

# Domain and function testing

# Methods and measures for functional software integration testing

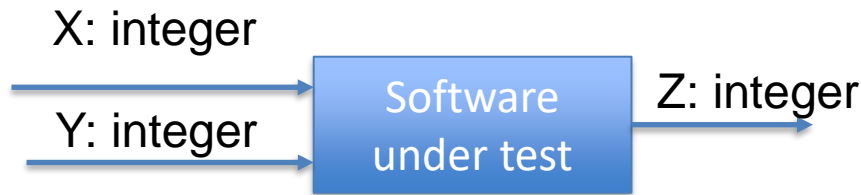
Methods and measures	Criteria	ASIL			
		A	B	C	D
Requirements based test	Test case have to be derived from the requirements. All SW component requirements are covered	++	++	++	++
External interface test	Test cases can be determined with the help of <b>equivalence classes</b> and <b>input partitioning</b> of the external interface. This can be completed by <b>boundary value analysis</b> . Boundary values for data types as well as plausible ranges of values for interface variables have to be considered.	+	++	++	++
Fault injection test	...	+	+	++	++
Error guessing test	...	+	+	++	++

# Outline

## ➤ Single variable

- Multiple variables
- Error guessing/Risk based
- Output coverage and testing

# The challenge



The total number of possible combinations of X and Y is  $2^{32} \times 2^{32}$

Combinatorial explosion!

- We need to **systematically** reduce the set of **all possible values** to few **manageable** subsets

# Partitioning of domain

- Identifying domain first
  - Linear (variables whose values can be mapped onto a number line, e.g., integer variables)
  - Non-linear (e.g., variables with enumeration values)
- Ideally, partitioning of input & output domain should result in non-overlapping or disjoint partitions

# Black-box domain testing approaches

- Identify input and output domains
- Generate test case to cover single variables in the domains and combination of the variables
  - Equivalence classes, boundary values
  - Decision tables, pairwise testing
  - Error guessing/risk-based

# Partitioning of domain of single variable

- Variable types
  - Linear domain variable (e.g., integer)
  - Linear domain variable with multiple ranges
  - String variable
  - Enumerated variable
  - Multidimensional variable

# Partitioning domain of single variable

- Step 1: Identify the variable
- Step 2: Determine the domain
  - All possible values of the variable
- Step 3: Identify risks
- Step 4: Partition the domain into equivalence classes based on the identified risks



# Linear domain variable

- Software application example

BankA issues Visa credit cards with credit limits in the range of \$4000 to \$40000. A customer is not to be approved for credit limits outside this range.

A customer can **apply for the card using an online application form** in which one of the fields requires that the customer type in his/her desired credit limit.

# Linear domain variable (cont')

- Identify the variable: “credit-limit”
- Determine the input domain:  $\$4000 \leq \text{credit-limit} \leq \$40000$
- Identify risks
  - Failure to process credit limit requests **between** \$4000 and \$40000 correctly
  - Failure to disapprove credit limit requests **less than** \$4000
  - Failure to disapprove credit limit requests **greater than** \$40000
  - Mishandling of **negative** credit limit requests
- Partition the input domain into equivalence classes based on risks
  - \$4000, \$40000
  - \$3999
  - \$4001
  - -\$4000

# Equivalence class testing

- Rationale: **complete** testing but **no redundancy**
- **Cover each partition at least once**
- E.g., to test whether software can identify the equilateral triangle
  - After using (5, 5, 5) as test input
  - We do not expect to learn much from using (6, 6, 6), (10, 10, 10) as test inputs

# Linear domain variable with multiple ranges

If Taxable Income Is Between:	The Tax Due Is:
0 - \$9,225	10% of taxable income
\$9,226 - \$37,450	\$922.50 + 15% of the amount over \$9,225
\$37,451 - \$90,750	\$5,156.25 + 25% of the amount over \$37,450
\$90,751 - \$189,300	\$18,481.25 + 28% of the amount over \$90,750
\$189,301 - \$411,500	\$46,075.25 + 33% of the amount over \$189,300
\$411,501 - \$413,200	\$119,401.25 + 35% of the amount over \$411,500
\$413,201 +	\$119,996.25 + 39.6% of the amount over \$413,200

1

# Linear domain variable with multiple ranges (cont')

- Identify risks
  - Failure to calculate the tax correctly for **each of the income sub-ranges**
  - Mishandling of low and high **boundaries of each of the sub-ranges**
  - Mishandling of values just **beneath and beyond low and high boundaries** respectively for each of the sub-ranges

# Equivalence class testing vs. boundary value testing

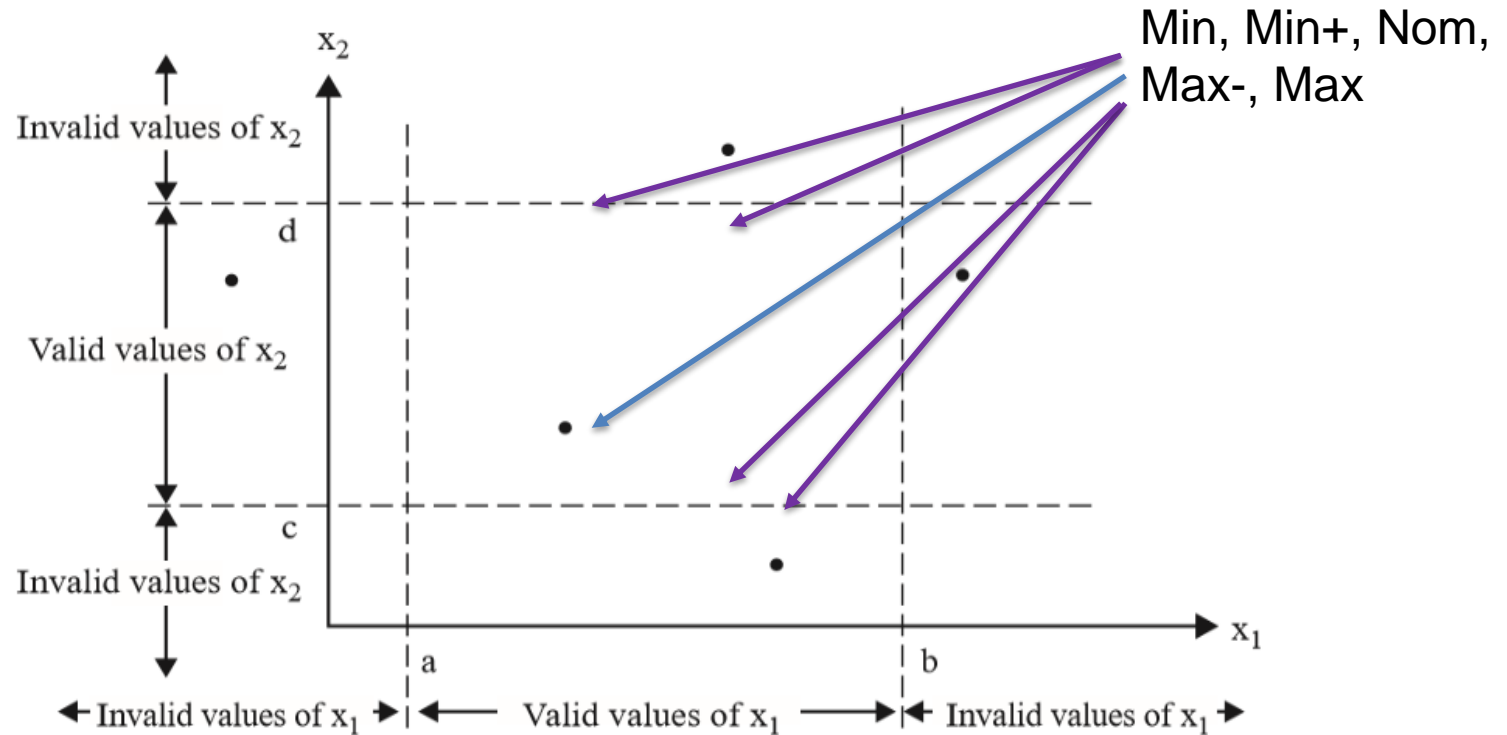


Figure 6.1 Traditional equivalence class test cases.

\*

\* Software Testing, A craftsman's approach, 4<sup>th</sup> Edition

# Boundary value testing

- Rationale: Errors tend to occur near the extreme values of an input variable, e.g.,

- Loop condition, may test for  $<$  when they should test  $\leq$

- The idea is to use input values

Min, min+, nom, max-, max

- Two variables (hold one in nom and vary the other one)

$\langle X_{1nom}, X_{2min} \rangle \langle X_{1nom}, X_{2min+} \rangle \langle X_{1nom}, X_{2nom} \rangle \langle X_{1nom}, X_{2max-} \rangle \langle X_{1nom}, X_{2max} \rangle$

$\langle X_{1min}, X_{2nom} \rangle \langle X_{1min+}, X_{2nom} \rangle \langle X_{1max-}, X_{2nom} \rangle \langle X_{1max}, X_{2nom} \rangle$

- Several variables (hold one in nom and vary the others)

# Robust boundary value testing

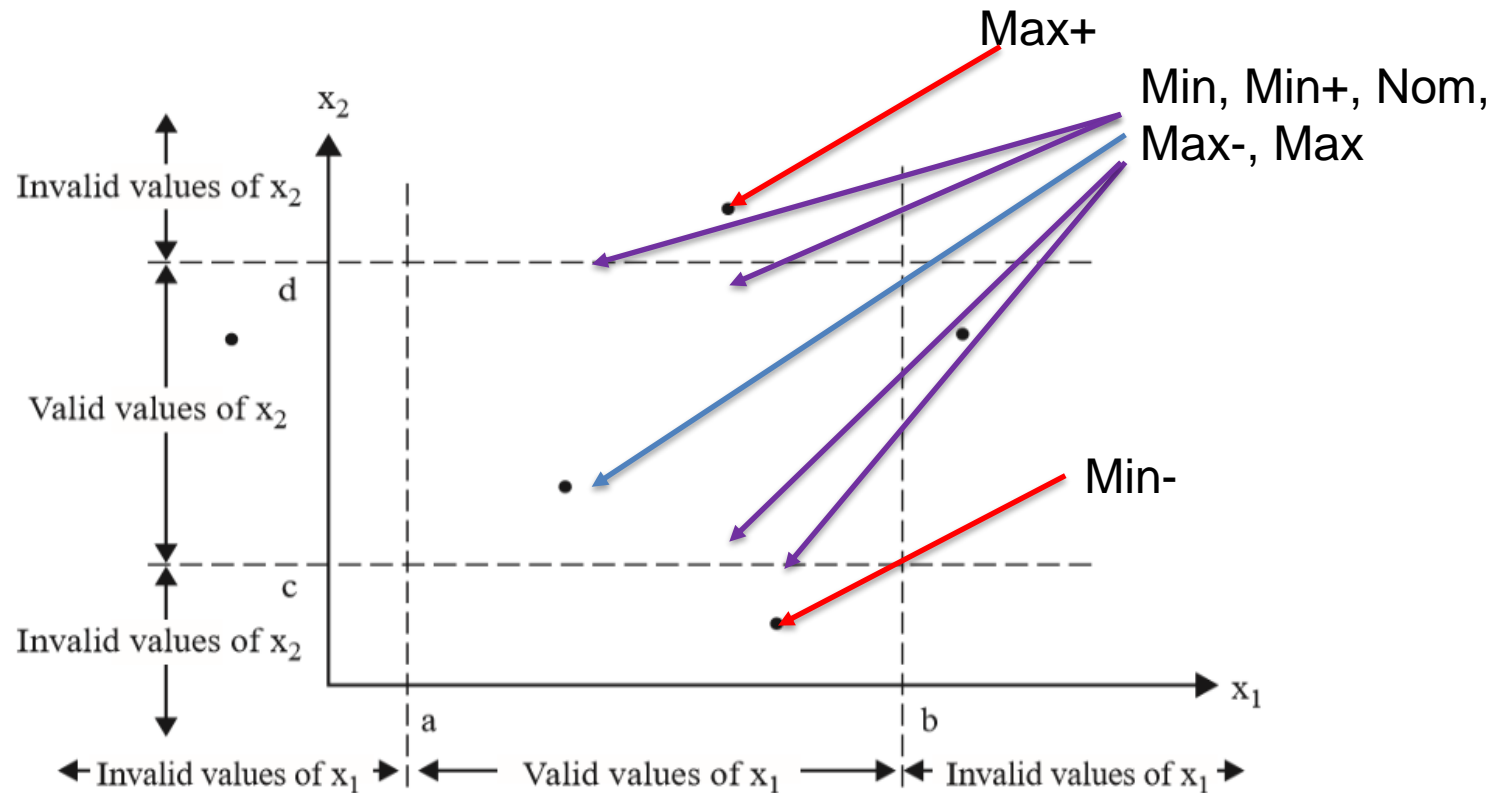


Figure 6.1 Traditional equivalence class test cases.

\*

\* Software Testing, A craftsman's approach, 4<sup>th</sup> Edition



# Special value testing

- Special value testing of the tax calculation system
  - Mishandling of **non-numbers**
  - Mishandling of **negative** incomes
  - Mishandling of **smallest and largest** value at the system level, and beyond
  - ...
- Another special value testing example
  - E.g., test how the system react if **29 Feb.** as the input of a date
- Most intuitive and least uniform (ad hoc)

# String variables

- For example, a string variable 's' has to have a letter for the first character; the rest of them can be any printable graphic ASCII characters.
- Domain
  - First character
    - (A-Z)  $65 \leq \text{ASCII (first character)} \leq 90$
    - (a-z)  $97 \leq \text{ASCII (first character)} \leq 122$
  - Remaining characters
    - $32 \leq \text{printable graphic ASCII characters} \leq 126$

# String variables (cont')

- Identify risks
  - Failure to process strings correctly that have **letters as their first character** (B3, Zb)
  - Failure to process **boundary characters** (A~) (the ASCII code of ~ is 126)
  - Mishandling of string values that have **nonletters** for the first character ([A)
  - Mishandling of characters that belong to the **extended ASCII set** (Aβ)

# Enumerated variable

- A variable which takes only a set of values/set of options

Effects

<input type="checkbox"/> Strikethrough	<input type="checkbox"/> Small Caps
<input type="checkbox"/> Double Strikethrough	<input type="checkbox"/> All Caps
<input type="checkbox"/> Superscript	Offset: 0% <input type="text"/>
<input type="checkbox"/> Subscript	<input type="checkbox"/> Equalize Character Height

- Risks
  - Failure to present font effect correctly that has a Superscript/Subscript/... option selected
  - Mishandling of **no option** selected
  - Mishandling of **multiple** options selected

# Multidimensional variables

- Multidimensional variable

There is more than one dimension of analyzing such a variable

- Software application example

To log in a system, the user must input the correct username and password. The username can have **five to fifteen characters**. Also, only **digits and lowercase characters** are allowed for the username.

- Username variable has two dimensions  
Length and String

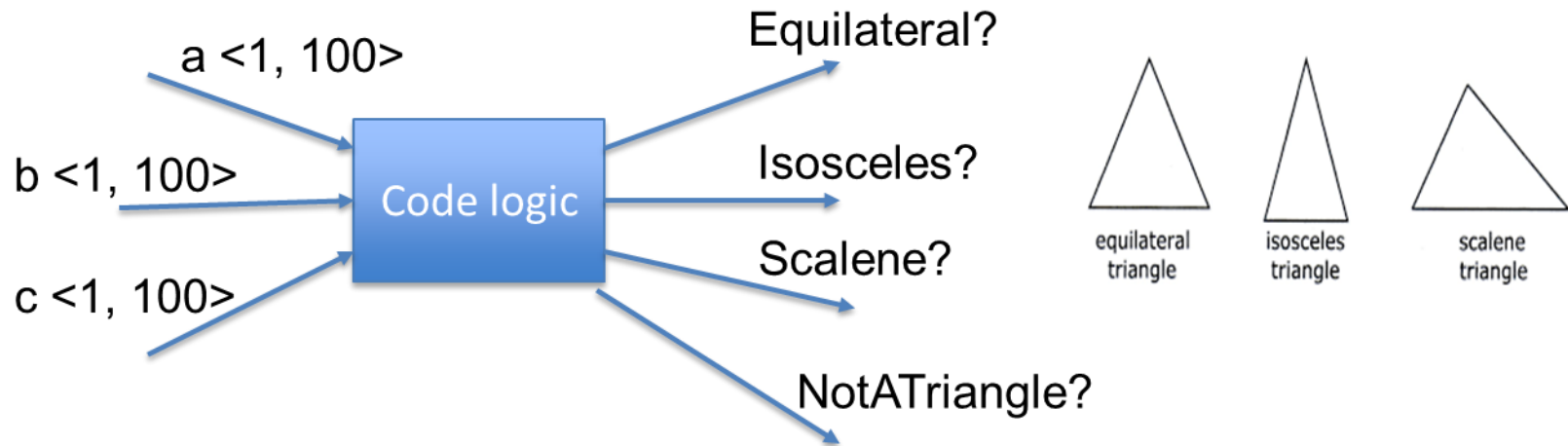
# Menti Exercise

Go to [www.menti.com](https://www.menti.com) and use the code **6438 5825**

# Outline

- Single variable
- **Multiple variables**
- Error guessing/Risk based
- Output coverage and testing

# Multivariable application example (1)



- Boundary value testing  $\langle 1, 0, 1 \rangle \langle 1, 0, 2 \rangle \langle 1, 0, 50 \rangle \langle 1, 0, 99 \rangle \langle 1, 0, 100 \rangle$   
 $\langle 1, 1, 1 \rangle \langle 1, 1, 2 \rangle \langle 1, 1, 50 \rangle \langle 1, 1, 99 \rangle \langle 1, 1, 100 \rangle \dots$
- Equivalence class testing  $\langle 1, 0, 1 \rangle \langle 1, 1, 0 \rangle \langle 101, 0, 1 \rangle \langle 101, 1, 0 \rangle \dots$
- Also need to analyze complex logical relationships of the input variables



# Decision table-based testing (cont')

<4, 1, 2>

<3, 3, 3>

<2, 2, 3>

Table 7.2 Decision Table for Triangle Problem

c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

\*

\* Software Testing, A craftsman's approach, 4<sup>th</sup> Edition

# Why test variable combination?

- All variables will **interact** as they are part of one functional unit and they have to unite to achieve the duty that the functional unit is designated for
- Hence most of these variables influence each other in some, or the other way
- Testing variables in combination may **reveal certain bugs that might not have been found by testing the variables in isolation**

# Multivariable application example (2)

- Online shopping system example
  - Parameters: Availability, Payment Method, Carrier, Delivery Schedule, Export Control

Availability	Payment	Carrier	Delivery Schedule	Export Control
Available	Credit	Mail	One Day	True
Not in Stock	Paypal	UPS	2-5 Working Days	False
Discontinued	Gift Voucher	Fedex	6-10 Working Days	
No Such Product			Over 10 Working Days	

$4 \times 3 \times 3 \times 4 \times 2 = 288$  combinations

# The challenge of testing the online shopping system example

- Suppose there is a bug, and **Credit** does not work well with **One Day delivery**
  - Any combination of inputs that include Credit and a One Day delivery will expose that bug
  - Interaction between 2 variables
- Suppose **Credit** does not work well with a **One Day delivery**, but only with **Fedex**
  - Any combination of inputs that include Credit, a One Day delivery, and not with Fedex will expose that bug
  - Interaction between 3 variables
- Do we really need to test all  $4 \times 3 \times 3 \times 4 \times 2 = 288$  combinations?

# Do we really need to test all combinations?

- The root cause analysis of many bugs shows they depend on the value of one variable (20%-68%)
- Most defects can be discovered in tests of the interactions between the values of two variables (65-97%)

Number of variables involved in triggering software faults\*

Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security
1	66	29	42	68	20
2	97	76	70	93	65
3	99	95	89	98	90
4	100	97	96	100	98
5		99	96		100
6		100	100		

\*<http://csrc.nist.gov/groups/SNS/acts/ftfi.htm>

# What is n-way test?

- Given any **n** variables (out of all the variables) of a system, every combination of values of these **n** variables is covered in at least one test
- For example, an application has 5 variables
  - 2-way test  
Every combination of values of any 2 variables (out of 5 variables) is covered in at least one test
  - 3-way test  
Every combination of values of any 3 variables (out of 5 variables) is covered in at least one test

# What is n-way test (cont')?

- An application with 5 input variables A, B, C, D, E
  - A has values A1, A2
  - B has values B1, B2, B3,
  - C has values C1, C2
  - D has values D1, D2, D3, D4
  - E has values E1, E2
- Full combinations  $2 \times 3 \times 2 \times 4 \times 2 = 96$  test cases
- 2-way test (56 combinations)
  - Every combination of values of any 2 variables
  - AB, AC, AD, AE, BC, BD, BE, CD, CE, DE
  - AB (A1-B1, A1-B2, A1-B3, A2-B1, A2-B2, A2-B3)
  - AC (A1-C1, A1-C2, A2-C1, A2-C2)
  - ...

# All-Pairs combination testing

- 2-way testing
  - Every combination of values of 2 variables is covered in at least one test
- All-pair combination/pairwise testing is condensed 2-way testing
  - Many pairs can be tested at the same time in one combination

A1-B1, B1-C2, C2-D2 -> A1-B1-C2-D2



# All-pair combination/pairwise testing approaches

- In Parameter Order
- Variation of In Parameter Order

# In Parameter Order

- Initialization phase
- Horizontal growth phase
- Vertical growth phase

# Example of In Parameter Order

- Variables
  - X (True, False)
  - Y (0, 5)
  - Z (P, Q, R)

# In Parameter Order - Initialization

- X (True, False)
- Y (0, 5)

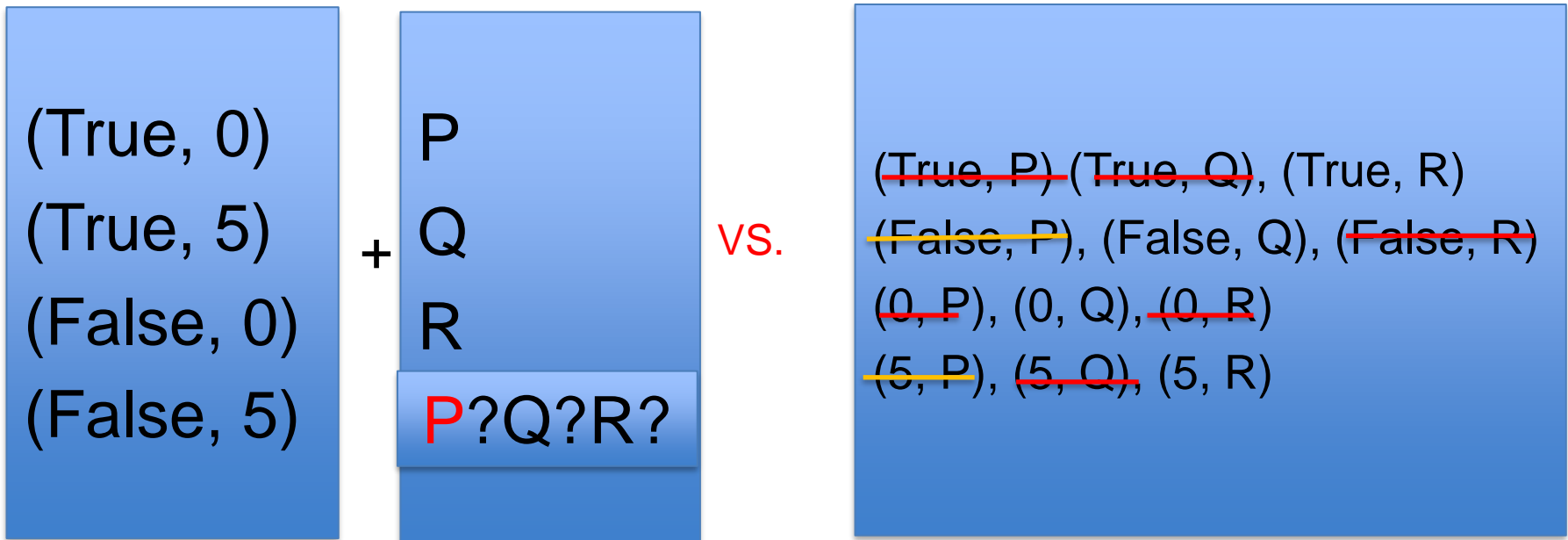
Find any two variables and generate test cases

- T= (True, 0)  
(True, 5)  
(False, 0)  
(False, 5)

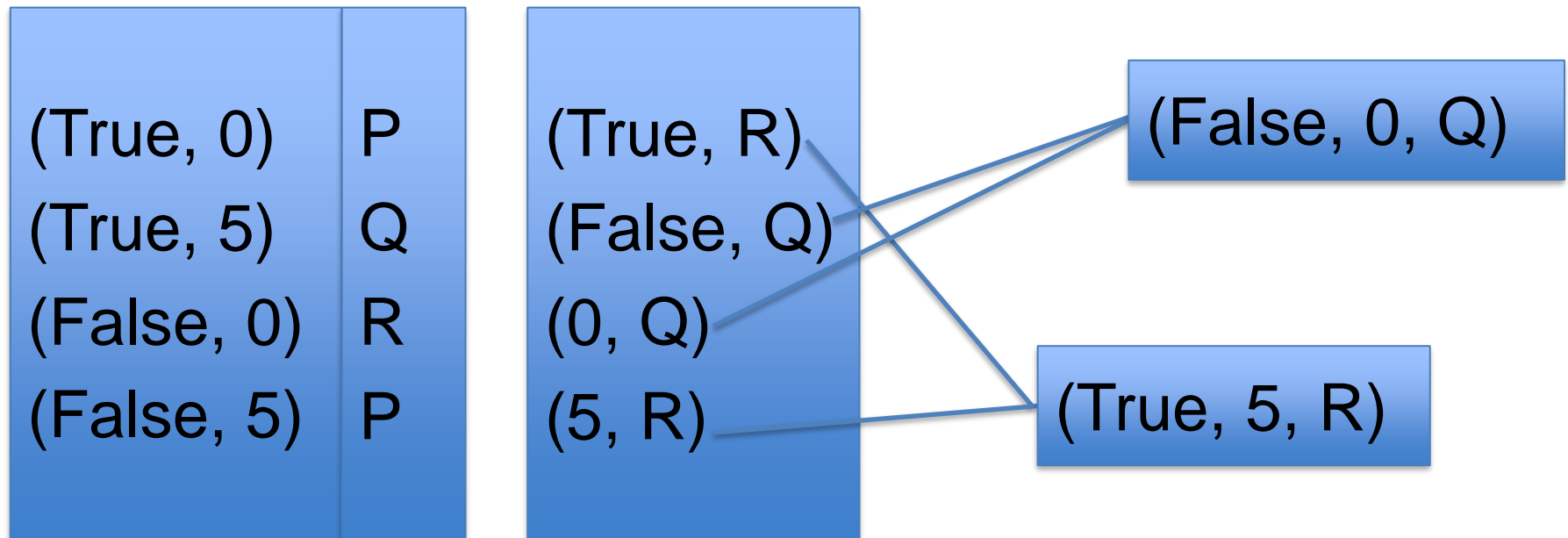
# In Parameter Order - Horizontal growth

- X (True, False)
  - Y (0, 5)
  - Z (P, Q, R)
- Add one more variable in each step to generate more combination
  - Remove duplications

All 2-way test combinations



# In Parameter Order - Vertical growth



Merge short combinations to longer ones

# A Variation of In Parameter Order

- Initialization phase
  - Order variables based on the number of their possible values
  - Generate test cases by full combinations of the first two variables
- Vertical growth phase
  - Add other variables one-by-one to generate a combination
  - Re-order certain variable values if needed
- Horizontal growth phase
  - If re-order cannot generate enough new combinations
  - Extend horizontally to add possibility of using re-ordering to generate new combinations

# A Variation of In Parameter Order - Example

- A car ordering application \*
- Variables
  - Order category (Buy, Sell)
  - Location (Oslo, Trondheim)
  - Car brand (BMW, Audi, Mercedes)
  - Registration numbers (Valid, Invalid)
  - Order type (E-Booking, In-store)
  - Order time (Working hours, Non-working hours)

\*[www.softwaretestinghelp.com/what-is-pairwise-testing/](http://www.softwaretestinghelp.com/what-is-pairwise-testing/)



# Variation of In Parameter Order – Initialization (reorder variables)

- Order variables based on their number of values
- The results after ordering
  - **Car brand (BMW, Audi, Mercedes)**
  - Order category (Buy, Sell)
  - Location (Oslo, Trondheim)
  - Registration numbers (Valid, Invalid)
  - Order type (E-Booking, In-store)
  - Order time (Working hours, Non-working hours)


# Vertical growth phase – iteration 1

Car Brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy				
BMW	Sell				
Audi	Buy				
Audi	Sell				
Mercedes	Buy				
Mercedes	Sell				

# Vertical growth phase – iteration 2

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo			
BMW	Sell	Trondheim			
Audi	Buy	Oslo			
Audi	Sell	Trondheim			
Mercedes	Buy	Oslo			
Mercedes	Sell	Trondheim			



# Vertical growth phase – iteration 2 (cont')


- After re-ordering

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo			
BMW	Sell	Trondheim			
Audi	Buy	Trondheim			
Audi	Sell	Oslo			
Mercedes	Buy	Oslo			
Mercedes	Sell	Trondheim			

# Vertical growth phase – iteration 3

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid		
BMW	Sell	Trondheim	Invalid		
Audi	Buy	Trondheim	Valid		
Audi	Sell	Oslo	Invalid		
Mercedes	Buy	Oslo	Valid		
Mercedes	Sell	Trondheim	Invalid		



# Vertical growth phase – iteration 3 (cont')

- After re-ordering

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid		
BMW	Sell	Trondheim	Invalid		
Audi	Buy	Trondheim	Valid		
Audi	Sell	Oslo	Invalid		
Mercedes	Buy	Oslo	Invalid		
Mercedes	Sell	Trondheim	Valid		

# Vertical growth phase – iteration 4

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	
BMW	Sell	Trondheim	Invalid	E-booking	
Audi	Buy	Trondheim	Valid	E-booking	
Audi	Sell	Oslo	Invalid	In Store	
Mercedes	Buy	Oslo	Invalid	E-booking	
Mercedes	Sell	Trondheim	Valid	In Store	

# Vertical growth phase – iteration 5

If re-order cannot generate enough new combinations

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	Working hours
BMW	Sell	Trondheim	Invalid	E-booking	Non-working hours
Audi	Buy	Trondheim	Valid	E-booking	Working hours
Audi	Sell	Oslo	Invalid	In Store	Non-working hours
Mercedes	Buy	Oslo	Invalid	E-booking	Working hours
Mercedes	Sell	Trondheim	Valid	In Store	Non-working hours



# Horizontal growth

Extend horizontally to add possibility of using re-ordering to generate new combinations

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	Working hours
BMW	Sell	Trondheim	Invalid	E-booking	Non-working hours
	Buy				Non-working hours
Audi	Buy	Trondheim	Valid	E-booking	Working hours
Audi	Sell	Oslo	Invalid	In Store	Non-working hours
	Sell				Working hours
Mercedes	Buy	Oslo	Invalid	E-booking	Working hours
Mercedes	Sell	Trondheim	Valid	In Store	Non-working hours

# All-pairs combination testing tools

- [ACTS](#) – ‘Advanced Combinatorial Testing System’, provided by NIST, an agency of the US Government
- [VPTag](#) free Pairwise Testing Tool

# Outline

- Single variable
- Multiple variables
- Error guessing/Risk based
- Output coverage and testing

# Error guessing / Risk-based

- Based on experience
  - Different types of faults
  - Probability of the faults
  - Consequence of the faults
  - ...

# Bach's risk-based testing\*

- Bach's heuristics is based on his experience as a tester
- Two risk analysis approach
  - Inside-out: What can go wrong there?
  - Outside-in: What things are associate with this kind of risk?
- Based on this experience he has identified
  - A generic risk list – things that are important to test
  - A risk catalogue – things that often go wrong to a specific domain

\*<http://www.satisfice.com/articles/hrbt.pdf>

# Bach's generic risk list

- Complex – large, intricate or convoluted
- New – no past history in this product
- Changed – anything that has been tampered with or “improved”
- Upstream dependency – a failure here will cascade through the system
- Downstream dependency – sensitive to failure in the rest of the system
- Critical – a failure here will cause serious damage

# Bach's generic risk list (cont')

- Precise – must meet the requirements exactly
- Popular – will be used a lot
- Strategic – of special importance to the users or customers
- Third-party – developed outside the project
- Distributed – spread out over time or space but still required to work together
- Buggy – known to have a lot of problems
- Recent failure – has a recent history of failures

# Bach's risk catalogs example

- Wrong files installed
  - Temporary files not cleaned up
  - Old files not cleaned up after upgrade
  - Unneeded file is installed
  - Needed file not installed
  - Correct file installed in the wrong space
- Files clobbered
  - Old file replaces newer file
  - User data file clobbered during upgrade
- ...



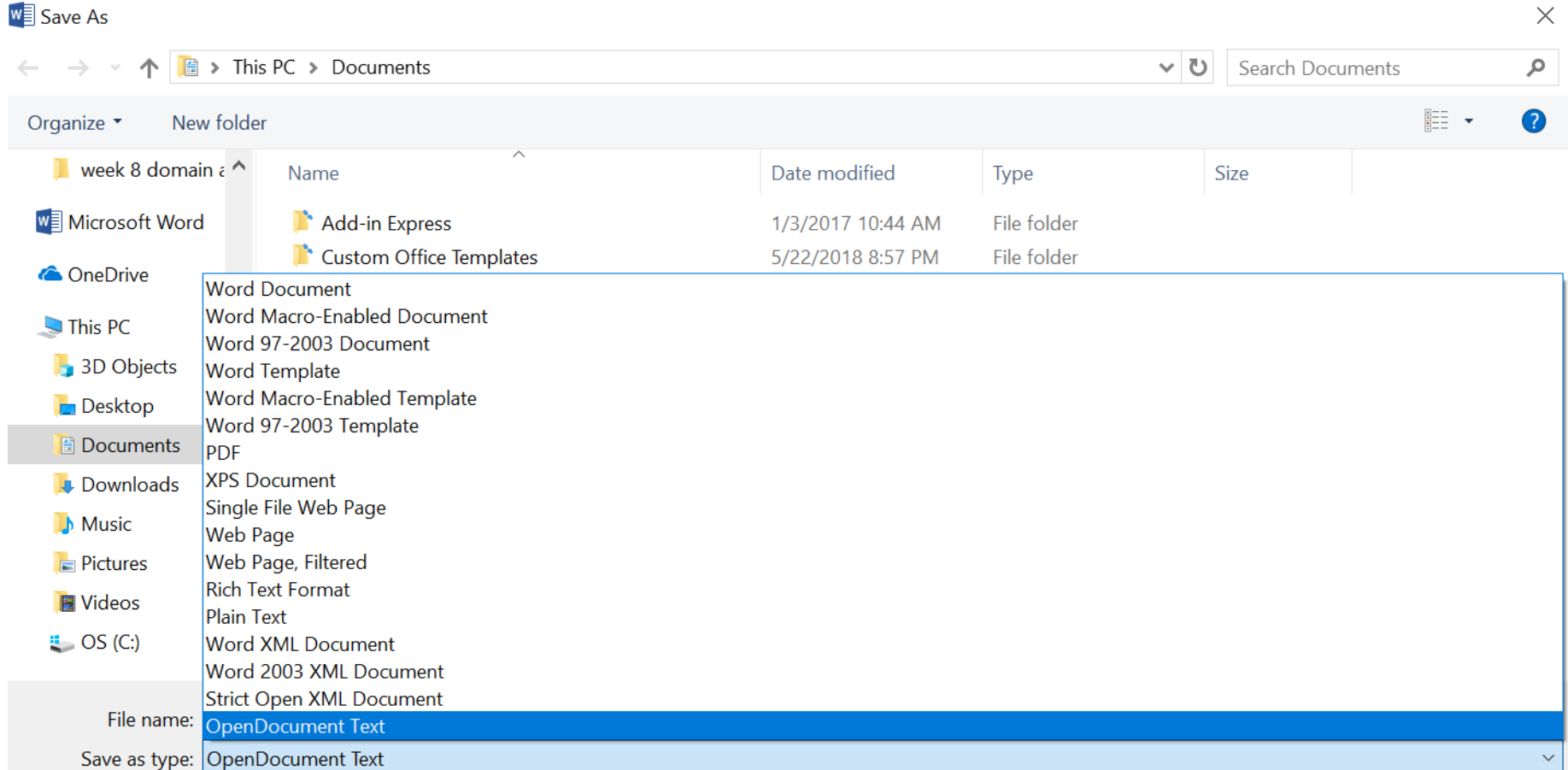
# Outline

- Single variable
- Multiple variables
- Random testing
- Error guessing/Risk based
- Output coverage and testing

# Output coverage

- All the coverage types that we have looked at so far have been related to input data
- It is also possible to define coverage based on output data
- Uncovered outputs help us define the new test cases

# Output coverage (cont')



# Output coverage (cont'')

- The main challenge with using output coverage measure is that output can be defined at several levels of details
  - An account summary
  - An account summary for a special type of customer
  - An account summary for a special event

# Summary

- We studied
  - How to identify the domains of **input & output** variables
  - How to derive function test cases based on single and multiple domain variables
- Can be applied to
  - Unit test
  - Integration test
  - System test
  - Acceptance test