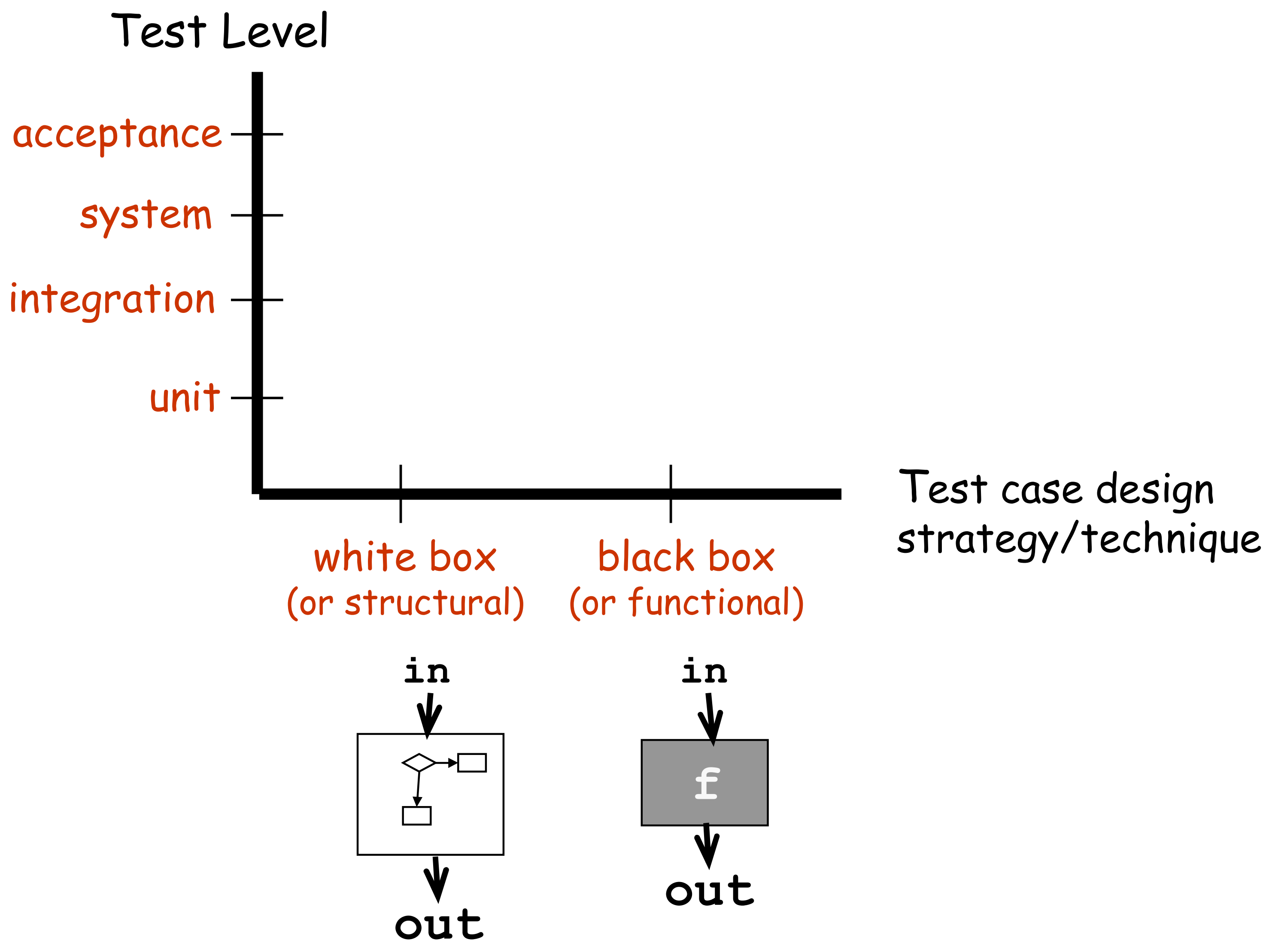
TVVS - Test, Verification and Validation of Software

Test case design

Ana Paiva

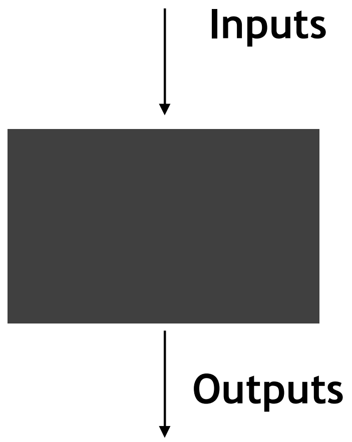
apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Test types (remember)



Test Level
acceptance
system
integration
unit
Test case design strategy/technique
white box (or structural)
black box (or functional)
in
out
in
f
out

Test case design strategies and techniques

Strategy	Tester's View	Knowledge sources	Techniques / Methods
Black-box testing (not code-based) (sometimes called functional testing)		Requirements document Specifications User manual Models Domain knowledge Defect analysis data Intuition Experience	Equivalence class partitioning Boundary value analysis Cause effect graphing Pairwise testing Error guessing Random testing State-transition testing Scenario-based testing

Black-box testing

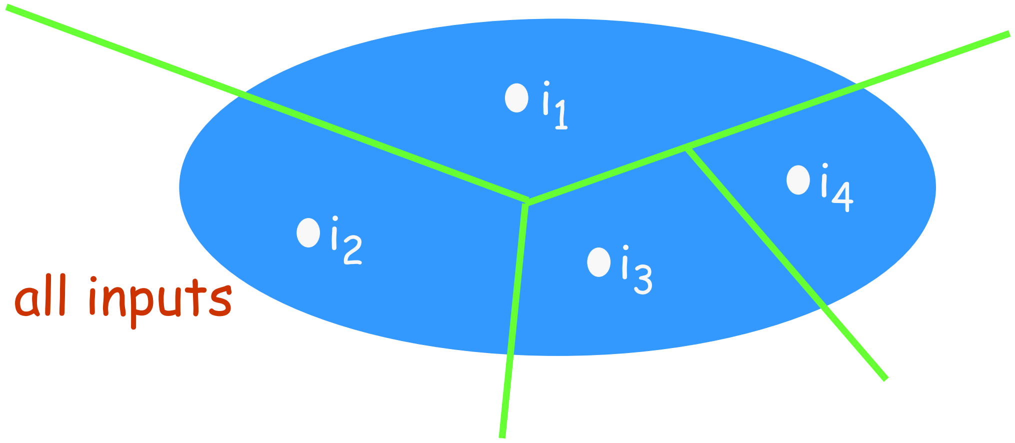
- Equivalence class (or domain) partitioning
- Boundary-value analysis
- Cause-effect graphing (or decision table)
- Pairwise testing
- Error guessing
- Risk based testing
- Testing for race conditions
- Random testing
- Requirements based testing
- Model-based testing (MBT)
- …

Equivalence class partitioning

Divide all possible inputs into classes (partitions) such that :

- There is a finite number of input equivalence classes
- You may reasonably assume that
  - the program behaves analogously for inputs in the same class
  - one test with a representative value from a class is sufficient
  - if the representative detects a defect
    then other class members would detect the same defect

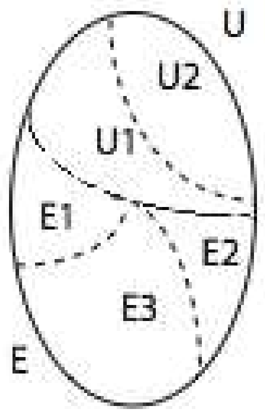
(Can also be applied to outputs)



Equivalence class partitioning

Faults targeted

- The entire set of inputs to any application can be divided into at least two subsets:
  - one containing all the expected, or legal, inputs (E) and
  - the other contains all unexpected or illegal inputs (U).
- Each of the two subsets, can be further subdivided into subsets on which the application must behave differently (e.g., E1, E2, E3, and U1, U2).



Systematic procedure for equivalence partitioning

- 1. Identify the input domain
- 2. Equivalence classing
- 3. Combine equivalence classes
- 4. Identify infeasible equivalence classes

Systematic procedure for equivalence partitioning

1. Identify the input domain: Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use.

- Environment variables, such as class variables used in the method under test and environment variables in Unix, Windows, and other operating systems, also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets.

Systematic procedure for equivalence partitioning

2. Equivalence classing: Partition the set of values of each variable into disjoint subsets. Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. Partitioning the input domain using values of one variable, is done based on the expected behavior of the program.

- Values for which the program is expected to behave in the “same way” are grouped together. Note that “same way” needs to be defined by the tester.

Sistematic procedure for equivalence partitioning

3. Combine equivalence classes: This step is usually omitted, and the equivalence classes defined for each variable are directly used to select test cases. However, by not combining the equivalence classes, one misses the opportunity to generate useful tests.

- The equivalence classes are combined using the multidimensional partitioning approach described earlier.

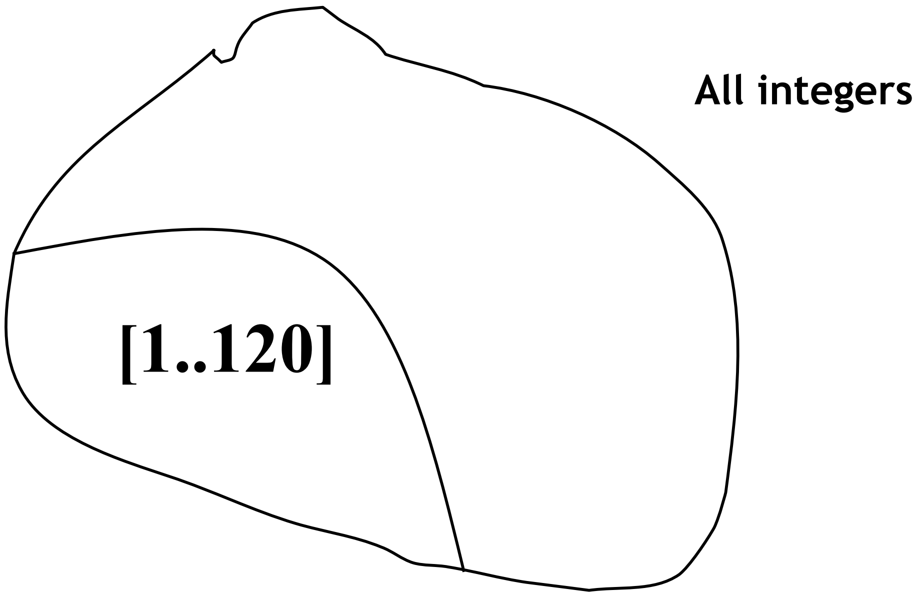
Sistematic procedure for equivalence partitioning

4. Identify infeasible equivalence classes: An infeasible equivalence class is one that contains a combination of input data that cannot be generated during the test. Such an equivalence class might arise due to several reasons.

- For example, suppose that an application is tested via its GUI, i.e., data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering only a palette of valid inputs. There might also be constraints in the requirements that render certain equivalence infeasible.

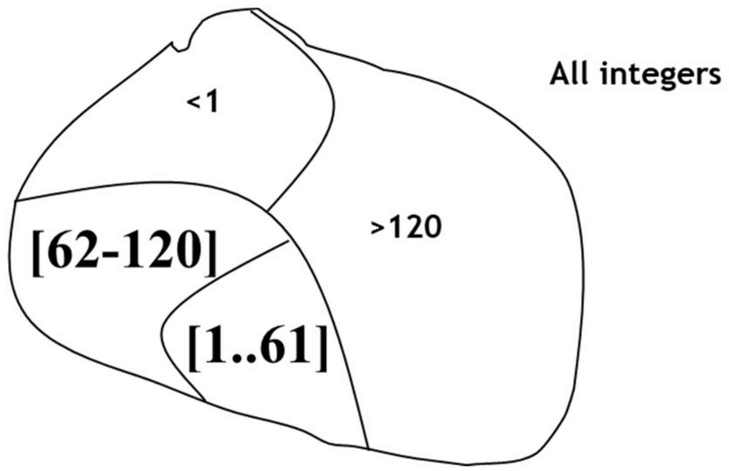
Equivalence class partitioning
Example 1

- Consider an application A that
  - inputs an integer denoted by age
  - the only legal age values are in the range [1..120]
- The input values are now divided into an E containing all integers in the range [1..120] and a set U containing the remaining integers.



Equivalence class partitioning
Example 1

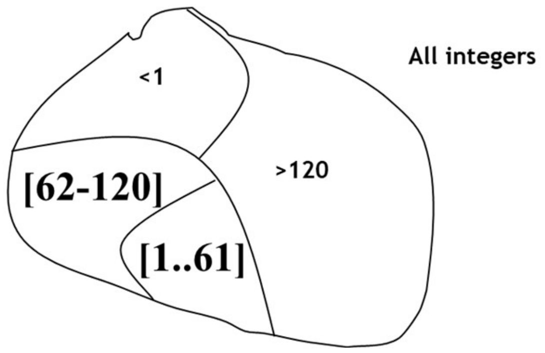
- Further, assume that
  - the application processes all values in the range [1..61] in accordance with requirement R1 and
  - those in the range [62..120] according to requirement R2.
- Thus, E is further subdivided into two regions depending on the expected behavior



- Similarly, it is expected that
  - all invalid inputs less than or equal to 1 are to be treated in one way
  - while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.

Equivalence class partitioning Example 1

- Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e.,
  - two regions containing expected inputs and
  - two regions containing the unexpected inputs.



- It is expected that any single test selected from the range [1..61] will reveal any fault with respect to R1.
- Similarly, any test selected from the region [62..120] will reveal any fault with respect to R2.
- A similar expectation applies to the two regions containing the unexpected inputs.

Equivalence class partitioning
Example 2

Test a function that calculates the absolute value of an integer *x*

- Equivalence classes :

|x|	Valid	Invalid
Input Type	integer	Non-integer
Class	{<0} {>=0}	{non-integer}

- Test Cases :

|x|	Expected result	Class
-10	10	{<0}
100	100	{>=0}
“XYZ”	Error	{non-integer}

Equivalence class partitioning
Example 3

- Test a program that
  - computes the *sum* of the first *N* integers, as long as this *sum* is less than *maxint*.
  - Otherwise, an error should be reported.
  - If *N* is negative, then it takes the absolute value *N*.

- Formally:

  Given integer inputs *N* and *maxint* compute result :

$$\text{result} = \sum_{K=0}^{|N|} k \quad \text{if this} <= \text{maxint}, \text{ error otherwise}$$

Equivalence class partitioning
Example 3

- Equivalence classes:

	Valid	Invalid
Input type	Int, Int	No-Int
Classes for |N|	{N < 0}, {N >= 0}	{no-int}
Classes for comparing **∑k with maxint**	∑k <= maxint ∑k > maxint	

- Test Cases

maxint	N	Result	Class
100	10	55	{N>=0}; ∑k <= maxint
100	-10	55	{N<0}; ∑k <= maxint
10	10	error	{N>=0}; ∑k > maxint
"XYZ"	9.1E4	error	{no-int}

Equivalence classes based on program output

- In some cases, the equivalence classes are based on the output generated by the program. For example, suppose that a program outputs an integer. It is worth asking:
  - Does the program ever generate a 0?
  - What are the maximum and minimum possible values of the output?
- These two questions lead to the two following equivalence classes based on outputs:
  - E1: Output value v is 0.
  - E2: Output value v is the maximum possible.
  - E3: Output value v is the minimum possible.
  - E4: All other output values.
- Based on the output equivalence classes one may now derive equivalence classes for the inputs. Thus, each of the four classes given above might lead to one equivalence class consisting of inputs.

Partitioning

- Unidimensional partitioning
  - Consider one input domain variable at a time.
  - Each input variable leads to a partition of the input domain.
  - We refer to this partitioning style as unidimensional equivalence or unidimensional partitioning.

  This type of partitioning is commonly used.

- Multidimensional partitioning
  - Consider the input domain as the set product of the input variables.
  - One partition consisting of several equivalence classes.
  - We refer to this method as multidimensional equivalence partitioning or multidimensional partitioning.

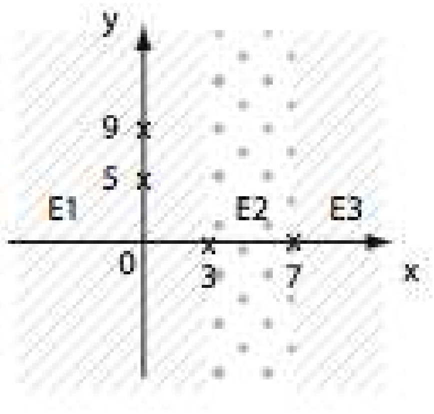
  Multidimensional partitioning leads to many equivalence classes that are difficult to manage manually. Many classes so created might be infeasible. Nevertheless, equivalence classes so created offer an increased variety of tests, as illustrated in the next section.

Partitioning Example

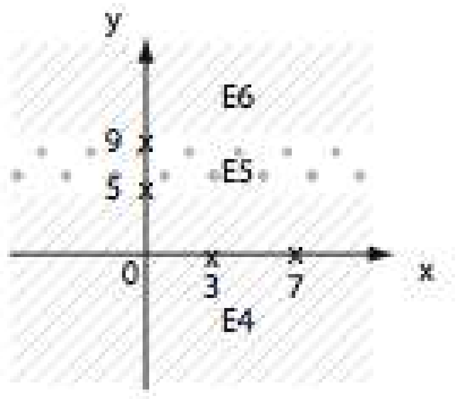
- Consider an application that requires two integer inputs *x* and *y*. Each of these inputs is expected to lie in the following ranges: $3 \leq x \leq 7$ and $5 \leq y \leq 9$.
- For unidimensional partitioning we apply the partitioning guidelines to *x* and *y* individually. This leads to the following six equivalence classes.
  - E1: $x<3$ E2: $3 \leq x \leq 7$ E3: $x>7$ (y ignored)
  - E4: $y<5$ E5: $5 \leq y \leq 9$ E6: $y>9$ (x ignored)
- For multidimensional partitioning we consider the input domain to be the set product X x Y. This leads to 9 equivalence classes.

Partitioning Example

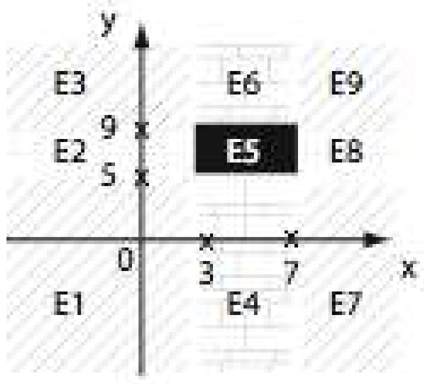
- 9 equivalent classes
  - E1: **x<3**, y<5
  - E3: **x<3**, y>9
  - E2: **x<3**, 5≤y≤9
  - E4: **3≤x≤7**, y<5
  - E5: **3≤x≤7**, 5≤y≤9
  - E6: **3≤x≤7**, y>9
  - E7: **x>7**, y<5
  - E8: **x>7**, 5≤y≤9
  - E9: **x>7**, y>9



(a)



(b)



(c)

Pairwise testing

Pairwise testing: A black box test design technique in which test cases are designed to execute all possible discrete combinations of each pair of input parameters.

- The number of pairwise test cases will be O(*mn*), where *m* and *n* are the number of possibilities for each of the two parameters with the most choices.

$$O(mn) = Max(X) \times Max(X \backslash Max(X))$$

and X is the set with the number of possibilities for every parameters

Pairwise testing

Parameter Name	Value 1	Value 2	Value 3	Value 4
Enabled	True	False		
Choice Type	1	2	3	
Category	a	b	c	d

Enabled	Choice Type	Category
True	$\boxed{3}$	a
True	1	d
False	1	c
False	2	d
True	2	c
False	2	a
False	1	a
False	$\boxed{3}$	b
True	2	b
True	$\boxed{3}$	d
False	$\boxed{3}$	c
True	1	b

X = {2,3,4}

$O(mn)$ = 4 x 3 = 12

Boundary value analysis

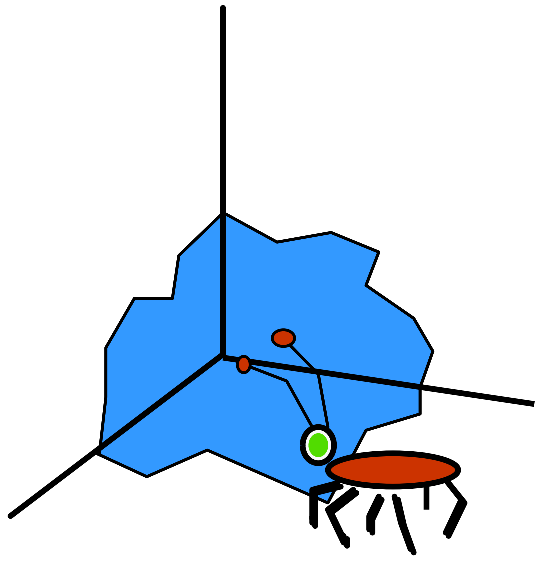
Based on experience/heuristics:

- Testing boundary conditions of equivalence classes is more effective, i.e., values directly on, above, and beneath edges of classes
  - If a system behaves correctly at boundary values, then it probably will work correctly at "middle" values
- Choose input boundary values as tests in input classes instead of, or additional to arbitrary values
- Choose also inputs that invoke output boundary values (values on the boundary of output classes)
- Example strategy as an extension of equivalence class partitioning:
  - Choose one (or more) arbitrary value(s) in each equivalent class
  - Choose values exactly on the lower and upper boundaries of equivalent classes
  - choose values immediately below and above each boundary (if applicable)

Boundary value analysis

"Bugs lurk in corners and congregate at boundaries."

[Boris Beizer, "Software testing techniques"]



Boundary value analysis
Example 1

Test the calculation of the absolute value of an integer

Valid equivalence classes :

|N|	Valid	Invalid
Classes	{<0}, {>=0}	{No-Int}

- Test cases :

class x < 0, arbitrary value:	x = -10
class x >= 0, arbitrary value	x = 100
classes x < 0, x >= 0, on boundary :	x = 0
classes x < 0, x >= 0, below and above:	x = -1, x = 1

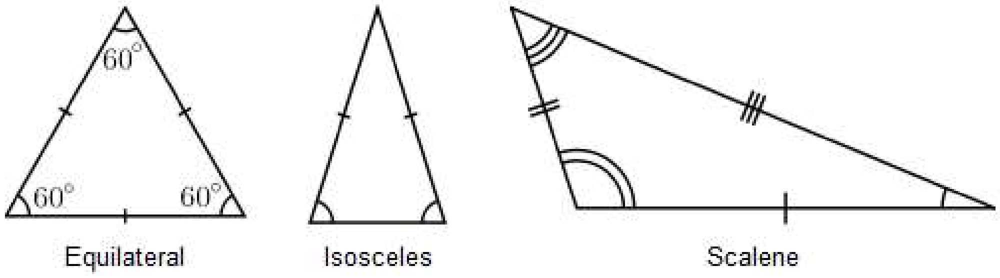
Boundary value analysis - exercise

- "A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene (all lengths are different), isosceles (two lengths are equal), or equilateral (all lengths are equal)."

Write a set of test cases to test this program.

Inputs: l_1, l_2, l_3 , integer, $l_i > 0$, $l_i < l_j + l_k$

Output: error, scalene, isosceles or equilateral



Boundary value analysis - exercise

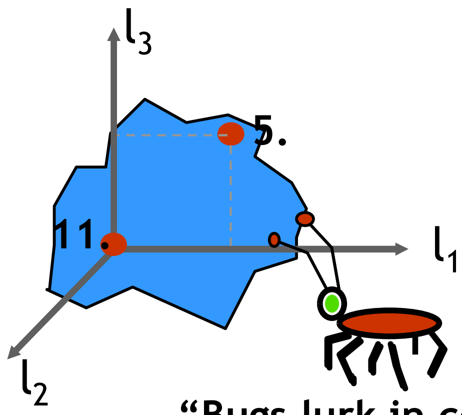
Test cases for:

valid inputs:

1. valid scalene triangle ?
2. valid equilateral triangle ?
3. valid isosceles triangle ?
4. 3 permutations of previous ?

invalid inputs:

5. side = 0 ? (boundary "plane")
6. negative side ?
7. one side is sum of others ? (boundary)
8. 3 permutations of previous ?
9. one side larger than sum of others ?
10. 3 permutations of previous ?
11. all sides = 0 ? (boundary "corner")
12. non-integer input ?



"Bugs lurk in corners and congregate at boundaries."

Boundary value analysis
Example 2

- Given inputs maxint and N compute result :

$$\text{result} = \sum_{K=0}^{|N|} k \quad \text{if this} <= \text{maxint}, \quad \text{error otherwise}$$

  - Valid equivalence classes :

condition	valid eq. classes	boundary values.
$\|N\|$	$N < 0$, $N \geq 0$	$N = -1, 0, 1$
maxint	$\sum k \leq$ maxint,	$\sum k =$ maxint-1,
	$\sum k >$ maxint	maxint,
		maxint+1

Boundary value analysis
Example 3: search routine specification

```
procedure Search (Key : ELEM ; T: ELEM_ARRAY;
      Found : out BOOLEAN; L: out ELEM_INDEX) ;

Pre-condition
            -- the array has at least one element
            T'FIRST <= T'LAST
Post-condition
            -- the element is found and is referenced by L
            ( Found and T (L) = Key)
or
            -- the element is not in the array
            ( not Found and
            not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

Boundary value analysis
Example 3 - input partitions

- P1 - Inputs which conform to the pre-conditions (valid)
  - array with 1 value (boundary)
  - array with more than one value (different size from test case to test case)
- P2 - Inputs where a pre-condition does not hold (invalid)
  - array with zero length
- P3 - Inputs where the key element is a member of the array
  - first, last and middle positions in different test cases
- P4 - Inputs where the key element is not a member of the array

Boundary value analysis
Example 3 - test cases (valid cases only)

Array	**Element**
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

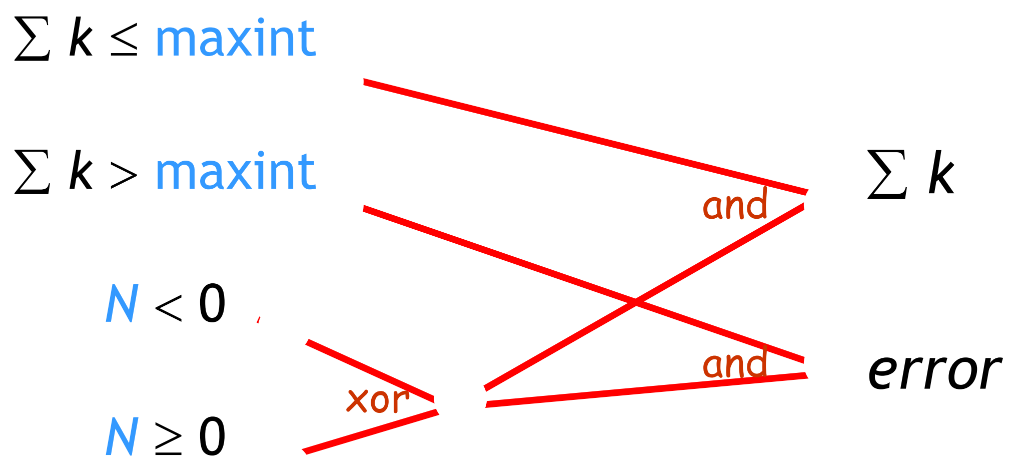
Cause-effect graphing

- Black-box technique to analyze combinations of input conditions
- Identify causes and effects in specification

$$\begin{tikzcd}
\text{causes} \arrow[d] & \text{effects} \arrow[d] \\
\text{inputs / initial state conditions} & \text{outputs / final state conditions}
\end{tikzcd}$$

- Make a Boolean Graph linking causes and effects
- Annotate impossible combinations of causes and effects
- Develop a decision table from a graph with each column a particular combination of inputs and outputs
- Transform each column into a test case

Cause-effect graphing
Example 2



Decision table

("truth table")

Each entry in de decision table is a 0 or a 1 depending on whether or not the corresponding condition is false or true

causes	$\sum k \leq$ maxint	1	1	0	0
(inputs)	$\sum k >$ maxint	0	0	1	1
	$N < 0$	1	0	1	0
	$N \geq 0$	0	1	0	1
effects	$\sum k$	1	1	0	0
(outputs)	*error*	0	0	1	1

Cause-effect graphing

- Systematic method for generating test cases representing combinations of conditions
- Differently from eq. class partitioning, we define a test case for each possible combination of conditions
- Combinatorial explosion of number of possible combinations
  - In the worst case, if *n* causes are related to an effect *e*, then the maximum number of combinations that bring *e* to a *1-state* is 2^n.

Error guessing

- Just ‘guess’ where the errors are ……
- Intuition and experience of tester
- Ad hoc, not really a technique
- But can be quite effective
- Strategy:
  - Make a list of possible errors or error-prone situations (often related to boundary conditions)
  - Write test cases based on this list

Risk based testing

- **Risk-based testing** (RBT) is a type of software testing that prioritizes the features and functions to be tested based on priority/importance and likelihood or impact of failure. More sophisticated 'error guessing'
- Try to identify critical parts of program (high risk code sections):
  - parts with unclear specifications
  - developed by junior programmer while his wife was pregnant ……
  - complex code :
    measure code complexity - tools available (McGabe, Logiscope,…)
- High-risk code will be more thoroughly tested
  (or be rewritten immediately ……)

Testing for race conditions

- Also called bad timing and concurrency problems
- Problems that occur in multitasking systems (with multiple threads or processes)
- A kind of boundary analysis related to the dynamic views of a system (state-transition view and process-communication view)
- Examples of situations that may expose race conditions:
  - problems with shared resources:
    - saving and loading the same document at the same time with different programs
    - sharing the same printer, communications port or other peripheral
    - using different programs (or instances of a program) to simultaneously access a common database
  - problems with interruptions:
    - pressing keys or sending mouse clicks while the software is loading or changing states
  - other problems:
    - shutting down or starting two or more instances of the software at the same time
- Knowledge used: dynamic models (state-transition models, process models)

Random testing

- Input values are randomly generated
- Do you know that a monkey using a piano keyboard could play a Vivaldi opera? Could the same monkey, using your application, discovery defects?
- Two kinds of tools
  - **Dumb monkeys** – low IQ; they can't recognize an error when they see one
  - **Smart monkeys** – generate inputs with some knowledge to reflect expected usage; get knowledge from state table or model of the AUT.
- Microsoft says that 10 to 20% of the bugs in Microsoft projects are found by these tools

Random testing

- Advantages
  - Good for finding system crashes
  - Particularly adequate for performance testing (it's not necessary to check the correctness of outputs)
  - No effort in generating test cases
  - Independent of updates
  - Increase confidence on the software when running several hours without finding errors
  - “Easy” to implement
- Disadvantages
  - Not good for finding other kinds of errors
  - Difficult to reproduce the errors (repeat test cases / sequence of inputs)
  - Unpredictable
  - May not cover special cases that are discovered by "manual" techniques

Deriving test cases from requirements and use cases

- Particularly adequate for system and acceptance testing
- From requirements:
  - You have a list of requirements
  - Define at least one test case for each requirement
  - Build and maintain a (tests to requirements) traceability matrix
- From use cases:
  - You have use cases that capture functional requirements
  - Each use case is described by one or more *normal* flow of events and zero or more *exceptional* flow of events
  - Define at least one test case for each flow of events (also called *scenario*)
  - Build and maintain a (tests to use cases) traceability matrix

State-transition testing

- Construct a state-transition model (state machine view) of the item to be tested (from the perspective of a user/client). E.g., with a state diagram in UML
- Define test cases to exercise all states and all transitions between states
  - Usually, not all possible paths (sequences of states and transitions), because of combinatorial explosion
  - Each test case describes a sequence of inputs and outputs (including input and output states), and may cover several states and transitions
  - Also test to fail – with unexpected inputs for a particular state
- We will talk about this techniques in more detail in the following lectures

Black box testing: Which One ?

- Black box testing techniques :
  - Equivalence partitioning
  - Boundary value analysis
  - Cause-effect graphing
  - Error guessing
  - …………
- Which one to use ?
  - None of them is complete
  - All are based on some kind of heuristics
  - They are complementary

Black box testing: which one ?

- Always use a combination of techniques
  - When a formal specification is available try to use it
  - Identify valid and invalid input equivalence classes
  - Identify output equivalence classes
  - Apply boundary value analysis on valid equivalence classes
  - Guess about possible errors
  - Cause-effect graphing for linking inputs and outputs

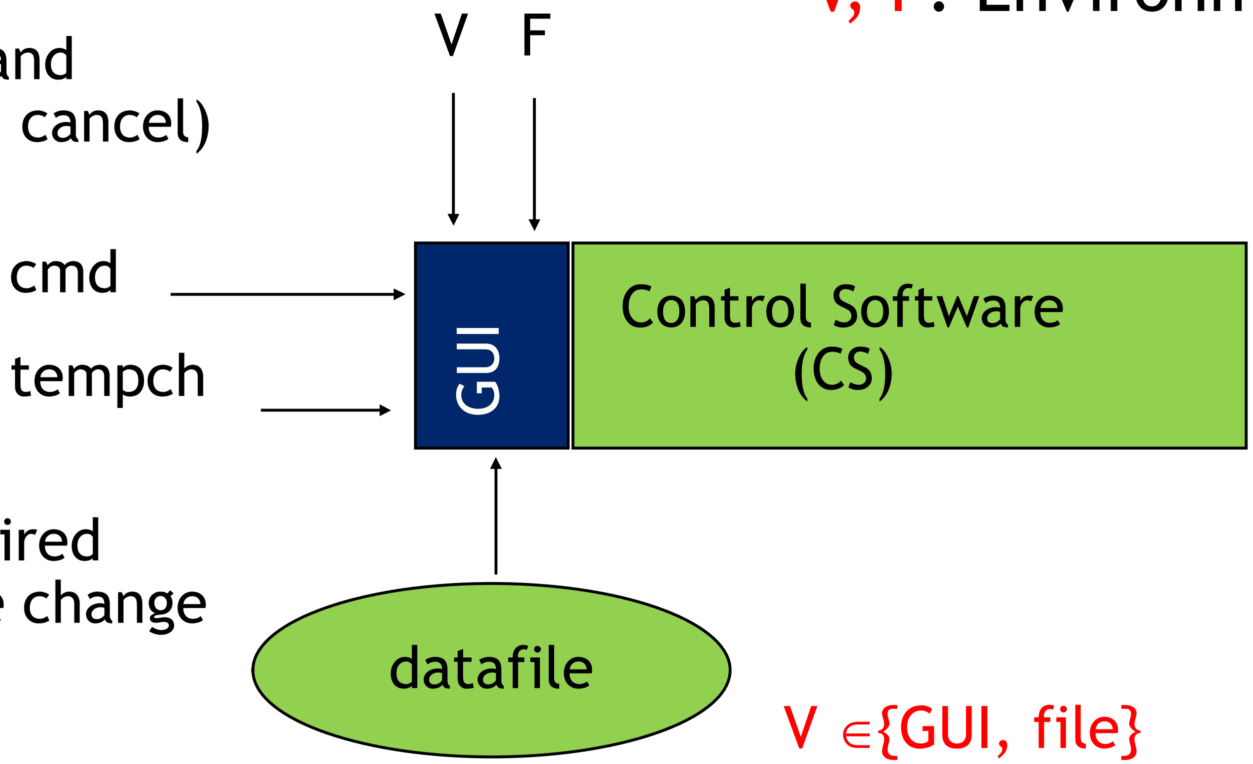
Exercise

- The control software of BCS, abbreviated as CS, allows a human operator to give one of three commands (*cmd*): change the boiler temperature (*temp*), shut down the boiler (*shut*), and cancel the request (*cancel*).
- Command temp causes CS to ask the operator to enter the amount by which the temperature is to be changed (*tempch*). Values of *tempch* are in the range [-10..10] in increments of 5 degrees Fahrenheit. A temperature change of 0 is not an option.
- BCS examines variable *V*. If *V* is set to *GUI*, the operator is asked to enter one of the three commands via a *GUI*. However, if *V* is set to *file*, BCS obtains the command from a command file.
- The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is *temp*. The file name is obtained from variable *F*.

Exercise

V, F: Environment variables

cmd: command
(temp, shut, cancel)



tempch: desired temperature change
(-10..10)

V $\in$ {GUI, file}

F: file name if V is set to “file.”

Exercise

- Identify input domain
- Identify equivalence classes
- Combine equivalence classes
- Discard infeasible equivalence classes
- Generate sample tests for BCS from the remaining feasible equivalence classes

Identify input domain

Variable	Type	Value(s)
V	Enumerated	GUI or file
F	String	A file name
cmd	Enumerated	temp, cancel, or shut
tempch	Enumerated	-10, -5, 5, or 10

S = V x F x cmd x tempch

Equivalence classing

Variable	Type	Value(s)	Class
V	Enumerated	GUI or file	{GUI},{file},{undefined}
F	String	A file name	f_valid, f_invalid
cmd	Enumerated	temp, cancel, or shut	{temp},{cancel},{shut},{c_invalid}
tempch	Enumerated	-10, -5, 5, or 10	{-10},{-5},{5},{10},{t_invalid}

S = V x F x cmd x tempch

Combine equivalence classes

Variable	Type	Value(s)	Class
V	Enumerated	GUI or file	{GUI},{file},{undefined}
F	String	A file name	{f_valid}, {f_invalid}
cmd	Enumerated	temp, cancel, or shut	{temp},{cancel},{shut},{c_invalid}
tempch	Enumerated	-10, -5, 5, or 10	{-10},{-5},{5},{10},{t_invalid}

S = V x F x cmd x tempch

Variables V, F, cmd and tempch have been partitioned into 3, 2, 4 and 2 sets, respectively

Set products of these four variables leads to a total of

3 x 2 x 4 x 5 = 120

Discard infeasable classes - example

- Carefully designed application might not ask for the values of **tempch** when **V=file** and **F** contains a file name that does not exist
  - $\#\{(\underbrace{\textbf{file},}\quad \underbrace{\textbf{f\_invalid},}\quad \underbrace{\text{temp},}\quad \underbrace{\textbf{t\_valid U t\_invalid}})\} =$

    $1 \times 1 \times 1 \times 5 = 5$

- GUI does not allow invalid values of temperature change to be input. More combinations infeasible
  - $\#\{(\underbrace{\textbf{GUI,}}\quad \underbrace{\text{F ,}}\quad \underbrace{\text{temp},}\quad \underbrace{\textbf{t\_invalid}})\} =$

    $1 \times 2 \times 1 \times 1 = 2$

  (…)

Some remaining feasible classes

{(GUI, – , temp, t_valid)}	= 8
{(GUI, –, cancel, NA)}	= 2
{(file, f_valid, temp, t_valid U t_invalid)}	= 5
{(undefined, NA, NA, NA)}	= 1
(…)	

‘–’ means that data can be input but is not used by the software

‘NA’ means that data cannot be input to the control software

References

- Practical Software Testing, Ilene Burnstein, Springer-Verlag, 2003
- Software Testing, Ron Patton, SAMS, 2001
- The Art of Software Testing, Glenford J. Myers, Wiley & Sons, 1979 (Chapter 4 - Test Case Design)
  - Classical
- Software testing techniques, Boris Beizer, Van Nostrand Reinhold, 2nd Ed, 1990
  - Bible
- Foundations of Software Testing, 1st edition, by Aditya P. Mathur, Pearson Education, 2008
- Testing Computer Software, 2nd Edition, Cem Kaner, Jack Falk, Hung Nguyen, John Wiley & Sons, 1999
  - Practical, black box only
- Software Engineering, Ian Sommerville, 6th Edition, Addison-Wesley, 2000
- Test Driven Development, Kent Beck, Addison Wesley, 2002
- http://www.swebok.org/
  - Guide to the Software Engineering Body of Knowledge (SWEBOK), IEEE Computer Society
- http://www.mccabe.com/iq_research_metrics.htm
- What Is a Good Test Case?, Cem Kaner, Florida Institute of Tecnology, 2003
- Software Unit Test Coverage and Adequacy, Hong Zhu et al, ACM Computing Surveys, December 1997