Software Testing

Course Code CSE-4017

Lecture 3: Functional Testing

Subject Guide: Dr. Aprna Tripathi

Content

- Software testing concepts
- Software testing Terminology
- Software testing process
- Levels of software testing
- Types of testing techniques
- Functional testing
  - BVA
  - EC
  - CEA
- Structural testing
  - Control flow testing
  - Data low testing
  - Coverage testing

Lecture Objectives:

- The objectives of this lecture are:
  - Explore the concept of Testing
  - Detailed understanding of bug and its variants
  - To understand the need of functional testing
  - To understand and develop test cases using functional testing techniques
    - Boundary Value Analysis
    - Equivalence Class Analysis
    - Decision Table Based Approach

Who should Do the Testing ?

- Testing requires the developers to find errors from their software.
- It is difficult for software developer to point out errors from own creations.
- Many organisations have made a distinction between development and testing phase by making different people responsible for each phase.

Why should We Test ?

- Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.
- In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

Error

- Error
  - A person act that generates an erroneous result.
  - Consequently, the term **failure** refers to a behavioral deviation from the user wants or the product specification.
  - **Error** refers to a missing or wrong person action resulting in certain fault being injected into software.
  - Errors also include error sources such as human misunderstandings, dissensions, misinterpretation and so on.

Bugs

- Even if we correctly discover all cases for placing words on the Scrabble board, it is very likely that we'll have some bugs when we code it
  - bugs are inevitable in any complex software system
  - a bug can be very visible or can hide in your code until a much later date
- we can hunt down the cause of a known bug using print statements or our IDE's **debugger** ... but how do we discover all of the bugs in our system, even those with low visibility?
  - ANSWER: testing and Quality Assurance practices

Bug

- A bug is the result of a coding error.
- Any Missing functionality or any action that is performed by the system which is not supposed to be performed is a Bug.
- Any of the following may be the reason for birth of Bug
    1. Wrong functionality
    2. Missing functionality
    3. Extra or unwanted functionality

Faults

- What is the difference between a fault and an error? What are some kinds of faults?
- **Error**: incorrect software behavior

  example: message box text said "Welcome null."
- **Fault**: mechanical or algorithmic cause of error
  - *example: account name field is not set properly.*
  - a fault is not an error, but it can lead to them
  - need requirements to specify desired behavior, and need to see system deviate from that behavior, to have a failure

Some types of faults

- algorithmic faults
  - design produces a poor algorithm
  - fail to implement the software to match the specification
  - subsystems don't communicate properly
- mechanical faults
  - earthquake
  - virtual machine failure (why is this a "mechanical" fault?)

Defect

- A defect is a deviation from the requirements.
- A defect does not necessarily mean there is a bug in the code, it could be a function that was not implemented but defined in the requirements of the software.

Error, Mistake, Bug, Fault and Failure

- People make **errors. A good synonym is mistake. This may be a syntax** error or misunderstanding of specifications. Sometimes, there are logical errors.
- When developers make mistakes while coding, we call these mistakes "**bugs".**
- A **fault is the representation of an error, where representation is the mode** of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc.
- A **failure occurs when a fault executes. A particular fault may cause** different failures, depending on how it has been exercised.

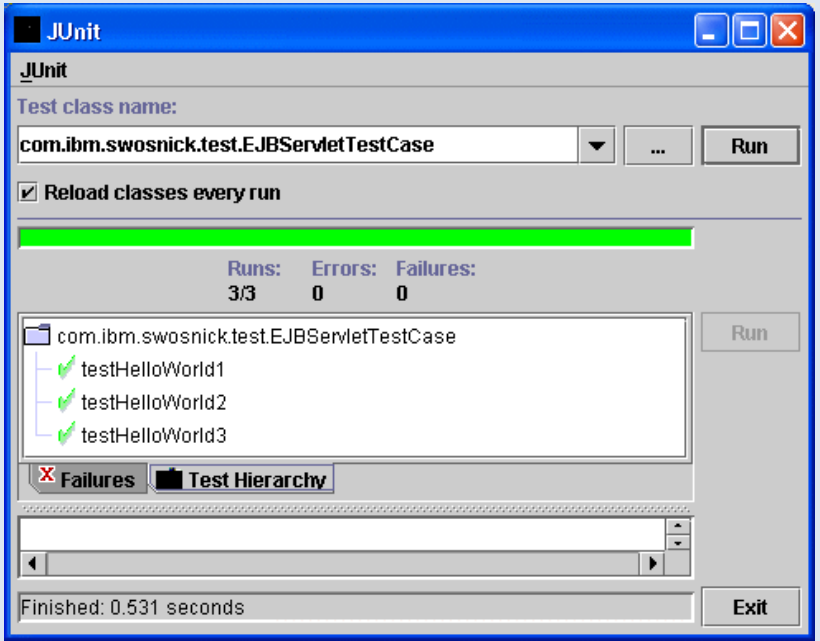
Defect, Failure and Fault

- Defect
  - Commonly refers to several troubles with the software products, with its external behavior or with its internal features.
- Failure
  - The incapacity of a system to conduct its required functions within clarified performance requirements.
- Fault
  - A false, wrong step, process or data definition in a software product.
  - **Fault** refers to an underlying condition within software that causes failure to happen.

Testing concepts

What is a test case? What is a failure? How are they related?

- **failure**: particular instance of a general error, which is caused by a fault
- **test case**: set of inputs and outputs to cause failures



Test cases

What are the five elements of a well-written test case, according to the authors? (Hint: one of these is an "oracle." What is this?)

- name: descriptive name of what is being tested
- location: full path/URL to test
- input: arguments, commands, input files to use
  - entered by tester or test driver
- oracle: expected output
- log: actual output produced

Test data and test cases

- *Test data* Inputs which have been devised to test the system
- *Test cases* Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

Test, Test Case and Test Suite

Test and **Test case** terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description.

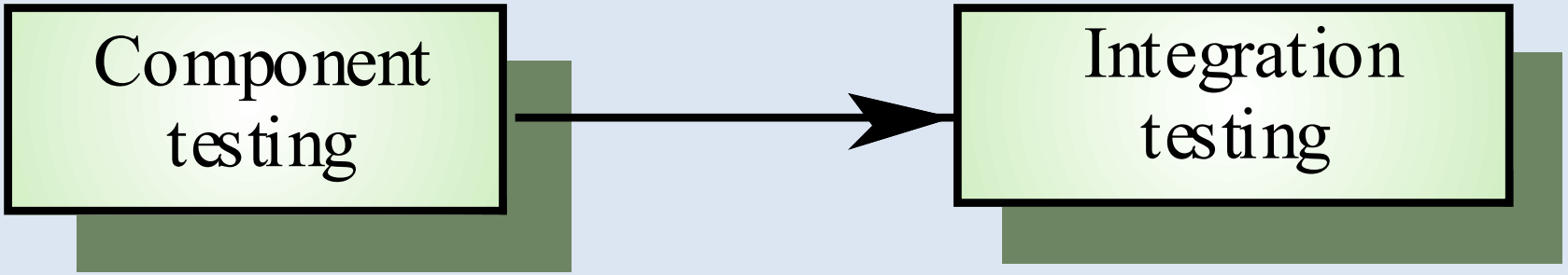
Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

The set of test cases is called a **test suite**. Hence any combination of test cases may generate a test suite.

The testing process

- Component testing / Unit Testing
  - Testing of individual program components
  - Usually the responsibility of the component developer (except sometimes for critical systems)
  - Tests are derived from the developer's experience
- Integration testing
  - Testing of groups of components integrated to create a system or sub-system
  - The responsibility of an independent testing team
  - Tests are based on a system specification

Testing phases



Software developer

Independent testing team

Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests.

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers.

Alpha Tests are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

Example

- Person A wanted from the software comp. B to design a program for library book issue and return
- Inside the comp. B, developer X implemented these two functionalities.
- SRS- Max 6 books can be issued
- What a programmer X has to implement?
- SRS and program === validation--
- If as a tester I am performing:
  - Max 5 books can be issued at any time, and I trying to issue 6th book from my account.
  - Are we able to supply all the user requirements.

  Result and user need=== verification

Verification and Validation

- **Verification is the process of evaluating a system or component to** determine whether the products of a given development phase *satisfy the conditions imposed at the start of that phase*. (black box, functional testing)
- **Validation is the process of evaluating a system or component during or at** the end of development process to determine whether it *satisfies the specified requirements* . (white Box, Structural Testing)
- **Testing= Verification + Validation**

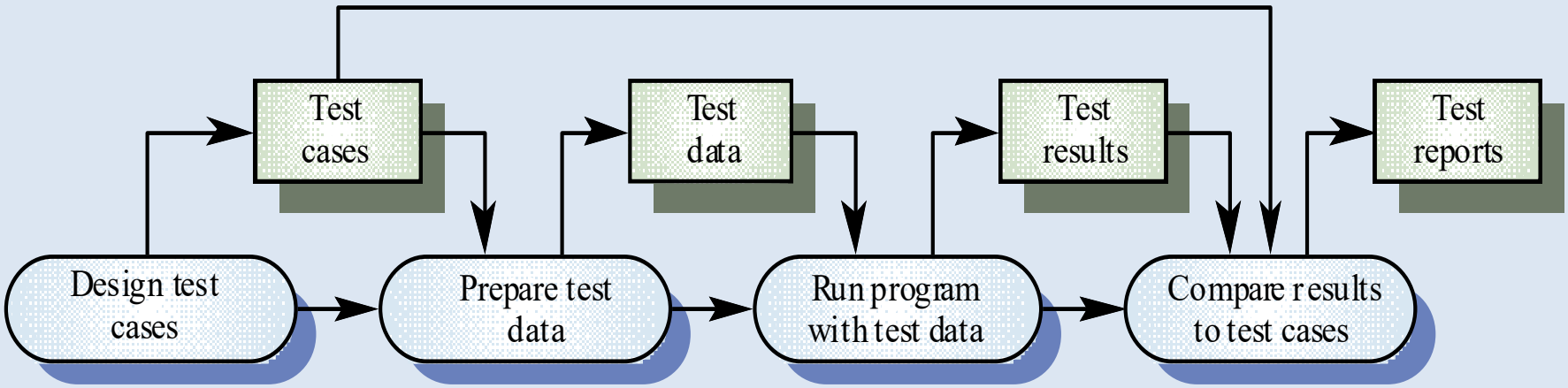
Defect testing

- The goal of defect testing is to discover defects in programs
- A *successful* defect test is a test which causes a program to behave in an anomalous way
- Tests show the presence not the absence of defects.

Testing priorities

- Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible
- Tests should exercise a system's capabilities rather than its components
- Testing old capabilities is more important than testing new capabilities
- Testing typical situations is more important than boundary value cases

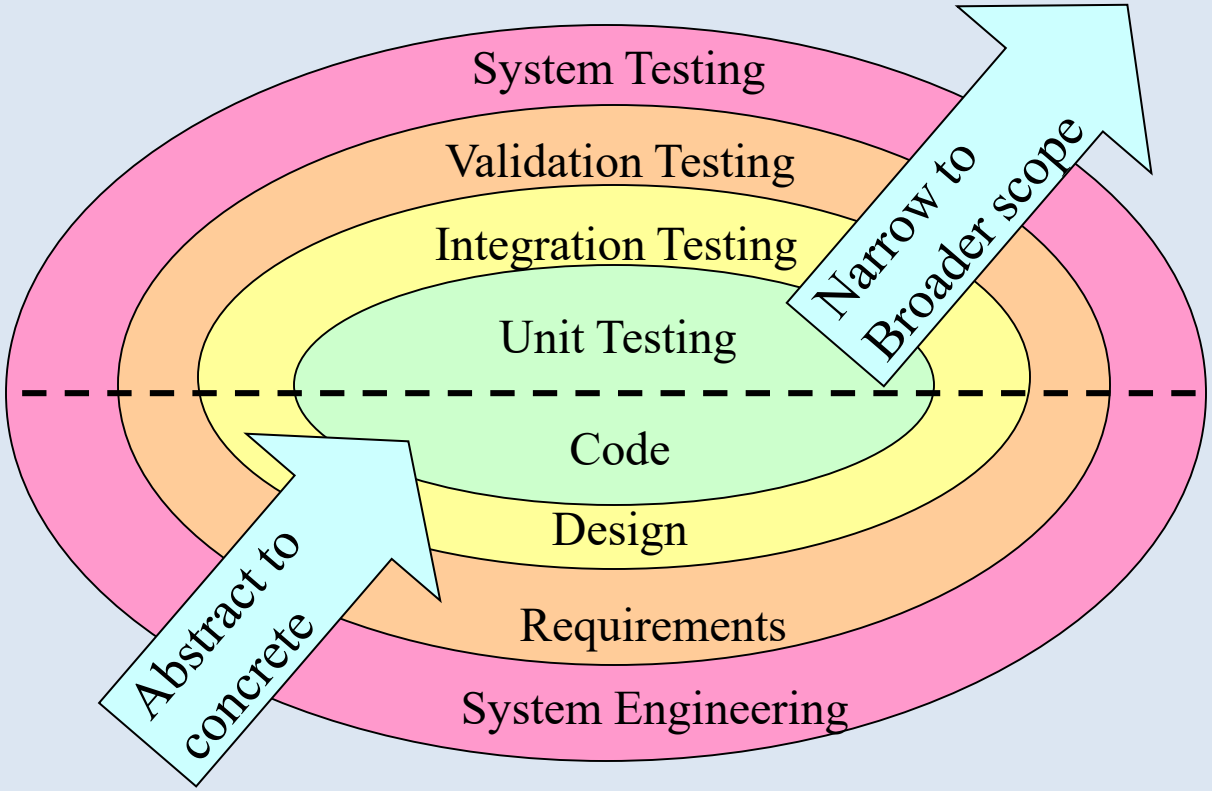
The defect testing process



Organizing for Software Testing

- Testing should aim at "breaking" the software
- Common misconceptions
  - The developer of software should do no testing at all
  - The software should be given to a secret team of testers who will test it unmercifully
  - The testers get involved with the project only when the testing steps are about to begin
- Reality: Independent test group
  - Removes the inherent problems associated with letting the builder test the software that has been built
  - Removes the conflict of interest that may otherwise be present
  - Works closely with the software developer during analysis and design to ensure that thorough testing occurs

A Strategy for Testing Conventional Software



Levels of Testing for Conventional Software

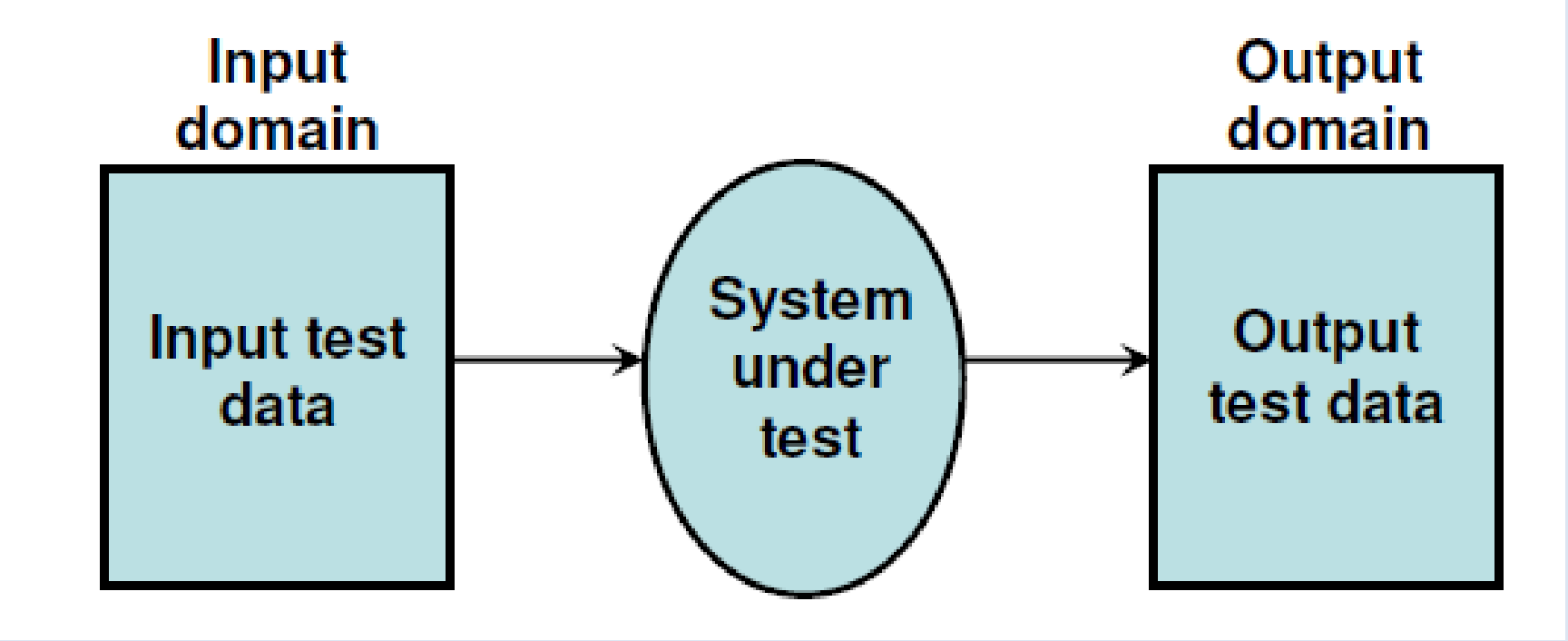
- Unit testing
  - Concentrates on each component/function of the software as implemented in the source code
- Integration testing
  - Focuses on the design and construction of the software architecture
- Validation testing
  - Requirements are validated against the constructed software
- System testing
  - The software and other system elements are tested as a whole

Testing Strategy applied to Conventional Software

- Unit testing
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
- Integration testing
  - Focuses on inputs and outputs, and how well the components fit together and work together
- Validation testing
  - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- System testing
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Black Box Testing

Functional Testing



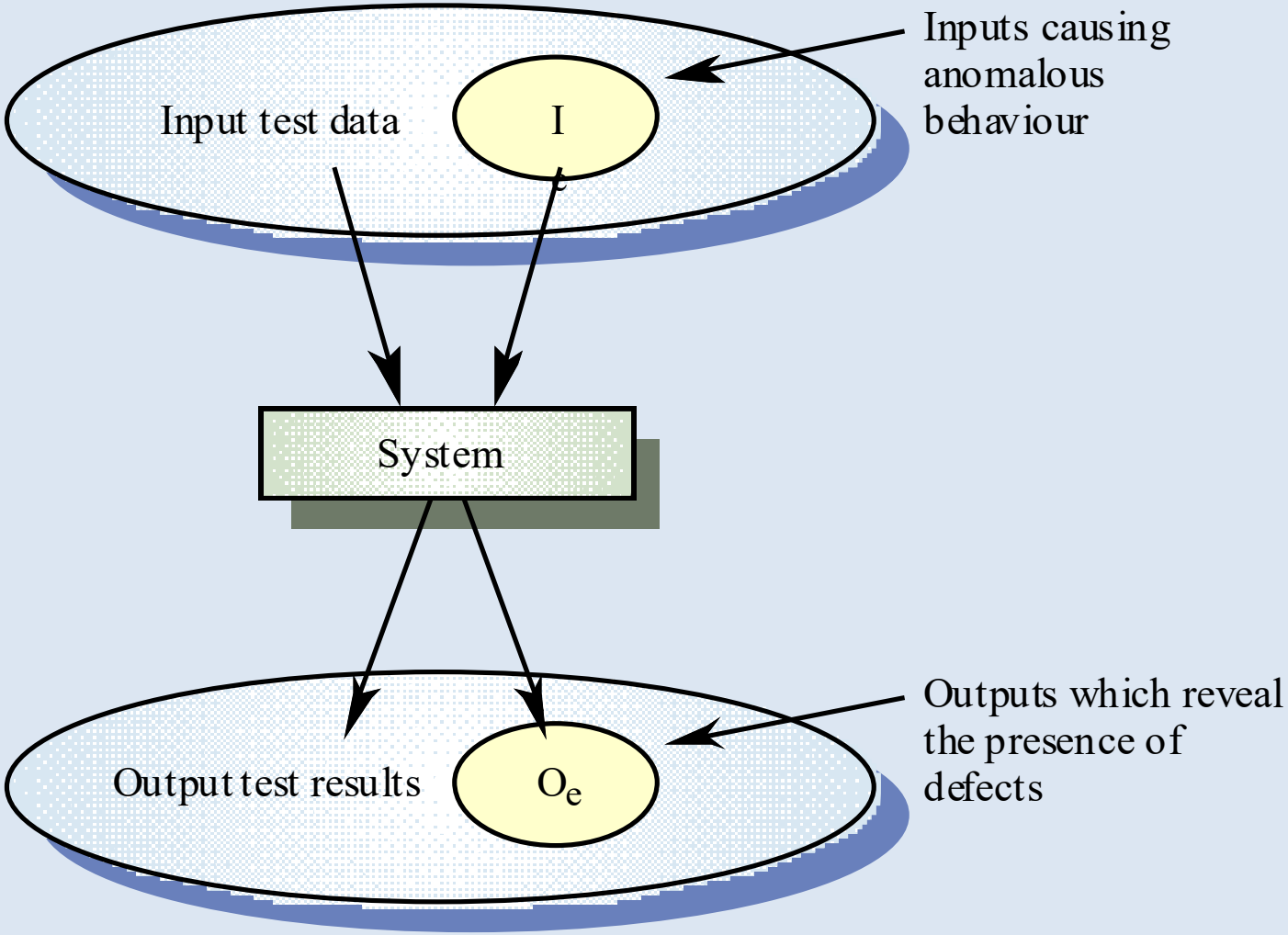
Functional Testing

- Any program can be considered to be a function
  - Program inputs form its domain
  - Program outputs form its range
- Boundary value analysis is the best known functional testing technique.
- The objective of functional testing is to use knowledge of the functional nature of a program to identify test cases.
- Historically, functional testing has focused on the input domain, but it is a good supplement to consider test cases based on the range as well.

Black-box testing

- Functional Testing Also called as Black Box Testing because for testing the program is considered as a ‘black-box’
- The program test cases are based on the system specification
- Test planning can begin early in the software process

Black-box testing



Functional Testing

- Functional Testing
  - Boundary Value Testing (BVT)
  - Equivalence Class Testing
  - Decision Table Based testing

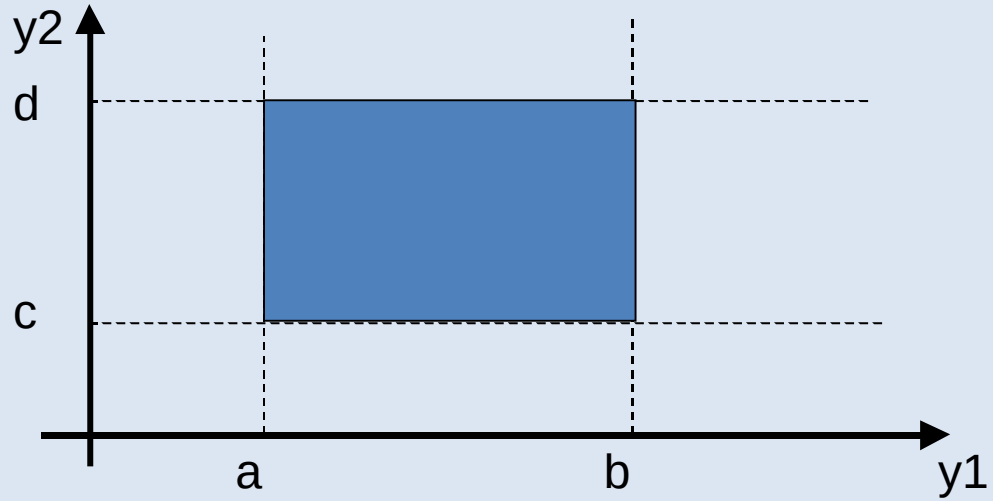
Boundary Value Analysis

- Boundary value analysis focuses on the boundary of the input space to identify test cases.
- The rationale behind boundary value analysis is that errors tend to occur near the extreme values of an input variable.
- Programs written in non-strongly typed languages are more appropriate candidates for boundary value testing.
- In our discussion we will assume a program P accepting two inputs x_1 andx_2 such that $a \leq y_1 \leq b$ and $c \leq y_2 \leq d$

Valid Input for Program P

- consider the following function:
  $f(y_1, y_2)$, where $a \le y_1 \le b$, $c \le y_2 \le d$

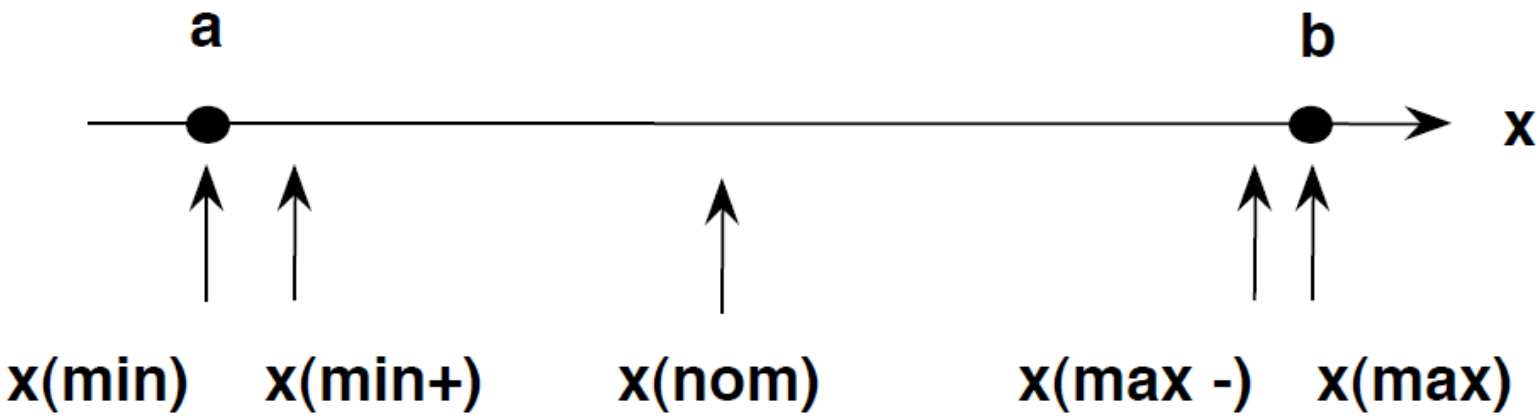
- boundary inequalities of *n* input variables define an *n*-dimensional input space:



Sum
X = 1- 10
Y = 1- 10
X=0 -34
Y=0 -12

Value Selection in Boundary Value Analysis

- The basic idea in boundary value analysis is to select input variable values at their:
  - Minimum
  - Just above the minimum
  - A nominal value
  - Just below the maximum
  - Maximum


a
b
x
x(min)
x(min+)
x(nom)
x(max -)
x(max)

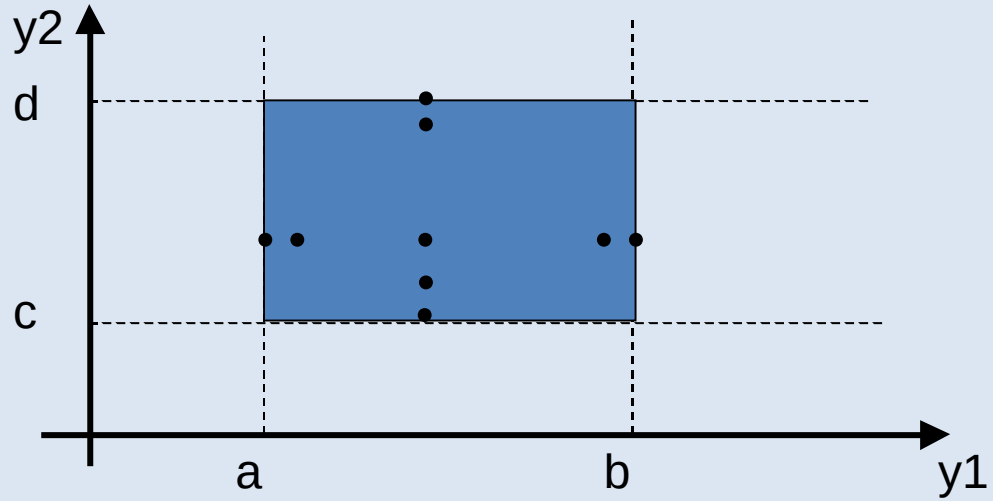
Single Fault Assumption

- Boundary value analysis is also augmented by the single fault assumption principle.

"Failures occur rarely as the result of the simultaneous occurrence of two (or more) faults"

- Method
  - by holding the values of all but one variable at their nominal values, and letting that variable assume its extreme values.

Boundary Value Analysis for Program P



$m, y_{2min}>, <y_{1nom}, y_{2min+}>, <y_{1nom}, y_{2nom}>, <y_{1nom}, y_{2max-}>,$
$_x>, <y_{1min}, y_{2nom}>, < y_{1min+}, y_{2nom}>, <y_{1max-}, y_{2nom}>,$
$_n> \}$

Example Test Cases Using Boundary Value Analysis

a,b,c representing the sides of triangle, we want to test the behavior
a,b,c all range is 1- 200

Case #	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Trianle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a Triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a Triangle

Generalizing Boundary Value Analysis

- The basic boundary value analysis can be generalized in two ways:
  - By the number of variables - (4n +1) test cases for n variables
  - By the kinds of ranges of variables
    - Programming language dependent
    - Bounded discrete
    - Unbounded discrete (no upper or lower bounds clearly defined)
    - Logical variables

Next Date Problem

- To find the next date
- Month-> 1-12
- Day-> 1-31
- Year -> 2000- 2020

Test ID	Day	Month	Year

Next Date Problem

- To find the next date
- Month-> 1-12
- Day-> 1-31
- Year -> 2000- 2020

Test ID	Day	Month	Year
1	1	6	2010
2	2	6	2010
3	15	6	2010
4	30	6	2010
5	31	6	2010
6	15	1	2010
7	15	2	2010
8	15	11	2010
9	15	12	2010
10	15	6	2000
11	15	6	2001
12	15	6	2019
13	15	6	2020

Limitations of Boundary Value Analysis

- Boundary value analysis works well when the program to be tested is a function of several *independent* variables that represent *bounded physical quantities*.

- Boundary value analysis selected test data with no consideration of the function of the program, nor of the semantic meaning of the variables.

- We can distinguish between physical and logical type of variables as well (e.g. temperature, pressure speed, or PIN numbers, telephone numbers etc.)

Independence Assumption and Efficacy of BVT

- Assumes that input variables are independent of one another,
  - i.e. the assumption that particular combinations of input variable values have no special significance
- If basic assumption is not true, then BVT may overlook important test requirements
- BVT is an instance of more general techniques such as equivalence class testing or domain testing.
- BVT tends to generate more test cases with poorer test coverage (with the independence assumption) than domain or equivalence testing.
- But, due to its simplicity, BVT test case generation can be easily automated.

Limitation of BVT

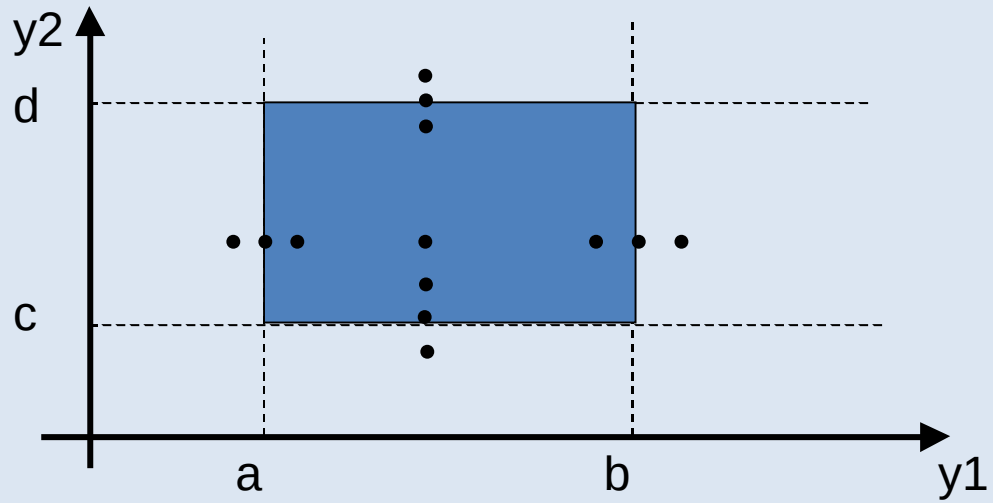
test cases for a variable x, where a <= x <= b

1. stress input boundaries
2. acceptable response for invalid inputs?
3. leads to exploratory testing (test hackers)
4. can discover hidden functionality

Robustness Testing

- Robustness testing is a simple extension of boundary value analysis.
- In addition to the five boundary value analysis values of variables, we add values slightly greater that the maximum (max+) and a value slightly less than the minimum (min-).
- The main value of robustness testing is to force attention on exception handling.
- In some strongly typed languages values beyond the predefined range will cause a run-time error.
- It is a choice of using a weak typed language with exception handling or a strongly typed language with explicit logic to handle out of range values.

Robustness Test Cases for Program P



Example

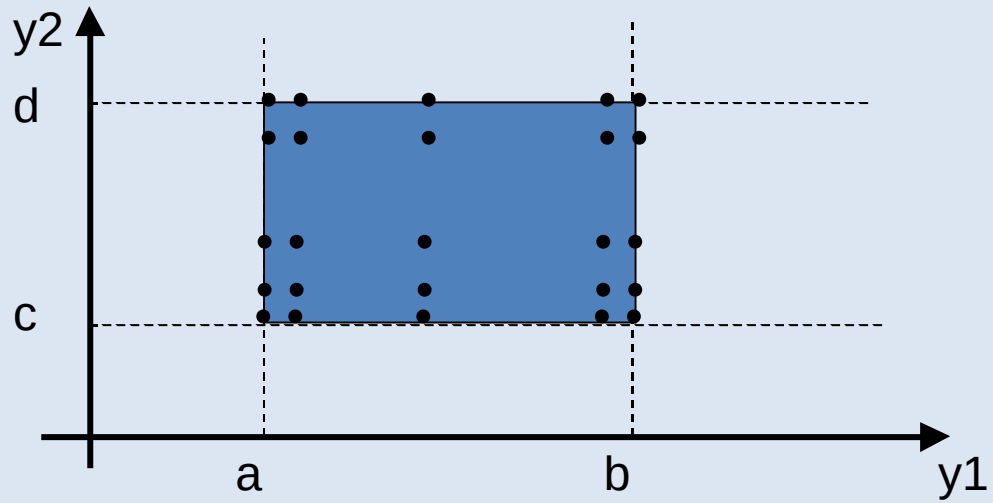
- A: 1 to 10
- B: 1 to 10
- No. of Test case = 6*n +1
- Where n is no. of variable

TestID	A	B	Output
1	1	5	Valid
2	2	5	Valid
3	5	5	Valid
4	9	5	Valid
5	10	5	Valid
6	0	5	InValid
7	11	5	InValid
8	5	0	InValid
9	5	1	Valid
10	5	2	Valid
11	5	9	Valid
12	5	10	Valid
13	5	11	InValid

Worst Case Testing

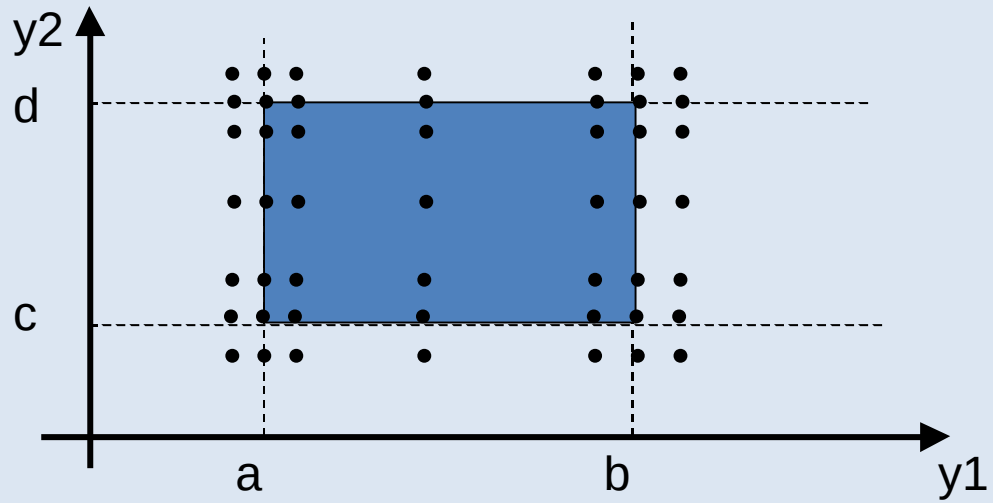
- In worst case testing we reject the single fault assumption and we are interested what happens when more than one variable has an extreme value.
- Considering that we have five different values that can be considered during boundary value analysis testing for one variable, now we take the Cartesian product of these possible values for 2, 3, … n variables.
- In this respect we can have 5^n test cases for n input variables.
- The best application of worst case testing is where physical variables have numerous interactions and failure of a program is costly.
- Worst case testing can be further augmented by considering robust worst case testing (i.e. adding slightly out of bounds values to the five already considered).

Worst Case Testing for Program P



Robust Worst Case Testing for Program P

Y1>> y1 min-, y1, min, y1min+m y1nom, y1max-,Y1max, y1max+



Special Value Testing

- Special value testing is probably the most widely practiced form of functional testing, most intuitive, and least uniform.

- Utilizes domain knowledge and engineering judgment about program's "soft spots" to devise test cases.

- Event though special value testing is very subjective on the generation of test cases, it is often more effective on revealing program faults.

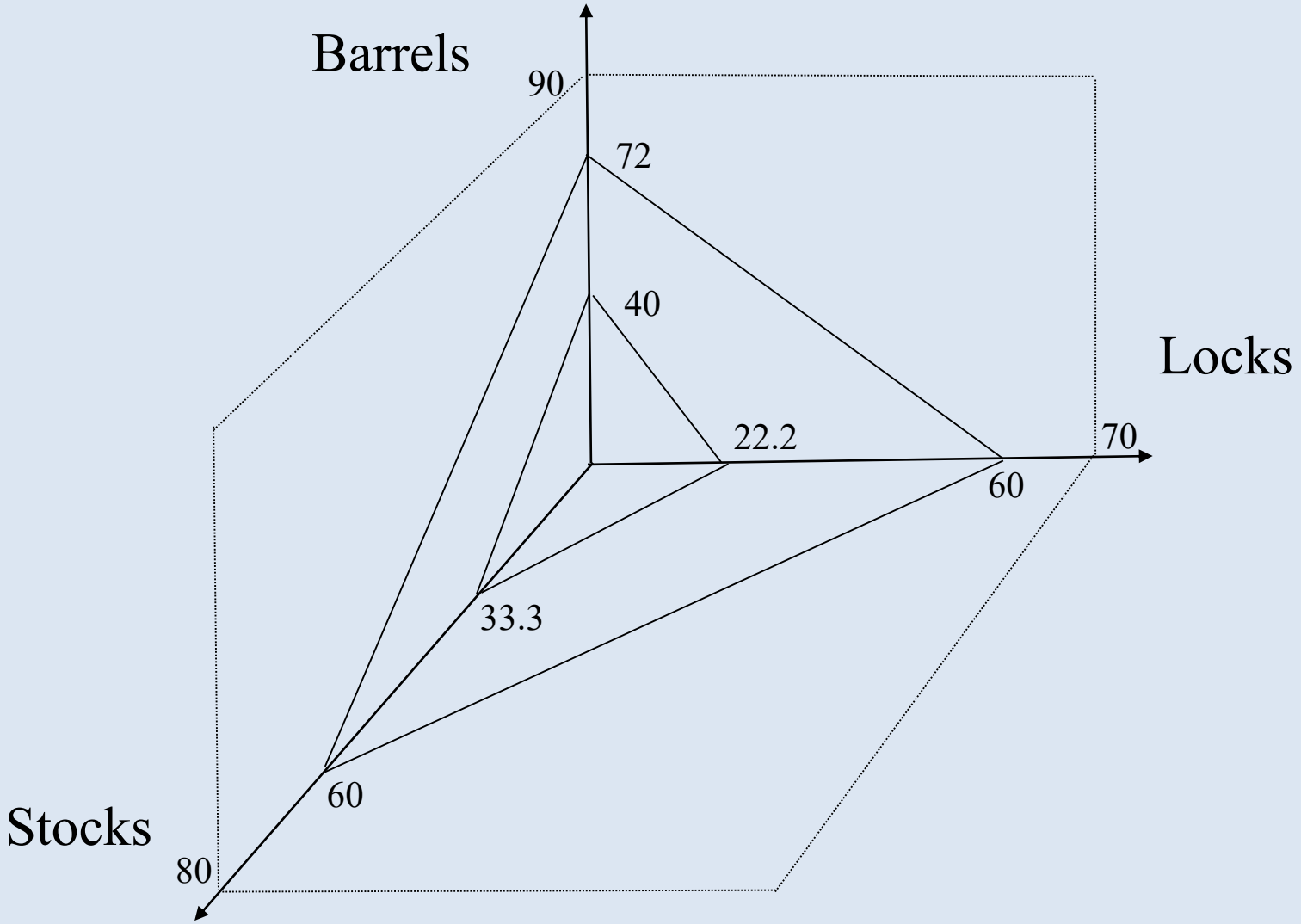
Guidelines for Boundary Value Testing

- With the exception of special value testing, the test methods based on the boundary values of a program are the most simple.
- Issues in producing satisfactory test cases using boundary value testing:
  - Truly independent variables versus not independent variables
  - Normal versus robust values
  - Single fault versus multiple fault assumption
- Boundary value analysis can also be applied to the output range of a program (i.e. error messages), and internal variables (i.e. loop control variables, indices, and pointers).

The Commission Problem

- Rifle salespersons in the Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri
- Lock = $45.00, stock = $30.00, barrel = $25.00
- Each salesperson had to sell at least one complete rifle per month ($100)
- The most one salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels
- Each salesperson sent a telegram to the Missouri company with the total order for each town (s)he visits
- $1 \leq$ towns visited ≤ 10, per month
- Commission: 10% on sales up to $1000, 15% on the next $800, and 20% on any sales in excess of $1800

Example Test Cases Using Output Range Values



Output Boundary Value Test Cases

Case #	Locks	Stocks	Barrels	Sales	Comm.	Comments
1	1	1	1	100	10	min
2	10	10	9	975	97.5	border-
3	10	9	10	970	97	border-
4	9	10	10	955	95.5	border-
5	**10**	**10**	**10**	**1000**	**100**	**border**
6	10	10	11	1025	103.75	border+
7	10	11	10	1030	104.5	border+
8	11	10	10	1045	106.75	border+

Output Special Value Test Cases

Case #	Locks	Stocks	Barrels	Sales	Comm.	Comment
1	10	11	9	**1005**	100.75	border+
2	18	17	19	**1795**	219.25	border+
3	18	19	17	**1805**	221	border+

NextDate Function

NEXTDATE is a function of three variables: month, day, and year, for years from 1812 to 2012. It returns the date of the next day.

NEXTDATE(Dec, 31, 1991) returns Jan 1 1992

NEXTDATE(Feb, 21, 1991) returns Feb 22 1991

NEXTDATE(Feb, 28, 1991) returns Mar 1 1991

NEXTDATE(Feb, 28, 1992) returns Feb 29 1992

Leap Year: Years divisible by 4 except for century years not divisible by 400. Leap Years include 1992, 1996, 2000. 1900 was not be a leap year.

Input Domain Test Cases

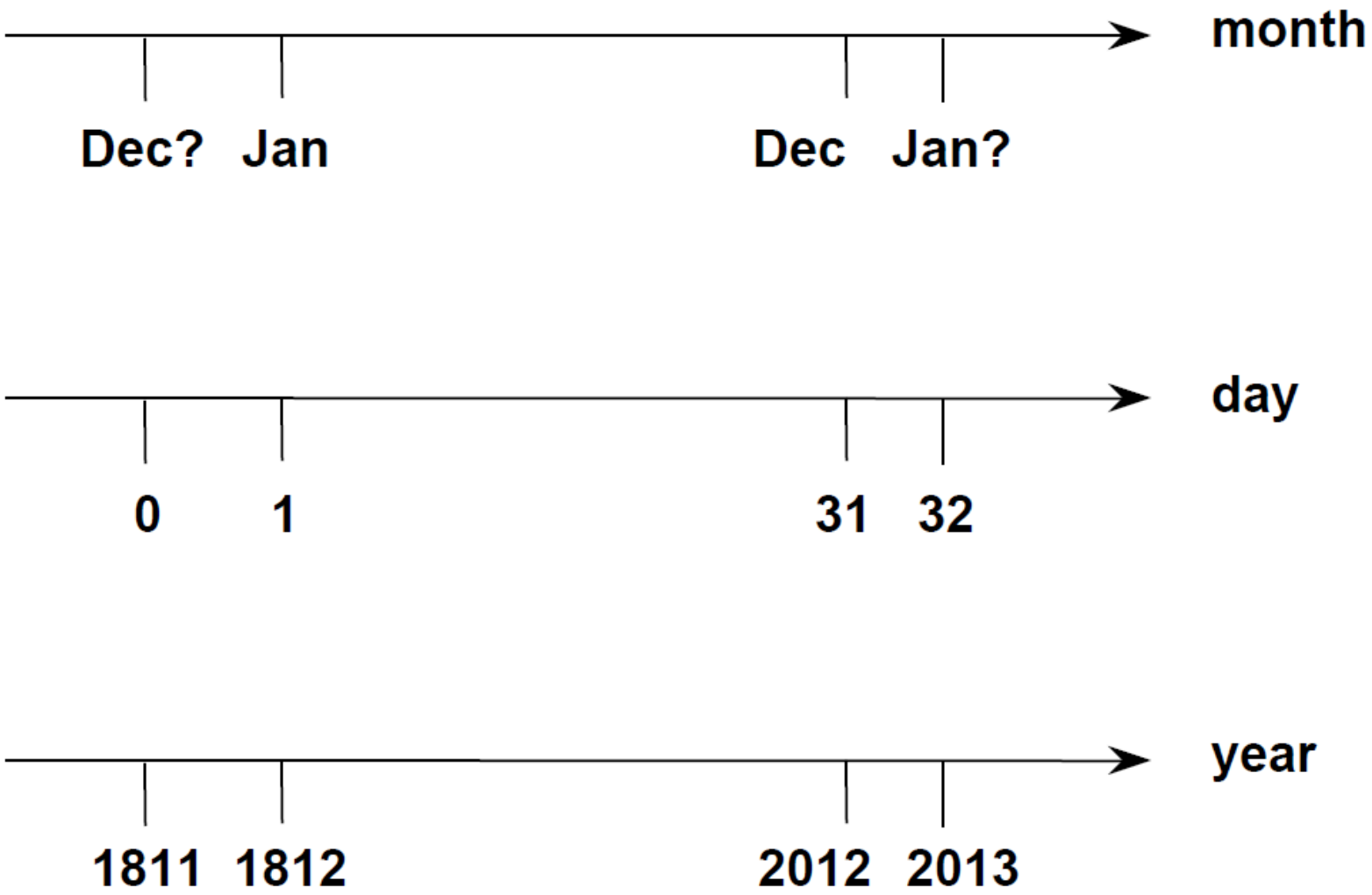
Test Case	Input Values: Month	Day	Year
ID-1	Jan	15	1912
ID-2	Feb	15	1912
ID-3	Jun	15	1912
ID-4	Nov	15	1912
ID-5	Dec	15	1912
ID-6	Jun	1	1912
ID-7	Jun	2	1912
ID-3	Jun	15	1912
ID-8	Jun	30	1912
ID-9	Jun	31	1912
ID-10	Jun	15	1812
ID-11	Jun	15	1913
ID-3	Jun	15	1912
ID-12	Jun	15	2011
ID-13	Jun	15	2012

Select test cases so that one variable assumes nominal and extreme values while others are held at nominal values.

Notice that Input Domain testing presumes that the variables in the input domain are independent; logical dependencies are unrecognized.

This typically results in "anomalies" like test case ID-9, which is logically impossible.

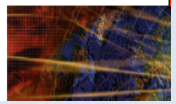
Robust Input Domain Test Cases



Robust Input Domain Test Cases

Robust Test Case	Test Case	Input Values: Month	Day	Year
RD-1	ID-5	Dec	15	1912
RD-2	ID-1	Jan	15	1912
RD-3	ID-2	Feb	15	1912
RD-4	ID-3	Jun	15	1912
RD-5	ID-4	Nov	15	1912
RD-6	ID-5	Dec	15	1912
RD-7	ID-1	Jan	15	1912
RD-8		Jun	0	1912
RD-9	ID-6	Jun	1	1912
RD-10	ID-7	Jun	2	1912
RD-11	ID-3	Jun	15	1912
RD-12	ID-8	Jun	30	1912
RD-13	ID-9	Jun	31	1912
RD-14		Jun	32	1912
RD-15		Jun	15	1811
RD-16	ID-10	Jun	15	1812
RD-17	ID-11	Jun	15	1913
RD-18	ID-3	Jun	15	1912
RD-19	ID-12	Jun	15	2011
RD-20	ID-13	Jun	15	2012
RD-21		Jun	15	2013

As with input domain testing, Robustness Testing presumes independent variables. The major difference is that robustness Testing also presumes that variables represent continuous functions. Notice that the Robustness Test Cases for Month and Day sometimes don't make sense, but those for Year do.



Special Value Test Cases

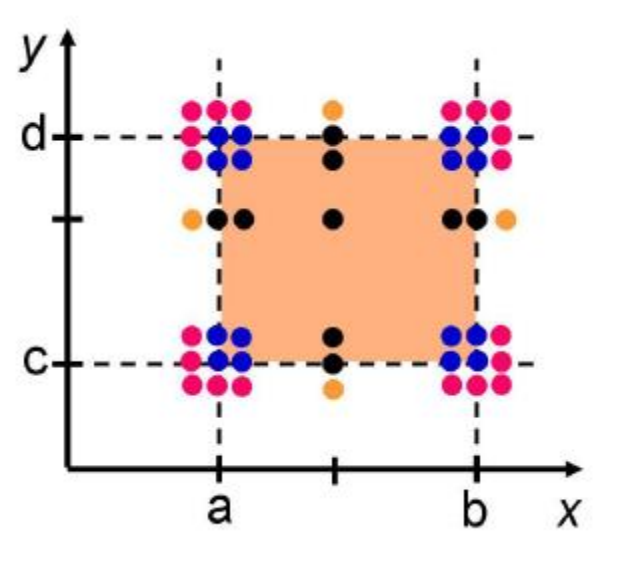
Test Case	Input Values			Reason for Test Case
	Month	Day	Year	
SV-1	Feb	28	1912	Leap day in a leap year
SV-2	Feb	28	1911	Leap day in a non-leap year
SV-3	Feb	29	1912	Leap day increment in a leap year
SV-4	Feb	28	2000	Leap day in the year 2000
SV-5	Feb	28	1900	Leap day in the year 1900
SV-6	Dec	31	1912	Change of year
SV-7	Jan	31	1912	Change of a 31 day month
SV-8	Apr	30	1912	Change of a 30 day month
SV-9	Dec	31	2012	Last day of defined interval

Boundary Value Testing: Pros and Cons

- straightforward test-case generation
  - no sense of covering the
- input domain
  - awkward for logical vars.
  - only independent input
  - domains
- not using white-box information

Boundary Value Testing (recap)

- boundary value testing:
  - choose extreme values.
- variants:
  - worst-case
  - robust
  - robust worst-case
  - other (non-standard)
- variants:
  - special value
  - random



Equivalence Class Testing

- I have a var1: 1-10
- Output: even/odd

Boundary value problems

- **What problems does boundary value testing have?**

Boundary value problems

- **What problems does boundary value testing have?**
- Boundary Value Testing derives test cases with
  - Serious gaps
  - Massive redundancy

Motivation for equivalence class testing

- **What are the motivations for equivalence class testing?**

Motivation for equivalence class testing

- **What are the motivations for equivalence class testing?**
- Avoid redundancy
  - Have fewer test cases
- Complete testing
  - Remove gaps

What assumptions are made?

- Program is a function from input to output
  - Input and/or output variables have well defined intervals
- For a two-variable function F(x1,x2)
  - $a \leq x1 \leq d$, with intervals [a,b), [b,c), [c,d]
  - $e \leq x2 \leq g$, with intervals [e,f), [f,g]

- Completeness
  - The entire set is represented by the union of the sub-sets
- Redundancy
  - The disjointness of the sets assures a form of non-redundancy
- Choose one test case from each sub-set

Equivalence partitioning

- Equivalence partitioning is a software testing technique that divides the input and/or output data of a software unit into partitions of data from which test cases can be derived.
- The equivalence partitions are usually derived from the requirements specification for input attributes that influence the processing of the test object.
- Test cases are designed to cover each partition at least once.

What can be found using equivalence partitioning?

- Equivalence partitioning technique uncovers classes of errors.

- Testing uncovers sets of inputs that causes errors or failures, not just individual inputs.

What can be partitioned?

- Usually it is the input data that is partitioned.
- However, depending on the software unit to be tested, output data can be partitioned as well.
- Each partition shall contain a set or range of values, chosen such that all the values can reasonably be expected to be treated by the component in the same way (i.e. they may be considered ‘equivalent’).

Recommendations on defining partitions

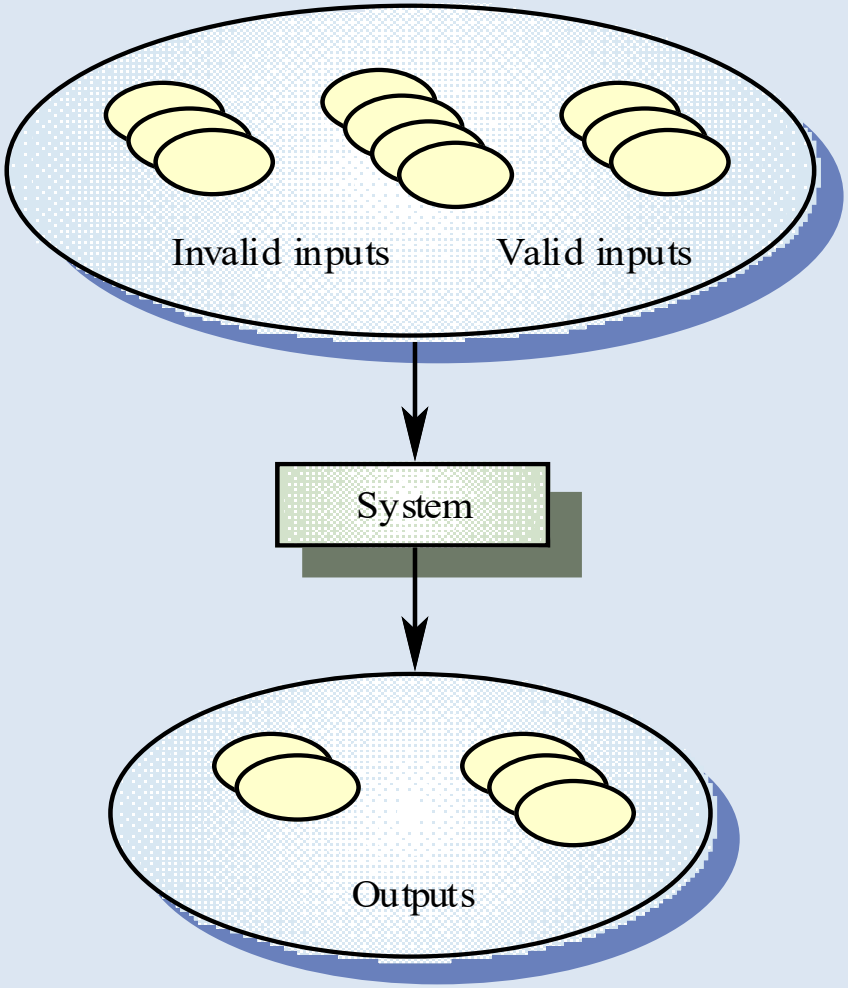
A number of items must be considered:

- All valid input data for a given condition are likely to go through the same process.
- Invalid data can go through various processes and need to be evaluated more carefully. For example:
  - a blank entry may be treated differently than an incorrect entry,
  - a value that is less than a range of values may be treated differently than a value that is greater,
  - if there is more than one error condition within a particular function, one error may override the other, which means the subordinate error does not get tested unless the other value is valid.

Equivalence partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

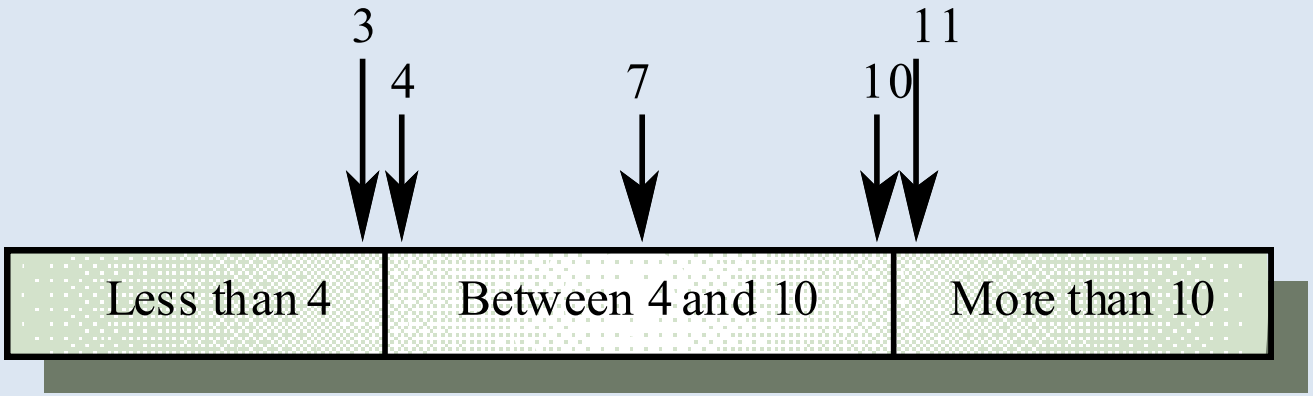
Equivalence partitioning



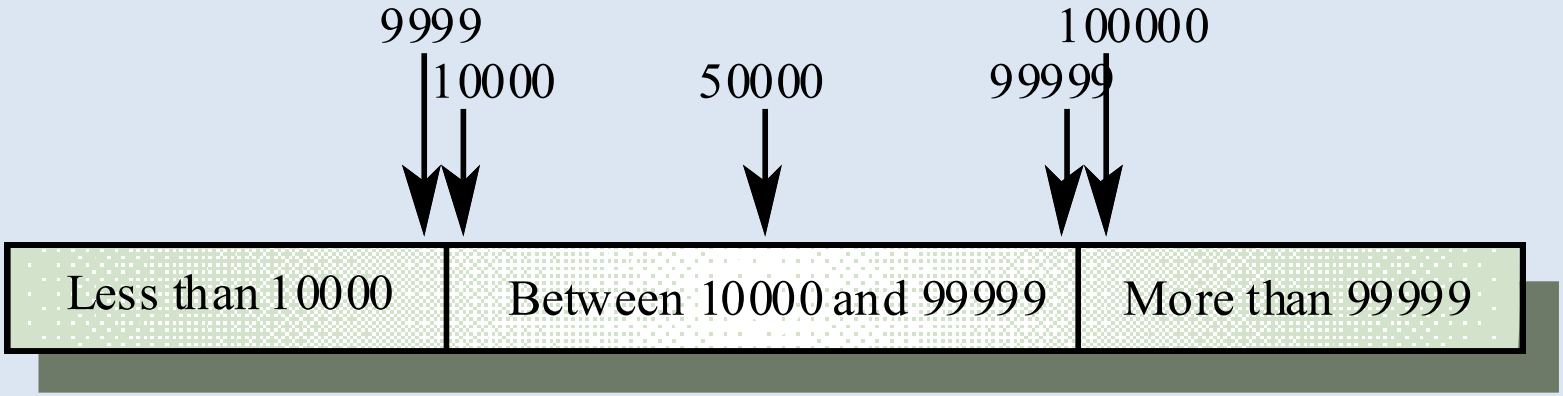
Equivalence partitioning

- Partition system inputs and outputs into ‘equivalence sets’
  - If input is a 5-digit integer between 10,000 and 99,999,
    equivalence partitions are <10,000, 10,000-99, 999 and >
    10, 000
- Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 10001

Equivalence partitions



Number of input values



Input values

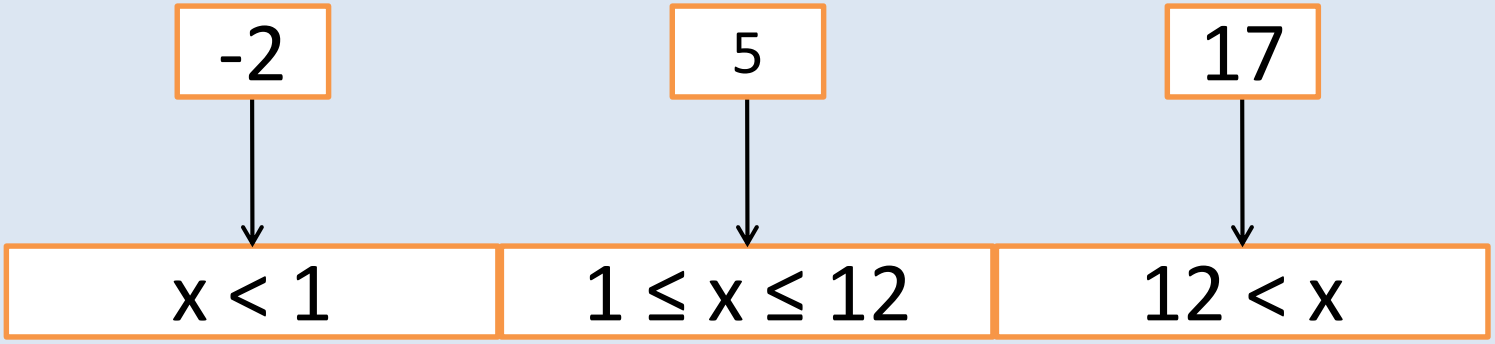
Equivalence partitioning example

- Example of a function which takes a parameter “month”.
- The valid range for the month is 1 to 12, representing January to December. This valid range is called a partition.
- In this example there are two further partitions of invalid ranges.

$x < 1$	$1 \leq x \leq 12$	$12 < x$

Equivalence partitioning example

- Test cases are chosen so that each partition would be tested.



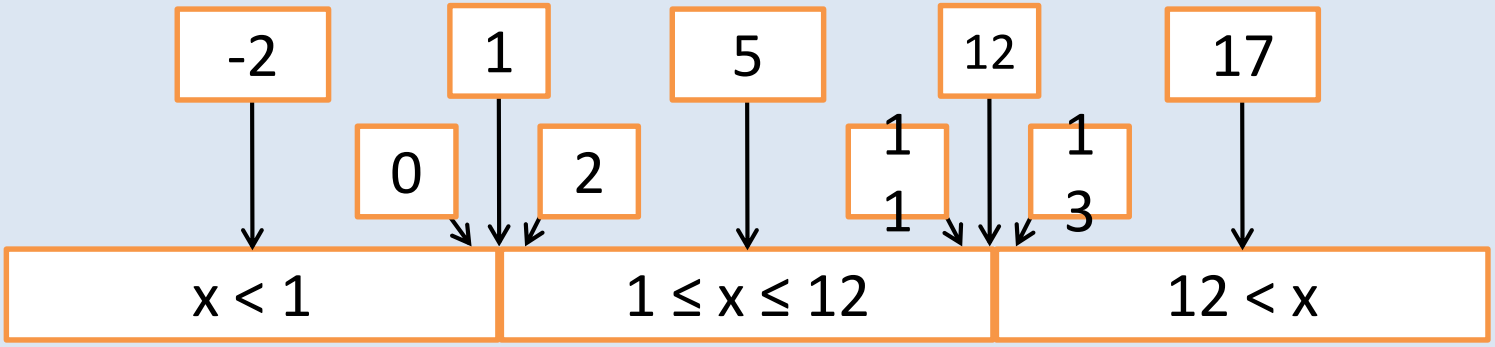
Boundary value analysis

- Equivalence partitioning is not a stand alone method to determine test cases. It is usually supplemented by ***boundary value analysis***.

- ***Boundary value analysis*** focuses on values on the edge of an equivalence partition or at the smallest value on either side of an edge.

Equivalence partitioning with boundary value analysis

- We use the same example as before.
- Test cases are supplemented with ***boundary values***.



Example

- For example consider a program with two input variables `size` and `weight`:
  - valid ranges:

    S1: $0 < \texttt{size} < 200$

    W1: $0 < \texttt{weight} < 1500$
  - corresponding invalid ranges might be:

    S2 : size ≥ 200

    S3 : size ≤ 0

    W2 : weight ≥ 1500

    W3 : weight ≤ 0

Test Cases Example (Traditional View)

Test Case	`size`	`weight`	Expected Output
TE1	100	750	what ever it should be
TE2	100	-1	invalid input
TE3	100	1500	invalid input
TE4	-1	750	invalid input
TE5	200	750	invalid input

Equivalence Test Cases for the Triangle Problem (Output Domain)

- In the problem statement we note that there are four possible outputs:
  - Not a Triangle
  - Isosceles
  - Equilateral
  - Scalene
- We can use these to identify output (range) equivalence classes:

R1= {<a, b, c> | the triangle with sides a, b, c, is equilateral}
R2= {<a, b, c> | the triangle with sides a, b, c, is isosceles}
R3= {< a, b, c> | the triangle with sides a, b, c, is scalene}
R4= {a, b, c> | sides a, b, c do not form a triangle}

- These classes yield the following simple set of test cases:

Sample Test Cases based on Output Domain

Test Case	a	b	c	Expected Output
OE1	5	5	5	Equilateral
OE2	2	2	3	Isosceles
OE3	3	4	5	Scalene
OE4	4	1	2	Not a Triangle

Equivalence Test Cases for the Triangle Problem (Input Domain)

- If we base the equivalence classes on the input domain, we will obtain a larger set of test cases. We can define the sets:

D1= {<a,b,c> | a=b=c}
D2= {<a,b,c> | a=b, a≠c}
D3= {<a,b,c> | a=c, a≠b}
D4= {<a,b,c> | b=c, a≠b}
D5= {<a,b,c> | a≠b, a≠c, b≠c}

- As a separate property we can apply the triangle property to see even if the input constitutes a triangle

D6= {<a, b, c> | a ≥ b+c}
D7= {<a, b, c> | b ≥ a+c}
D8= {<a, b, c> | c ≥ a+b}

- If we wanted also we could split D6 into

D6'={<a, b, c> | a = b+c} and
D6''= {<a, b, c> | a > b+c}

Equivalence Test Cases for the NextDate Problem (Input Domain)

- Nextdate is a function of three variables, month, day, and year and these have ranges defined as:

$$1 \leq \text{month} \leq 12$$
$$1 \leq \text{day} \leq 31$$
$$1812 \leq \text{year} \leq 2012$$

- We will examine below the valid, invalid equivalence classes, strong, and weak equivalence class testing.

Traditional Test Cases

- The valid equivalence classes are:

M1= {month | $1 \leq$ month ≤ 12}
D1= {day | $1 \leq$ day ≤ 31}
Y1= {year | $1812 \leq$ year ≤ 2012}

The invalid equivalence classes are:
M2= {month | month < 1}
M3= {month | month > 12}
D2= {day | day < 1}
D3= {day | day > 31}
Y2= {year | year < 1812}
Y3= {year | year > 2012}

These classes yield the following test cases, where the valid inputs are mechanically selected from the approximate middle of the valid range:

Traditional Test Cases

Case ID	Month	Day	Year	Expected Output
TE1	6	15	1912	6/16/1912
TE2	-1	15	1912	Invalid
TE3	13	15	1912	Invalid
TE4	6	-1	1912	Invalid
TE5	6	32	1912	Invalid
TE6	6	15	1811	Invalid
TE7	6	15	2013	Invalid

Choice of Equivalence Classes

- If we more carefully chose the equivalence relation, the resulting equivalence classes will be more useful

M1= {month | month has 30 days}
M2= {month | month has 31 days}
M3= {month | month is February}
D1= {day | $1 \leq$ day ≤ 28}
D2= {day | day = 29}
D3= {day | day = 30}
D4= {day | day=31}
Y1= {year | year = 1900}
Y2= {year | $1812 \leq$ year ≤ 2012 AND year $\neq 1900$ AND (year = 0 mod 4}
Y3= {year | $1812 \leq$ year ≤ 2021 AND year $\neq 0$ mod 4}

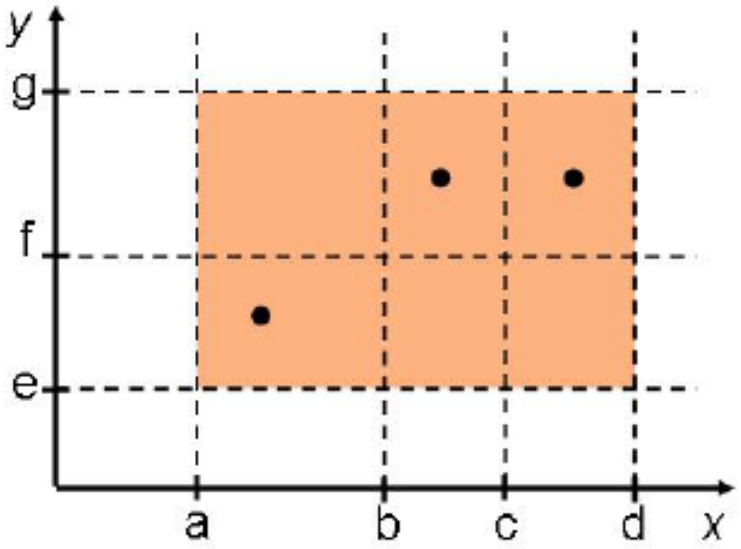
Types of Equivalence Class Testing

- There are four types of equivalence class testing
  1. Weak Normal equivalence class testing
  2. Strong Normal equivalence class testing
  3. Weak Robust equivalence class testing
  4. Strong Robust equivalence class testing

- Normal ECT : only valid entries are considered while determining the test cases
- We consider invalid values in Robust ECT.
- Weak ECT adopts the single fault assumption
- Strong adopts the multiple fault assumption

Weak EC

- Define equivalence classes on the domain (range) of input (output) for each variable: (independent input)
- cover equivalence classes for the domain of each variable: single fault assumption
- how many test-cases are needed?
- also called: (equivalence, category) partition method

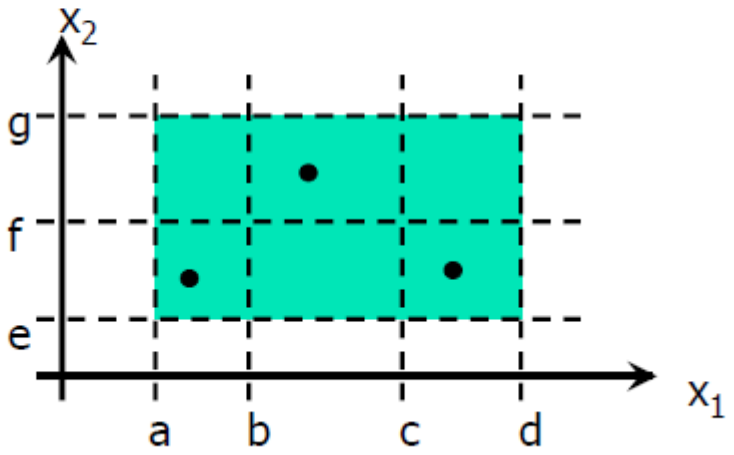


Weak EC

- **What is the number of test cases for weak-normal testing?**

Number of test cases =

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



Cont'd…

- What is the minimal number of tokens that are needed to be put in an m n grid such that each row and column contains at least one token?

Cont'd…

- What is the minimal number of tokens that are needed to be put in an m n grid such that each row and column contains at least one token?
- max(m,n):
  - Put token number i at
  - (max(i ;m); max(i ; n)).
- how many test-cases are needed?
  - $\max_x |S_x|$

Example- Statement

Spec. Write a program that takes
three inputs: gender (boolean), age([18-55]), salary ([0-10000])
and output the total mortgage for one person

Mortgage = salary * factor,
where factor is given by the following table.

Category	**Male**	**Female**
Young	(18-35 years) 75	(18-30 years) 70
Middle	(36-45 years) 55	(31-40 years) 50
Old	(46-55 years) 30	(41-50 years) 35

Weak EC Testing

Category	Male	Female
Young	(18-35 years) 75	(18-30 years) 70
Middle	(36-45 years) 55	(31-40 years) 50
Old	(46-55 years) 30	(41-50 years) 35

- age: difficult!
- salary: [0-10000]
- male: as strange as boundary value!

Weak EC Testing

Category	Male	Female
Young	(18-35 years) 75	(18-30 years) 70
Middle	(36-45 years) 55	(31-40 years) 50
Old	(46-55 years) 30	(41-50 years) 35

- age: difficult! [18-30], [31-35], [36-40], [41,45], [46-50], [51-55]
- salary: [0-10000]
- male: as strange as boundary value! true, false

Weak EC Testing

if (male) **then return**

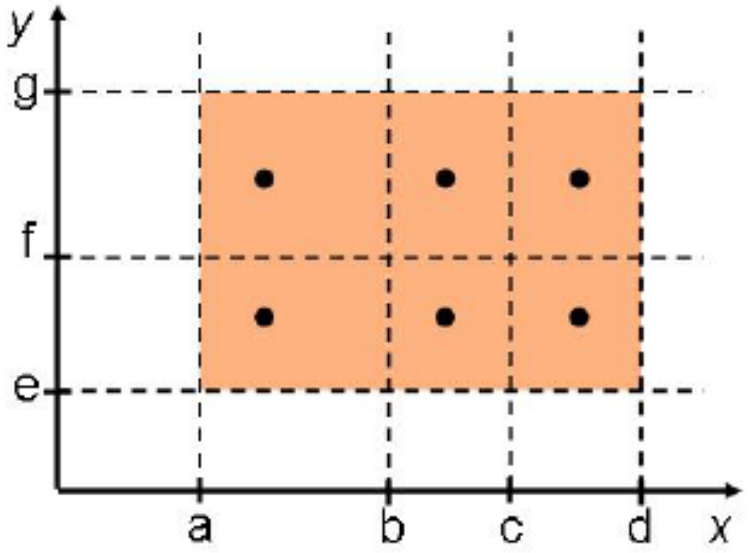
$((18 \leq age < 35)?(75 * salary) : (31 \leq age < 40)?(55 * salary) : (30 * salary))$

else return $((18 \leq age < 30)?(75 * salary) : (31 \leq age < 40)?(50 * salary) : (35 * salary))$

Gender	Age	Salary	Output	Correct Out.	Pass/Fail
male	20	1000	75*1000	75*1000	P
female	32	1000	50*1000	50*1000	P
male	38	1000	55*1000	50*1000	P
female	42	1000	35*1000	35*1000	P
male	48	1000	30*1000	30*1000	P
female	52	1000	35*5000	too late!	F

Strong Normal EC Testing

- cover the all combinations of equivalence classes for the domain of all variables:
  multiple fault assumption
- number of test-cases? $\prod_x \mid S_x \mid$

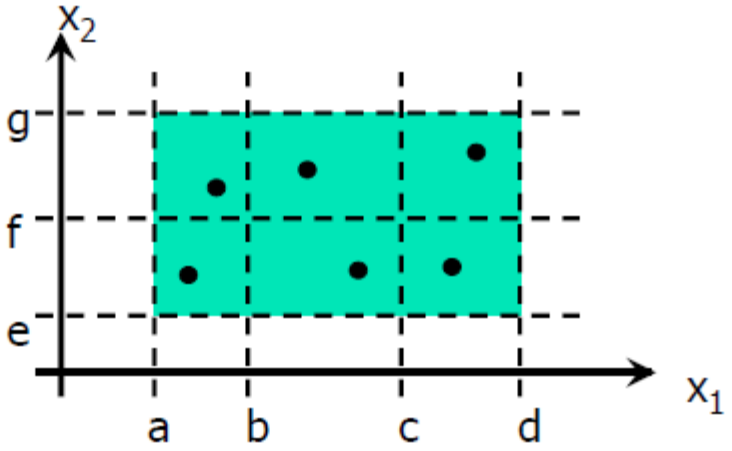


Strong Normal EC Testing

- **What is the number of test cases for strong-normal testing?**

Number of test cases =

$\times$ / [[v : 1 .. #variables • number_equivalence_classes ($variable_v$)]]



Strong Normal EC Testing

Category	Male	Female
Young	(18-35 years) 75	(18-30 years) 70
Middle	(36-45 years) 55	(31-40 years) 50
Old	(46-55 years) 30	(41-50 years) 35

- age: [18-30], [31-35], [36-40], [41,45], [46-50], [51-55]
- salary: [0-10000]
- male: true, false

Strong Normal EC Testing

if (male) **then return**

$((18 \leq age < 35)?(75 * salary) : (31 \leq age < 40)?(55 * salary) : (30 * salary))$

else return $((18 \leq age < 30)?(75 * salary) : (31 \leq age < 40)?(50 * salary) : (35 * salary))$

Gender	Age	Salary	Output	Correct Out.	Pass/Fail
female	20	1000	75*1000	70*1000	F
female	32	1000	50*1000	50*1000	P
female	38	1000	50*1000	50*1000	P
female	42	1000	35*1000	35*1000	P
female	48	1000	35*1000	35*1000	P
female	52	1000	35*5000	too late!	F

Strong Normal EC Testing

if (male) **then return**

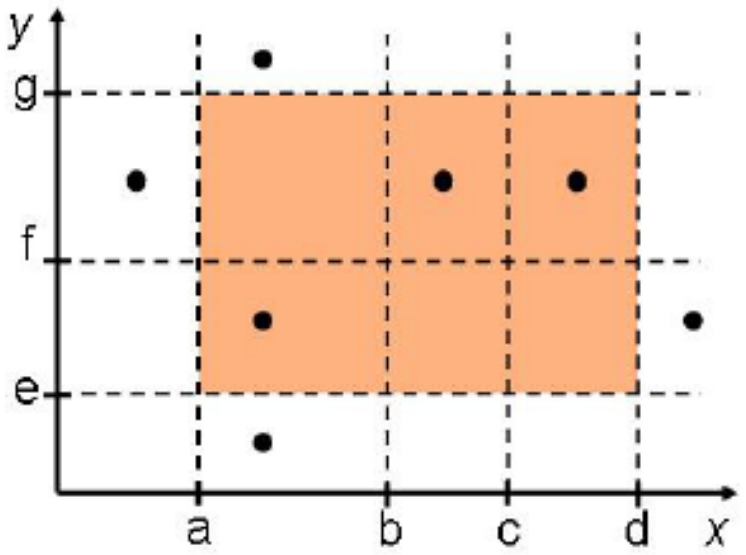
$((18 \leq age < 35)?(75 * salary) : (31 \leq age < 40)?(55 * salary) : (30 * salary))$

else return $((18 \leq age < 30)?(75 * salary) : (31 \leq age < 40)?(50 * salary) : (35 * salary))$

Gender	Age	Salary	Output	Correct Out.	Pass/Fail
male	20	1000	75*1000	75*1000	P
male	32	1000	50*1000	75*1000	F
male	38	1000	55*1000	50*1000	P
male	42	1000	30*1000	55*1000	F
male	48	1000	30*1000	30*1000	P
male	52	1000	30*1000	30*1000	P

Weak Robust EC

- includes weak normal; adds out of range test-cases for each variable
- number of test-cases?
  $(\max_x \mid S_x \mid) + 2 * n$



Weak Robust EC

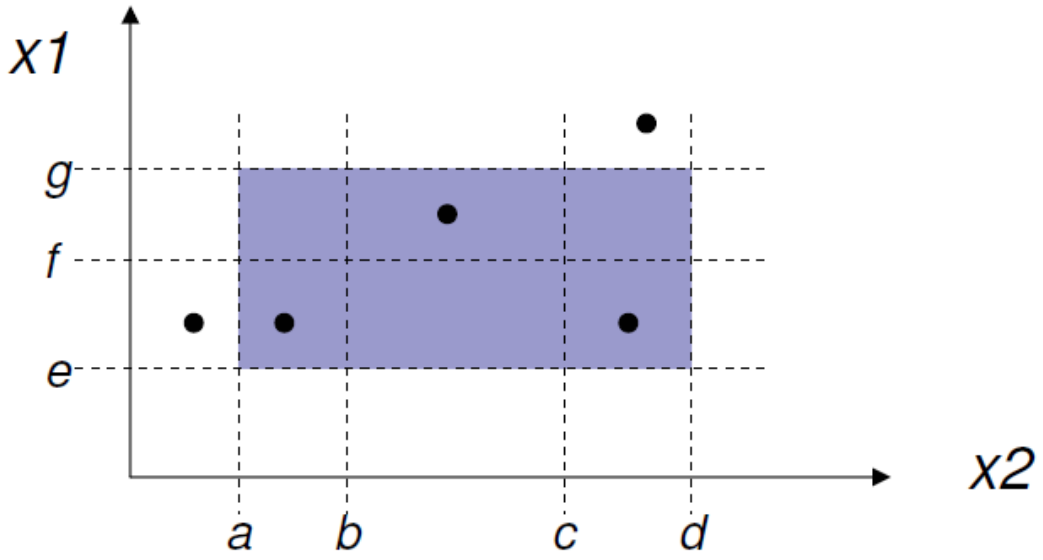
- **What is the number of test cases for weak-robust testing?**

Number of test cases =

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

+

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



Weak Robust EC Testing

if (male) **then return**

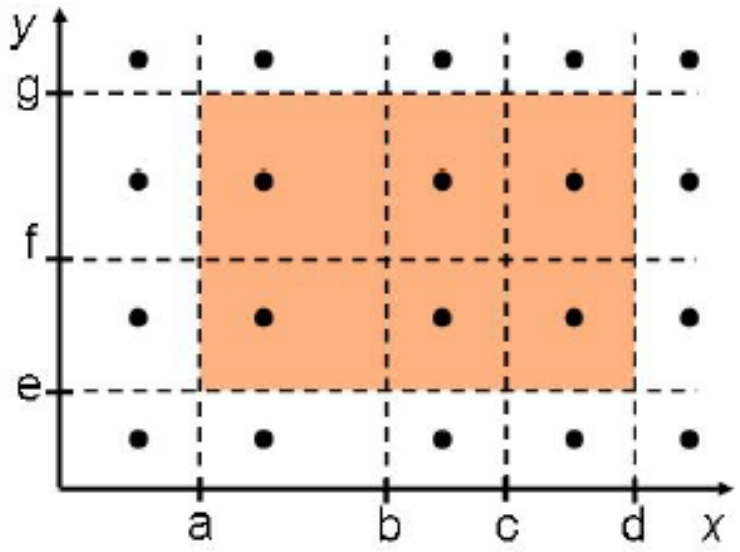
$((18 \leq age < 35)?(75 * salary) : (31 \leq age < 40)?(55 * salary) : (30 * salary))$

else return $((18 \leq age < 30)?(75 * salary) : (31 \leq age < 40)?(50 * salary) : (35 * salary))$

Gender	Age	Salary	Output	Correct Out.	Pass/Fail
male	17	1000	30*1000	too young!	F
female	56	1000	35*1000	too late	F
male	36	-1	55*-1	0	F
female	36	10001	50*10001	50*10000	F

Strong Robust EC

- Same as strong normal but also checks for all out of range combinations
- number of test-cases?
  $\prod_x(|\ S_x\ | +2)$

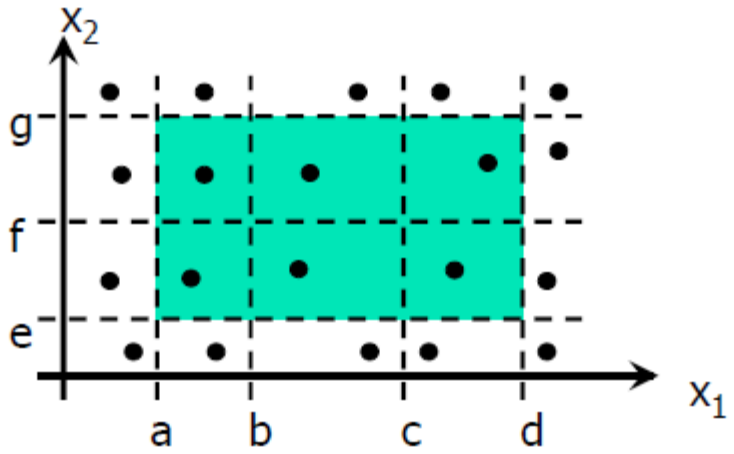


Strong Robust EC

- **What is the number of test cases for strong-robust testing?**

Number of test cases =

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



Strong Robust EC

if (male) **then return**

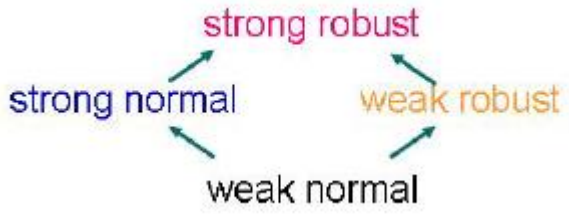
$((18 \leq age < 35)?(75 * salary) : (31 \leq age < 40)?(55 * salary) : (30 * salary))$

else return $((18 \leq age < 30)?(75 * salary) : (31 \leq age < 40)?(50 * salary) : (35 * salary))$

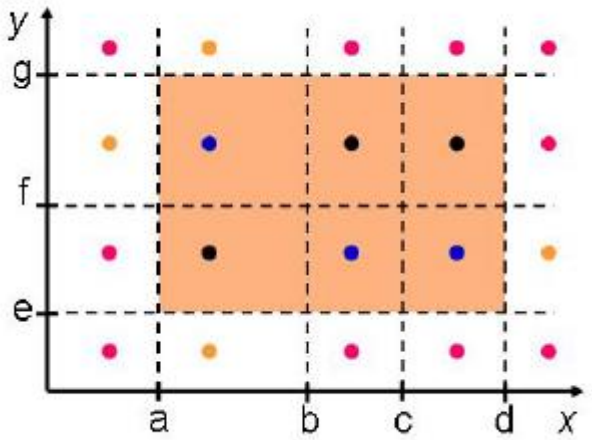
Mostly similar faults to Weak Robust EC:

Gender	Age	Salary	Output	Correct Out.	Pass/Fail
male	17	1000	30*1000	too young!	F
female	56	1000	35*1000	too late	F
female	17	1000	35*1000	too young!	F
male	56	1000	30*1000	too late	F
male	36	-1	55*-1	0	F
female	36	10001	50*10001	50*10000	F

A Brief Comparison



$A \rightarrow B$: Test-cases of A (faults detected by A) is a subset of those of B.



References

- Roger S. Pressman. *Software Engineering: A Practioner's Approach* (Sixth Edition, International Edition). McGraw-Hill, 2005.

- Ian Sommerville. *Software Engineering* (Seventh Edition). Addison-Wesley, 2004.