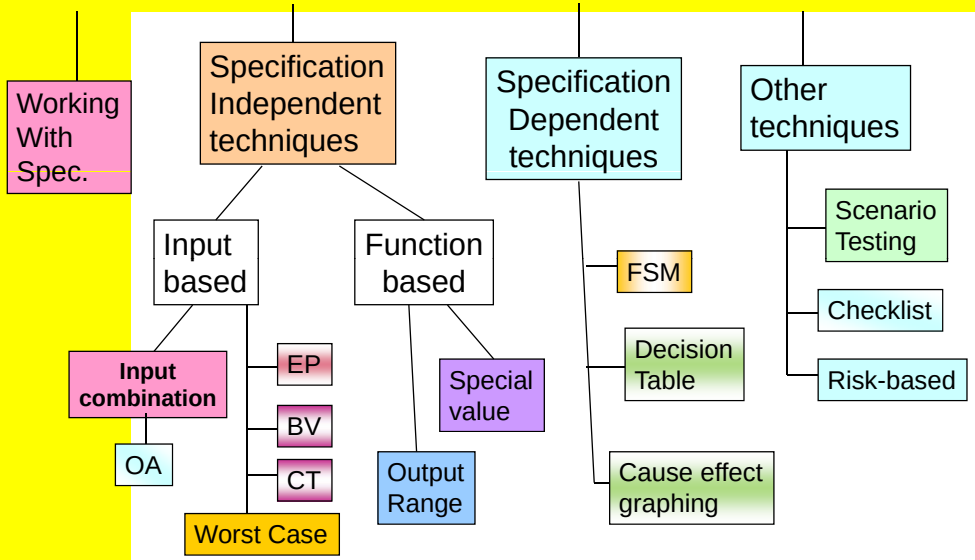
Course Structure

1. Software Quality Assurance
2. Testing Fundamentals
3. Code-based Techniques
4. Specification-based Techniques-Part I
5. Inspection Technique
6. Test Tools
7. Measuring Software Quality
8. TDD

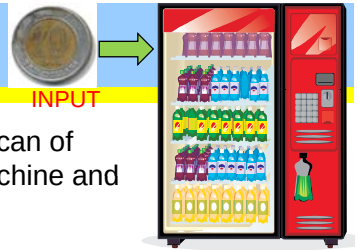
Spec.-based Testing



Learning Objectives

- understand the characteristics of specification-based testing, its advantages and disadvantages;
- select test cases using specification-based techniques:
  - input domain testing
  - Orthogonal array testing
  - equivalence partitioning testing
  - boundary value testing
  - classification tree
  - worst case testing
  - special value testing
  - output range
  - state machine testing
  - decision table testing
  - cause effect graphing
  - scenario testing
  - statistical testing
  - risk-based testing

Example Black Box



For a soda vending machine, to buy a can of Coke, we insert money into the machine and get our soda.

From testing perspective:
- The coins that we insert into the machine are INPUT.
- The soda is an OUTPUT.
- The buttons to select type of soda and the slot where we insert our coins are the USER INTERFACE (UI).
- The internals of the vending machine is like a **black box**, because we have no idea about the concrete mechanism that exchanges coins for soda.

What is Spec-based Testing?

Also called **Black-box testing**, or **functional testing**.

- done without any knowledge of the program
- attempts to derive sets of inputs that will fully exercise all the ***functional requirements*** of a system.

Can't see the code

Identify test cases from the specification.

Spec-based Testing

Black-box testing can be applied at unit test, integration test, and system test level.

At unit testing, each unit provide some services (or functions, such as sorting a list of numbers). We can apply black-box testing without looking at the source code.

At system testing, we look at higher level functions of the whole system. Again, no looking at source code.

Attempts to find these errors:
1 incorrect or missing functions,
2 interface errors,
3 errors in data structures, external database access,
4 initialization and termination errors.

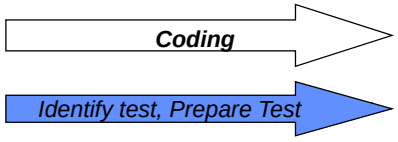
Specification-based Testing

Advantages:

☺ Independent of the implementation. If implementation changes, no need to change the test cases.

☺ Identify faults missed by code-based testing.

☺ Test case development can occur in *parallel* with code development.



Disadvantages:

☹ Difficulties in deriving tests:
- Spec. expressed in an untestable fashion
- Spec. has no structure and no sense of ordering
- Spec. may be ambiguous and imprecise due to the use of natural language

☹ Parts of the code may not be fully exercised by black-box tests, e.g., a true/false condition in the specification may be implemented with many LOC.

☹ Cannot test "hidden" function implemented in the code, but not specified in the spec..

A Common Situation

At UAT
Developer: "Here is the system built according to your specification."

User: "This isn't the software we need!"

Requirement defects are the most disastrous and costly!

High Quality Spec. is Critical

- The most costly errors are those made early in the development cycle
- Misunderstandings about the requirements specification lead to early mistakes
- Programmers need the specification to tell what is needed
- Also, maintenance actions must be based on requirements specification.

"Better specification enable better testing, and better testing requires better specifications" Anonymous

Common Problems with Spec.

⌘ Missing description (**incomplete**)
Does not cover ALL
Feature set, Use cases, Usage scenarios
⌘ Contradictory description (**inconsistent**)
⌘ Confusing **(ambiguous)** –multiple meaning
Force the developers to make assumption on their own! *Their assumption may be different from that of the client/user.*

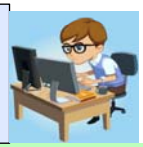
Some Facts about requirements:

⌘ Half of the features provided in most software are never used.
⌘ More than half of the effort on most software systems is wasted.
⌘ 80% of requirements errors found in system testing are a result of incorrect facts about the requirements or omitted requirements.

An Example Specification from a real project

"The exception information will be in the ABC file, too."

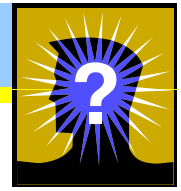
Programmer's interpretation:
"Another *place* the exception information appears is the ABC file."



But, what the customer mean:
"Another type of information that appears in the ABC file is the exception information."

In fact, this information was not duplicated elsewhere. The customer's "too" led to the loss of valuable and unrecoverable information!

Exercise 1



Identify some of the problems with the following specification:

1. For 90% of cases, the response time of on-line request should be less than 1 second and less than 4 seconds for the rest.
2. For 90% of cases, the response time of wildcard search must be within 10 seconds.
3. Write a program that inputs a sequence of integers and outputs the sorted version of this sequence.

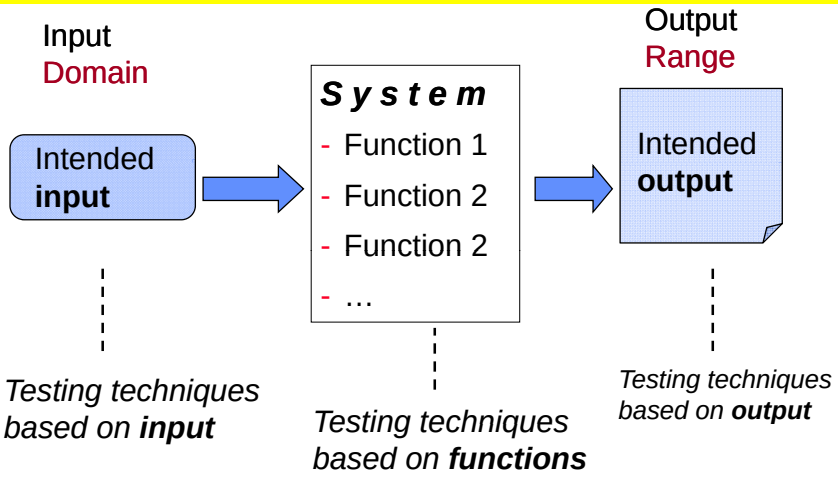
"We should take full responsibility for failing to meet our customer's needs.
For a retail store, it should take full responsibility if it fails to satisfy the customer with its product. It can't blame the customer for failing to clearly express his or her needs."
Al Davis

System Specifications

- System spec. consists of functional spec. and non-functional spec.
- Functional specifications can be formal (mathematical), or informal (in a natural language like English)
- Functional specification usually contains at least 3 kinds of information:
  - The intended **inputs**
  - The corresponding intended **actions**
  - The corresponding intended **outputs**

Spec-based Testing mainly identify test cases for testing functional spec.

When we Test . . .



3 kinds of Black-box testing methods

Based on the 3 kinds of information:

- **Input coverage tests**, based on an analysis of the *intended inputs*, independent of their actions or outputs
- **Output coverage tests**, based on an analysis of the *intended outputs*, independent of their inputs or actions
- **Functionality coverage tests**, based on an analysis of the *intended actions/functions*, with or without their inputs and outputs.

Classification of Spec-based Testing (1)

Testing techniques that are Independent of specification technique:

Based on the input

- ✓ Input domain testing
- ✓ Equivalence partitioning (等价类划分)
- ✓ Boundary value analysis (边界值分析法)
- ✓ Worst case testing

Based on the Function to be Computed (or Output)

- ✓ Special-value testing
- ✓ Output domain coverage

Classification of Spec-based Testing (2)

Testing techniques that are Dependent on the specification technique:

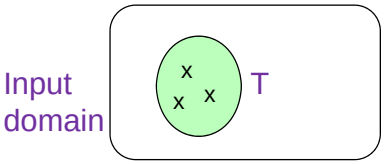
- ➢ Algebraic
- ➢ Axiomatic
- ✓ State Machines
- ✓ Decision Tables
- ✓ Cause-effect

We will not cover algebraic and axiomatic testing as they require math. background knowledge that is outside the scope of this course.

The specification must be provided in certain format. Otherwise, these testing techniques cannot be used.

Test Selection Problem

- The input domain of a program consists of all possible inputs that could be taken by the program.
- Ideally, the test selection problem is to select a **subset T** of the input domain such that the execution of T will reveal all errors.
- In practice, the test selection problem is to select a subset of T within budget such that it reveals as many errors as possible.



Input Domain Testing

Exhaustive or combinatorial testing (组合测试):

- Test every possible data value, valid and invalid, in every **possible combination**.
- Good for data-driven application: order-processing, form-based, transaction-based applications.

Problems

☹ Impossible to test all combinations
☹ May overlook rare combinations, miss obscure paths
☹ Generally covers only the 'expected' inputs

Too Many Combinations

- **Combinations of configuration parameter values**
  For example, telecoms software may be configured to work with different types of call (local, long distance, inter), billing (caller, phone card, 800), access (ISDN, VOIP, PBX), and server for billing (Win Server, Linux/MySQL, Oracle).
- **Combinations of input data values**
  For example, a word processing application may allow the user to select 10 ways to modify some highlighted text: subscript, superscript, underline, bold, italic, strikethrough, emboss, shadow, small caps, or all caps. With 10 binary inputs, 1,024 combinations (2^{10}).
- **Combinations of test sequences**

➔ For example, in mobile application, there are many steps and each step has many choices.
☹ If there are k choices in each step, after n steps there are k x k x .. x k = $\mathbf{k^n}$ different test sequences.
☹ It is not practical to test all these sequences!!

Combinatorial Example: Find window

The Find window takes 3 inputs:

- **Find what** : a text string
- **Match case:** yes or no
- **Direction:** up or down

We simplify this by considering only 3 values for the text string, say "Lee", "Yip" and "Chan".

Combinatorial Example

1. How many combinations of these 3 variables are possible?
   - **Find what** has 3 values (Lee, Yip, Chan) (L Y C)
   - **Match case** has 2 values (Yes / No) (Y N)
   - **Direction** has 2 values (Up / Down) (U D)

2. List ALL the combinations of these 3 variables (total 12 cases)

L Y U	Y Y U	C Y U
L Y D	Y Y D	C Y D
L N U	Y N U	C N U
L N D	Y N D	C N D

Ref. Cem Kaner & James Bach's notes

Test Cases

	Input			Output
TC	Find what	Match case	Direction	Expected result
1	Lee	yes	up	find Lee
2	Yip	no	down	find Yip
3	abc	yes	up	'invalid input'

Another Example: Ordering Pizza

$6x2^{17}x2^{17}x2^{17}x4x3x2x2x5x2$
= WAY TOO MUCH TO TEST

Simplified pizza ordering:

6x4x4x4x4x3x2x2x5x2
= 184,320 possibilities

Still too many cases to test!

Ref: From slides of Rick Kuhn and Raghu Kacker, Combinatorial Testing

Reducing the test cases

Simplified pizza ordering:

6x4x4x4x4x3x2x2x5x2
= 184,320 possibilities

2-way tests:	32
3-way tests:	150
4-way tests:	570
5-way tests:	2,413
6-way tests:	8,330

If all failures involve 5 or fewer parameters, then we can have confidence after running all 5-way tests.

Ref: From slides of Rick Kuhn and Raghu Kacker, Combinatorial Testing

How to minimize the number of test cases?

Try to sample from the input space in way that assures a certain level of combination coverage.

1. Test the most common or important combination, and then vary one or more parameters for the next test.
   All singles: *all individual valid values tested*
   For the **Find window** example, use 3 test cases:
   (L, Y, U)
   (M, N, D)
   (C, Y, U)
2. Test all pair-wise combinations (given any 2 parameters, every combination of values of these two parameters are covered in at least one test case)
   All pairs (2-way) → *all pairs of valid values*

   For the ordering Pizza example: All pairs=> 32 test cases

How to minimize the number of test cases?

3. ***All triples (3-way)*** → *all triplets of valid values*
   For the ordering Pizza example,
   All 3-way=> 150 test cases

4. ***All N-tuples*** → *all combinations of valid values*
   (N is the number of input parameters.)
   Every possible combination of values of the parameters must be covered
   Too many, *Not practical!*

   For the ordering Pizza example,
   All N-tuples=> $6x2^{17}x2^{17}x2^{17}x4x3x2x2x5x2$ test cases!

Example 2-way Interaction Fault in code

The fault occurs when: pressure < 10 & volume > 300 (2-way interaction)

This fault is also called **double-mode fault**.

```
if (pressure < 10) {
     // do something
     if (volume > 300)  {faulty code!  BOOM!}
     else { good code, no problem}
}
else {
     // do something else
}
```

Error detection rates for interaction of 1 to 6

Results showed that across a variety of domains, all failures could be triggered by a maximum of 6-way interactions

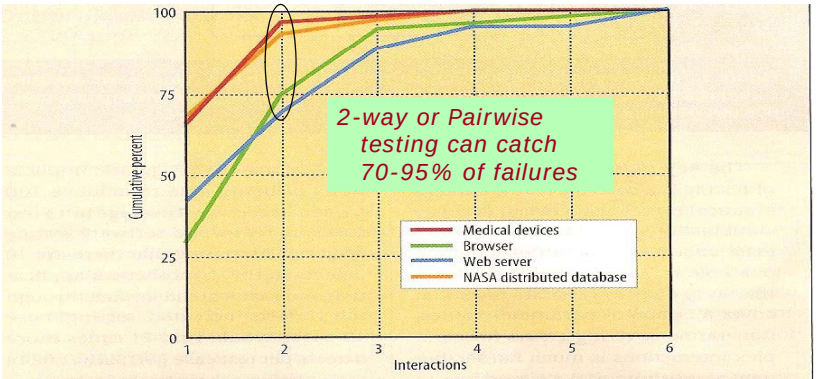


Figure 2. Cumulative error detection rate for fault-triggering conditions. Many faults were caused by a single parameter value, a smaller proportion resulted from an interaction between two parameter values, and progressively fewer were triggered by three-, four-, five, and six -way interactions.

Orthogonal Array (正交表)Testing

(Test all-pairs combinations, i.e. 2-way interactions)

- **Pairwise testing** method finds **double-mode faults** : 2 parameters conflicting with each other.

Example of double-mode fault: *one parameter overshadowing another, inhibiting required processing of that other parameter*

(1) when input is control-C, the return key will not work.

(2) When the font is Times and the menus are on the right, the tables don't line up properly.

- Most defects **are not** a result of complex interactions such as, *"When the background is blue and the font is Times and the images are large and … then the tables don't line up properly."*

Terminology

Orthogonal Array. Two-dimensional arrays that possess the interesting quality that by choosing any 2 columns in the array we receive an *even distribution* of all the pair-wise combinations of values in the array.

OAs. Written as OA(N, k^s, t), or OA(N, k^s) if its strength is two. OA with N runs, k parameters, s levels, and strength t.

Runs. number of rows that are the potential test cases.

Parameters. number of columns that are parameters of interest.

Levels. The number of options or values for each parameter.

Strength. The number of columns it takes to see each option equally often. (For pairwise combination, the strength is 2)

- The number of runs (or test cases) is dependent on the number of **parameters**, **levels**, and **strength**.
- Example, OA(9,4^3) means we have 9 runs (**test cases**) required to cover 4 parameters with 3 options each (see example on next page).
- Since the strength is assumed 2, this OA covers all pairs of parameters.
- OA(64,6^4,3) means we have 64 runs required to cover all three-way combinations (strength 3) of six test parameters with four options each.

OA(9, 4^3)

Each parameter has 3 values

Number of input parameters

	A	B	C	D
1	0	0	0	0
2	0	1	1	2
3	0	2	2	1
4	1	0	1	1
5	1	1	2	0
6	1	2	0	2
7	2	0	2	2
8	2	1	0	1
9	2	2	1	0

← 4 parameters: (A, B, C, D)

9 test cases

Each parameter has 3 options (possible values), represented by 0, 1, 2.

The above OA covers every pair of A-B, every pair of A-C, every pair of A-D, every pair of B-C, every pair of B-D, and every pair of C-D.

Orthogonal Array Testing Strategy (OATS)

Steps:

1. Decide the number of parameters to test.
2. Decide which options to test for each parameter (how many values).
3. Find a suitable OA with the smallest number of runs to cover all parameters and options. (http://www.research.att.com/~njas/oadir/)
4. Map the parameters and options onto the array.
5. A given test situation may not need all the parameters or all the options that a particular OA provides.
   Choose values for any options that are not needed (call them **leftovers**) from the valid remaining options and delete any columns that are not needed.
6. Transcribe the runs into test cases, adding additional combinations as needed.

Many OAs have been published

- See: **http://www.research.att.com/~njas/oadir/**
- If there is not a specific OA for our test situation, we can use one that is **similar**, delete extra columns, and choose values for the extra options consistent with our test situation.

$OA(12, 11^2)$

11111111111
01011100010
00101110001
10010111000
01001011100
00100101110
00010010111
10001001011
11000100101
11100010010
01110001001
10111000100

$OA(18, 7^3)$

0000000
1111110
2222220
0012120
1120200
2201010
0102211
1210021
2021101
0220111
1001221
2112001
0121022
1202102
2010212
0211202
1022012
2100122

Example of Applying OATS

We need to test a system that run on 2 types of CPU, 2 different browsers, 2 operational network modes: internal intranet and modem remote, and 3 types of employee.

CPU	Browser	Network	Type of Employee
Brand X	IE	Internal	Salaried
Brand Y	Firefox	Modem	Hourly
			Part-Time
2	2	2	3

All combination requires 2x2x2x3=24 tests

Example

Use the following steps to identify a minimal set of tests to cover all pairs.

1. **Decide the number of parameters to test. We chose the four parameters shown in previous slide.**
2. **Decide which options to test for each parameter. The options for 4 parameters are listed in previous slide.**
3. **Find a suitable OA with the smallest number of runs to cover all parameters and options. A suitable OA is $OA(9,4^3)$. A suitable array has at least the number of parameters and options with a minimum of leftovers.**

   First 3 parameters each have one leftover option**!**

Example

4. Map the parameters and options onto the array. The leftovers are highlighted below (shaded).

CPU Browser Network Employee

	A	B	C	D
1	Array			
2	A	C	D	E
3	0	0	0	0
4	0	1	1	2
5	0	~~2~~ 0	~~2~~ 0	1
6	1	0	1	1
7	1	1	~~2~~ 1	0
8	1	~~2~~ 1	0	2
9	~~2~~ 0	0	~~2~~ 0	2
10	~~2~~ 1	1	0	1
11	~~2~~ 0	~~2~~ 0	1	0

0=Brand X
1=Brand Y

0=Salaried
1=Hourly
2=PT

Example

5. Choose values for any leftovers from the valid remaining options and delete any columns that are not needed.
   We assign alternating values for each parameter.

	A	B	C	D
1	Array			
2	A	C	D	E
3	0	0	0	0
4	0	1	1	2
5	0	0	0	1
6	1	0	1	1
7	1	1	1	0
8	1	1	0	2
9	0	0	1	2
10	1	1	0	1
11	0	0	1	0

Example

5a. Delete any rows that have no unique pairs. (i.e., test case which tests pair of values that are already tested by other test cases)

	A	B	C	D
1	Array			
2	A	C	D	E
3	0	0	0	0
4	0	1	1	2
5	0	0	0	1
6	1	0	1	1
7	1	1	1	0
8	1	1	0	2
9	0	0	0	2
10	1	1	0	1
11	0	0	1	0
12				

Row 11 contains no new pairs in the row. That is, values for (A,B)=(0,0) have been covered by row 3; values for (A,C)=(0,1) have been covered by row 4, values for (B,C)=(0,1) have been covered by row 6, etc.
This row can be deleted.

Reanalysis after deleting a row

After deleting row 11, reanalyze the array to produce the Table below.

	A	B	C	D
1	Array			
2	A	C	D	E
3	0	0	0	0
4	0	1	1	2
5	0	0	0	1
6	1	0	1	1
7	1	1	1	0
8	1	1	0	2
9	0	0	0	2
10	1	1	0	1
11				

This 0 was selected arbitrary earlier. We can choose 1 instead of 0 here.

Example

5b. Rearrange cell values to create a row with no unique pairs. If this cannot be done, then quit; otherwise, go back to step 5a.
Row 10 shows that the B:D columns pair is unique (only one occurrence). Move that pair combination to row five by assigning the value one to B5 (*note that this is the spreadsheet cell location*) as shown in previous Table.

	A	B	C	D	E
1	Array				
2	A	C	D	E	
3	0	0	0	0	
4	0	1	1	2	
5	0	1	0	1	
6	1	0	1	1	
7	1	1	1	0	
8	1	1	0	2	
9	0	0	0	2	
10	1	1	1	1	
11					

The results of reanalysis after reassigning cell B5

Example

Continuing steps 5a and 5b produces the results shown below with 6 resulting test cases.

	A	B	C	D
1	Array			
2	A	C	D	E
3	0	0	0	0
4	0	0	1	2
5	1	0	1	1
6	1	1	1	0
7	1	1	0	2
8	0	1	0	1
9				

6 **test cases** rather than 24.

6. Transcribe the runs into **test cases**, adding combinations as needed.

A	C	D	E
CPU	Browser	Network	Type of Employee
Brand X	IE	Internal	Salaried
Brand X	IE	Modem	Part-Time
Brand Y	IE	Modem	Hourly
Brand Y	Firefox	Modem	Salaried
Brand Y	Firefox	Internal	Part-Time
Brand X	Firefox	Internal	Hourly

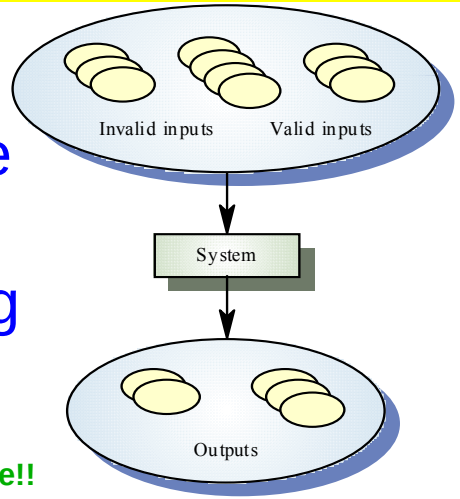
Summary

- Empirical research suggests that all software failures caused by interaction of few parameters (between 2 to 6)
- OA is useful for
  - ✓ integration testing of software components,
  - ✓ web e-commerce systems,
  - ✓ communication protocols, and
  - ✓ configuration testing

- Not useful
  - Small number of parameters, where exhaustive testing is possible
  - No interaction between parameters, so interaction testing is pointless.

Testing tool:
- From NIST called ACTS, http://csrc.nist.gov/acts
- http://www.pairwise.org/

Equivalence Partitioning (EP) Testing



All Inputs are not alike!!

Adapted from Ian Sommerville "Software Engineering"

Example: Square Root Specification

To calculate the square root of a real number:
Input: real number
Output: real number
(1) For an input of 0 or greater, the positive square root of the input shall be returned.
(2) For an input of less than 0, a value of 0 will be returned and output 'illegal input'.

Values<0 will be rejected, ≥0 will be accepted and process.

There are 2 input classes:

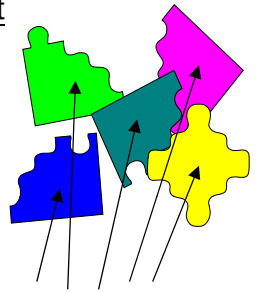
all possible values that are <0

all those that are ≥0

From the point of view of the requirements, all values within such a class are ***equivalent*** (but not equal).

Equivalence Partitioning Testing (1)

Partition: a collection of mutually disjoint subsets whose union is the entire set.
(A set of **equivalent classes**)



Why important?

1. disjointedness assures a form of **non-redundancy**
2. the entire set implies a form of **completeness**

⌘ Elements in the same equivalent class have something in common.

Identify test cases by using one element from each equivalence class.

Equivalence Partitioning Testing (2)

⌘ Identify interesting classes of input, where each class is representative of many possible tests.

⌘ Two tests belong to the same equivalence class if we expect the **same result (pass/fail)** of each. Testing multiple members of the same equivalence class is redundant and waste time!

⌘ EP testing is appropriate

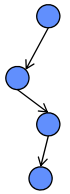
- ✓ when input data is defined in terms of ranges and sets of discrete values (ordered set).
- ✓ when the program function is complex. The complexity of the function can help identify useful equivalence classes

What is *Equivalence*?

– ***Intuitive Similarity***: two test values are equivalent if they are **so similar** to each other that it seems pointless to test both.

– ***Specified As Equivalent:*** two test values are equivalent if the specification says that the system handles them in the same way.

– ***Equivalent Paths:*** two test values are equivalent if they would drive the program down the same path



But, 2 values might take the program down the same path but have very different effects (e.g. word processor's output may be the same but may yield different results from different printers.)

Example Equivalence partitioning

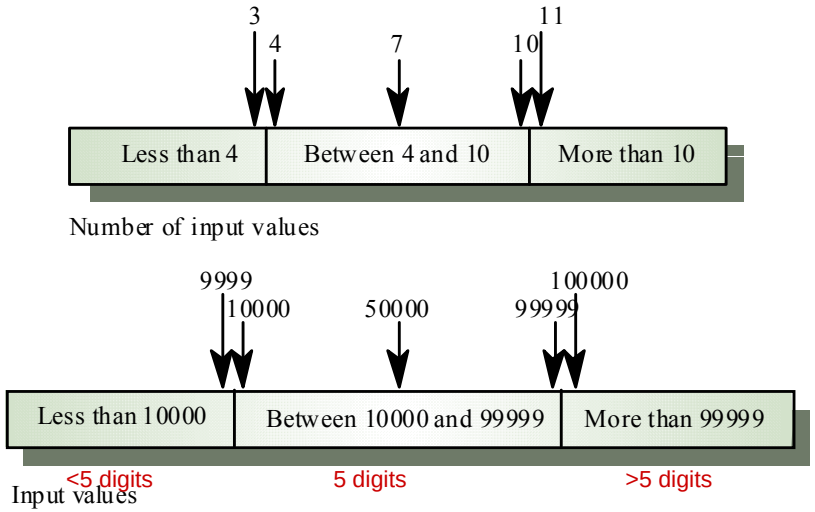
A program accepts 4 to 10 five-digit integer inputs
(e.g., 14617, 92210, 25999, 45411)

- Partition system inputs into 'equivalence sets'
  The equivalence partitions for *five-digit integer* are:
  - *< 10,000 (invalid integer)*
  - *10,000-99,999 (valid integer)*
  - *> 99,999 (invalid integer)*
- Then, we choose one input from each partition:
  - 05000, 59999, 120001

Another partition:

- *Input with **4 to 10** 5-digit integers:*
  (12340, 98980, 23456, 49821, 98765, 22222)
- *Input with **<4** 5-digit integers: (12340, 98980, 23456)*
- *Input with **>10** 5-digit integers:*
  (12340, 98980, 23456, 49821, 98765, 22222, 34341, 56565, 77777, 59090, 67890)

Example Equivalence partitioning



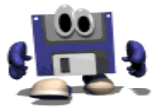
Equivalence Partitioning Testing (3)

⌘ Running one test from each *equivalence class* will avoid wasting time repeating what is virtually the same test.

2 steps of applying EP testing:

(1) Identify equivalence classes (EC)

(2) Identify test cases



1. Identifying Equivalence Classes (1)

➔ if the external input specifies a range of valid values, define 1 valid EC within the range, and 2 invalid EC (outside each end of the range)

E.g. the sale item count can be from 1 to 999

EC: [1-999], [<1], [>999]

TC 1: sale item count = 21

TC 2: sale item count =0 (invalid)

TC 3: sale item count = 1001(invalid)

➔ if the external input specifies the number (N) of valid values, define 1 valid EC and 2 invalid ECs (less than N, and more than N)

E.g. if the input requires 4 book titles

EC: [4], [<4], [>4]

TC 1: 4 book titles (valid)

TC 2: 2 book titles (invalid)

TC 3: 10 book titles (invalid)

1. Identifying Equivalence Classes (2)

➔ if the external input must belong to a group, define 1 valid EC within the set and 1 invalid EC outside the set

e.g.1 Type of vehicle must be {Bus, Taxi, Truck, Cycle}

EC: [Bus, Taxi, Truck, Cycle], [any other values]

TC 1: Taxi (randomly pick a value from the set)

TC 2: Trailer (invalid)

e.g. 2 Enter the name of a country

TC 1: any country name - China

TC 2: input that is not a country name –Earth

➔ if the external input specifies a ***list*** of possible inputs and the program responds differently to each, each input in the list is its own equivalence class. The invalid equivalence class includes any inputs not on the list.

e.g. Program asks "Are you sure? (Y/N)"

EC: [Y], [N], [any other values]

TC 1: Y

TC 2: N

TC 3: X

1. Identifying Equivalence Classes (3)

➔If the external input specifies a 'must be' situation, define one valid EC and one invalid EC.

e.g., The first character of a field must be alpha

EC: [all strings with the first character alpha], [all other strings not start with alpha]

TC 1: hare234 TC 2: 123hare

➔ if the external input items must calculate to a certain value or range, define one valid EC that satisfy the required value or range, and one invalid EC that does not.

e.g., Enter the three angles of a *triangle*

EC: [all 3 values which add to 180], [any 3 values not add to 180]

TC 1: (20, 60, 100)

TC 2: (100, 100, 20) (invalid)

➔ For each external input, if there is reason to believe that elements in an EC are not handled in an identical manner by the program, subdivide the EC into smaller ECs.

➔ This depends on the experience and skill of the tester

⌘ If we know that the system may react in different ways to invalid input, we must build several invalid ECs

2. Identifying Test Cases

⌘ Until all **valid** ECs have been covered by test cases, write a new test case covering as many of the uncovered ECs as possible.

⌘ Until all **invalid** ECs have been covered by test cases, write a test case that covers one, and only one, of the uncovered invalid ECs.

If multiple invalid ECs are tested in the same test case, some of those tests may never be executed because the first test may mask other tests or terminate execution of the test case.

Use more test cases rather than one test case to test many invalid ECs.

Summary of Applying EP (1)

1. For each requirement, determine the inputs to the requirement.
   - Inputs are anything that influences the functionality or the behavior of the requirement, and any kind of data that is needed in order to process the requirement.

Example: **"if a file is to be deleted, the system prompts the user with a confirmation request, where the user can confirm or cancel deletion."**

Answers to this request: "confirm" or "cancel". The answer either causes the file to be deleted or keeps the file. It influences the behavior, so it is an input.

2. If the requirement does not contain all the information necessary to identify its inputs, identify where other relevant information is located (other documents, etc).
   - For each input, divide the input domain into EC.

Summary of Applying EP (2)

3. When it is not possible to create ECs just by looking at one input, because the effect of one input depends on the effect of another input. The requirement is too complex because it contains several requirements.

Break the requirement down into several smaller ones:

- If the requirement consists of several steps. Write down each of these steps.
- Write a separate requirement for each of these steps, using the information from the original requirement.

Example requirement: **The user can choose to delete a file and is then asked to confirm that they really want to delete it.**

How many inputs?

Summary of Applying EP (3)

Two inputs:

- **The user's decision to delete a file**
- **The user's confirmation of that action.**

The behavior depends on the combination of both inputs.
We can identify two steps, corresponding to the inputs:

1. the user can choose to delete a file, and
2. the user has to confirm deletion.

We thus write a requirement for each of these steps:

- **The user can choose to delete a file**
- **if a file is to be deleted, the system prompts a confirmation request, where the user can confirm or cancel deletion.**

EC Example: A Banking Application

The customer can dial the bank from a PC, provide a 6-digit password, and follow with a series of keyword commands that activate the banking function.

Input:

Area code:	blank or 3-digit number
Prefix:	4-digit number, not beginning with 0 or 1
Suffix:	4-digit number
password:	6-character alphanumeric
commands:	'check', 'deposit', 'pay'

EC Example

Input	Type	Note
Area code	boolean	the area code may or may not be present
	range	values between 200 and 999; check that it is in the range.
Prefix	range	specified value >2000
Suffix	value	4-digit length
Password	boolean	password may or may not be present
	value	6-character alphanumeric; check it is in the range.
Commands	set	'check', 'deposit', 'pay'; check if the command is a valid one or not

EC Example

Input	Type	Equivalence Class
Area code	boolean	1. No area code given
		2. Area code given
	range	1. Valid - in range specified
		2. Invalid - greater
		3. Invalid - less than
Prefix	range	1. Valid - between 2000 and 9999
		2. Invalid - >9999
		3. Invalid <2000
Suffix	value	1. Valid - 4-digit number
		2. Invalid - 5-digit number
		3. Invalid - 3-digit number
Password	boolean	1. No password
		2. Password given
	value	1. Valid - 6-character string
		2. Invalid - 5-character string
		3. Invalid - 7-character string
Commands	set	1. Valid - command in command set
		2. Invalid - command not in command set

Example test cases

TC	Area code	Prefix	Suffix	Password	Command
1	852	2766	7252	abc123	'check'
2	604	1345	1234	hello	abc

We combine several ECs together to form a test case.

How to combine these different ECs?

There are 2 ways: **WEC** and **SEC**

2 Types of EC Testing

(1) Weak Equivalence Class (WEC) Testing

- Use one variable from each EC in a test case
- We will always have the same number of weak EC test cases as there are classes in the partition with the largest number of subsets.

(2) Strong Equivalence Class (SEC) Testing

- Based on the Cartesian Product of the partition.
- Achieve 'completeness': we cover all the EC, and we have one of each possible combination of inputs.
- Strong EC testing assumes that the variables are independent when the Cartesian Product is taken. If there are any dependencies, these will generate impossible test cases
- SEC is more **thorough** than WEC.

Example WEC and SEC

Assume that the airfare depends on the age of the passenger, high or low session, and time of the day.
The input consists of 3 values: (*age*, *date*, *time*)

Input equivalent partitions:

Age: [1 to 4], [5 to 18], [19 to 60]
[>60] --- *(4 partitions)*

Date: [April to September],
[Oct. to March],
[Christmas period],
[Easter holiday] --- *(4 parti)*

Time: [10 am to 8pm],
[8pm to 10am] --- *(2 parti)*

Test cases using WEC:
[2, May, 11am]
[6, Oct, 10pm]
[21, Christmas, noon]
[65, Easter, 2am]
Every partition has been tested at least once.
Total test case for **WEC**: 4

Total test case for **SEC**:
4*4*2=32

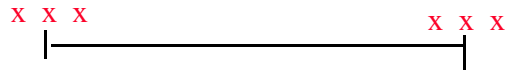
How to reduce this?

Based on spec

Boundary Value Testing

(Counterpart of Domain Testing, which is based on source code)

Boundary Value Testing (1)



- Many errors tend to occur at the **boundaries** of the input domain (e.g., off-by-one defects, forget to handle empty set).
- BV testing complements equivalence partitioning. Rather than select any element in an equivalence class, we select those at the ''edge" of the class.
- We aim to detect “boundary” errors.

Example:
A method M is supposed to compute a function f1 when condition **x <= 0** and function f2 otherwise. However, M has an error such that it computes f1 for **x < 0** and f2 otherwise.

BV Example

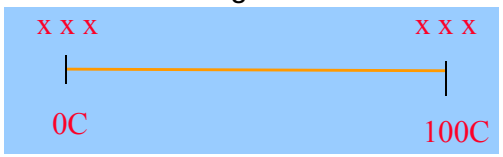
A program simulates water in a chemical plant. Water freezes at 0C and boils at 100C.

=> Meaningful partitions are: (<0), (0-100), and (>100).

If we use EP testing, we will have 3 test cases: one value from each of the three partitions. The reason being that if the program works at 12C, it should also work at 55C.

If we use BV testing, select the following test cases:

-1C, 0C, 1C,
99C, 100C, 101C



Boundary Value Testing (2)

- Boundary conditions may be subtle and difficult to identify.
- BV also incorporates ***negative testing***, as it recommends test cases outside the boundary.

Example: months 0 and 13 are good boundary cases.

Steps:
- Partition the input domain
- Identify the boundaries for each partition
- Select test data such that each boundary value occurs in at least one test input

Identifying BV Test Cases

1. If an input condition specifies a range bounded by values a and b, select the following tests:
   - ➔ a, values just above a, values just below a
   - ➔ b, values just above b, values just below b

   That is, we use values (a-1), a, (a+1), (b-1), b, (b+1).

2. If an input condition specifies a number of values, select tests that exercise the min. and max. numbers. Also, values just above and below min. and max. (That is, (n-1), n, and (n+1)). *See the example on previous page.*

Boundary Value Testing (3)

Advantages:

- –expose any errors that affect an entire equivalence class.
- –expose errors that mis-specify a boundary.
  - coding errors (off-by-one errors) or typing mistakes (such as entering 57 instead of 75 that defines the boundary).
  - Mis-specification result from ambiguity or confusion about the decision rule that defines the boundary.

Disadvantages:

- ☹ Would not work for Boolean variables and logical variables
- ☹ Do not test dependencies among the variables
- ☹ Do not consider combinations of input/output conditions!

BV Example: Business Rule

Business rules for a banking application:

Rule 1. No account balance can be less than *credit limit*.

Test case: try to request a transfer an amount in excess of the current balance plus that credit limit.

Use *boundary analysis*: try 1 cent less than the limit, exactly the limit, and 1 cent above the limit.

If the balance in the account is $1000 and the credit limit is $5000, then try

$5999.99
$6000.00
$6000.01

BV Example: Business Rule

Another Business rules for a banking application:

Rule 2. No more than 10 transfers can be made per day or no orders can be submitted above 100,000.

Test case: generate 10+1 transfer requests with amounts below the maximum amount (99999) and at least one above the maximum amount (100001).

Exercise 2

Consider the following Business rules :

A 5% discount is applied whenever the total purchase is $1000 or more

What test cases to use based on BV testing?

Can you spot the error if we implement Discount like this?

```
Public class Discount {
   public double getDiscount(double
   amount) {
      if (amount > 1000.00) {
            return 0.05 * amount;
      } else {
            return 0.00;
      }
   }
}
```


Domain Testing vs. BV Testing

Program —*Domain Testing*→ White-box test cases testing the Boundary of the input domain as defined by the **program**

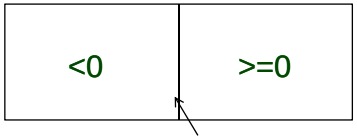
Specification —*Boundary Value Testing*→ Black-box test cases testing the Boundary of the input domain as defined by the **spec**.

Domain Testing is based on source code, rather than the spec

Example

Spec says "2 set of numbers: >=0 and <0"

EC from Spec.

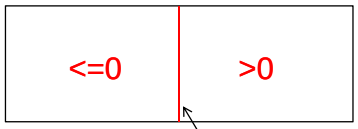


boundary from Spec.

But, the program:

```
if input >0 then
   ....
else .....
```

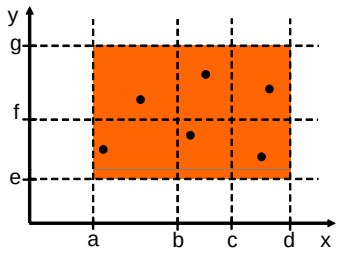
EC from program



boundary from program

EP summary

#tests: small to moderate
Usage: some study of requirements needed
When to use:
- independent inputs
- ordered sets
- when suspecting computational errors
- may easily be combined with BV testing



BV testing summary

#tests: moderate to very many
Usage: straightforward, easy to implement
When to use:
- independent inputs
- ordered sets
- when suspecting boundary errors (such as miscoding of boundaries or ambiguity in definition of the valid/invalid sets.)

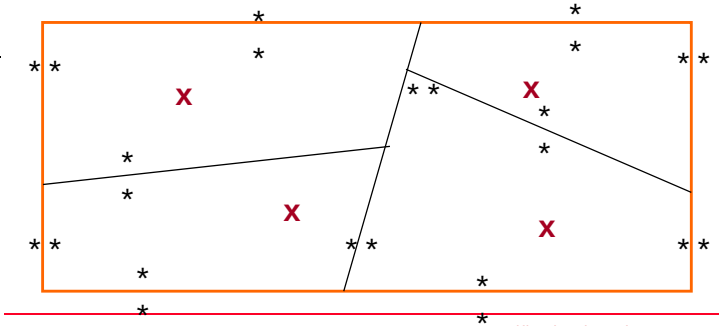
Comparing EP and BV Testing

- BV testing has no recognition of data or logical dependencies. They are easy to automate.
- EP testing pays attention to data dependencies and to the function itself. Thus, more thought is required to identify the EC.
- In an ordered set, ***boundaries*** mark the point or zone of transition from one EC to another.

Assume that domain can be partitioned into 4 EC shown right.

X are the test cases for EP testing

***** are the test cases for BV testing



Classification-tree Method

- ⌘ Aims to identify all the feasible combinations of classes as test cases
- ⌘ Enables systematic test case design– minimize the chance of overlooking some important test cases.
- ⌘ Can apply to informal and formal specification.
- ⌘ Easy to understand from the graphical representation

The size of the classification tree can be a measure of a system's complexity. The number of test cases from the CTM can help to estimate the testing effort and development effort.

Tool available (Classification-Tree Editor CTE XL): www.systematic-testing.com

Classification-tree Method

Classifications are the input parameters and environment condition of the module that will affect its execution behavior. Can look at the interface description (or input to the system)

Classifications are different criteria for partitioning the input domain of the problem to be tested.

p-type: based on explicit input **p**arameter to the system
Example: input of invoice amount (> 0, =0, <0),
input of approval (yes or no)
e-type: do not depend on the input; based on **e**nvironmental condition
Example: the status of Customer Master File (does not exist, exist).

Classes are the non-overlapping subsets of the values of the corresponding category. Given by values or intervals.

Testers will manually create classes for each classification, based on the requirement specification or functional specification.

Classification-tree Method

1. Decompose the specification into modules that can be tested independently.
2. For each module selected for testing, repeat the following steps:
   1. Identify a set of classifications and classes for the module.
   2. Construct a classification tree from the identified classifications and classes
   3. Construct a test case table from the classification tree
   4. Construct test cases:
      - ⌘ Identify all the *feasible combinations* of classes from the test case table. Each combination of classes represents a test case.
      - ⌘ For each test case, identify concrete test values to serve as test data.

Example: TRADE System

A goods trading system, TRADE, shall be developed for the HK Wholesale.

For each credit purchase, the system accepts the customer and purchase details to determine whether this purchase should be approved.

Function of TRADE:

1. Accept the customer number.
2. Check the customer number against the Customer Master File. If the customer number exists, go to step 3. Otherwise, reject the transaction and stop.
3. Check the status of the customer. If it is deactivated, go to step 4. Otherwise, go to step 5.
4. Check whether the transaction is supported by special management approval. If yes. Accept the transaction and stop. Otherwise, reject the transaction and stop.
5. Compare the credit limit and the invoice amount. If the credit limit is less than the invoice amount, reject the transaction. Otherwise, accept the transaction.

Details of customer number, invoice amount, and management approval are manually entered by users.

Other details, such as customer's status and credit limit, are automatically retrieved from the Customer Master File by TRADE.

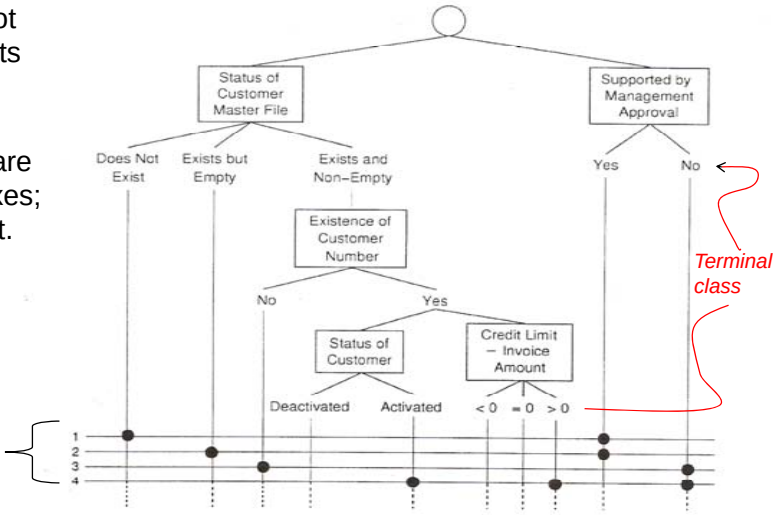
Classification and Classes of TRADE

	Classifications	Associated Classes
e-type	Status of Customer Master File	Does not exist, Exists but empty, Exists and non-empty
p-type	Existence of Customer Number	Yes, No
	Status of Customer	Deactivated, Activated
	Credit Limit – Invoice Amount	< 0, = 0, > 0
	Supported by Management Approval	Yes, No

Part of Classification-tree for TRADE

The top node (root node) represents the whole input domain.

Classifications are enclosed in boxes; **classes** are not.



Drawing the Classification-tree

From CT, test cases can be expressed in the test-case table:

1. Draw the grids of the test-case table under CT. The columns of the table correspond to the terminal nodes of CT. The rows correspond to test cases.
2. Construct a test case in the test-case table by selecting a combination of classes in CT as follows:
   a) Select one and only one child class of each top-level classification
   b) For every child classification of each selected class, recursively select one and only one child class.

How many test cases?

- **Minimum** criterion: each terminal class is include at least once in a test case. (our example: 3)
- **Maximum** criterion: all permitted combinations of terminal classes are included. (our example: 3x2x2x3x2 = 72)

  Note: some combinations may not be possible!! So the feasible combination is normally less than the maximum number.
- **Good rule**: total number of terminal classes gives sufficient test coverage – test all values of terminal classes (our example: 3+2+2+3+2 = 12)

Observations

- More than one correct classification tree may be constructed – depend on the tester

  Which one is better?

  Look at number of classification and classes, ease of understanding, effectiveness in generating valid test cases.
- How to apply CTM to complex specification?

  Decompose it into smaller modules that can be independently tested.
- Incorrect classification are caused by a partial understanding of the problem domain of the specification.
- **Common problems:** overlapping classes, not enough decomposition (divide a class into sub-classes)

Guideline of Using CTM

- For defining classification, think about p-type & e-type.
- For defining classes, think about the "boundary cases."
- Should not construct the CT based on "control-flow" – CT is based is data-oriented, black-box testing.
- Avoid using control-flow information unconsciously for constructing the classification.

- There may be some logical dependencies between classes. Keep this in mind when preparing the test cases.
- Combination Rule: Some classifications and classes may be more important than others. Example: ***size*** and ***form*** are more relevant to testing than ***color*** in a computer vision system. Try to include all possible combinations of ***size*** and ***form***

Classification Tree provides a visual representation of system requirements.

Worst Case Testing

- Try to test when more than one variable has an extreme value.
- More thorough than boundary value testing
- But this requires more effort, as all combinations are tested.
- Similar to the **strong equivalence class** testing in terms of selecting many test cases

Example: test the accounting package with the largest amount and a user name with the longest string.

Functional Testing

A function is something the system can do (also called features / commands)

⌘ Function testing is usually the first black box test to use.

"Test what it can do"

Fundamental goal: Test each function thoroughly, one at a time.

⌘ Demonstrate each required function

⌘ No reference is made to the implementation (i.e., the source code)

Some Function Testing Tasks

(1) Identify each function and sub-function
- From specifications or the draft user manual, design, externally observable behavior, walking through user interface, searching the program for command names

(2) From this information, build the function list

Example: For each category of function *(e.g. Editing)*
- **For Individual functions** *(e.g. Cut, Paste, Delete), identify*
  - Inputs to the function (Variable: try Maximum, Min, special cases)
  - Outputs of the function
  - Possible scope of the function *(e.g. Delete word, Delete paragraph)*
  - Options of the function *(e.g. configure the program to Delete the contents of a row of a table, leaving a blank row)*
  - Circumstances under which the function behaves differently *(e.g. deleting from a word processor configured to track and display changes or not to track changes)*

Some Function Testing Tasks

(3) When building or using the function list:
- Determine how we would know if the function worked.
- Identify variables used by the functions and test their boundaries.
- Identify environmental variables that may constrain the function under test.
- Check that each function does what it's supposed to do (**positive test**) and does not do what it's not supposed to do (**negative test**).

Function Testing Example

How to ensure we adequately test the function?

Step 1: physically partition the functional specification into separate requirements (note: the requirements may not be independent)

Step 2: create test cases for each partitioned requirement.

Example:

"Given as input two integers x and y, output all the numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."

Function Testing Example:
Requirements Partitioning

"Given as input two integers x and y"

R1: accept 2 integers as input.

"output all the numbers"

R2: output zero or more (integer) numbers.

"smaller than or equal to x"

R3: All numbers output must be less than or equal to the first input number.

"*evenly divisible by y"*

R4: All numbers output must be evenly divisible by the second number.

"all the number"

R5: output must contain all numbers that meet both R3 and R4.

"If either x or y is zero, then output zero."

R6: output must be zero (only) in the case where either first or second input integer is zero.

Function Testing Example:
Test case selection

We model each partitioned requirement as independent.

We create separate test cases for each partitioned requirement.

For our example, we choose test cases for R6:

R6 Test1: 0 0 (both zero)

R6 Test2: 0 1 (x zero, y not)

R6 Test3: 1 0 (y zero, x not)

R6 Test4: 1 1 (neither zero)

and other test cases. . .

Primary Uses of Function Testing

- **Check coverage**
  - we check our test plan or set of test cases against the function list, for coverage of every feature, and every option and subfunction of every feature.
- Useful for **initial testing** of the product
  - Learn the capabilities of the product
  - Fast scan for serious problems that should be addressed early
- Best for assessing **capability** rather than reliability

Risks of Function Testing

- **Problems** with using this testing method only :
  - Significant redundancies among test cases (may test the same program path over and over again).
  - Not able to test behavior not specified
  - Misses feature interactions
  - Misses load-related issues, interaction with background tasks, effects of interrupts
  - Often focuses on the features without considering boundaries or other tests of special data
  - Doesn't address user tasks—whether the customer can actually achieve the benefits promised by the program

Special-value Testing

- Tester uses his domain knowledge, experience with similar programs, etc to devise test cases
- Very dependent on the ability of the tester.
- Highly subjective. But, it often results in a set of test cases which is more effective in revealing faults than other methods.
- Select test data on the basis of features of the function to be computed.
- Properties of the function to be computed can aid in selecting input which will indicate the accuracy of the computed solution.

  e.g., Feb. 29 for Y2K problem

Examples of Special-value

Zeroes or nulls: these have the property that they do not generate a change in value for addition or subtraction.

Ones: these do not generate changes in values for multiplication and division.

- Always include these values in our test!

Example: a function that sorts a collection of numbers.

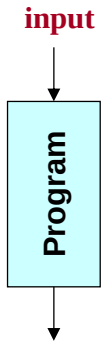
- We should test using

  (1) the empty set

  (2) a set of one item.

Output Range Coverage

- Each function has an associated output range.
- Select test cases that will cause the **extremes** of each of the output ranges to be achieved.
- Check for (1) maximum and minimum output conditions and (2) that all categories of error messages have been tested.
- In general, constructing such test data requires knowledge of the function to be computed and, hence, expertise in the application area.
- Must analyse requirements to figure out what input is required to produce each output - this can be a complex and time consuming analysis.



Output Range Coverage

- This technique can be very effective in finding problems, because it requires a deep understanding of the requirements
- But, this technique is difficult to use.

Example

A program that will print the class list.

Each class can have at most 60 students and no fewer than 10 students.

Test cases:

- Try to print a class of 60 students
- Try to print the class with 10 students.

Output boundary

Exhaustive Output Testing

Have one test for every possible output.

Practical more often than exhaustive input testing, because programs are often written to reduce or summarize input data.

But still impractical in general – most programs have an infinite number of different possible outputs.

Example: The requirements say:
"output 1 if two input integers are equal, 0 otherwise." [only 2 possible output!!]
2 integer inputs – doing input partitioning, we have many test cases:
- Numbers equal
- Numbers not equal
- First number zero/ positive/ negative
- Second number zero/ positive/ negative.

But, we can do exhaustive output testing with only 2 test cases : output 1, output 0.

Output Partition Testing


Output range

- **Output partitioning** is like input partitioning, only we analyse the possible outputs.
- Similar to input partitioning, we partition all the possible outputs into a set of equivalence classes which have something in common.
- Once we have the output partitions, we must design inputs to cause outputs in each class.
- This is a ***difficult and time consuming task***.
- Sometimes, we discover that we cannot find such an input – *this implies an **error** or **oversight** in either the requirements or in the partition analysis.*

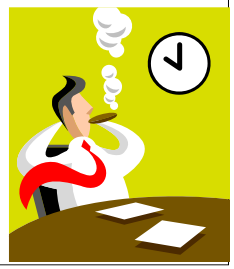
Warnings & Errors

The man is smoking and leaving smoke rings into the air.
His girlfriend gets irritated with the smoke and says:

"Can't you see the warning written on the cigarettes packet, smoking is injurious to health!"

The man replies back:

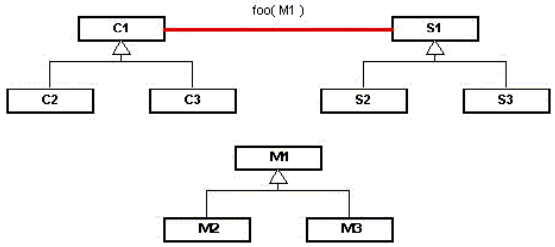
"I am a programmer.
We don't worry about warnings,
we only worry about errors."



Supplementary Notes

Example 2 of OATS (pairwise testing) (1)

Given an object-oriented system that contains a client class (C1) with 2 subclasses (C2 and C3).
These client classes interact with a server class hierarchy consisting of class S1 with subclasses S2 and S3.
The server class contains a method **`foo()`** which takes an instance of class M1 as a parameter. M1 has 2 subclasses, M2 and M3.



Example 2 (2)

To test all combinations of the classes involved, it would take 27 test cases (3 clients that can each send 3 messages to 3 servers -- 3 x 3 x 3 = 27).

- That assumes that the method **`foo()`** can be tested with only a single test case.
- But, in most circumstances, it will take many test cases to test a particular method.

Thus, there are many more test cases in practices!

Example 2 (3)

Using the OATS technique:

1. There are 3 independent variables (the client, the server, and the message class).
2. Each variable can take on 3 values.
3. Ideally, we would use an array that contains three levels and three parameters (an OA(?, 3^3)).

   However, no such published array exists. Therefore, we need to look for the smallest array that will handle our problem.

   OA(9,4^3) will work. It has the three levels for the values and four parameters is more than enough for the three variables.

Example 2 (4)

4. Mapping the values onto the array:
   A. For Client, C1=0; C2=1; C3=2.
   B. For Server, S1=0; S2=1; S3=2.
   C. For Message, M1=0; M2=1; M3=2.

OA(9,4^3)

	A	B	C	D
1	0	0	0	0
2	0	1	1	2
3	0	2	2	1
4	1	0	1	1
5	1	1	2	0
6	1	2	0	2
7	2	0	2	2
8	2	1	0	1
9	2	2	1	0

	Client	Server	Message
Test 1	C1	S1	M1
Test 2	C1	S2	M2
Test 3	C1	S3	M3
Test 4	C2	S1	M2
Test 5	C2	S2	M3
Test 6	C2	S3	M1
Test 7	C3	S1	M3
Test 8	C3	S2	M1
Test 9	C3	S3	M2

5. There are no "left over" Levels.
   But, we notice that there was an extra parameter in OA(9,4^3). This parameter can simply be ignored. We still get an even distribution of the pair-wise combinations.
6. We end up with 9 test cases.

Another Pairwise coverage Example

An application that is intended to run on a variety of platforms comprised of 5 components: an operating system (Windows XP, Apple OS X, Red Hat Linux), a browser (Internet Explorer, Firefox), protocol stack (IPv4, IPv6), a processor (Intel, AMD), and a database (MySQL, Sybase, Oracle),

There are 3 x 2 x 2 x 2 x 2 = 48 possible platforms.

With only 10 tests, it is possible to test every component interacting with every other component at least once, i.e., all possible pairs of platform components.

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHL	IE	IPv6	AMD	MySQL
8	RHL	Firefox	IPv4	Intel	Sybase
9	RHL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

Another EC Example: Phone System

Consider a program which dials a phone given a valid telephone number.

Equivalent classes based on the phone number:

- C1: local call (8 digits)
- C2: long distance (001+3 digits + 7 digits)
- C3: spaces as separators
- C4: hyphens as separators
- C5: () around the area code
- C6: Police call (911)
- C7: Directory assistance (1083)
- C8: too long
- C9: too short
- C10: invalid characters

Example 2 of CT

Computer vision system: determine the size of different objects passing the camera of the system on a conveyor belt. Inputs are various building blocks.

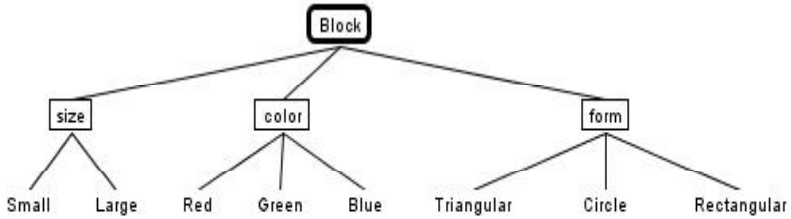
What classification (or aspects)?

Size, Color and Form of the building blocks.

Classification for size: small, large building blocks

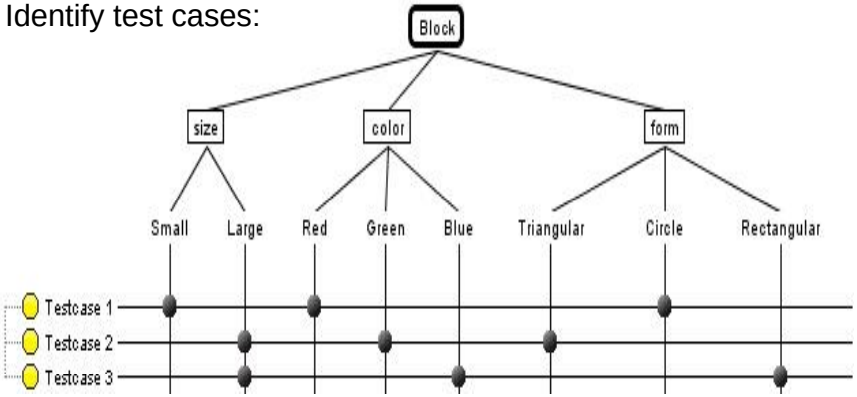
Classification for color: red, green, blue

Classification for form: triangular, circle, rectangular



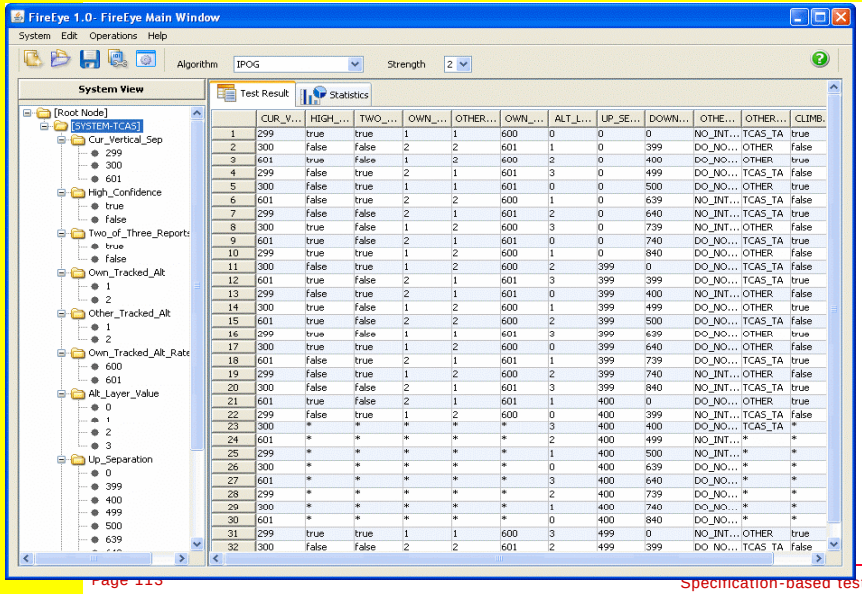
Example 2 of CT

Identify test cases:



We can continue, and have subclass under Triangular class: Shape of Triangle

ACTS Tool



Defining a new system