PA2516: Verification and Validation

Lecture 3: Coverage and Intervention I: Specification based

Kai Petersen, PhD
Blekinge Institute of Technology

Dynamic V&V - Software Testing (some definitions)

- “Software testing is the process of uncovering evidence of defects in software systems” McGregor 2002
- “A set of activities conducted with the intent of finding errors in the software”
- “Testing shows the presence, not the absence of bugs” E.W. Dijkstra
- Testing is the process of exercising a software component using a selected set of test cases, with the intent of (1) revealing defects, and (ii) evaluating quality.

What is software testing (cont.)?

- Testing is the a sampling based execution of test objects under specified conditions with the aim of checking the test results against desired behavior.
- Testing is a control function that does not include fault correction.
- Through testing failures are discovered (not the causes – bugs)
- Debugging is the activity of localizing and correcting the defects

Software testing … some more definitions

AGRESSION TESTING: If this doesn't work, I'm gonna kill somebody.

COMPRSSION TESTING: []

CONFESSION TESTING: Okay, Okay, I did program that bug.

CONGRSSIONAL TESTING: Are you now, or have you ever been a bug?

DEPRESSION TESTING: If this doesn't work, I'm gonna kill myself.

EGRESSION TESTING: Uh-oh, a bug... I'm outta here.

DIGRESSION TESTING: Well, it works, but can I tell you about my truck...

EXPRESSION TESTING: #@%^&*!!!, a bug.

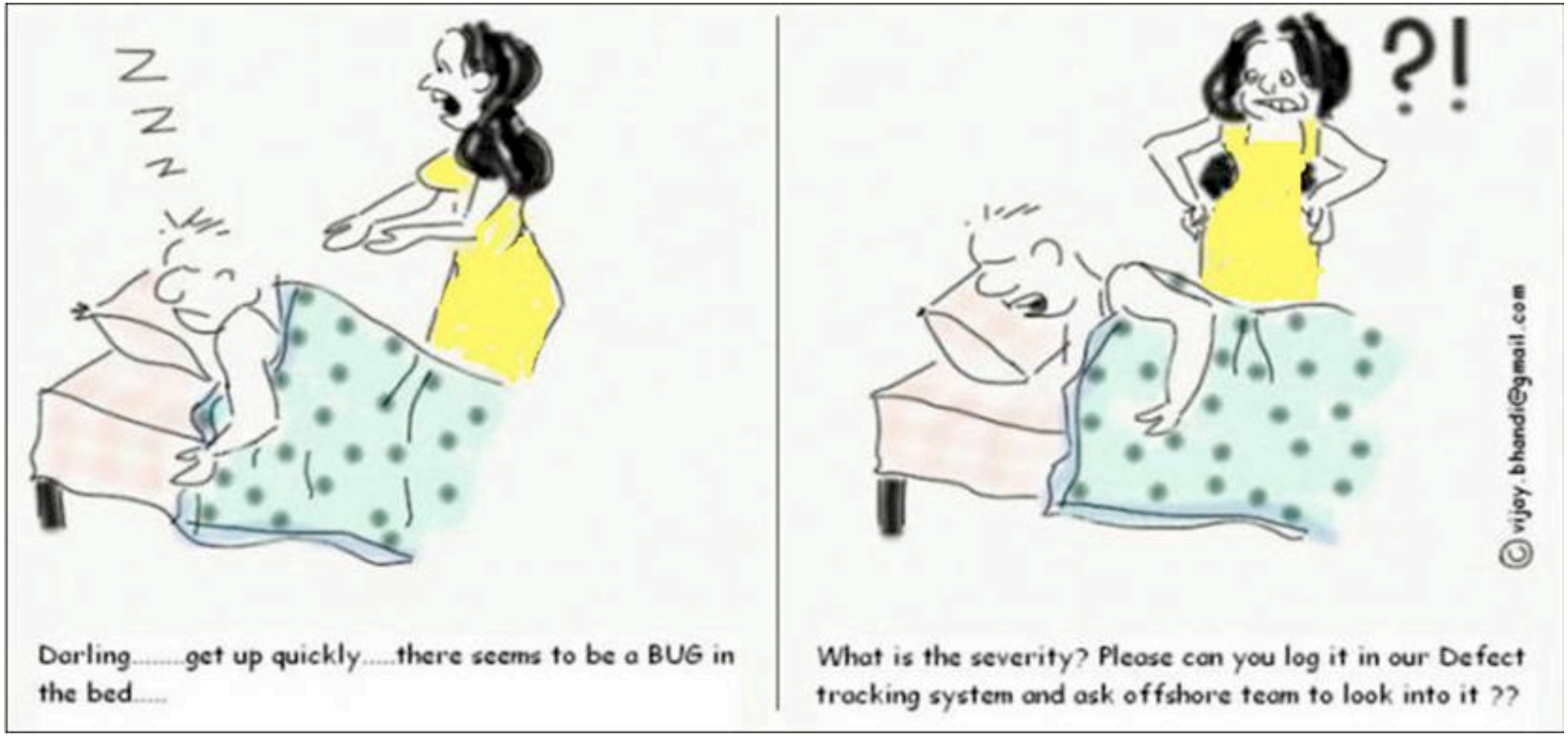
OBSESSION TESTING: I'll find this bug if it's the last thing I do.

OPRESSION TESTING: Test this now!

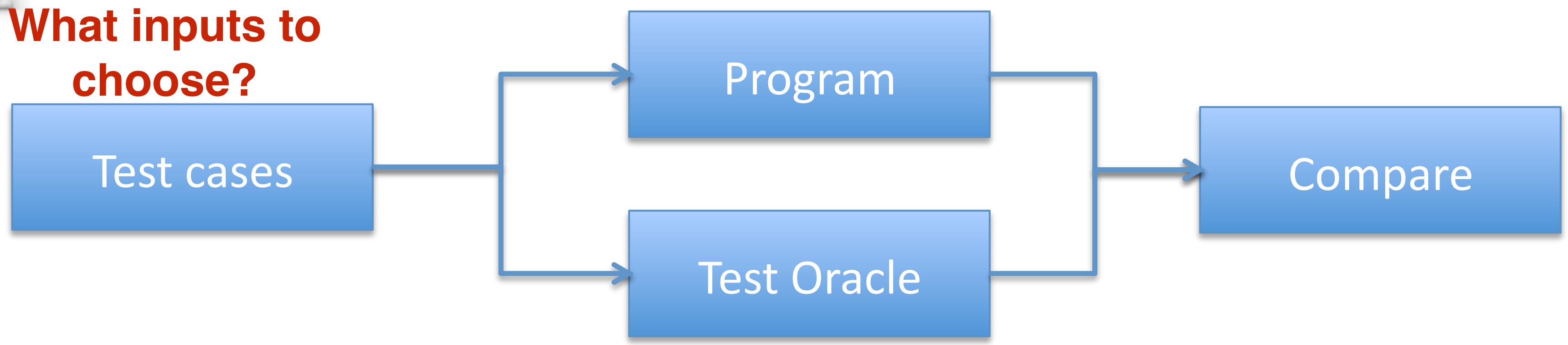
REPRESSION TESTING: It's not a bug, it's a feature.

SECCESSION TESTING: The bug is dead! Long lives the bug!

SUGGESTION TESTING: Well, it works but wouldn't it be better if...


Z Z z
?!
© vijay.bhandi@gmail.com
Darling.........get up quickly......there seems to be a BUG in the bed.....
What is the severity? Please can you log it in our Defect tracking system and ask offshore team to look into it ??

How testing works



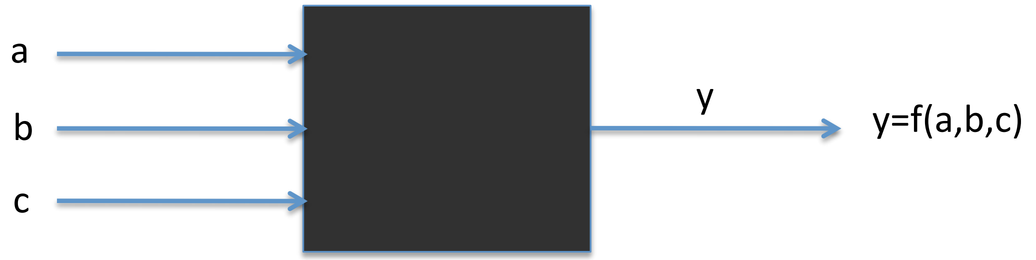
- Test oracle - knows the desired behaviour for given inputs
    - may not be the system under test
    - in an ideal case the oracle should be generated
    - in the typical case the test oracle is a human
- Problem:
    - Requirements might be described in a poor (non-testable) way
    - Requirements may be incomplete (i.e. if the requirements drive the testing, then we do not know if our testing is “complete” wrt. requirements coverage)
- Wrong requirements lead to wrong desired behaviors

Specification based

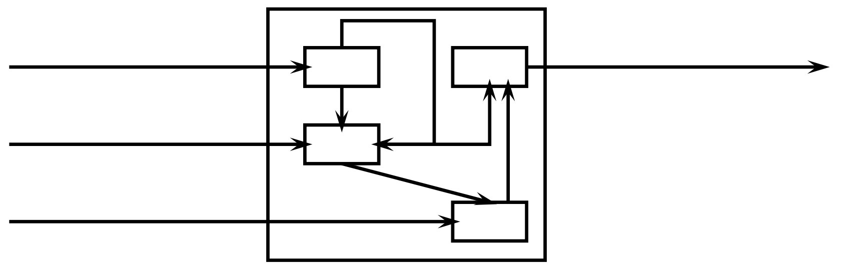
Specification (and interfaces) are the sources of test cases

Two sources for generating test cases

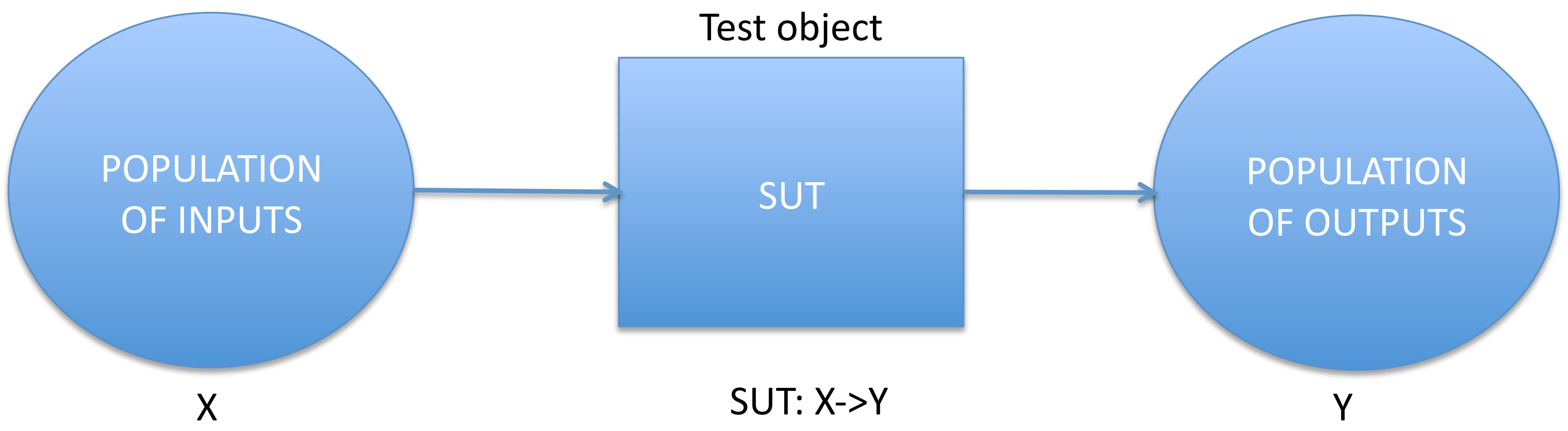
- **Specification based testing (black-box testing)**
  - Test-case selection based on requirements specification
  - Structure of the program is not considered
  - Desired behavior is determined based on the specification



- **Structural testing (White-box testing)**
  - Test case selection based on code/ system structure
  - Desired behavior determined based on the specification

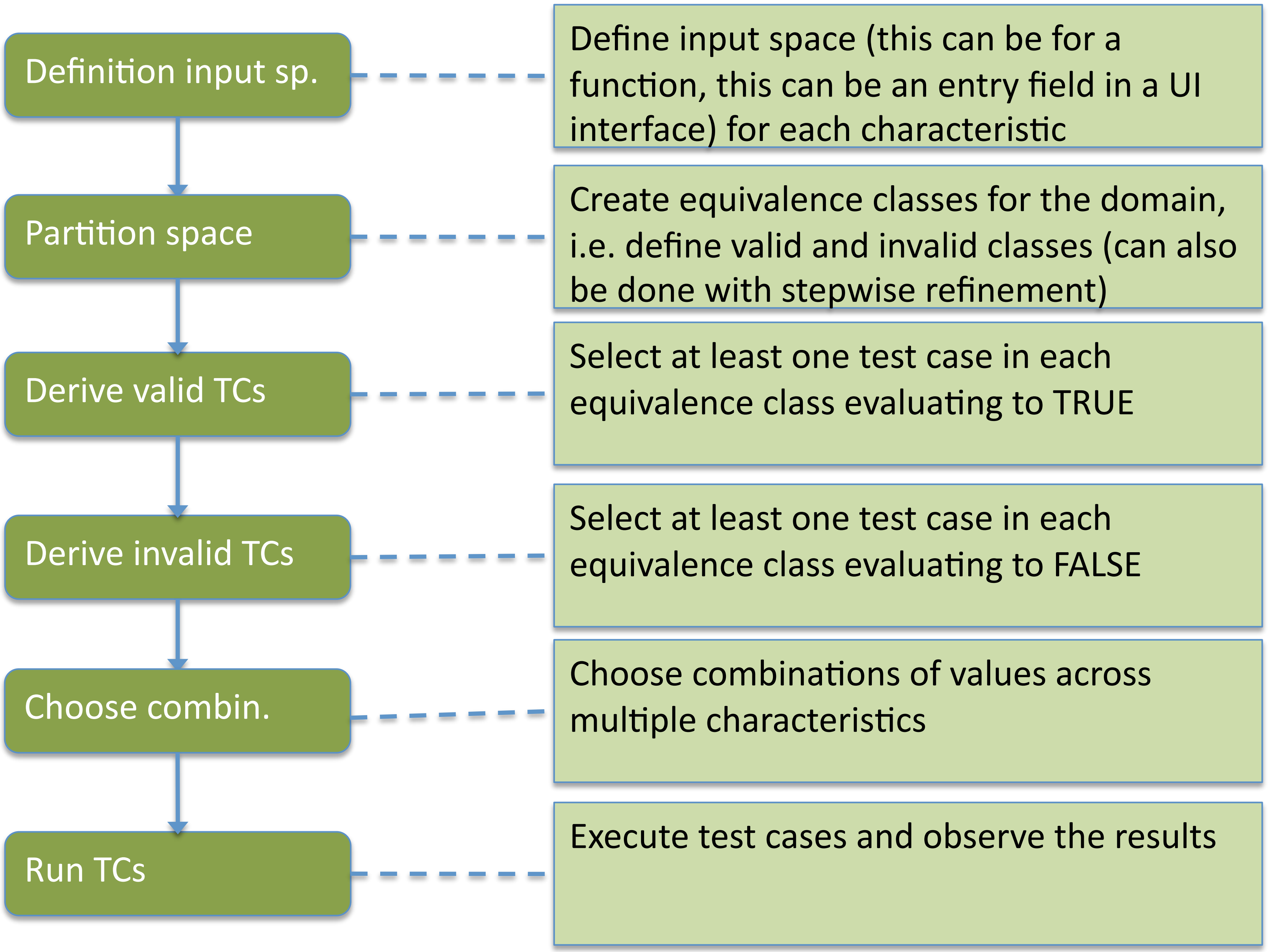


Exhaustive testing is not possible



- Input space (possible values of input parameters)
  - Example: Test a 64 Bit floating point number
  - Assumption on performance: We are able conduct 100 million different computations on the floating point number (incl. comparison with desired result) per second
  - Test of all computations will be astronomically high

Process



Define input space (this can be for a function, this can be an entry field in a UI interface) for each characteristic

Create equivalence classes for the domain, i.e. define valid and invalid classes (can also be done with stepwise refinement)

Select at least one test case in each equivalence class evaluating to TRUE

Select at least one test case in each equivalence class evaluating to FALSE

Choose combinations of values across multiple characteristics

Execute test cases and observe the results

Defining the input space and partitions

On identifying characteristics and input space

- **Functionality-based:** Identify intended functionality of the system
  - requires domain knowledge
  - deep insights in intended functionality yield good knowledge about expected results (oracle)
  - availability of testable requirements is helpful
  - incomplete requirements for complex systems make it hard to identify good input/expected result pairs
  - requirements are often too abstract to directly lead to input parameters
  - Example: Function for drawing a triangle might take specific characteristics of triangles into consideration (e.g. Equilateral, Isosceles and Scalene)

On identifying characteristics and input space cont.

- **Interface-based:** Identify possible classes from interfaces (e.g. method signature, data fields, etc.)
  - easy to apply and leads directly to executable test cases
  - incomplete information about intended usage
  - combinations of values that make sense are not known as different interfaces are looked at in isolation
  - Example: Function for drawing a triangle just treats the three parameters as integer values (0 often a special value)

Specification based testing: Input partitioning

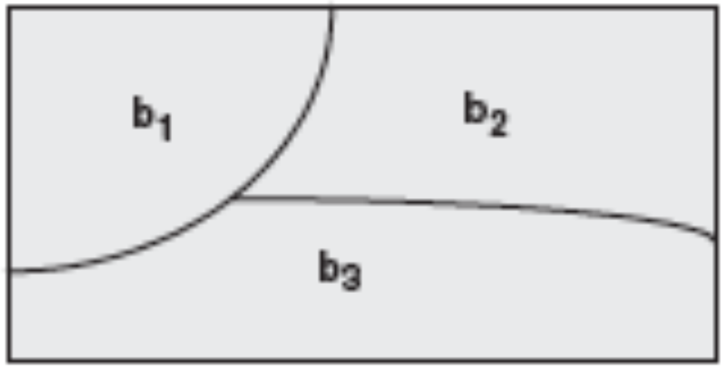
- Divide the input space into blocks (called equivalence classes) where for test inputs in a partition the system behaves in an identical way
- Consequence: Test inputs within a partition are equally good/suited to test the partition/block
- Mathematically: Partitions should be disjoint, and complete.

Definition: The partition defines a set of equivalence classes called blocks. The blocks are pair-wise disjoint (Block i ∩ Block j =⃠ for i ≠ j).

The joint set of Blocks makes up the total input domain D.

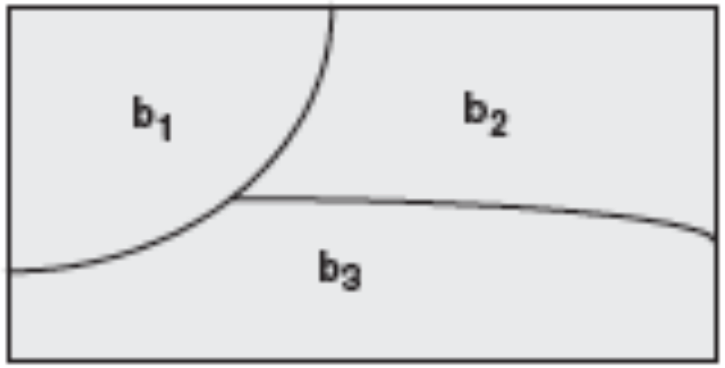
All inputs from one equivalence class lead to the same test result (i.e. PASS or FAIL)

$$a \in A \Rightarrow Test(a) = PASS \Rightarrow \forall b \in A \Rightarrow Test(b) = PASS$$
$$a \in A \Rightarrow Test(a) = FAIL \Rightarrow \forall b \in A \Rightarrow Test(b) = FAIL$$



Example: A software has multiple characteristics

- **C1: Order of file X (sorted, inverse sorted, arbitrar**
- C2: Password minimum length
- C3: Password restrictions on characters
- etc.



- Example C1
    - b1 = Sorted in ascending order
    - b2 = Sorted in descending order
    - b3 = Sorted in arbitrary order

Why defective?

If the length of the file list is 0 or 1, it will belong to all blocks (b1, b2, and b3), i.e. blocks are not disjoint

Example: Cont.

- Better definition of equivalence classes:
  - File F sorted ascending
    - b1: True (valid class)
    - b2: False (invalid class)
  - File F sorted descending
    - b1: True (valid class)
    - b2: False (invalid class)
  - Files with length 1 and 0 are in True (the same block b1), arbitrary sorting

Why should we care so much about disjointness and completeness?

Helps to make lack of clarity in rationales for partitions explicit

Helps in input domain modelling (consistent generation of tests, automation)

Rules for creating partitions 1

- Codomain
  - Specifies input conditions in an ordered codomain/values
  - one valid equivalence class
  - two invalid equivalence classes
  - **Example:**
    - Range 1 <= Value <= 49
    - Valid equivalence class: 1<=Value<=49
    - Invalid equivalence class: Value <1, Value > 49
- Data structure
  - Can define structure if upper and lower bounds are given
  - one valid equivalence class
  - two invalid equivalence classes
  - **Example:**
    - A list contains 1 to 155 elements
    - Valid equivalence class: 1 to 155 elements
    - Invalid equivalence class: 0, > 155 elements

Rules for creating partitions 2

- Input is restricted by conditions
  - one valid equivalence class and one invalid equivalence classes
  - **Example:**
    - Input is an integer
    - Valid: $x \in$ IN, Invalid: : $x \notin$ IN
- Enumeration
  - one valid equivalence class
  - one invalid equivalence class
  - **Example:**
    - Enumeration: {red, yellow, green}
    - Valid equivalence class: {red, yellow, green}
    - Invalid equivalence class: Values not in {red, yellow, green}

Deriving valid and invalid test cases

- Several alternatives of combinatorial combinations are possible for tests:
  - All combinations coverage (ACoC)
  - Each choice coverage (ECC)
  - Pair-wise coverage (PWC)
  - T-wise coverage (TWC)
  - Base choice coverage (BCC)

Combinatorial testing

- Example: Let us assume three characteristics with blocks q1= [A,B], q2 = [1,2,3], and q3 = [x,y], B1 = 2, B2 = 3, B3 = 2

All combinations coverage (ACoC): All combinations of blocks from all characteristics must be used

Tests:

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

Will have a unique test for each combination of the blocks for each partition. More tests than necessary impractical; test needed = 2*3*2 = 12

Combinatorial testing

- Example: Let us assume three characteristics with blocks [A,B], [1,2,3], and [x,y]

Each Choice Coverage (ECC): One value from each block has at least to be used

Tests:

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

Leaves a lot of freedom to the tester to interpret (i.e. what combinations to choose), i.e. it is relatively weak and also has a relatively low coverage

Therefore we would like to put requirements on how to pair: tests needed = Max i=1 to Q Bi = 3

Combinatorial testing

- Example: Let us assume three characteristics with blocks [A,B], [1,2,3], and [x,y]

Pair-wise Coverage (PWC): A value from each block for each characteristic must be combined with a value from every block for each other characteristic

First, we do the pairing for each characteristic

Now we can combine them in several ways, given that one test case can cover several pairs

Tests:

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

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

Natural extensions: require t values instead of pairs to be combined;

Combinatorial testing

- Example: Let us assume three characteristics with blocks [A,B], [1,2,3], and [x,y]

t-wise Coverage (TWC): A value from each block for each group of t characteristics must be combined, if t = number of characteristics, then it is equal to ACoC

Expensive, going beyond pairs (t=2) does not seem to help much

Combinatorial testing

- Example: Let us assume three characteristics with blocks [A,B], [1,2,3], and [x,y]

Base Choice Coverage (BCC): A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

Base choice test = (A, 1, x)

Constant = 1, x. Non-base choice = B, = (B, 1, x)
Constant = 1, A. Non-base choice = x, = (A, 1, y)
Constant = A, x. Non-base choice = 2, 3, = (A, 2, x) and (A,3,x)

Note: Several strategies to choose the base choice, e.g. simplest, first in ordering, most likely from user perspective

Exercise

- Specification: Employees receive 50 % of their monthly salary as Christmas gratification when they are with the company for more than 3 years. Employees that are with the company for more than 5 years receive 75 %. In case they are with the company more than 8 years gratification of 100 % is provided. In addition we distinguish different brackets (G1 = normal employees, G2 = middle management, G3 = higher management). Bracket G2 receives a gratification of 50 %, G3 receives a gratification of 100 %. G1 receives a gratification of 0 %.

Characteristics and input space

- Characteristic 1: Time within the company (duration in years)
- Characteristic 2: Brackets for types of employees

Equivalence classes

Characteristic	valid equivalence classes	Invalid equivalence classes
1. With the company	1.1 **0 <= x <= 3** 1.2 **3 < x <= 5** 1.3 **5 < x <= 8** 1.4 **8 < x <= 50**	1.a: x<0 1.b: x>50
2. Bracket	2.1: g=G1 2.2: g=G2 2.3: g=G3	2a: not in {G1, G2, G3}

- High level classes:
  - 0<=x<=50 => TRUE
  - bracket = {G1, G2, G3} => TRUE
- Sub-partitions are defined in each of the classes
- Choose what to test by doing combinations, see earlier slides

Example: Applying ECC combinatorial testing

Test case	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**
With company	2	4	7	12	-1	83	7
Bracked	G1	G2	G3	G2	G1	G3	G4
Covered Eq. Class	**1.1 2.1**	**1.2 2.2**	**1.3 2.3**	**1.4 2.2**	**1.a**	**1.b**	**2.a**

Boundary value analysis

- Problem: In equivalence classes we assume that all values within the equivalence class are equally well suited to discover defects
  - Is this a valid assumption?

Boundary value analysis

- Entryfield x in an User interface expects a signed integer
- Three equivalence classes:
  - **valid Equivalence Class 1:** [-16384, 16383]
  - **Invalid Equivalence Class 1a:** < -16384
  - **Invalid Equivalence Class 1b:** > 16383
- **Test cases**
  - **From equivalence classes**

    13566, -17000, 18000
  - **Boundary Values**

    -16385, -16384, -16383 (lower bound)

    16383, 16384, 16385 (upper bound)
  - **Special values**

    0 -1 +1

Boundary value analysis cont.

- Entryfield x in an User interface expects a signed integer
- Refinement of equivalence classes:
  - **valid Equivalence Class 1.1:** [-16384, 0[
  - **valid Equivalence Class 1.2:** 0
  - **valid Equivalence Class 1.3:**]0, 16383]
  - **Invalid Equivalence Class 1a:** Value < -16384
  - **Invalid Equivalence Class 1b:** Value > 16383
  - **Invalid Equivalence Class 1c:** no number

- **Test Cases**

-12345	0	11111	-17000	18000	„ten“

Decision tables

- Good overview of complex decisions and their consequences can be achieved

- Conditions and actions – if..then relationship

Conditions (IF)	**Condition pointer**
Action (Then)	**Action pointer**

Complex decision problem

- In the swamps there live four different tribes, ASIS, BELAS, CEDIS, and DRUIDS. Research has shown that there are four different characteristics that allow to distinguish the tribues. An inhabitant of the swamps can (but does not have to) manusel, have a knelt, lopsel, or nopel.
- We know that only the ASIS have a knelt and manusel.
- If someone does not have a knelt and nopels then he is a BELA.
- An inhabitant with a Knelt, who does not manusel, is a CEDI if he always nopels.
- Everyone who does not have a knelt and lopsels, never nopels, and always manusels is a CEDI for sure; if he would not manusel then he would be a DRUDI
- It is typical for DRUDIS that they neither manusel nor nopel; but they have a knelt.
- Inhabitants that have a knelt, do not lopsel and do not nopel are DRUDIS if they manusel, and CEDIS if they do not manusel.

Conditions

- Conditions:
  - has knelt, manusel, lopsel, nopel
- Action/Result:
  - Is ASIS, is BELA, is CELA, is DRUDI
- Construction:
  - 4 Conditions = 2^4 condition pointers
  - Now the actions have to be assigned to the condition pointers based on the specification.

Decision table constructed

- Decision tables include redundancies
- Example: R1 and R2 lead to the same result
- Redundancies can be eliminated to arrive at the most simple decision table for the given problem (that then can be more easily implemented).
- Rules: Two rules (in this case R1 to R16) can be combined only if
  - They lead to the same action
  - only one condition pointer is different (this is also referred to as the "irrelevance pointer"

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
has knelt	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
nopels	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
manusels	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
lopels	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
ASIS	X	X			X	X										
BELA									X	X	X	X				
CEDI			X	X									X			X
DRUDI							X	X						X	X	

Decision tables (remove redundancy)

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
has knelt	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
nopels	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
manusels	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
lopels	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
ASIS	X	X			X	X										
BELA									X	X	X	X				
CEDI			X	X									X			X
DRUDI							X	X						X	X	

	R1	R3	R5	R7	R9	R11	R13	R14	R15	R16
has knelt	Y	Y	Y	Y	N	N	N	N	N	N
nopels	Y	Y	N	N	Y	Y	N	N	N	N
manusels	Y	N	Y	N	Y	N	Y	Y	N	N
lopels	-	-	-	-	-	-	Y	N	Y	N
ASIS	X		X							
BELA					X	X				
CEDI		X					X			X
DRUDI				X				X	X	

Removing more redundancy

	R1	R3	R5	R7	R9	R11	R13	R14	R15	R16
has knelt	Y	Y	Y	Y	N	N	N	N	N	N
nopels	Y	Y	N	N	Y	Y	N	N	N	N
manusels	Y	N	Y	N	Y	N	Y	Y	N	N
lopels	-	-	-	-	-	-	Y	N	Y	N
ASIS	x		x							
BELA					x	x				
CEDI		x					X			X
DRUDI				x				X	X	

	R1	R3	R7	R9	R13	R14	R15	R16
has knelt	Y	Y	Y	N	N	N	N	N
nopels	-	Y	N	Y	N	N	N	N
manusels	Y	N	N	-	Y	Y	N	N
lopels	-	-	-	-	Y	N	Y	N
ASIS	x							
BELA				x				
CEDI		x			X			X
DRUDI			x			X	X	

Implementation

```
If R1 THEN …
ELSE IF R3 THEN …
ELSE IF R7 THEN …

… etc.
```

Summary

- Exhausting testing is not possible
- Need to analyse the input space and partition it wisely
- Multiple strategies are available to choose and combine values