



AARHUS UNIVERSITET

# **Software Engineering and Architecture**

Systematic Testing  
Equivalence Class Partitioning

# What is reliability?

## Definition: Reliability (ISO 9126)

The capability of the software product to maintain a specified level of performance when used under specified conditions.

- If there is a defect, it may fail and is thus not reliable.
- Thus:
  - Reduce number of defects
  - ... will reduce number of failures
  - ... will increase reliability



- True?
  - Find example of **removing a defect does not increase the system's reliability at all**
  
  - Find example of **removing defect 1 increase reliability dramatically while removing defect 2 does not**



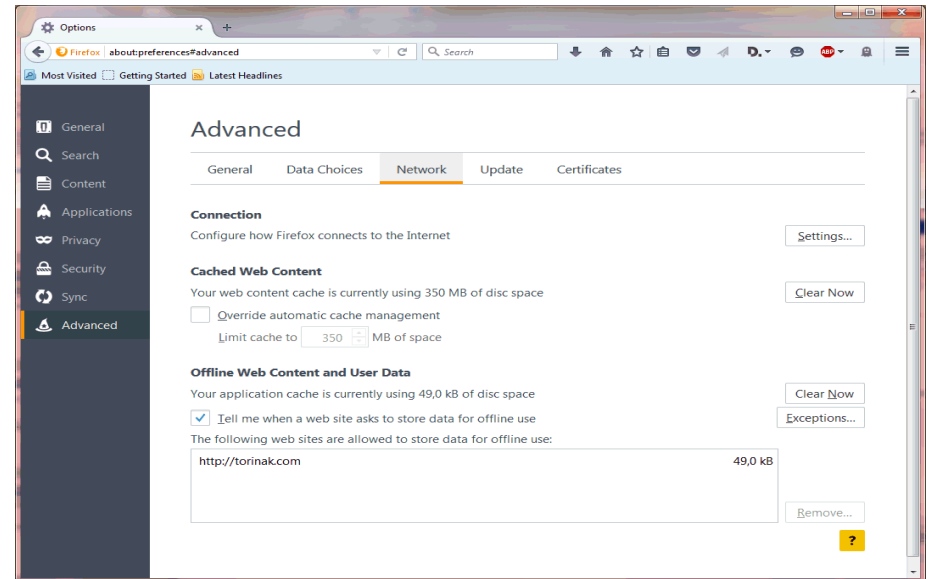
# All defects are not equal!

AARHUS UNIVERSITET

- So – given a certain amount of time to find defects (you need to sell things to earn money!):
  
- What kind of defects should you correct to get best *return on investment*?

# Example

- Firefox has an enormous amount of possible configurations! Imagine testing *all possible combinations!*
- Which one would you test most thoroughly?





AARHUS UNIVERSITET

# Testing Techniques

# One viewpoint

- The probability of defects is a function of the code *complexity*. Thus we may identify three different testing approaches:
  - **No testing**. Complexity is so low that the test code will become more complex. Example: set/get methods
  - **Explorative** testing: “gut feeling”, experience. TDD relies heavily on ‘making the smart test case’ but does not dictate any method.

## Definition: Systematic testing

Systematic testing is a planned and systematic process with the explicit goal of finding defects in some well-defined part of the system.



# Destructive!

- Testing is a *destructive process!!!*
  - In contrast to almost all other SW processes which are constructive!
- Human psychology
  - **I want to be a success**
    - **The brain will deliver!**
      - (ok, X-factor shows this is not always the case...)
- *I will prove my software works*
- *I will prove my software is really lousy*





- When testing, reprogram your brains

- ***I am a success if I find a defect.  
The more I find, the better I am!***

- There are a lot of different testing techniques.

- Types:

## Definition: **Black-box testing**

The *unit under test* (UUT) is treated as a black box. The only knowledge we have to guide our testing effort is the specification of the UUT and a general knowledge of common programming techniques, algorithmic constructs, and common mistakes made by programmers.

## Definition: **White-box testing**

The full implementation of the unit under test is known, so the actual code can be inspected in order to generate test cases.

- Here – two black box techniques:
  - Equivalence class partitioning
  - Boundary value analysis



AARHUS UNIVERSITET

# Equivalence Class Partitioning

- `Math.abs(x)`: Absolute value of  $x$ 
  - If  $x$  is not negative, return  $x$ .
  - If  $x$  is negative, return the negation of  $x$ .
- Will these five test cases ensure a reliable implementation?

Unit under test: <code>Math.abs</code>	
Input	Expected output
$x = 37$	37
$x = 38$	38
$x = 39$	39
$x = 40$	40
$x = 41$	41

# Two problems

- A) what is the probability that  $x=38$  will find a defect that  $x=37$  did not expose?
- B) what is the probability that there will be a defect in handling negative  $x$ ?

Unit under test: Math.abs	
Input	Expected output
$x = 37$	37
$x = 38$	38
$x = 39$	39
$x = 40$	40
$x = 41$	41

```
public static int abs(int x) { return x; }
```

- *We can find a **single** input value that represents a large set of values when it comes to finding defects!*

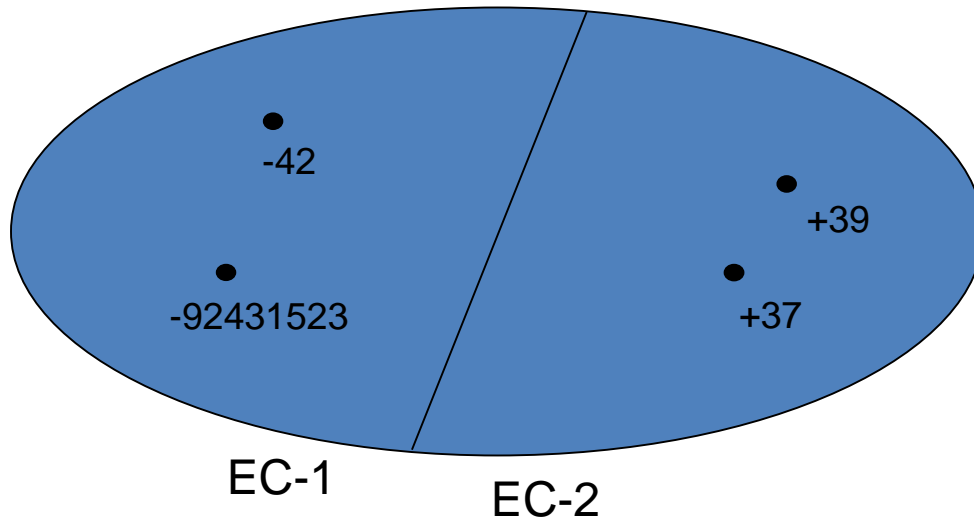
## Definition: Equivalence class (EC)

A subset of all possible inputs to the UUT that has the property that if one element in the subset demonstrates a defect during testing, then we assume that all other elements in the subset will demonstrate the *same* defect.

An EC is a (sub)set.  
Da: Delmængde

# For Math.abs

- ECs are subsets of the full input set.



Unit under test: Math.abs	
Input	Expected output
$x = 37$	37
<del><math>x = 38</math></del>	<del>38</del>
<del><math>x = 39</math></del>	<del>39</del>
<del><math>x = 40</math></del>	<del>40</del>
<del><math>x = 41</math></del>	<del>41</del>
$X = -42$	42



# The argumentation

- The specification will force an implementation where (most likely!) all *positive* arguments are treated by one code fragment and all *negative* arguments by another.
  - Thus we only need two test cases:
    - 1) one that tests the positive argument handling code
    - 2) one that tests the negative argument handling code
- For 1)  $x=37$  is just as good as  $x=1232$ , etc. Thus we simply *select a representative element* from each EC and generate a test case based upon it.



- Systematic testing ...
- **does *not* mean**
  - *Systematically find all defects and guaranty none are left!!!*
- **does mean**
  - *Systematically derive a **small set of test cases with high probability** of finding many defects!!!*



# Specifically

- Systematic testing cannot in any way counter *malicious programming*
  - *Virus, easter eggs, evil-minded programmers*
  - *(really really lousy programmers)*

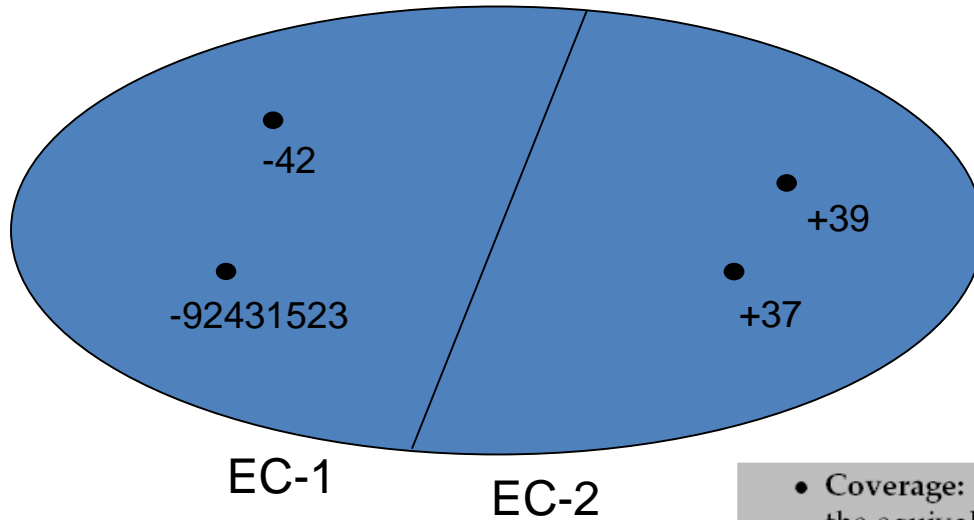


# A sound EC partitioning

- For our ECs to be *sound*:

- **Coverage:** Every possible input element belongs to at least one of the equivalence classes.
- **Representation:** If a defect is demonstrated on a particular member of an equivalence class, the same defect is assumed to be demonstrated by any other member of the class.

- Coverage? Representation?



- **Coverage:** Every possible input element belongs to at least one of the equivalence classes.
- **Representation:** If a defect is demonstrated on a particular member of an equivalence class, the same defect is assumed to be demonstrated by any other member of the class.

- The classic blunder at exams!
  - *Representation = two values from the same EC will result in the **same behaviour** in the algorithm*
- Argue why this is completely wrong!
  - Consider for instance the Account class' deposit method...
    - `account.deposit(100);`
    - `account.deposit(1000000);`

# Documenting ECs

- Math.abs is simple. Thus the ECs are simple.
- This is often **not** the case! Finding ECs require deep thinking, analysis, and iteration.
- Document them!
- **Equivalence class table:**

Condition	Invalid ECs	Valid ECs
absolute value of $x$	–	$x > 0[a1]$ $x \leq 0[a2]$

# Invalid/Valid ECs

- Some input values make an algorithm *give up*, *bail out*, or *compute answer immediately*:
  - `file.open("nonexistingfile");`
  - `g.moveUnitTo([mountain])`
- Input values that leads to *abnormal processing/giving up* we classify as belonging to ***invalid ECs***.
- Those input values that process normally we say belong to ***valid ECs***.



Bail out fast



AARHUS UNIVERSITET

# Finding the ECs

Often tricky...



## Example: Backgammon move validation

- **validity = v ( move, player, board-state, die-value )**
  - is Red move (B1-B2) valid on this board given this die?
- Problem:
  - multiple parameters: player, move, board, die
  - complex parameters: board state
  - coupled parameters: die couples to move!
    - EC boundary is not a constant!

- To find the ECs you will need to look carefully at the specification and especially all the **conditions** associated.
  - Conditions express choices in our algorithms and therefore typically defines disjoint algorithm parts!
  - And as the test cases should at least run through all parts of the algorithms, it defines the boundaries for the ECs.
- ... And consider typical programming techniques
  - Will a program contain if's and while's here?



# Partitioning Heuristics

- 2) If you have a *set, S, of values* specification
  - make  $|S|+1$  ECs
    - $[1]..[|S|]$  one for each member in S                      valid
    - $[|S|+1]$  for a value outside S                                      invalid
  
- Ex. PayStation accepting coins
  - Set of {5, 10, 25} cents coins

Condition	Invalid ECs	Valid ECs
Allowed coins	$\notin \{5, 10, 25\}[a1]$	$\{5\}[a2]; \{10\}[a3]; \{25\}[a4]$

# Partitioning Heuristics

- 3) If you have a *boolean condition* specification
  - make **two** ECs
- Ex. the first character must be a letter

Condition	Invalid ECs	Valid ECs
Initial character of identifier	non-letter [a1]	letter [a2]

- the object reference must not be null
  - *you get it, right?* 😊



# Partitioning Heuristics

- 4) If you question the representation property of any EC, then repartition!
  - Split EC into smaller ECs
- Examples: shortly 😊



# Computation Heuristics

AARHUS UNIVERSITET

- Computations have their own pitfalls: 0 and 1

If the specification of the unit under test defines an *arithmetic computation*, then

- **Addition and subtraction:** *If a computation includes addition or subtraction, select one valid EC for the neutral element 0, and one valid EC for all other elements.*
- **Multiplication and division:** *If a computation includes multiplication or division, select one valid EC for the neutral element 1, and one valid EC for all other elements.*

- Specification:

```
// return a*x + b
public double linearFunction(double a, double x, double b) {
    return a*x + b;
}
```

- But: No condition => only one test case => a=x=b=0!

```
// return a*x + b
public double linearFunction(double a, double x, double b) {
    return 0.0;
}
```

Condition	Invalid ECs	Valid ECs
multiplication (a*x)	-	a = 1 [a1]; a != 1 [a2]
	-	x = 1 [a3]; x != 1 [a4]
addition (+b)	-	b = 0 [b1]; b != 0 [b2]

Note: the neutral element ECs need not be tested at all!





# Computation Heuristics

AARHUS UNIVERSITET

- Example (or a strong hint 😊) – HotCiv attacks
  - Implementation 1:
    - Value = attackStrength \* dieValue;
  - Implementation 2:
    - Value = attackStrength;
  - What happens if you *choose 1 as die value in all your test cases???*
  - Using the heuristics:
    - Condition
    - Die value
- Valid EC
- {1} [A], {2,3,4,5,6} [B]



AARHUS UNIVERSITET

# From ECs to Test cases

- For disjoint ECs you simply pick an element from each EC to make a test case.
- **Extended test case table**

ECs covered	Test case	Expected output
[a1]	$x = 201$	+201
[a2]	$x = -87$	+87

- Augmented with a column showing which ECs are covered!

Condition	Invalid ECs	Valid ECs
absolute value of x	-	$x > 0[a1]$ $x \leq 0[a2]$

- ECs are seldom disjoint – so you have to *combine* them to generate test cases.
- Ex. Chess board validation

Condition	Invalid ECs	Valid ECs
Column	< 'a' [a1]; > 'h' [a2]	'a'-'h' [a3]
Row	< 1 [b1]; > 8 [b2]	1-8 [b3]

ECs covered	Test case	Expected output
[a1], [b1]	(' ',0)	illegal
[a2], [b1]	('i',-2)	illegal
[a3], [b1]	('e',0)	illegal
[a1], [b2]	(' ',9)	illegal
[a2], [b2]	('j',9)	illegal
[a3], [b2]	('f',12)	illegal
[a1], [b3]	(' ',4)	illegal
[a2], [b3]	('i',5)	illegal
[a3], [b3]	('b',6)	legal

# Combinatorial Explosion

- Umph! Combinatorial explosion of test cases 😞.
- Ex.
  - Three independent input parameters  $\text{foo}(x,y,z)$
  - Four ECs for each parameter
    - That is:
      - x's input set divided into four ECs,
      - y's input set divided into four ECs,
    - etc.
- Question:
  - How many test cases?



# Combinatorial Explosion

AARHUS UNIVERSITET

- Answer:  $4^3 = 64$  test cases...
- TC1: [ECx1],[ECy1],[ECz1]
- TC2: [ECx1],[ECy1],[ECz2] ...
- TC4: [ECx1],[ECy1],[ECz4] ...
- TC5: [ECx1],[ECy2],[ECz1] ...
- TC9: [ECx1],[ECy3],[ECz1] ...
- ...
- TC64: [ECx4],[ECy4],[ECz4]

- Often, generate test cases like this is better:
  1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.
  2. Until all invalid ECs have been covered, define a test case whose element only lies in a single invalid ECs.

Condition	Invalid ECs	Valid ECs
Column	< 'a' [a1]; > 'h' [a2]	'a'-'h' [a3]
Row	< 1 [b1]; > 8 [b2]	1-8 [b3]

Rule 2

Rule 1

ECs covered	Test case	Expected output
[a1], [b3]	(' ',5)	illegal
[a2], [b3]	('j',3)	illegal
[a3], [b1]	('b',0)	illegal
[a3], [b2]	('c',9)	illegal
[a3], [b3]	('b',6)	legal

# Why Rule 2?

- Due to **masking**
  - One correct test *masks* a later incorrect test
- Ex.

```
public class ChessBoard {  
    public boolean valid(char column, int row) {  
        if ( column < 'a' ) { return false; }  
        if ( row < 0 ) { return false; }  
        return true;  
    }  
}
```

```
assertThat(b.valid(' ',0), is(false));
```

False positive

- Test case ( ' ',0) will pass which is expected
  - Code deemed correct, but this is a **wrong** conclusion!
    - It was the 'column' test that returned 'false' so the defect in the row conditions is **masked**





# Why Rule 1?

- You may combine as *many* valid ECs as possible and cover it with only a single test case until all valid ECs are exhausted...
- Why?
- Because I must assume that all elements from ECs are used to compute the final result. Any defects will thus most likely show up even if the defect only relate to one element.
- Ex. `bankdayNumberInYear(int month, int day)`
  - return  $30 * \text{month} + \text{day}$

Jan = 1; Feb = 2



AARHUS UNIVERSITET

# The Process



1. Review the requirements for the UUT and identify *conditions* and use the heuristics to find ECs for each condition. ECs are best written down in an *equivalence class table*.
2. Review the produced ECs and consider carefully the representation property of elements in each EC. If you question if particular elements are really representative then repartition the EC.
3. Review to verify that the coverage property is fulfilled.
4. Generate test cases from the ECs. You can often use Myers heuristics for combination to generate a minimal set of test cases. Test cases are best documented using a *test case table*.
5. Review the generated test cases carefully to find what you missed – and iterate!



AARHUS UNIVERSITET

# An Example

Phew – let's see things in practice



```
public interface weekday {  
  
    /** calculate the weekday of the 1st day of the given month.  
     * @param year the year as integer. 2000 means year 2000 etc. Only  
     * years in the range 1900-3000 are valid. The output is undefined  
     * for years outside this range.  
     * @param month the month as integer. 1 means January, 12 means  
     * December. Values outside the range 1-12 are illegal.  
     * @return the weekday of the 1st day of the month. 0 means Sunday,  
     * 1 means Monday etc. up til 6 meaning Saturday.  
     */  
    public int weekday(int year, int month)  
        throws IllegalArgumentException;  
}
```

- Which heuristics to use on the spec?
  - **Range:** *If a condition is specified as a range of values, select one valid EC that covers the allowed range, and two invalid ECs, one above and one below the end of the range.*
  - **Set:** *If a condition is specified as a set of values then define an EC for each value in the set and one EC containing all elements outside the set.*
  - **Boolean:** *If a condition is specified as a “must be” condition then define one EC for the condition being true and one EC for the condition being false.*



# Process step 1

- Result

Condition	Invalid ECs	Valid ECs
year	$< 1900 [y1]; > 3000 [y2]$	$1900 - 3000 [y3]$
month	$< 1 [m1]; > 12 [m2]$	$1 - 12 [m3]$

# Process step 2

- Review and consider representation property?

Condition	Invalid ECs	Valid ECs
year	$< 1900 [y1]; > 3000 [y2]$	$1900 - 3000 [y3]$
month	$< 1 [m1]; > 12 [m2]$	$1 - 12 [m3]$



- Nice to be an astronomer 😊

Condition	Invalid ECs	Valid ECs
year ( $y$ )		$\{y   y \in [1900; 3000] \wedge y \% 400 = 0\}$ [ $y3a$ ] $\{y   y \in [1900; 3000] \wedge y \% 100 = 0 \wedge y \notin [y3a]\}$ [ $y3b$ ] $\{y   y \in [1900; 3000] \wedge y \% 4 = 0 \wedge y \notin [y3a] \cup [y3b]\}$ [ $y3c$ ] $\{y   y \in [1900; 3000] \wedge y \% 4 \neq 0\}$ [ $y3d$ ]

- What about the months? Let's play it safe...

Condition	Invalid ECs	Valid ECs
month		1 – 2 [ $m3a$ ]; 3 – 12 [ $m3b$ ]



AARHUS UNIVERSITET

- Coverage?

# Process step 3



# Process step 4

AARHUS UNIVERSITET

- Generate, using Myers

ECs covered	Test case	Expected output	Invalid ECs	Valid ECs
[y3a], [m3a]	y = 2000; m = 2	-	$\{y y \in [1900; 3000] \wedge y\%400 = 0\} [y3a]$ $\{y y \in [1900; 3000] \wedge y\%100 = 0 \wedge y \notin [y3a]\} [y3b]$ $\{y y \in [1900; 3000] \wedge y\%4 = 0 \wedge y \notin [y3a] \cup [y3b]\} [y3c]$ $\{y y \in [1900; 3000] \wedge y\%4 \neq 0\} [y3d]$	$\{y y \in [1900; 3000] \wedge y\%400 = 0\} [y3a]$ $\{y y \in [1900; 3000] \wedge y\%100 = 0 \wedge y \notin [y3a]\} [y3b]$ $\{y y \in [1900; 3000] \wedge y\%4 = 0 \wedge y \notin [y3a] \cup [y3b]\} [y3c]$ $\{y y \in [1900; 3000] \wedge y\%4 \neq 0\} [y3d]$
[y3b], [m3b]	y = 1900; m = 5	-		
[y3c], [m3b]	y = 2004; m = 10	5		
[y3d], [m3a]	y = 1985; m = 1	-		
[y1], [m3b]	y = 1844; m = 4	[exception]		
[y2], [m3b]	y = 4231; m = 8	[exception]		
[m1], [y3d]	y = 2003; m = 0	[exception]		
[m2], [y3b]	y = 2004; m = 13	[exception]		

Condition	Invalid ECs	Valid ECs
month		1 – 2 [m3a]; 3 – 12 [m3b]

Condition	Invalid ECs	Valid ECs
year	< 1900 [y1]; > 3000 [y2]	1900 – 3000 [y3]
month	< 1 [m1]; > 12 [m2]	1 – 12 [m3]

[y3c]

- Conclusion: *Only 8 test cases* for a rather tricky alg.



AARHUS UNIVERSITET

# Example 2



```
/** format a string representing of a double. The string is always  
6 characters wide and in the form ###.##, that is the double is  
rounded to 2 digit precision. Numbers smaller than 100 have '0'  
prefix. Example: 123 -> '123.00'; 2,3476 -> '002.35' etc. If the  
number is larger or equal to 999.995 then '***.**' is output to  
signal overflow. All negative values are signaled with '----.---'  
*/  
public String format(double x);
```

- Note! Most of the conditions do **not** really talk about x but on the output.
- Remember: *Conditions in the specs!!!*

- What do we do?

```
/** format a string representing of a double. The string is always  
6 characters wide and in the form ###.##, that is the double is  
rounded to 2 digit precision. Numbers smaller than 100 have '0'  
prefix. Example: 123 -> '123.00'; 2,3476 -> '002.35' etc. If the  
number is larger or equal to 999.995 then '***.**' is output to  
signal overflow. All negative values are signaled with '---.--'  
*/  
public String format(double x);
```

- Range? Boolean? Set?
- Valid / Invalid ECs?

# My analysis

Condition	Invalid ECs	Valid ECs
overflow / underflow 2 digit rounding  prefix          output suffix	$\geq 1000.0 [a1]; < 0.0 [a2]$	$0.0 \leq x < 1000.0 [a3]$ (,00x round up) [b1]; (,00x round down) [b2] no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4] '.yx' suffix ( $x \neq 0$ ) [d1] '.x0' suffix ( $x \neq 0$ ) [d2] exact '.00' suffix [d3]



# Test cases

ECs covered	Test case	Expected output
[a1]	1234.456	'***. **'
[a2]	-0.1	'—.-'
[b1], [c1], [d1]	212.738	'212.74'
[b2], [c2], [d2]	32.503	'032.50'
[b1], [c3], [d3]	7.995	'008.00'
[b2], [c4], [d1]	0.933	'000.93'

Condition	Invalid ECs	Valid ECs
overflow / underflow 2 digit rounding	$\geq 1000.0$ [a1]; $< 0.0$ [a2]	(,00x round up) [b1]; (,00x round down) [b2]
prefix		no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4]
output suffix		'yx' suffix ( $x \neq 0$ ) [d1] 'x0' suffix ( $x \neq 0$ ) [d2] exact '.00' suffix [d3]





# Process Recap

AARHUS UNIVERSITET

1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.
2. Until all invalid ECs have been covered, define a test case whose elements lie in a single invalid ECs.

Condition	Invalid ECs	Valid
overflow / underflow 2 digit rounding	$\geq 1000.0$ [a1]; $< 0.0$ [a2]	(,00x round (,00x round down) [b2]
prefix		no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4]
output suffix		'yx' suffix ( $x \neq 0$ ) [d1] 'x0' suffix ( $x \neq 0$ ) [d2] exact '.00' suffix [d3]

ECs covered	Test case	Expected output
[a1]	1234.456	'***.***'
[a2]	-0.1	'-.-'
[b1], [c1], [d1]	212.738	'212.74'
[b2], [c2], [d2]	32.503	'032.50'
[b1], [c3], [d3]	7.995	'008.00'
[b2], [c4], [d1]	0.933	'000.93'

# Process Recap

1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.
2. Until all invalid ECs have been covered, define a test case whose elements lie in a single invalid ECs.

Condition	Invalid ECs	Valid
overflow / underflow 2 digit rounding	$\geq 1000.0$ [a1]; $< 0.0$ [a2]	(,00x round (,00x round down) [b2]
prefix		no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4]
output suffix		'yx' suffix ( $x \neq 0$ ) [d1] 'x0' suffix ( $x \neq 0$ ) [d2] exact '.00' suffix [d3]

ECs covered	Test case	Expected output
[a1]	1234.456	'***.***'
[a2]	-0.1	'-.-'
[b1], [c1], [d1]	212.738	'212.74'
[b2], [c2], [d2]	32.503	'032.50'
[b1], [c3], [d3]	7.995	'008.00'
[b2], [c4], [d1]	0.933	'000.93'

# Process Recap

1. Until all valid ECs have been covered, define a test case that covers a uncovered valid ECs as possible.
2. Until all invalid ECs have been covered, define a test case whose elements lies in a single invalid ECs.

Condition	Invalid ECs	Valid ECs
overflow / underflow	$\geq 1000.0$ [a1]; $< 0.0$ [a2]	
2 digit rounding		(,00x round up) [b1], (,00x round down) [b2]
prefix		no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4]
output suffix		'yx' suffix ( $x \neq 0$ ) [d1] 'x0' suffix ( $x \neq 0$ ) [d2] exact '.00' suffix [d3]

ECs covered	Test case	Expected output
[c1]	1234.456	'***. **'
[a2]	-0.1	'-.-'
[b1], [c1], [d1]	212.738	'212.74'
[b2], [c2], [d2]	32.503	'032.50'
[b1], [c3], [d3]	7.995	'008.00'
[b2], [c4], [d1]	0.933	'000.93'

# Process Recap

1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.
2. Until all invalid ECs have been covered, define a test case whose elements lie in a single invalid ECs.

Condition	Invalid ECs	Valid ECs
overflow / underflow	$\geq 1000.0$ [a1]; $< 0.0$ [a2]	
2 digit rounding		(,00x round up) [b1]; (,00x round down) [b2]
prefix		no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4]
output suffix		'yx' suffix ( $x \neq 0$ ) [d1] 'x0' suffix ( $x \neq 0$ ) [d2] exact '.00' suffix [d3]

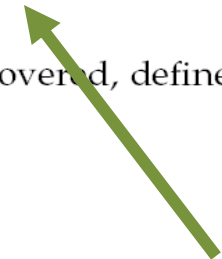
ECs covered	Test case	Expected output
[a1]	1234.456	'***. **'
[a2]	-0.1	'-.-'
[b1], [c1], [d1]	212.738	'212.74'
[b2], [c2], [d2]	32.503	'032.50'
[b1], [c3], [d3]	7.995	'008.00'
[b2], [c4], [d1]	0.933	'000.93'



# Process Recap

AARHUS UNIVERSITET

1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.
2. Until all invalid ECs have been covered, define a test case whose elements lie in a single invalid ECs.



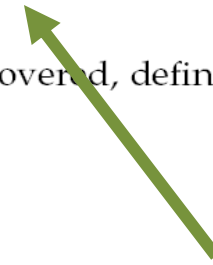
Condition	Invalid ECs	Valid ECs
overflow / underflow	$\geq 1000.0$ [a1]; $< 0.0$ [a2]	(,00x round up) [b1]; (,00x round down) [b2]
2 digit rounding		no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4]
prefix		'yx' suffix ( $x \neq 0$ ) [d1] 'x0' suffix ( $x \neq 0$ ) [d2] exact '.00' suffix [d3]
output suffix		

ECs covered	Test case	Expected output
[a1]	1234.456	'***. **'
[a2]	-0.1	'-.-'
[b1], [c1], [d1]	212.738	'212.74'
[b2], [c2], [d2]	32.503	'032.50'
[b1], [c3], [d3]	7.995	'008.00'
[b2], [c4], [d1]	0.933	'000.93'



# Process Recap

1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.
2. Until all invalid ECs have been covered, define a test case whose elements lie in a single invalid ECs.



Condition	Invalid ECs	Valid ECs
overflow / underflow	$\geq 1000.0$ [a1]; $< 0.0$ [a2]	
2 digit rounding		(,00x round up) [b1]; (,00x round down) [b2]
prefix		no '0' prefix [c1] exact '0' prefix [c2] exact '00' prefix [c3] exact '000' prefix [c4]
output suffix		'yx' suffix ( $x \neq 0$ ) [d1] 'x0' suffix ( $x \neq 0$ ) [d2] exact '.00' suffix [d3]

ECs covered	Test case	Expected output
[a1]	1234.456	'***. **'
[a2]	-0.1	'-.-'
[b1], [c1], [d1]	212.738	'212.74'
[b2], [c2], [d2]	32.503	'032.50'
[b1], [c3], [d3]	7.995	'008.00'
[b2], [c4], [d1]	0.933	'000.93'



# Boundary value analysis



# Boundary value analysis

- Experience shows that test cases focusing on *boundary conditions* have high payoff.
- Some that spring into my mind are
  - *iteration over a C array at its max size*
  - *"off by one" errors in comparisons*
    - *if ( x <= MAX\_SIZE )* and not *if ( x < MAX\_SIZE )*
  - *null as value for a reference/pointer*



# Complements EP analysis

## Definition: **Boundary value**

A boundary value is an element that lies right on or next to the edge of an equivalence class.

- Ex. Formatting has a strong boundary between EC [a1] and [a2]
  - 0.0 (-> '000.00') and -0.000001 (-> '----.---')
- It is thus very interesting to test  $x=0.0$  as boundary.



AARHUS UNIVERSITET

# Discussion



# A few key points

## **Key Point: Observe unit preconditions**

*Do not generate ECs and test cases for conditions that a unit specifically cannot or should not handle.*

## **Key Point: Systematic testing assumes competent programmers**

*Equivalence partitioning and other testing techniques rely on honest and competent programmers that are using standard techniques.*

## **Key Point: Do not use Myers combination heuristics blindly**

*Myers heuristics for generating test cases from valid and invalid ECs can lead to omitting important test cases.*

- Equivalence Class Partitioning
  - EC = set of input elements where each one will show same defect as all others in the same set (representation)
  - Find ECs, use Myers to generate test cases.
- Boundary analysis
  - be sceptical about values at the boundaries
  - Especially on the boundary between valid and invalid ECs