its input only two of them are independent



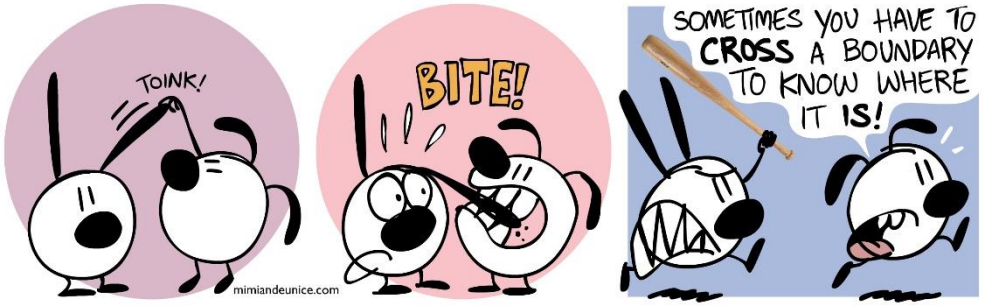
# ECT: Summary

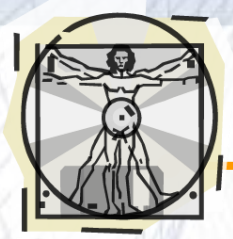
- Decide how many independent variables you have
- For each, how many distinct partitions (ranges) for each variable you have
- Select number of test cases based on weak, strong, or robustness criteria  **Coming next**
- Create test cases using acceptable values assigned from each distinct partitions for each variable involved
- Review test cases: remove redundant, add tests for problems perceived, etc.
- Repeat above steps until you are satisfied with the test cases


UNIVERSITY OF CALGARY

# Section 2

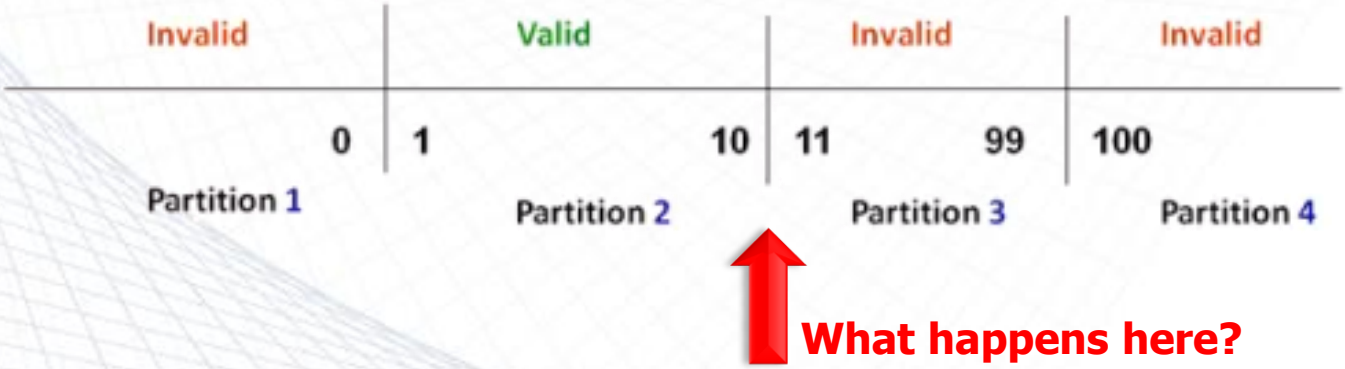
## Boundary Value Testing

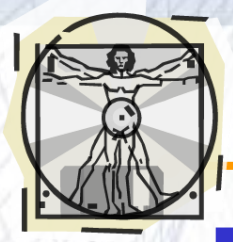




# Boundary Value Testing (BVT)

- **Motivation:** Experience indicates that, sometimes, programmers make mistakes in processing values at or near the boundaries of equivalence classes
- Simpler but complementary to previous techniques
- Example: Flight reservation app



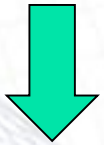


# Error at the Boundaries

- Programmers make mistakes in **processing values at and near the boundaries** of equivalence classes

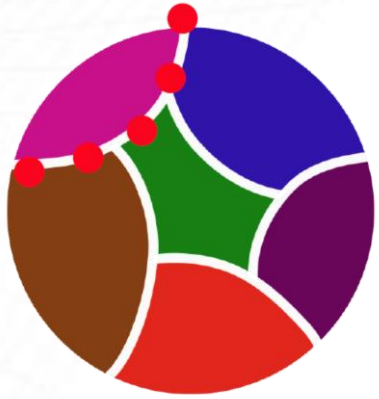
```
If (x<0)                    //Should have been If (x<=0)
        M1();
Else
        M2();
```

Equivalent Classes = (x<0 ; x>=0)

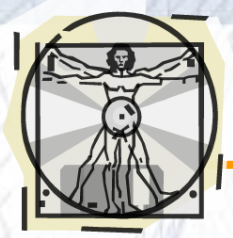


Tests = (x=-4, x=7)

**Only x=0 (the boundary case) can reveal the bug!**

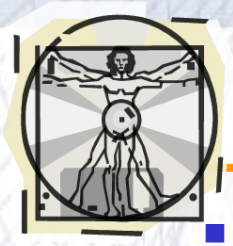


**In this example, the value x=0, lies at the boundary of the equivalence classes x≤0 and x>0**



# Boundary Value Analysis

- While equivalence partitioning selects tests from within equivalence classes, boundary value analysis focuses on tests **at and near** the boundaries of equivalence classes
- Tests derived using either of the two techniques may **overlap**
- **Boundary Value Testing (BVT) Approach:** Select tests at or near boundary values

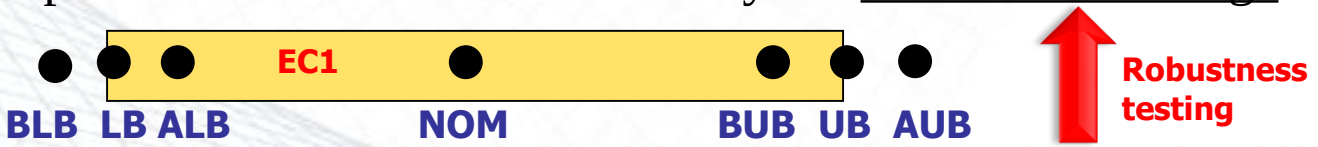


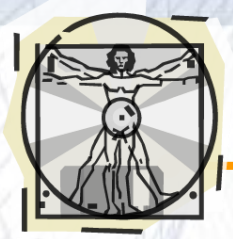
# BVT: How to

- For each equivalence class, setting values for input variable just below their min, at their min, just above the min, a nominal value, just below their max, at their max and just above their max
- A typical strategy for all input variables: Holding the values of all but one variable at their nominal values, letting one variable take each of the above combinations

| | |
|---|---|
| BLB | : a value just below the lower bound |
| LB | : the value on the lower boundary |
| ALB | : a value just above the lower boundary |
| NOM | : a nominal value |
| BUB | : a value just below the upper bound |
| UB | : the value on the upper bound |
| AUB | : a value just above the upper bound |

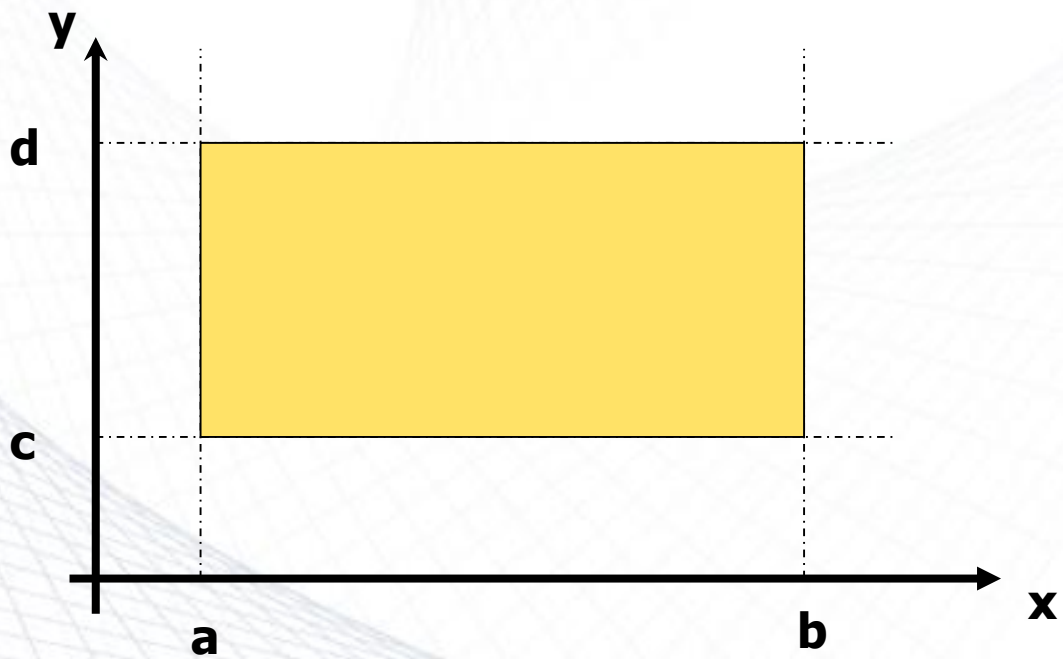
- We can drop the BLB and AUB because they are <u>out of the EC range</u>





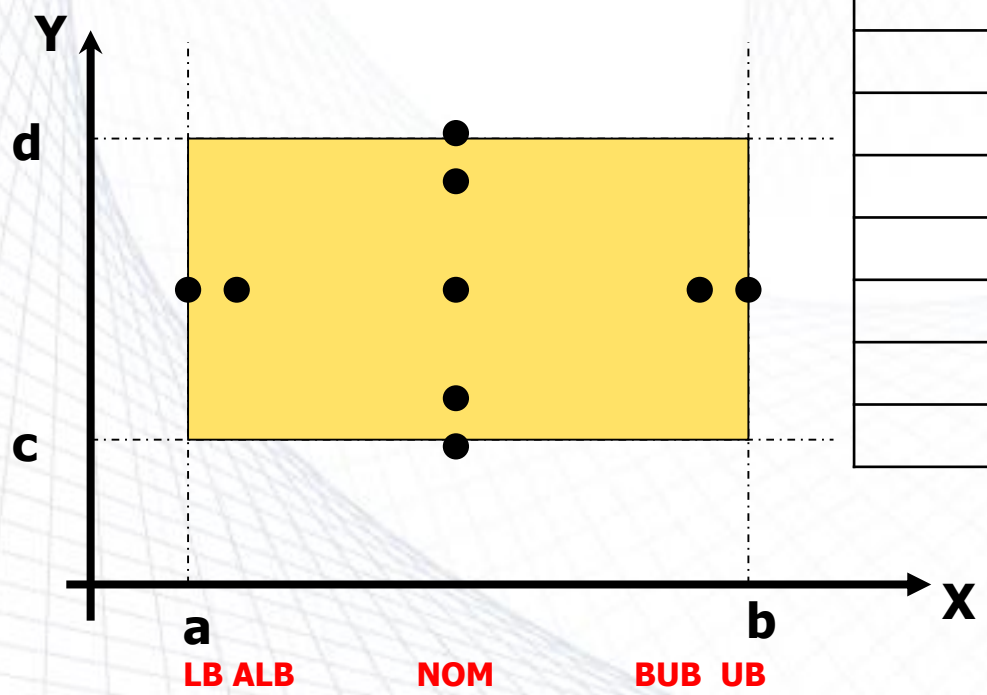
# BVT: Example

- Assume a function F, with two variables x and y, domains X and Y
- Boundaries: a <= x <= b, c <= y<= d

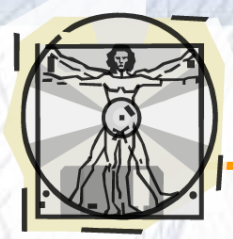


# BVT: Example (cont'd)

- Number of test cases=9




| Test cases (TC) | x | y |
|---|---|---|
| 1 | x_NOM | y_LB |
| 2 | x_NOM | y_ALB |
| 3 | x_NOM | y_NOM |
| 4 | x_NOM | y_BUB |
| 5 | x_NOM | y_UB |
| 6 | x_LB | y_NOM |
| 7 | x_ALB | y_NOM |
| 8 | x_BUB | y_NOM |
| 9 | x_UB | y_NOM |



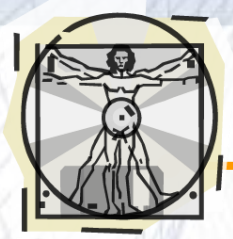
# BVT: Limitations

- How many test cases is required for a function with **n** variables?

  **Total 4n + 1 test cases**

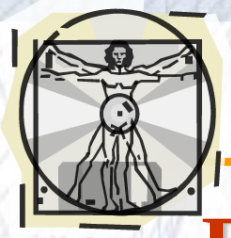
- **Works well** with variables that **represent bounded physical quantities**
- **No consideration of the nature of the function** and the meaning of variables
- Simple technique that is amenable to robustness testing

Later



# Boundary Value Hints

- Check if the boundary conditions for variables are set correctly
- Check if the inequality boundary conditions can be changed to equality or not
- Check if the *counter* variables allow departure from the loop correctly or not



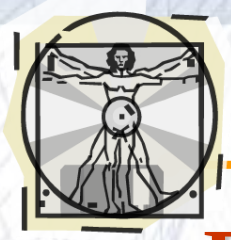
# Boundary Value Hints /1

## Example (1):

- If the program expects an upper case letter A-Z, Test @ (ASCII code just below the code for A) and [ (character just beyond Z)

## Example (2):

- If the program needs a specific number of inputs, give it that many, one more and one fewer



# Boundary Value Hints /2

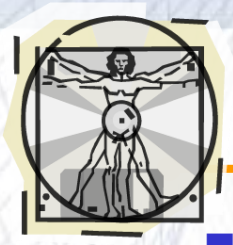
## Example (3):

- If a counter is set to have values:

  0<i< 10, check if 0 and 10 are correct boundary values

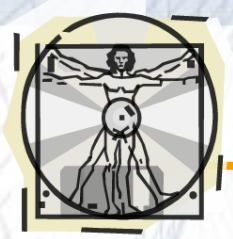
## Example (4):

- When reading from or writing to a disk file, check the first and last character in the file



# Example

- Suppose we are testing a software module of an editor system that allows a user to enter new font identifiers into a font database. The input specification for the module states that a font identifier should
  - (i) consist of alphanumeric characters;
  - (ii) the range for the total number of characters is between 3 and 15; and,
  - (iii) the first two characters must be letters
- When preparing test cases we will focus on selecting “equivalence classes” and “boundary values” for the inputs



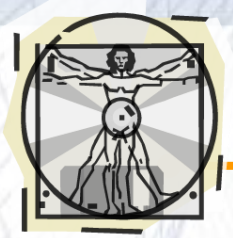
# Example (cont'd)

- Identify input equivalence classes for this example

**Spec. 1:**
**C1. Font name is alphanumeric, valid**
**C2. Font name is not alphanumeric, invalid**

**Spec. 2:**
**C3. Font identifier has between 3 and 15 characters, valid**
**C4. Font identifier has less than 3 characters, invalid**
**C5. Font identifier has greater than 15 characters, invalid**

**Spec. 3:**
**C6. The first 2 characters are letters, valid**
**C7. The first 2 characters are not letters, invalid**



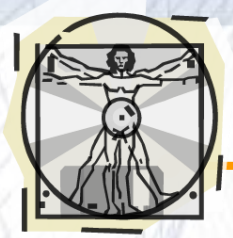
# Example (cont'd)

- Identify boundary values for this example

**BLB : a value just below the lower bound**
**LB : the value on the lower boundary**
**ALB : a value just above the lower boundary**
**NOM : a nominal value**
**BUB : a value just below the upper bound**
**UB : the value on the upper bound**
**AUB : a value just above the upper bound**

**For our example module the values for the bounds groups are:**

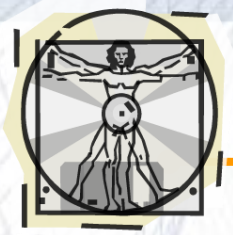
**BLB — 2** **BUB — 14**
**LB — 3** **UB — 15**
**ALB — 4** **AUB — 16**
**NOM — anything between 4 and 14**



# Example (cont'd)

- Typical set of test cases for this module based on the equivalence classes and boundary values

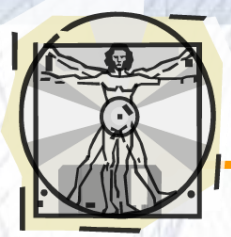
| Case | Input values | Relevant Conditions | Pass/Fail criteria |
|---|---|---|---|
| 1 | abc1 | C1; C3; C6; NOM | Pass |
| 2 | ab1 | C1; C3; C6; LB | Pass |
| 3 | abcdef123456789 | C1; C3; C6; UB | Pass |
| 4 | abcde123456789 | C1; C3; C6; BUB | Pass |
| 5 | 12* | C7 | Fail |
| 6 | ¥©* | C2 | Fail |
| 7 | ab | C4 | Fail |
| 8 | abcdedgh1234567890 | C5 | Fail |



# BB Testing Variations

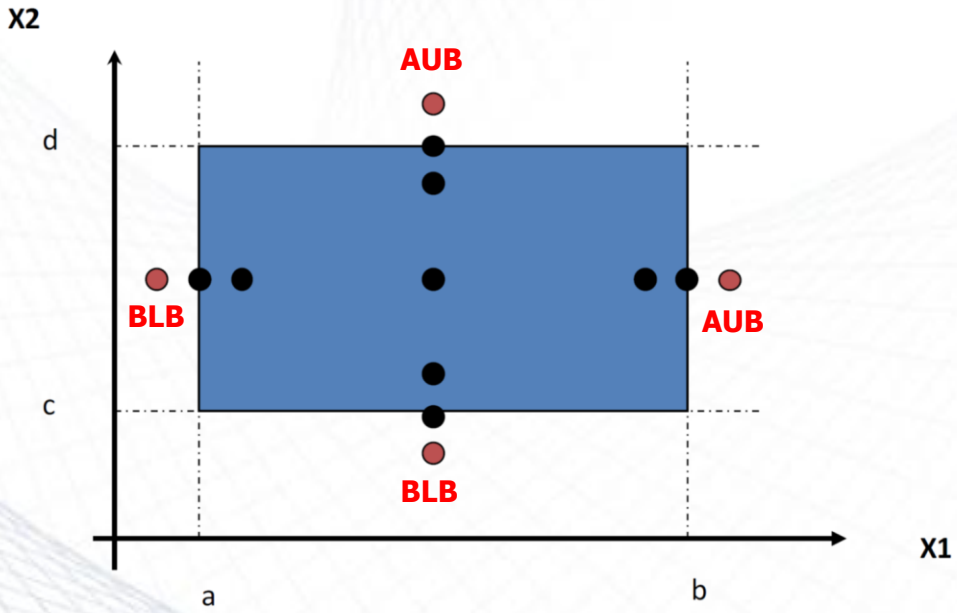
- Robustness testing
- Worst case testing
- Category-partition testing
- Decision tables
- …

- Model-based testing  **Will be discussed later**



# Robustness Testing

- In Robustness testing the BLB and AUB are also included in test cases



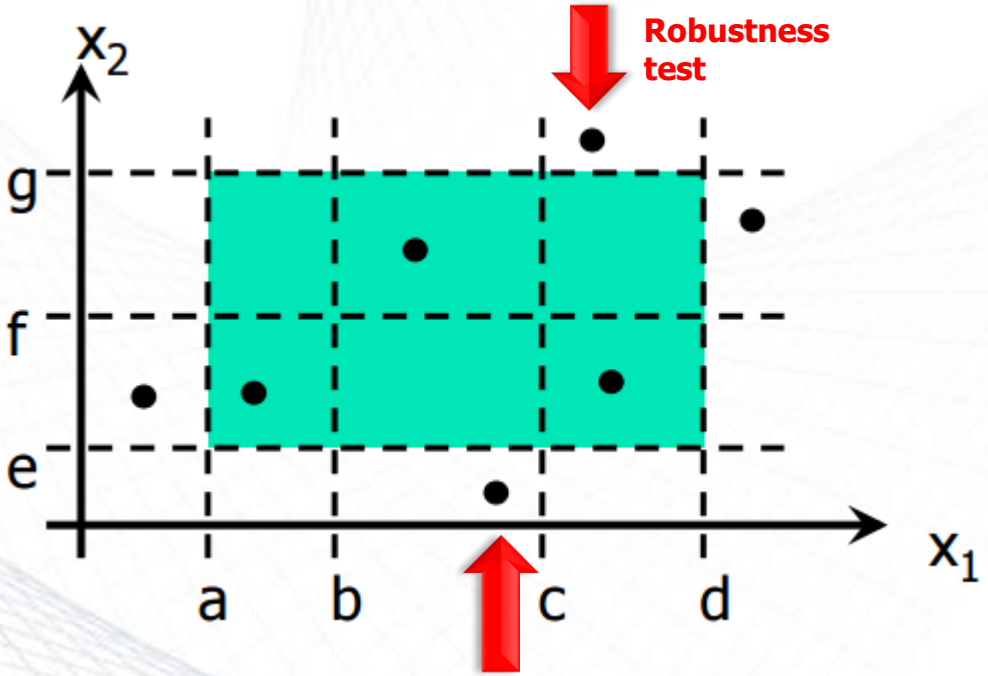
# Weak-Robust ECT

Number of test cases =

max / [[ v : 1 .. #variables • number_equivalence_classes (variable$_v$) ]]

+

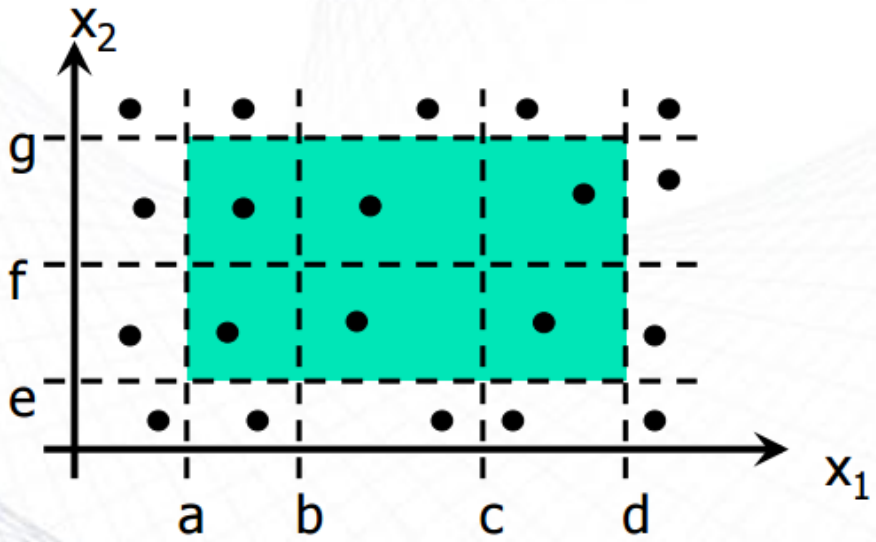
+/ [[ v : 1 .. #variables • number_invalid_bounds (variable$_v$) ]]

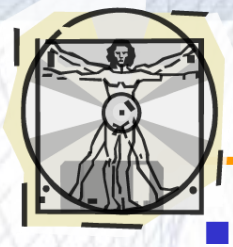


# Strong-Robust ECT

Number of test cases =

$$\times / [[\ v : 1 .. \#variables \bullet \quad number\_equivalence\_classes\ (variable_v) + number\_invalid\_bounds\ (variable_v)\ ]]$$

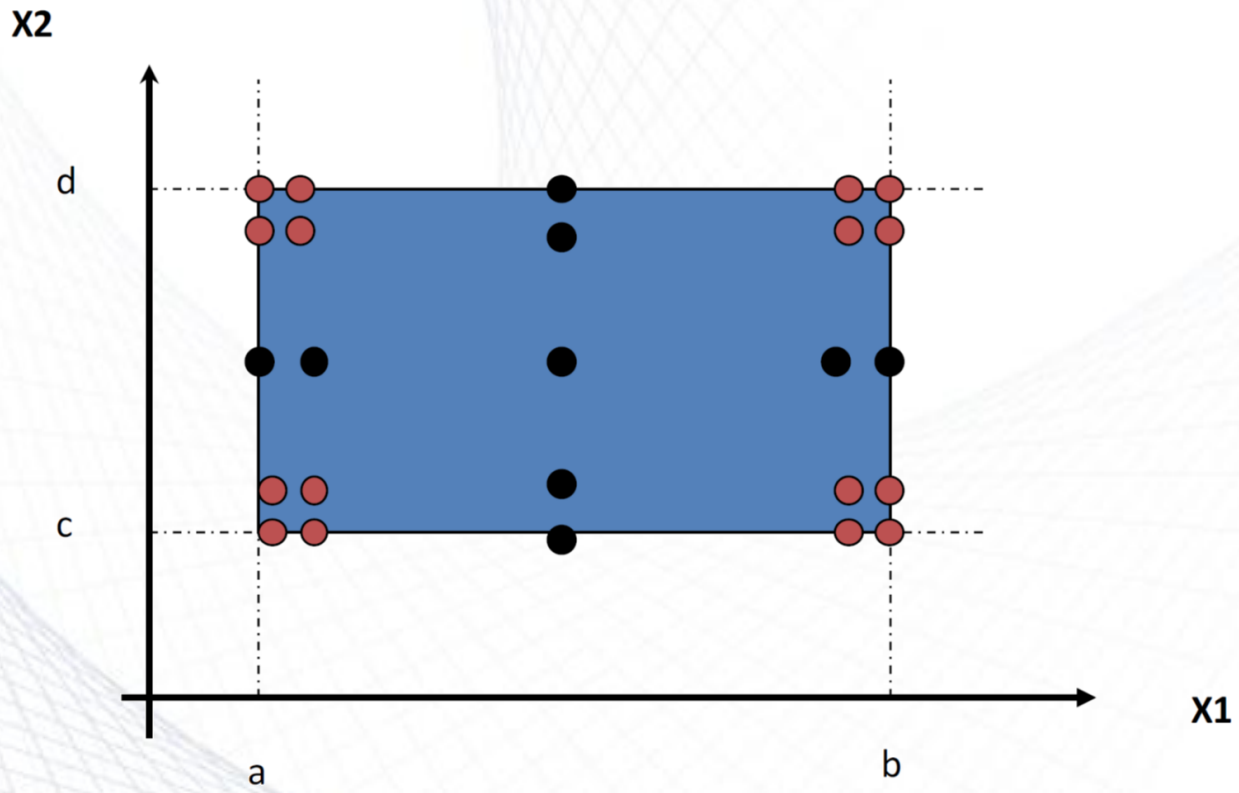




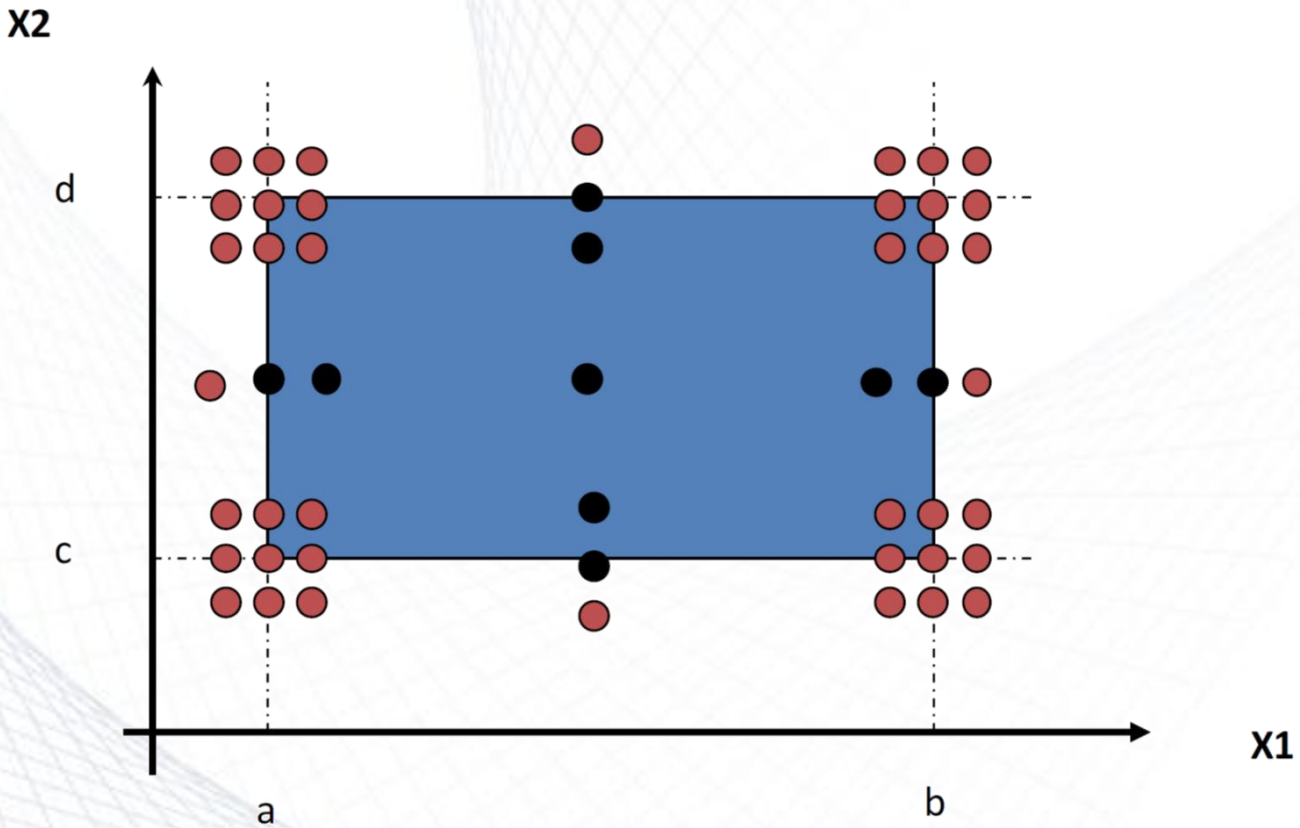
# Worst Case Testing (WCT)

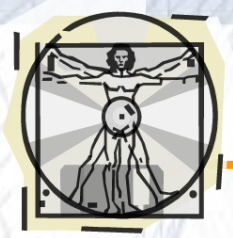
- Boundary value analysis makes the common assumption that failures, **most of the time**, originate from one fault related to **an extreme value**
- What happens when **more than one variable** has an extreme value?
  - Idea comes from electronics in circuit analysis
- Cartesian product of {LB, ALB, NOM, BUB, UB}
- Clearly **more thorough** than boundary value analysis, but **much more effort. How much?**
  - **5**n test cases**
- Good strategy when physical variables have numerous interactions, and where failure is costly

# WCT for 2 Variables



# Robust WCT for 2 Variables





# Category-Partition Testing

- Makes it easier to define special case categories
- Combines boundary value analysis and equivalent classes with **domain expertise**
- You can define constraints as well

- **Example:** We have a text processing software product (T), developed totally within our software house. It must work with operating systems 1 and 2 and printer systems A, B, C, and D. What systems should we test?

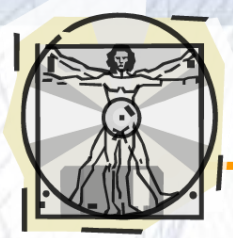
  Systems that represent all combinations of operating systems and printer systems with the text processing product: 1AT, 1BT, 1CT, 1DT, 2AT, 2BT, 2CT, 2DT and the product T itself. Some combinations may be illegal (i.e. can be ignored using domain expertise)

SCHULICH School of Engineering


UNIVERSITY OF CALGARY

# Section 3

## Testing Using Decision Tables



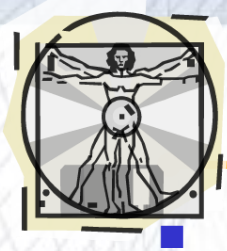
# Decision Tables

- Can help us deal with combination of inputs which produce different results

| Conditions | Rule 1 | Rule 2 | Rule 3 | Rule 4 |
|---|---|---|---|---|
| FLY FROM | F | T | F | T |
| FLY TO | F | F | T | T |
| ***OUTCOME*** FLIGHTS BUTTON | F | F | F | T |

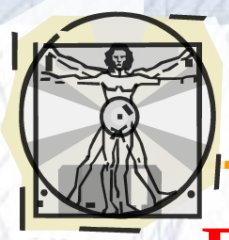

**"FLY FROM" and "FLY TO" should be selected to allow the "flight button" become active**

**Some kind of acceptance test**



# Testing Using Decision Table

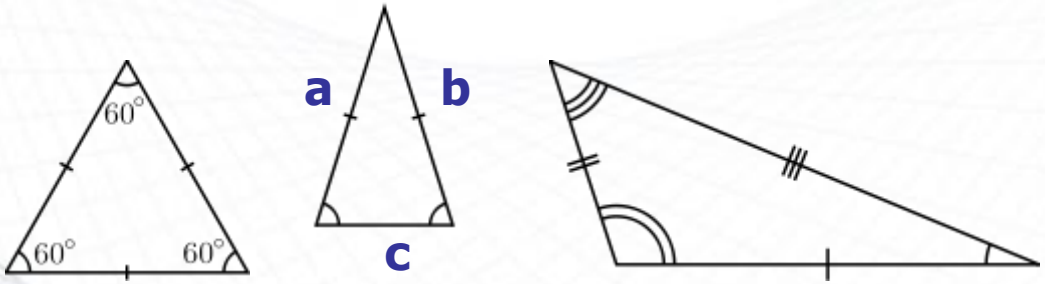
- The significance of this technique becomes immediately clear as the number of inputs increases
- Number of possible combinations is given by 2**n , where n is the number of inputs
- For n = 10, which is very common in the web based testing, having a typical fill form, the number of combinations will be 1,024
- Obviously, some combinations are meaningless and/or you cannot test all but you will choose a rich sub-set of the possible combinations using decision based testing technique



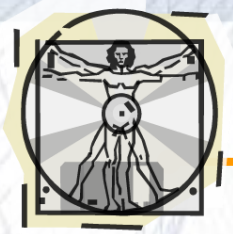
# Testing Using Decision Table

## Example

- Let's design a decision table for a program which needs to identify the type of a triangle given its edge (side) sizes



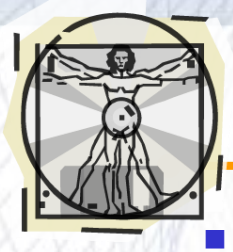

Equilateral Isosceles Scalene



# Testing Using Decision Table

- To help express *test requirements* in a directly usable form
- Easy to understand and support the systematic and automated and derivation of test cases
- Ideal for describing **situations** in which a number of combinations of **actions** are taken under varying sets of conditions, e.g., control systems

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **c1: a, b, c triangle?** | N | Y | Y | Y | Y | N | N | N |
| **c2: a = b?** | – | Y | Y | N | N | Y | Y | N |
| **c3: a = c?** | – | Y | N | Y | N | Y | N | Y |
| **c4: b = c?** | – | Y | N | N | N | N | N | N |
| **a1: not a triangle?** | X | | | | | | | |
| **a2: Scalene** | | | | | | | | X |
| **a3: Isosceles** | | | | | X | | X | X |
| **a4: Equilateral** | | X | | | | | | |
| **a5: Impossible** | | | X | X | | X | | |



# Decision Tables - Structure

- There is a Condition section which lists *conditions* and their combinations
- Conditions express relationships among *decision variables*
- There is an Action section which lists *responses* to be produced when corresponding combinations of conditions are true
- There will one test case for each test rule from the decision table

**Conditions**

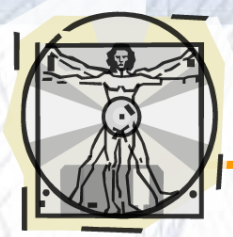
**Decisions**

**Actions**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| c1: a, b, c triangle? | N | Y | | | | | | | | |
| c2: a = b? | – | | Y | | | | N | | | |
| c3: a = c? | – | | | Y | | N | | | | |
| c4: b = c? | – | Y | N | Y | N | Y | N | Y | N | |
| a1: not a triangle? | X | | | | | | | | | |
| a2: Scalene | | | | | | | | | | X |
| a3: Isosceles | | | | | X | | X | X | | |
| a4: Equilateral | | X | | | | | | | | |
| a5: Impossible | | | X | X | | X | | | | |

**rule**

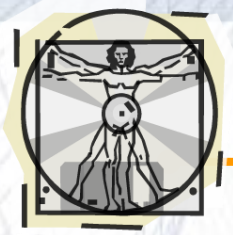
**Q. Are we missing conditions a+b >c, etc. ?**



# Testing Using Decision Table

- Converting decision tables to conventional truth table
- If the condition block in our decision table has common Y/N (T/F) for many conditions, it is better to convert it to a conventional truth table
- The triangle example, in the truth table, a "-" means "don't care"

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| c1: a, b, c triangle? | N | Y | | | | Y | | | |
| c2: a = b? | – | | | Y | | | | N | |
| c3: a = c? | – | Y | | | N | | Y | | N |
| c4: b = c? | – | Y | N | Y | N | Y | N | Y | N |
| a1: not a triangle? | X | | | | | | | | |
| a2: Scalene | | | | | | | | | X |
| a3: Isosceles | | | | | X | | X | X | |
| a4: Equilateral | | X | | | | | | | |
| a5: Impossible | | | X | X | | X | | | |



# Testing Using Decision Table

| 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 |
| a1: Not a triangle | X | X | X | | | | | | | | |
| a2: Scalene | | | | | | | | | | | X |
| a3: Isosceles | | | | | | | X | | X | X | |
| a4: Equilateral | | | | X | | | | | | | |
| a5: Impossible | | | | | X (?) | X (?) | | X (?) | | | |

A test rule (test cases)

11 test cases...

Test cases...

- One test case for each test rule from the corresponding truth table
- "?": cannot provide test data (i.e., the test case is impossible to execute)

# Example: Decision Table

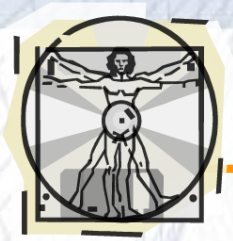
- Bank account transaction processing
  - Conditions (total 4):
    - Does the named person hold the credit card entered, and is the other information correct?
    - Is it still active or has it been cancelled?
    - Is the person within or over their limit?
    - Is the transaction coming from a normal or a suspicious location?
  - Actions:
    - Should we approve the transaction?
    - Should we call the cardholder (e.g., to warn them about a purchase from a strange place)?
    - Should we call the vendor (e.g., to ask them to seize the cancelled card)?

# Example: Decision Table

- Decision table for 4 conditions (2**4=16 cases)

| Conditions | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Real account? | Y | Y | Y | Y | Y | Y | Y | Y | N | N | N | N | N | N | N | N |
| Active account? | Y | Y | Y | Y | N | N | N | N | Y | Y | Y | Y | N | N | N | N |
| Within limit? | Y | Y | N | N | Y | Y | N | N | Y | Y | N | N | Y | Y | N | N |
| Location okay? | Y | N | Y | N | Y | N | Y | N | Y | N | Y | N | Y | N | Y | N |
| Actions | | | | | | | | | | | | | | | | |
| Approve? | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| Call cardholder? | N | Y | Y | Y | N | Y | Y | Y | N | N | N | N | N | N | N | N |
| Call vendor? | N | N | N | N | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |

Collapse



# Example: Decision Table

- Collapsing decision table: combining columns
- How?

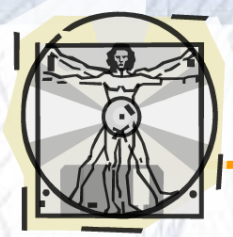
| Conditions | 1 | 2 | 3 | 5 | 6 | 7 | 9 |
|---|---|---|---|---|---|---|---|
| Real account? | Y | Y | Y | Y | Y | Y | N |
| Active account? | Y | Y | Y | N | N | N | - |
| Within limit? | Y | Y | N | Y | Y | N | - |
| Location okay? | Y | N | - | Y | N | - | - |
| Actions | | | | | | | |
| Approve? | Y | N | N | N | N | N | N |
| Call cardholder? | N | Y | Y | N | Y | Y | N |
| Call vendor? | N | N | N | Y | Y | Y | Y |

SCHULICH School of Engineering


UNIVERSITY OF CALGARY

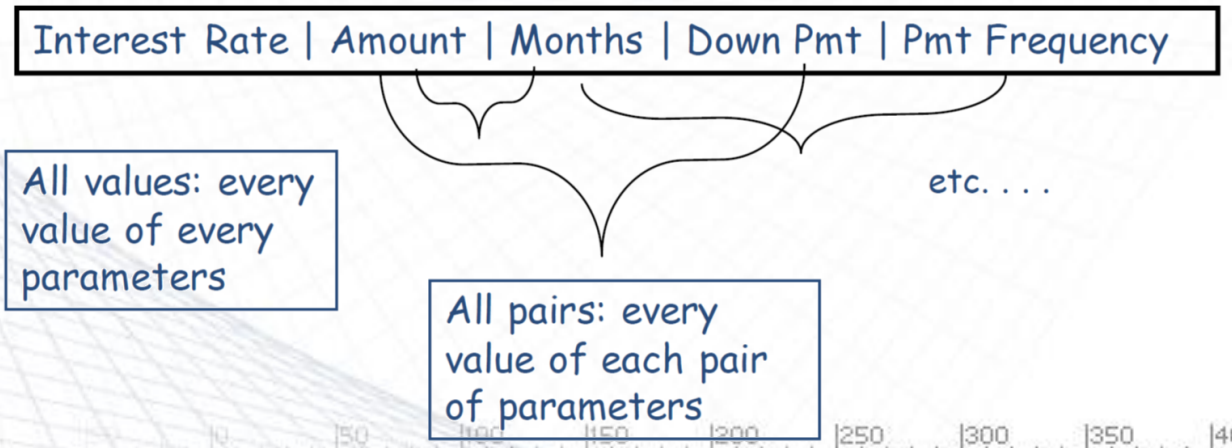
# Section 4

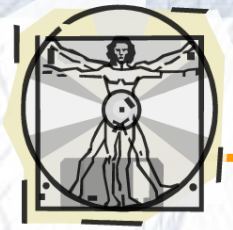
## Combinatorial Testing



# Combinatorial Testing

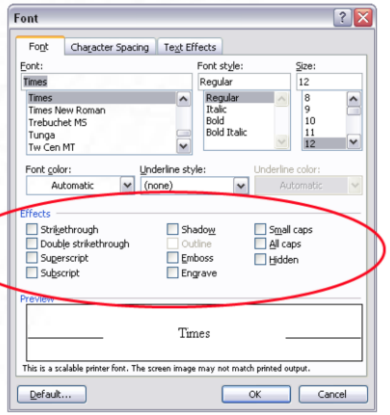
- Assumption: a main cause of software failure is interaction faults, specially in mission critical systems
- Combinatorial testing do
  - Combine values systematically but not exhaustively
  - **Rationale**: Most unplanned interactions are among just two or a few parameters or parameter characteristics
  - Look for interactions (or semantics) of the inputs

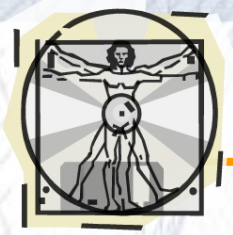




# Combinatorial Testing

- Typical examples
- **System Configuration:** Software applications are often designed to work in a variety of environments. Combinations of factors such as the operating system, network connection, and hardware platform, lead to a variety of environments. Each environment corresponds to a given set of values for each factor, known as a test configuration.
- **Software Configuration:** Set up options
- **Control room operation:** control room of a powerplant involves hundreds of switches that can be set manually





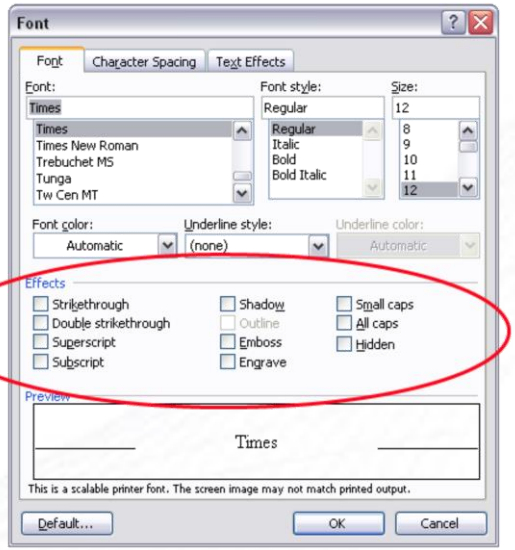
# Combinatorial Testing

- There are 10 effects, each can be On or Off
- All combinations is $2^{10}$ = 1,024 tests
- What if our budget is too limited for these tests?
  - The weak approach: test each effect independently:

    10 effects 1-2 tests each →

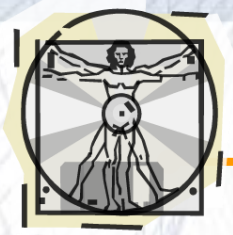
    total 10-20 tests → too weak?
- A middle ground?

  Let's look at **t-way** interactions




Why not testing 2- or 3- or t- of them together?

Pairwise (2-way) testing finds about 50% to 90% of flaws

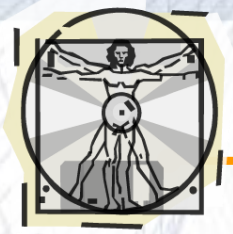


# Combinatorial Testing

- If a set has *n* elements, the number of *t*-way independent combinations is equal to the binomial coefficient
- In this example, number of 3-way interactions $\binom{10}{3} = \frac{10^!}{3^! (7)^!} = 120$
- Naively, 120 x $2^3$ = 960 tests
- Not much saved then 1024 vs. 960!

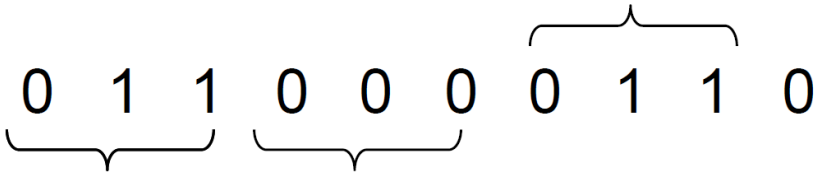
$$\binom{n}{t} = \frac{n^!}{t^! (n-t)^!}$$

Jakob Bernoulli
(1645-1705)

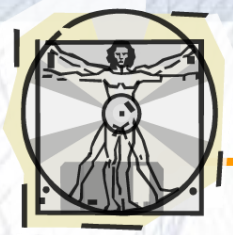


# Combinatorial Testing

- But we can design tests wiser to not test duplicate triples
- Each test exercises many triples:

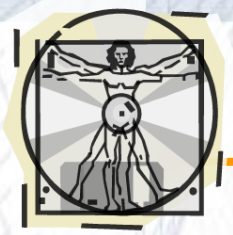



- Since we can pack 3 triples into each test, we need no more than (960/3)=320 tests
- The smallest number of tests we need $\binom{10}{3} = 120$



# Terminology

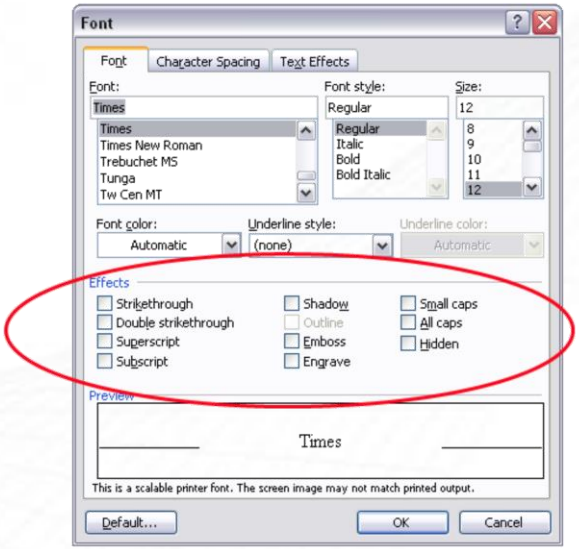
- *k* factors *(*parameters/configurations)
- *t* interactions (strength)
- pair-wise → *t*=2 **t-way testing**
- *l* levels (possible values for each factor, e.g. binary)
- **Covering array:** CA(*k*,*t*,*l*)
- A set of concrete test cases that covers all *t*-way interactions among the *k* factors with all combinations of *l* levels of values



# Minimum Set Covering Array

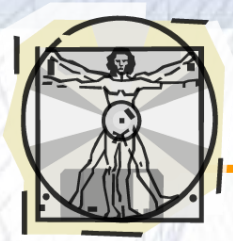
- Each row is a test, each column is a parameter

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 |




**All triples in only 13 tests, covering 960 combinations**

**Unfortunately, finding minimum covering arrays is NP hard**



# NIST's Covering Array Tool

NIST National Institute of Standards and Technology
Information Technology Laboratory

SEARCH: Search

CONTACT SITE MAP

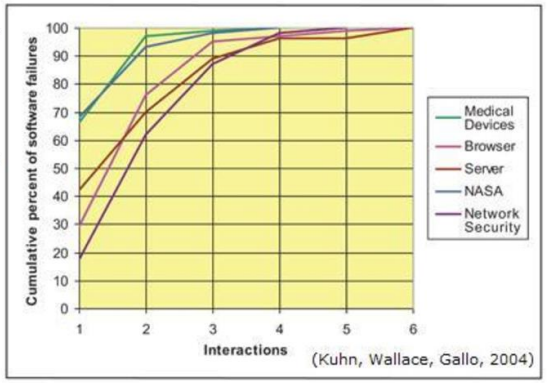
Computer Security Division
Computer Security Resource Center

CSRC Home | About | Projects / Research | Publications | News & Events

- Home
- Research Program & People
- Downloads
- Publications & Talks
- Software Failure Data
- Cybersecurity Testing
- Modeling and Simulation
- Coverage Measurement Tool
- Automated Test Generation
- Covering Array Library
- Event Sequence Testing
- Case Studies and examples
- Covering Arrays vs. Orthogonal Arrays
- Coverage Measurement

CSRC HOME > GROUPS > SNS > ACTS

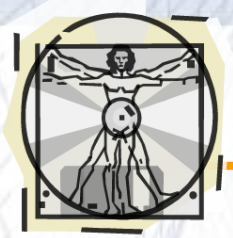
## AUTOMATED COMBINATORIAL TESTING FOR SOFTWARE (ACTS)




Combinatorial testing is a proven method for more effective software testing at lower cost. The key insight underlying combinatorial testing's effectiveness resulted from a series of studies by NIST from 1999 to 2004. NIST research showed that most software bugs and failures are caused by one or two parameters, with progressively fewer by three or more. This finding, referred to as the interaction rule, has important implications for software testing because it means that testing parameter combinations can provide more efficient fault detection than conventional methods. New algorithms compressing combinations into a small number of tests have made this method practical for industrial use, making it possible to do better testing at lower cost.

Read more

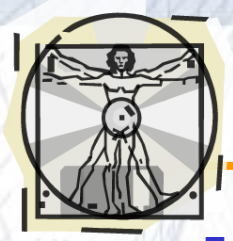
UNIVERSITY OF CALGARY



# Example

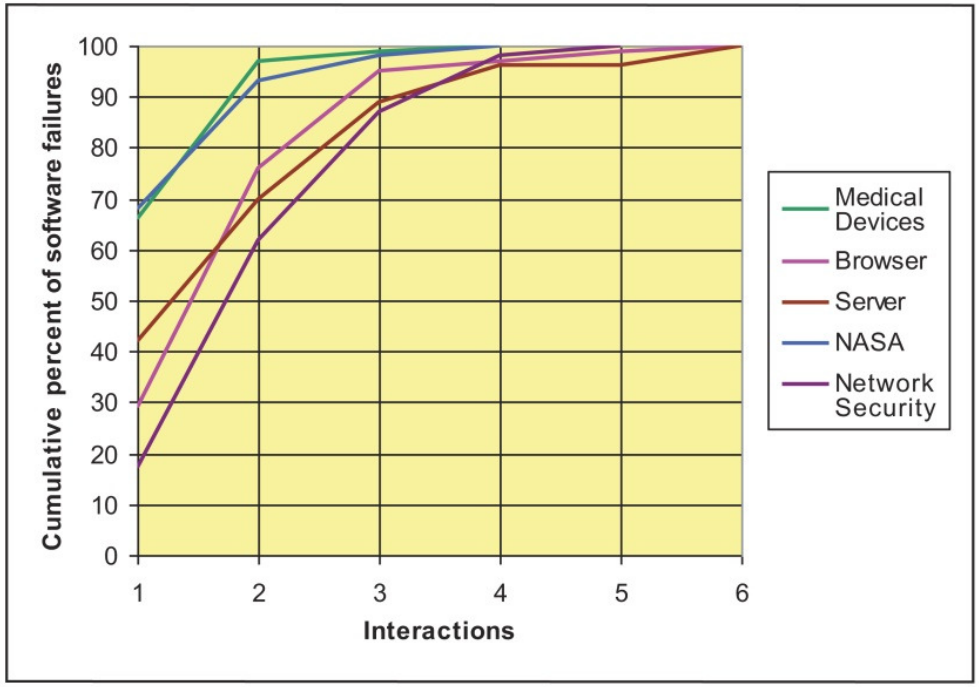
- A system with 34 on-off switches
- $2^{34} = 1.7 \times 10^{10}$ possible inputs tests
- What if we know no failure involves more than 3 switch settings interacting?

3-way interactions, need only **33** tests
4-way interactions, need only **85** tests



# Combinatorial Testing

- If all faults are triggered by the interaction of *t* or fewer variables, then testing all *t*-way combinations can provide strong assurance
- In FDA medical devices, the most complex failure reported required 4-way interaction to trigger
- In avionics software applications maximum interactions for fault triggering has been 6



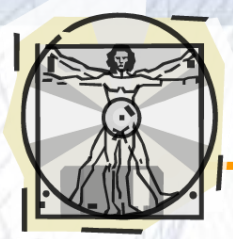

If all faults are triggered by the interaction of *t* or fewer variables, then testing all *t*-way combinations can provide strong assurance

SCHULICH School of Engineering


UNIVERSITY OF CALGARY

# Section 5

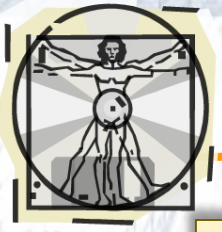
## Profile-based Testing



# Definition: Profile

- **Profile** for a phenomenon is a set of disjoint alternatives called "elements" or "features" that represent that phenomenon together with their occurrence probabilities.
- **Example:**
  - If alternative A occurs 30% of time; B occurs 20% of time and C occurs 50% of time, the profile is

| | |
|---|---|
| A | 0.3 |
| B | 0.2 |
| C | 0.5 |



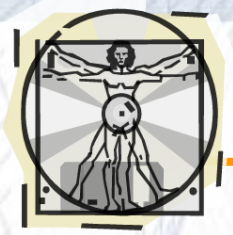
# Definition: Operational Profile

- **Operational Profile** is the set of operations (operation names and frequencies) and their probabilities of occurrences

- **Example:**

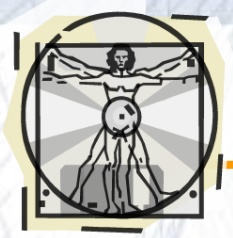
| Operation | Occurrence rate (operations per hour) |
|---|---|
| Phone number entry | 10,000 |
| Add subscriber | 50 |
| Delete subscriber | 50 |
| Process voice call, no pager, answer | 18,000 |
| Process voice call, no pager, no answer | 17,000 |
| Process voice call, pager, answer | 17,000 |
| Process voice call, pager, answer on page | 12,000 |
| Process voice call, pager, no answer on page | 10,000 |
| Process fax call | 15,000 |
| Audit section of phone number database | 900 |
| Recover from hardware failure | 0.1 |
| Total | 100,000 |

*Example From Musa's book*



# Representation

- Representing Operational Profile:
  - Tabular representation
  - Graphical representation

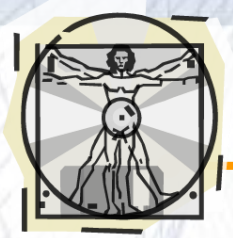


# Representation: Tabular

- Tabular representation is composed of a list of operation names and their probability of occurrence.

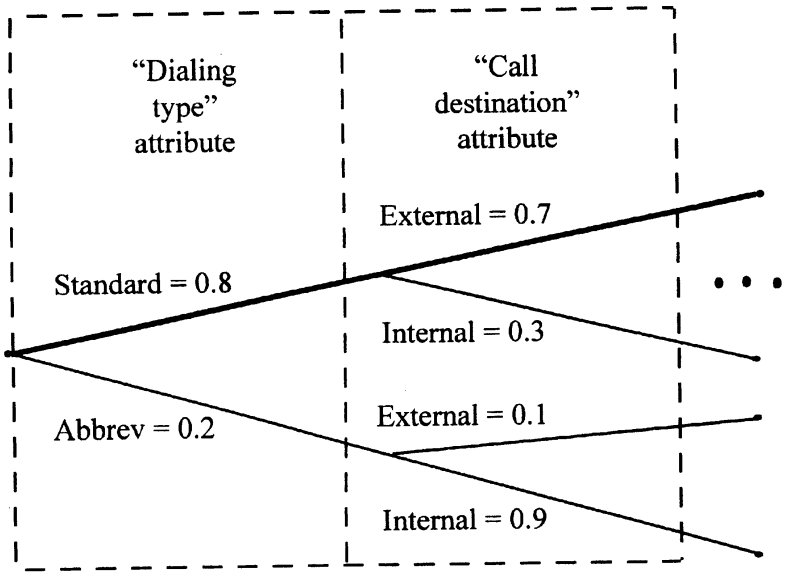
| Command | Transactions per month | Occurrence probability |
|---|---|---|
| relocate | 780 | 0.0153 |
| remove | 90 | 0.0017 |
| add -s staff | 70 | 0.0014 |
| update | | 0.0010 |
| add -s manager | 25 | 0.0005 |
| add -s secretary | 5 | 0.0001 |

Table from *Reliability Engineering Handbook*



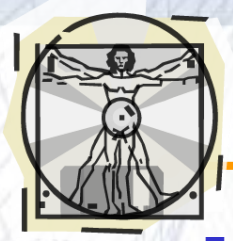
# Representation: Graphical

- Graphical representation is composed of
  - ***Nodes:*** attributes of operations
  - ***Branches:*** values of attributes
  - ***Occurrence probability:*** associated with the branches




Graphical representation can be generated from tabular representation and vice versa.

Figure from *Musa's book*



# Example 1

- An embedded software system has been developed for a VCR system. The system has 5 functions, namely: Stop (STOP) – Rewind (REWIND) – Play (PLAY) – Fast Forward (FF) and Record (REC). We want to create an operational profile for this VCR system in order to test the above 5 functions. The following table shows the usages scenario of each function depending on the starting state (rows) data that has been collected through analysis of customers' behaviour. The 5 functions are shown in the columns. Create the operational profile for this system.

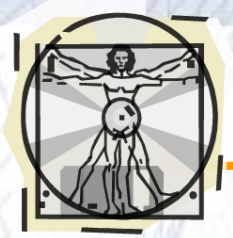
| Functions / Initial state | STOP | REWIND | PLAY | FF | REC |
|---|---|---|---|---|---|
| Standby | 0 | 0.20 | 0.50 | 0.20 | 0.1 |
| Rewind | 0.30 | 0 | 0.60 | 0.10 | 0 |
| Play (counter < 15000) | 0.50 | 0.30 | 0 | 0.20 | 0 |
| Fast forward | 0.50 | 0.10 | 0.40 | 0 | 0 |
| Record | 1 | 0 | 0 | 0 | 0 |
| Play (counter >=15000) | 0.80 | 0.15 | 0 | 0.05 | 0 |

# Example 1

- Operational profile for the VCR system

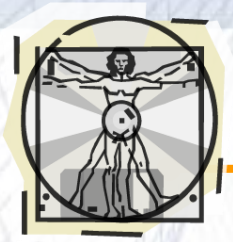
| Functions / Initial state | STOP | REWIND | PLAY | FF | REC | |
|---|---|---|---|---|---|---|
| Standby | 0 | 0.20 | 0.50 | 0.20 | 0.10 | |
| Rewind | 0.30 | 0 | 0.60 | 0.10 | 0 | |
| Play (counter < 15000) | 0.50 | 0.30 | 0 | 0.20 | 0 | |
| Fast forward | 0.50 | 0.10 | 0.40 | 0 | 0 | |
| Record | 1 | 0 | 0 | 0 | 0 | |
| Play (counter >=15000) | 0.8 | 0.15 | 0 | 0.05 | 0 | Sum |
| SUM | 3.10 | 0.75 | 1.50 | 0.55 | 0.10 | 6 |
| Normalize | 0.517 | 0.125 | 0.250 | 0.092 | 0.016 | 1 |

Test resources should be distributed among the programs operating the buttons based on their relative usage weights
Note: Criticality of operation/function may be considered, too



# Example 2

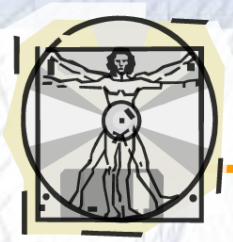
- An embedded software system has been developed for an elevator system
- The system has several operations, namely: UP (to go up) – DOWN (to go down) – HOLD OPEN (hold the doors open) and HOLD CLOSE (hold the doors closed)
- The user can also select the touch panel number 1-10 to select the floor he/she wants to go, in combination with the above buttons
- We want to create an operational profile for this system in order to test the above operations



# Example 2

- The following table shows the usage scenario of each operation depending on the starting state (rows) data that has been collected through analysis of customers' behaviour
- The operations are shown in the columns. For every row, the total probability is one
- **For passengers waiting outside the elevator:**

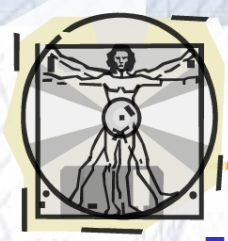
| Functions<br>Initial state of the elevator | UP | DOWN |
|---|---|---|
| Standby | 0.5 | 0.5 |
| Occupied; moving up; current location upper floors | 0.9 | 0.1 |
| Occupied; moving down; current location upper floors | 0.1 | 0.9 |
| Occupied; moving up; current location lower floors | 0.9 | 0.1 |
| Occupied; moving down; current location lower floors | 0.1 | 0.9 |



# Example 2

- **For passengers inside the elevator:**

| Functions<br>Initial state of the elevator | Select floor 1 to 10 | HOLD OPEN | HOLD CLOSE |
|---|---|---|---|
| Standby | 0.9 | 0.05 | 0.05 |
| Occupied; moving up; current location any | 0.8 | 0.1 | 0.1 |
| Occupied; moving down; current location any | 0.8 | 0.1 | 0.1 |



# Example 2

- Create test profile for this system by combining the two tables

| Functions<br>Initial state of the elevator | UP | DOWN | Select floor 1 to 10 | HOLD OPEN | HOLD CLOSE | |
|---|---|---|---|---|---|---|
| Standby | 0.5 | 0.5 | 0.9 | 0.05 | 0.05 | |
| Occupied; moving up; upper floors | 0.9 | 0.1 | 0.8 | 0.1 | 0.1 | |
| Occupied; moving up; lower floors | 0.9 | 0.1 | 0.8 | 0.1 | 0.1 | |
| Occupied; moving down; upper floors | 0.1 | 0.9 | 0.8 | 0.1 | 0.1 | |
| Occupied; moving down; lower floors | 0.1 | 0.9 | 0.8 | 0.1 | 0.1 | |
| SUM | 2.5 | 2.5 | 4.1 | 0.45 | 0.45 | Sum 10 |
| profile | 0.25 | 0.25 | 0.41 | 0.045 | 0.045 | 1 |

Test resources should be distributed among the programs operating the buttons based on their relative usage weights

# Conclusions

- Black Box partitioning techniques give us a
  - Systematic approach to identify values to be tested per input/feature
  - And couple of potential ways to combine them (weak/strong/robust, etc.)
- But ... test suite size grows very rapidly with number of categories
- Can we use a non-exhaustive approach?
  - Pairwise (and t-way) combinatorial, category-partitions and profile based testing do