AARHUS UNIVERSITET

Software Engineering and Architecture

Systematic Testing
Equivalence Class Partitioning

AARHUS UNIVERSITET

What is reliability?

Definition: Reliability (ISO 9126)

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

AARHUS UNIVERSITET

Discussion

- 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**

AARHUS UNIVERSITET

All defects are not equal!

- So – given a certain amount of time to find defects (you need to sell things to earn money!):

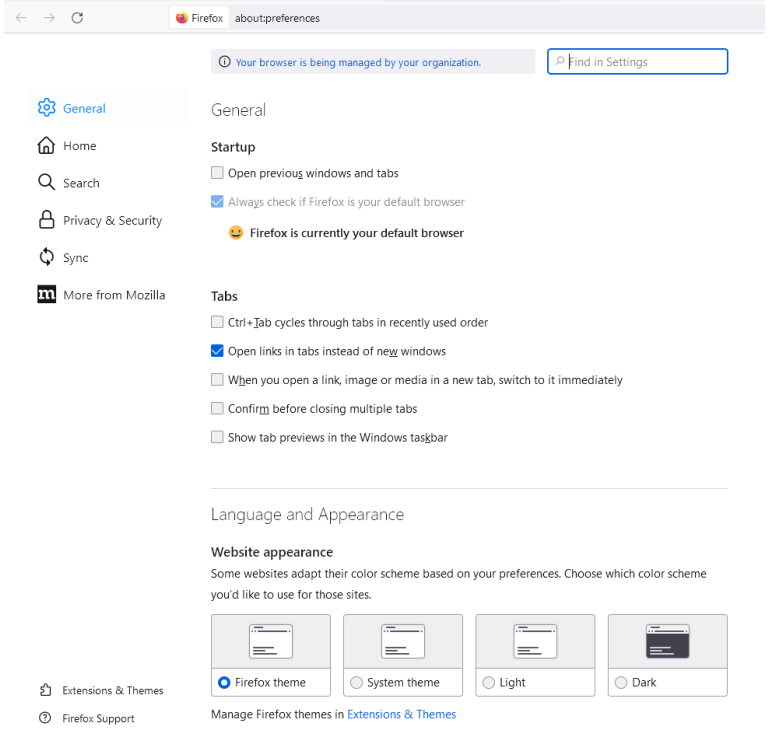
- What kind of defects should you correct to get best *return on investment?*

AARHUS UNIVERSITET

Example

- Firefox has an enormous amount of possible configurations! Imagine testing *all possible combinations!*

- Which one would you test most thoroughly?



Testing Techniques

AARHUS UNIVERSITET

One viewpoint

- The probability of defects is a function of the code *complexity*. Thus we may identify (at least) three different testing approaches:
  - **No testing**. Complexity is so low that the test code will become more complex or longer. 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.

AARHUS UNIVERSITET

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*

Morale:

- When testing, reprogram your brains

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

Our focus

- 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.

- In our course – two **black box** techniques:
  - Equivalence class partitioning (EC)
  - Boundary value analysis (which is actually associated with EC)

AARHUS UNIVERSITET

Equivalence Class Partitioning

Often just called EC testing…

Consider

- Math.abs(x): Absolute value of x;
  - Examples: abs(-2) = 2; abs(7) = 7;
  - 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: Math.abs	
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?

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

- B) what is the probability that there will be a defect in handling negative x?

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

Core insight

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

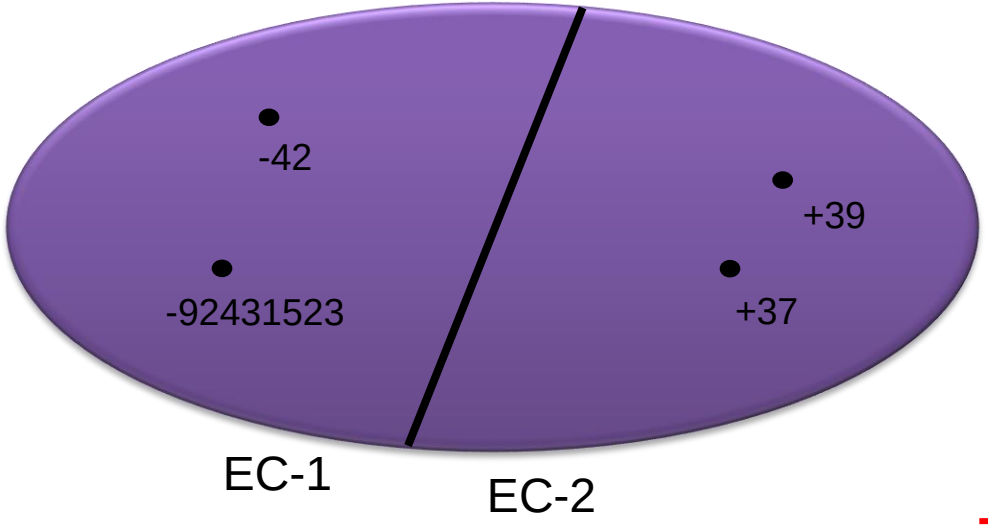
> Definition: **Equivalence class (EC)**
>
> A subset of all possible inputs to the UUT that has the property that if one element in the subset demonstrates a defect during testing, then we assume that all other elements in the subset will demonstrate the *same* defect.

An EC is a (sub)set of all inputs.
Da: Delmængde

- *So – we only need **one test case** representing that set*

For Math.abs

- ECs are subsets of the *full input set.*



Unit under test: Math.abs	
Input	Expected output
x = 37	37
~~x = 38~~	~~38~~
~~x = 39~~	~~39~~
~~x = 40~~	~~40~~
~~x = 41~~	~~41~~
X = - 42	42

AARHUS UNIVERSITET

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.

Note

- Systematic testing ...

- **does *not* mean**
  - *Systematically find all defects and guaranty none are left!!!*
    - *If you need proofs, you need my colleagues research ☺*

- **does mean**
  - Systematically derive a *small set of test cases with high* ***probability*** *of finding many defects!!!*

AARHUS UNIVERSITET

Specifically

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

AARHUS UNIVERSITET

A Sound EC partitioning

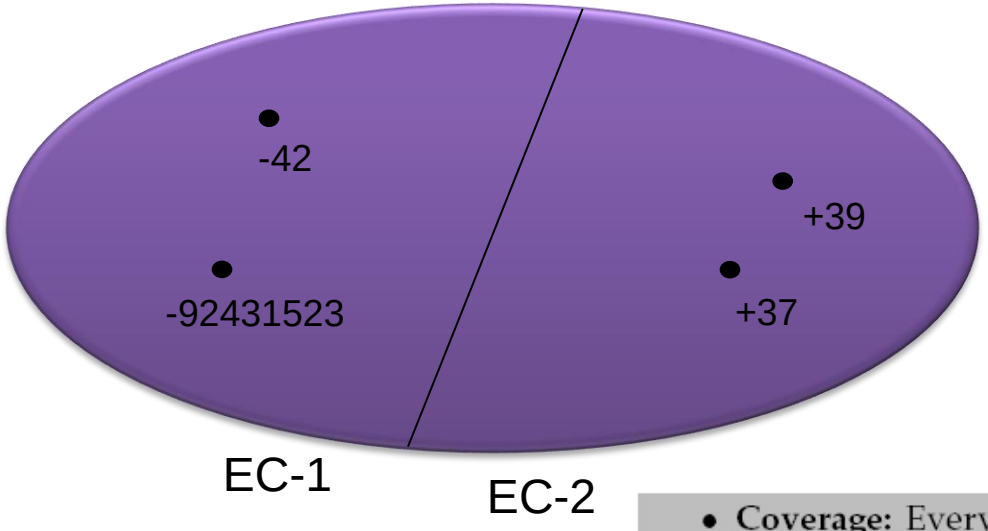
- For our ECs to be *sound:*

  - **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.

- *Exercise*
  - *Why the word 'assumed' in the above statement?*
    - *Why not 'guarantied' ?*

Exercise

- 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.

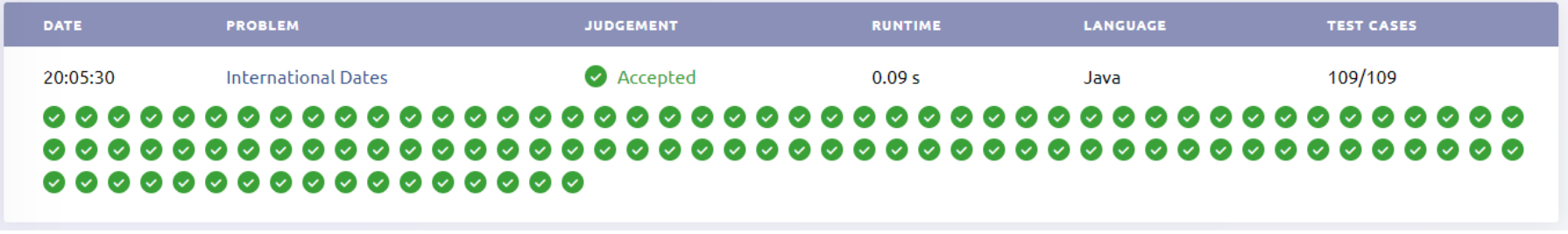
Exercise

- 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);

AARHUS UNIVERSITET

SideBar

- Open.kattis // Exercise **International Dates**



DATE	PROBLEM	JUDGEMENT	RUNTIME	LANGUAGE	TEST CASES
20:05:30	International Dates	Accepted	0.09 s	Java	109/109

- *109 test cases for five lines of code???*
- My analysis:
  - **3 test cases is enough!**

EC Covered	TestCase	Expected
[o1][f2][s1]	13/07/1963	EU
[o2][f1][s2]	08/13/2027	US
[o3][f1][s1]	12/12/1987	either

```
public static String classifyFormat(String dateString) {
  String parts[] = dateString.split( regex: "/");
  int p1 = Integer.parseInt(parts[0]);
  int p2 = Integer.parseInt(parts[1]);
  // US MM/DD - bail out if DD > 12
  if (p2 > 12) return "US";
  // EU DD/MM - bail out if DD > 12
  if (p1 > 12) return "EU";
  return "either";
}
```

Documenting ECs

- Math.abs is simple. Thus the ECs are simple.

- This is often **not** the case! Finding ECs requires deep thinking, analysis, and iteration.
- Document them!
- **Equivalence class table:**

Label the EC

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

AARHUS UNIVERSITET

Invalid/Valid ECs

- Some input values make an algorithm *give up*, *bail out, throw exception*, or *compute answer immediately:*
  - file.open("nonexisingfile");
  - game.usePower(FINDUS) (and findus mana =0)

- Input values that leads to *abnormal processing/bail out,* we classify as belonging to ***invalid ECs*.**

- Those input values that process normally/all the way, we say belong to ***valid ECs.***
  - Those cases where the 'method is run to conclusion'

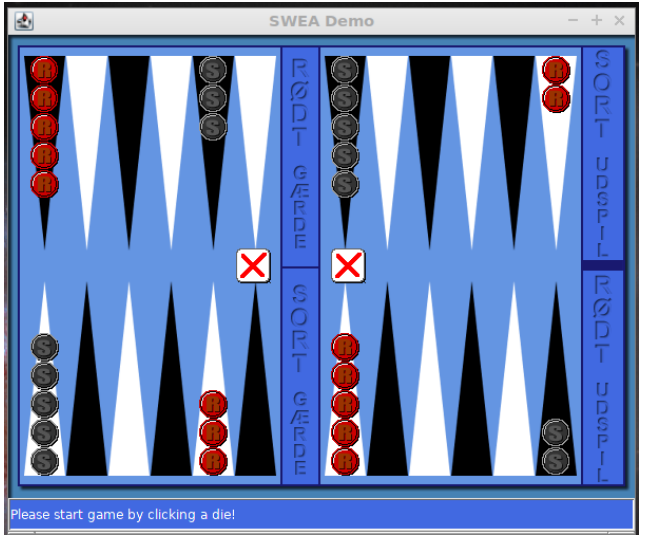
Finding the ECs

Often tricky…

Reality

Example: Backgammon move validation

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



AARHUS UNIVERSITET

A process

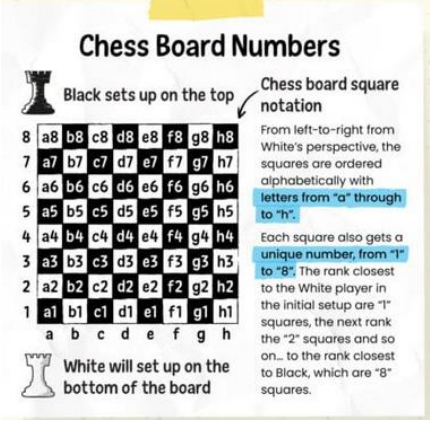
- To find the ECs you will need to look carefully at the specification and especially all the **conditions** associated.
  - Conditions express choices in our algorithms and therefore typically defines disjoint algorithm parts!
    - If (expr) {} else {}
  - And as the test cases should at least run 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

- 1) If you have a *range of values* specification
  - make **three** ECs:
    - [1] in range — valid
    - [2] above range — invalid
    - [3] below range — invalid

- Ex. Standard chess notation/ is “a8” or “x17” valid positions?
  - **Column range: a-h; Row range: 1-8**

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



AARHUS UNIVERSITET

Partitioning Heuristics

- 2) If you have a *set, S, of values* specification
  - make **|S|+1** ECs
    - [1] .. [|S|] one EC for each member in S valid
      - Each with only that particular member in the set
    - [|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	$\notin \{5, 10, 25\}[a1]$	$\{5\}[a2]; \{10\}[a3]; \{25\}[a4]$

  - Note: Not just *two sets!*

AARHUS UNIVERSITET

Partitioning Heuristics

- 3) If you have a *boolean condition* specification
  - make **two** ECs

- Ex. the first character of identifier 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?☺*

AARHUS UNIVERSITET

Partitioning Heuristics

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

AARHUS UNIVERSITET

From ECs to Test cases

AARHUS UNIVERSITET

Test case generation

- For disjoint ECs you simply pick an element from each EC to make a test case.
- Document in a **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	–	$x > 0[a1]$ $x \leq 0[a2]$

AARHUS UNIVERSITET

Usual case

- 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

AARHUS UNIVERSITET

Combinatorial Explosion

- Umph! Combinatorial explosion of test cases ☹.
- Ex.
  - **Three** independent input parameters 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?

AARHUS UNIVERSITET

Combinatorial Explosion

- 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]

AARHUS UNIVERSITET

Myers' Heuristics

- Often, it is better to generate test cases like this:

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]

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

Rule 2

Rule 1

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**

AARHUS UNIVERSITET

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*month+day

Jan = 1; Feb = 2

AARHUS UNIVERSITET

The Process

The Summary

Revisited…

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

Example

```
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;
}
```

Exercise

- 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	$< 1\ [m1]; > 12\ [m2]$	$1 - 12\ [m3]$

AARHUS UNIVERSITET

Damn – Leap years!

- Nice to be an astronomer ☺

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

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

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

AARHUS UNIVERSITET

Process step 3

- Coverage?

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

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

AARHUS UNIVERSITET

Process step 4

- Generate, using Myers rule 1 and rule 2

ECs covered	Test case	Expec
$[y3a], [m3a]$	$y = 2000; m = 2$	
$[y3b], [m3b]$	$y = 1900; m = 5$	
$[y3c], [m3b]$	$y = 2004; m = 10$	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]
$[m2], [y3c]$	$y = 2004; m = 13$	[exception]

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

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 $[m2]$	$1 - 12$ $[m3]$

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

Example 2

AARHUS UNIVERSITET

Formatting

```
/** 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!!!*

Exercise

- 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?

AARHUS UNIVERSITET

My analysis

Condition	Invalid ECs	Valid ECs
overflow / underflow	≥ 1000.0 $[a1]$; < 0.0 $[a2]$	0.0 <= x < 1000.0 [a3]
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 $[d3]$

range

boolean

set

set

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	≥ 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 $[d3]$

AARHUS UNIVERSITET

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 eleme lies in a single invalid ECs.

Condition	Invalid ECs	Valid
overflow / underflow	≥ 1000.0 $[a1]$, < 0.0 $[a2]$	
2 digit rounding		(,00x round
		(,00x round down) $[b2]$
prefix		no '0' prefix $[c1]$
		exact '0' prefix $[c2]$
		exact '00' prefix $[c3]$
		exact '0.00' 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'

AARHUS UNIVERSITET

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 eleme lies in a single invalid ECs.

Condition	Invalid ECs	Valid
overflow / underflow	≥ 1000.0 $[a1]$; < 0.0 $[a2]$	
2 digit rounding		(,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'

AARHUS UNIVERSITET

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 eleme lies in a single invalid ECs.

Condition	Invalid ECs	Valid ECs
overflow / underflow	≥ 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'

AARHUS UNIVERSITET

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 eleme lies in a single invalid ECs.

Condition	Invalid ECs	Valid ECs
overflow / underflow	≥ 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	'***.**'
[illegible]2]	-0.1	'—.–'
[illegible]1], [illegible]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'

AARHUS UNIVERSITET

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 eleme lies in a single invalid ECs.

Condition	Invalid ECs	Valid ECs
overflow / underflow	≥ 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'

AARHUS UNIVERSITET

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 eleme lies in a single invalid ECs.

Condition	Invalid ECs	Valid ECs
overflow / underflow	≥ 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'

AARHUS UNIVERSITET

Often, Alternative Analysis Possible

- I used ‘set’ heuristics to divided the ‘prefixed 0’ ECs
- A range heuristics could be applied just as well
  - Condition
    - Whole part of decimal number x, let call it w
  - Intervals
    - w in [100..999] [b1]
    - w in [10..99] [b2]
    - w in [1..9] [b3]
    - w is [0] [b4]
- (Note: all these are VALID ECs)
- Will lead to the exact same test cases…

AARHUS UNIVERSITET

Computations

"Code without if's"
… also requires some care

AARHUS UNIVERSITET

Computation Heuristics

- 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.

Example

- Specification:

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

- But: No conditions =>
  - Myers analysis will give only one test case =>
    - Could just be **a = x = b =0**
- **Thus will not detect that the function below is wrong!**

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

Example

- Specification:

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

- Our computation heuristics thus leads to:

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.

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!

- Testcase: a=7; x= 3; b=9 => output = 30.

AARHUS UNIVERSITET

Computation Heuristics

- Example/Hint – *SigmaStone attacks with field support*

> The resulting *attack strength* of a minion is the attack value of the minion itself, and to this value is added *field support*: +1 for each friendly fielded minion of the *same* class (and only them) up to a maximum of 4, and finally is added *boosts*

- +1 for every friendly minion of same class
  - Range [0..4] + >= 5
  - Implementation 1:
    - Value = attackStrength + sumOfFriendlyOfSameClass();
  - Implementation 2:
    - Value = attackStrength; // missed the summation here
- Picking 0 as representative element is problematic!

Boundary value analysis

AARHUS UNIVERSITET

Boundary value analysis

- Experience shows that test cases focusing on *boundary conditions* have high payoff.

- Some that spring into my mind are
  - *”off by one” errors in comparisons*
    - *if (x <= MAX_SIZE)* and not *if (x < MAX_SIZE)*
  - *null as value for a reference/pointer*

AARHUS UNIVERSITET

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

Other Reliability Techniques

Testing is not the only way…

AARHUS UNIVERSITET

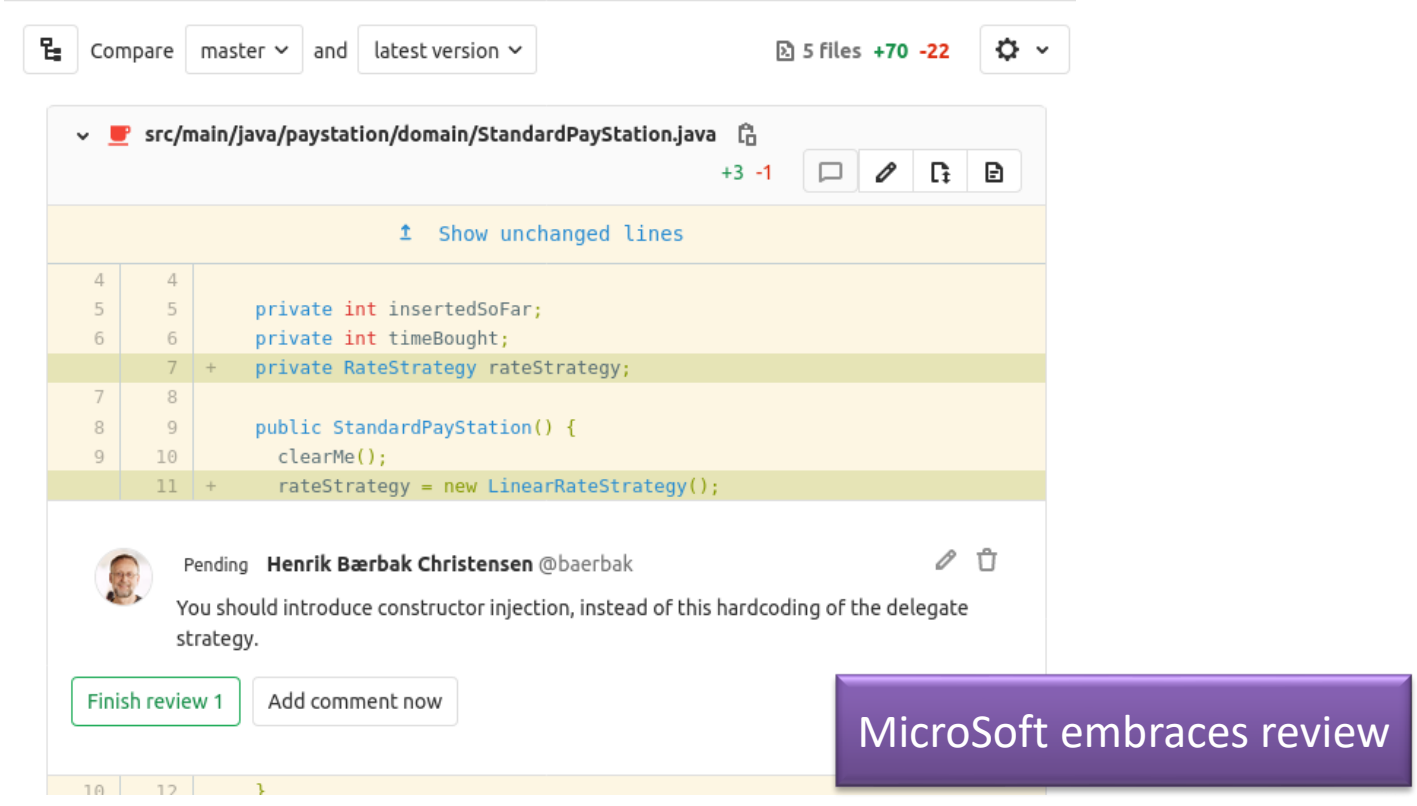
Systematic Review

- Systematic Review
  - A formalized, systematic, process of *people reading the code and identifying anomalies*
    - Pro:
      - Can find defects that no testing ever can
        - » Wrong comments
        - » Architectural flaws (‘strategy is used incorrectly here’)
      - Important learning
        - » Learn by reviewing the code of master coders
    - Con:
      - Manual/human/**slow/expensive**
      - Regression is too expensive
      - Bottleneck in releasing software

AARHUS UNIVERSITET

Systematic Review

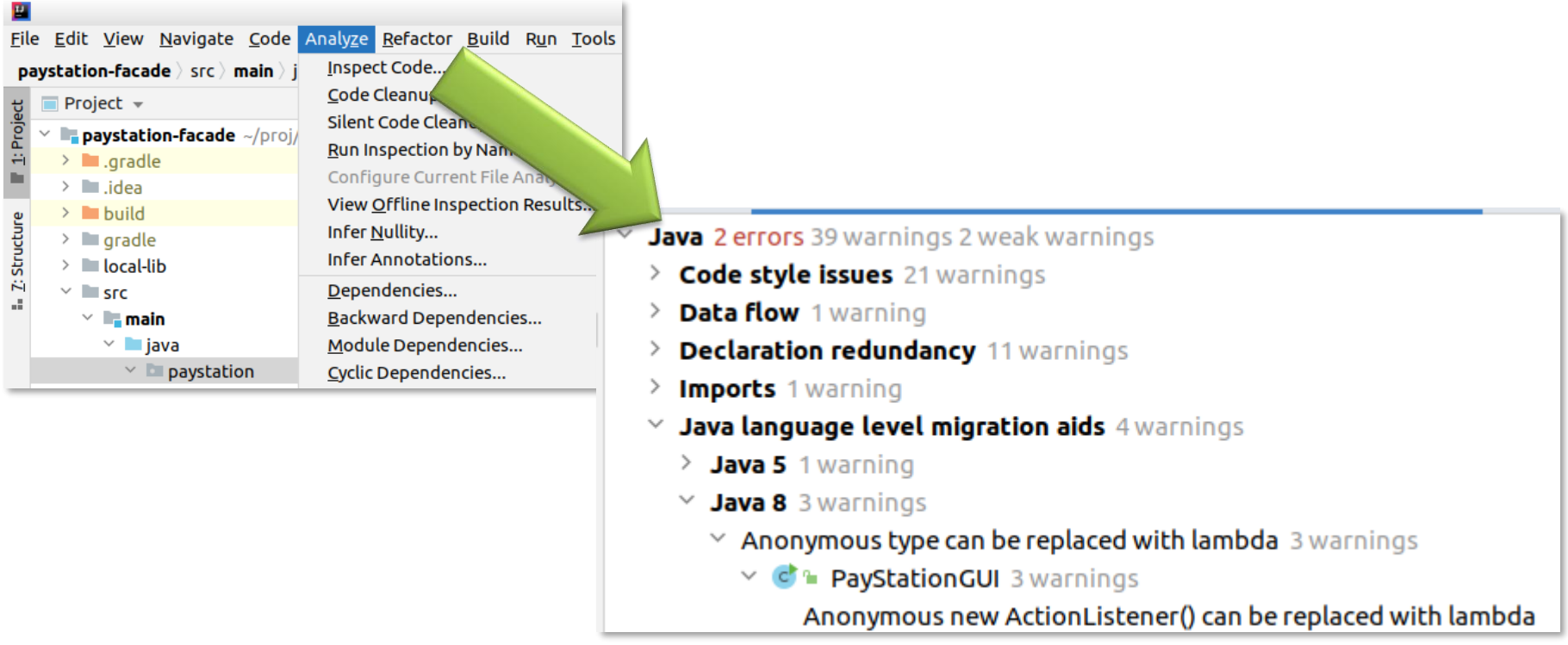
- Supported by GitHub, etc



AARHUS UNIVERSITET

Static Analysis

- Static Analysis
  - A review, but a **program** does it, using heuristics encoded



Static Analysis

- Static Analysis
  - Pro
    - Fast and automated
    - Especially suited to find *security flaws*
  - Con
    - Restricted in the types of failures it can find
      - Primarily language level errors, not on architectural level
    - Overload of errors
      - I review many and find they are not errors but ‘as I want it’
      - But they pop up every time I run the analysis => **error blindness**
        - » ***The one that I should have picked up drowns in all the information…***

AARHUS UNIVERSITET

Formal Methods

- The biggest gun in the cupboard
- Mathematical proof that our algorithm is correct
- Much research (also here at CS), but so far (I think)
  - Techniques do not scale well
  - Math may be even harder to read than code
    - That is, the error may be in the proof!
  - Math is a model, often ignoring physical properties
    - No, the Stack is **not *infinite!*** No, memory is **not infinite!** Etc…

AARHUS UNIVERSITET

Discussion

AARHUS UNIVERSITET

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.

Summary

- 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 skeptical about values at the boundaries
  - Especially on the boundary between valid and invalid ECs