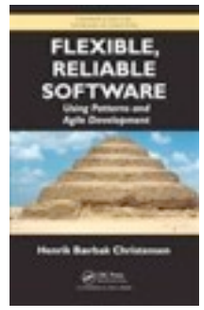
Theme 2
Program Design and Testing
Systematic Testing

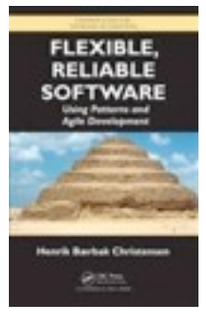
Learning Objectives

- to present a few systematic testing techniques that increase the chance of finding defects while keeping the number of test cases low.

Discussion

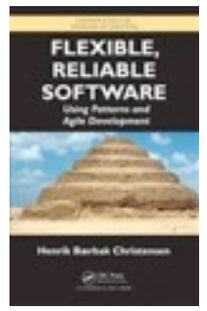


- Remember:
  If there is a defect, software may fail and is not reliable.
- Discussion
  - Find example of removing a defect does not increase the system's reliability at all
  - Find example of removing defect 1 increase reliability dramatically while removing defect 2 does not



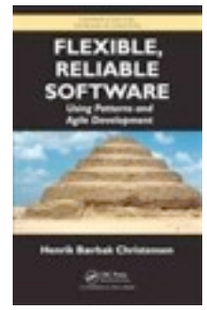
Question:
What kind of defects should you correct to get best return on investment?

Terminology



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

Testing Techniques



- **Black-box testing**
  The unit under test 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 programming, algorithms, and common mistakes made by programmers
- **White-box testing**
  The fill implementation of the unit under test is known, the actual code can be inspected in order to generate test cases.

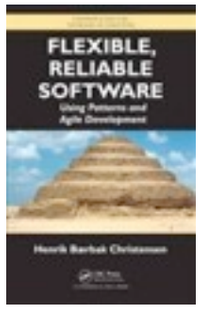
Equivalence Class Partitioning

- 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: Math.abs	
Input	Expected output
x = 37	37
x = 38	38
x = 39	39
x = 40	40
x = 41	41

Example



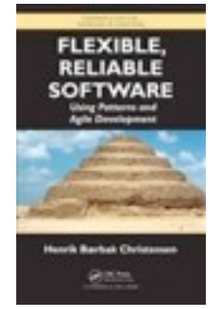
1. What is the probability that x=38 will find a defect that x=37 did not expose?

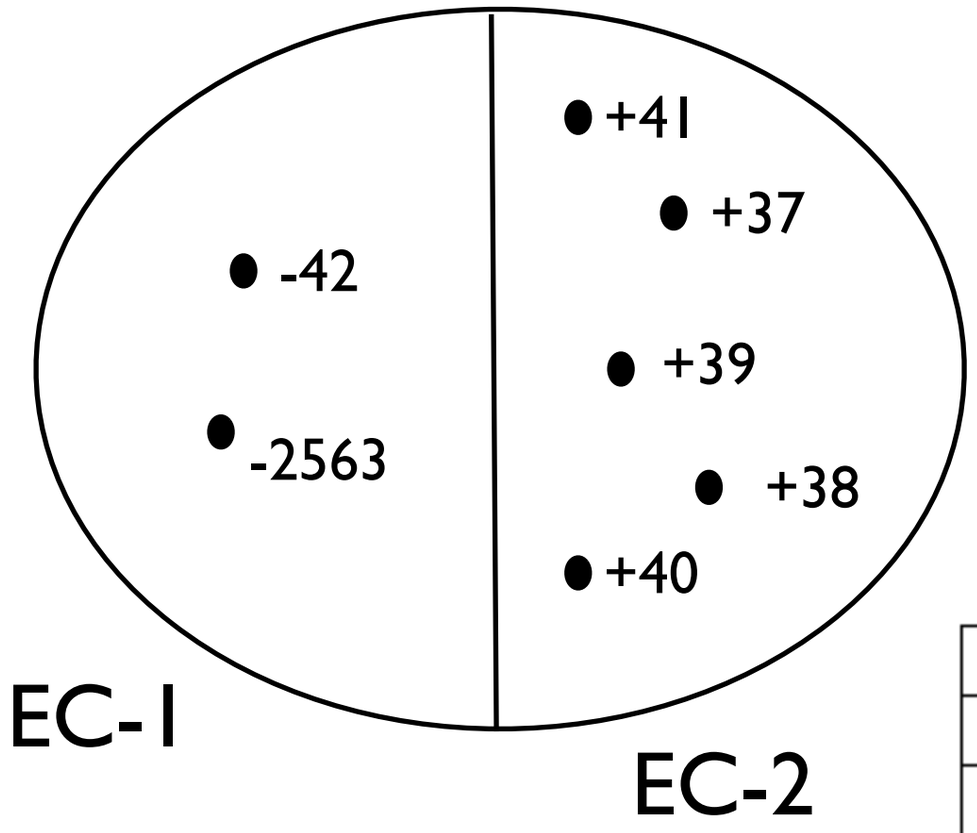
Unit under test: Math.abs	
Input	Expected output
x = 37	37
x = 38	38
x = 39	39
x = 40	40
x = 41	41

2. What is the probability that there will be a defect in handling negative x?

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

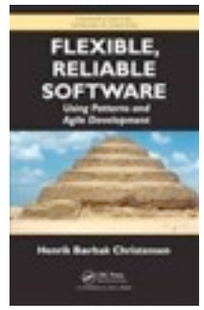
Example (continued)





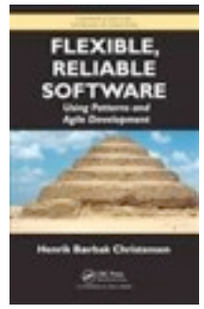
Unit under test: Math.abs	
Input	Expected output
x = 37	37
~~x = 38~~	~~38~~
~~x = 39~~	~~39~~
~~x = 40~~	~~40~~
~~x = 41~~	~~41~~
x = -42	42
~~x = -2563~~	~~2563~~

Conclusion



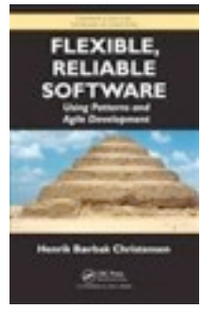
- The specification will force an implementation where (most likely!) all positive arguments are treated by one code fragment and all negative arguments by another.
  - Thus we only need two test cases:
    1. one that tests the positive argument handling code
    2. one that tests the negative argument 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.

Equivalence Class



- We can find a single input value that represents a large set of values!
- **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.

Remark



- Systematic testing ...

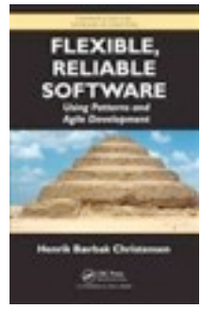
  does not mean

  - systematically find all defects and guaranty none are left!!!

  does mean

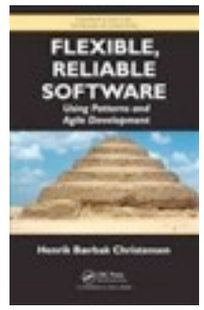
  - systematically derive a small test case with high probability of finding many defects!!!

Soundness



- Partitioning is sound if two properties are fulfilled:
  - **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.

Notation

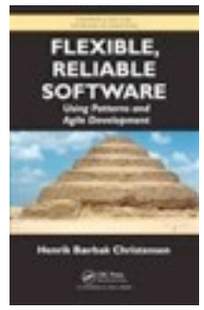


- Equivalence class table

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[1]$ $x \leq 0[2]$

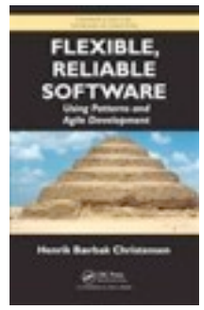
- Input values that leads to abnormal processing/giving up, we classify as belonging to invalid ECs.
- Those input values that process normally we say belong to valid ECs.

Process



- Finding a good set of ECs is often a difficult process requiring skills and experience!!
  - set of guidelines, heuristics
  - finding ECs is an iterative process!!!
    - take small steps!

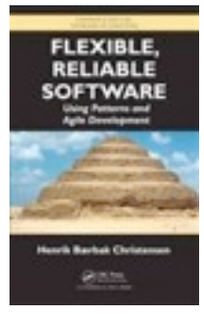
Reality Example



- Example: Backgammon move validation
  - validity = **v (move, player, board-state, die-value)**
    - is Red move (B1-B2) valid on this board given this die?

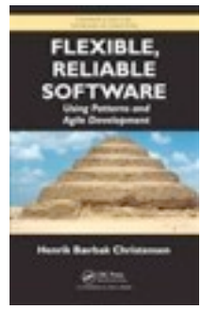
- Problem:
  - multiple parameters: *player, move, board, die*
  - complex parameters: *board state*
  - coupled parameters: *die couples to move!*

Guidelines



- Look for conditions in the specification of the UUT.
  - 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.
- Consider typical programming techniques
  - will a program contain if's and while's here?

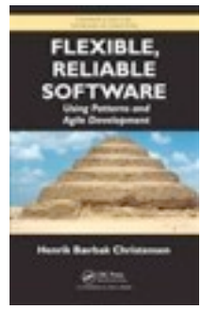
Heuristics: Range of Values



- If you have a *range of values* specification
  - make **3** ECs
    - [1] in range valid
    - [2] above range invalid
    - [3] below range invalid
  - Example: Standard chess notation
    Are "a8" or "x17" valid positions?

Condition	Invalid ECs	Valid ECs
Column	< 'a' [1]; > 'h' [2]	'a'–'h' [3]
Row	< 1 [4]; > 8 [5]	1–8 [6]

Heuristics: Set of Values

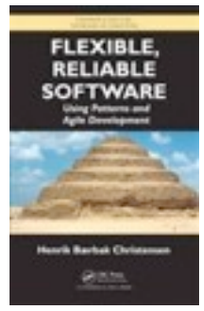


- If you have a *set*, S, *of values* specification:
  - make **|S|+1** ECs
    - [1]..[|S|] one for each member in S valid
    - [|S|+1] for a value outside S invalid

- Example: PayStation accepting coins
  - Set of {5, 10, 25} cents coins

Condition	Invalid ECs	Valid ECs
Allowed coins	$\notin \{5, 10, 25\}[1]$	$\{5\}[2]; \{10\}[3]; \{25\}[4]$

Heuristics: Boolean Condition

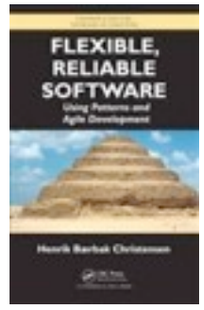


- If you have a *boolean condition* specification
  - make **2** ECs

- Example: the first character must be a letter

Condition	Invalid ECs	Valid ECs
Initial character of identifier	non-letter [1]	letter [2]

Test-case Generation



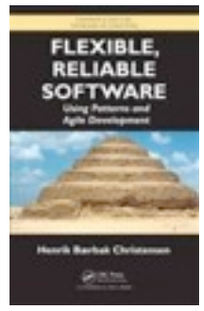
- Combination of conditions
  need to combine ECs to generate test cases
- Example: chess board validation (i.e., valid positions)

Condition	Invalid ECs	Valid ECs
Column	< 'a' [1]; > 'h' [2]	'a'–'h' [3]
Row	< 1 [4]; > 8 [5]	1–8 [6]

ECs covered	Test case	Expected output
[1], [4]	(' ',0)	illegal
[2], [4]	('i',-2)	illegal
[3], [4]	('e',0)	illegal
[1], [5]	(' ',9)	illegal
[2], [5]	('j',9)	illegal
[3], [5]	('f',12)	illegal
[1], [6]	(' ',4)	illegal
[2], [6]	('i',5)	illegal
[3], [6]	('b',6)	legal

Extended test case table

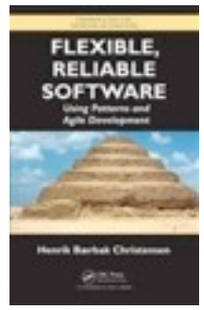
Myers Heuristic



- To limit the number of test cases, Myers suggests
  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 EC.

Condition	Invalid ECs	Valid ECs
Column	< 'a' [1]; > 'h' [2]	'a'–'h' [3]
Row	< 1 [4]; > 8 [5]	1–8 [6]

	ECs covered	Test case	Expected output
Rule 2	[1], [6]	(' ',5)	illegal
	[2], [6]	('j',3)	illegal
	[3], [4]	('b',0)	illegal
	[3], [5]	('c',9)	illegal
Rule 1	[3], [6]	('b',6)	legal

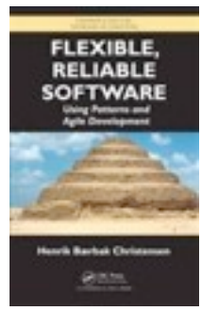


Why Rule 2?

- Letting only one condition be invalid at a time, avoid test cases that pass due to masking.
- Example

```
/** Demonstration of masking of defects.
*/
public class ChessBoard {
  public boolean valid(char column, int row) {
    if ( column < 'a' ) { return false; }
    if ( row < 0 ) { return false; }
    return true;
  }
}
```

Test case (‘ ‘,0) will pass which is expected
Code deemed correct, but this is a wrong conclusion!



The Process

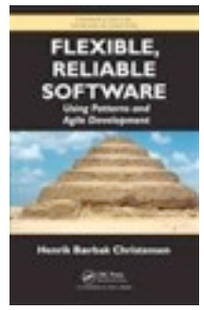
1. Review the requirements for the UUT and identify *conditions* and use the heuristics to find ECs for each condition. ECs are best written down in an *equivalence class table*.

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*.

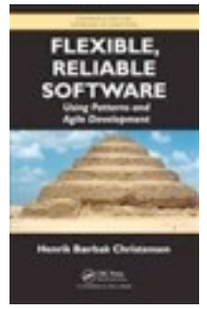
Example




```
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;
}
```

Example: Process Step 1



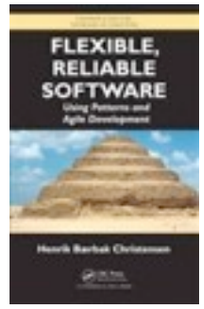
Which heuristic to use on the specification?
[Remember: range, set or boolean?]

Condition	Invalid ECs	Valid ECs
year	< 1900 [1]; > 3000 [2]	$1900 - 3000$ [3]
month	< 1 [4]; > 12 [5]	$1 - 12$ [6]

Representation property??

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.

Example: leap years!!



Condition	Invalid ECs	Valid ECs
year (y)		$\{y \mid y \in [1900; 3000] \wedge y\%400 = 0\}$ [3a] $\{y \mid y \in [1900; 3000] \wedge y\%100 = 0 \wedge y \notin [3a]\}$ [3b] $\{y \mid y \in [1900; 3000] \wedge y\%4 = 0 \wedge \notin [3a] \cup [3b]\}$ [3c] $\{y \mid y \in [1900; 3000] \wedge y\%4 \neq 0\}$ [3d]

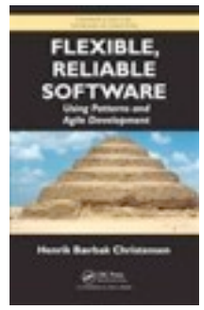
Weekday calculation is influenced by leap years w.r.t. months

Condition	Invalid ECs	Valid ECs
month		$1 - 2$ [6a]; $3 - 12$ [6b]

Coverage??

Every possible input element belongs to at least one of the equivalence classes.

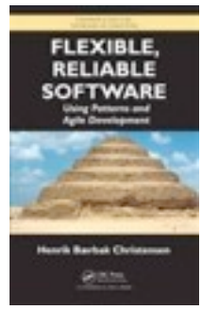
Example: Process step 4



Generate using Myers heuristics test cases

ECs covered	Test case	Expected output
$[3a], [6a]$	$y = 2000; m = 2$	-
$[3b], [6b]$	$y = 1900; m = 5$	-
$[3c], [6b]$	$y = 2004; m = 10$	5
$[3d], [6a]$	$y = 1985; m = 1$	-
$[1]$	$y = 1844; m = 4$	[exception]
$[2]$	$y = 4231; m = 8$	[exception]
$[4]$	$y = 2004; m = 0$	[exception]
$[5]$	$y = 2004; m = 13$	[exception]

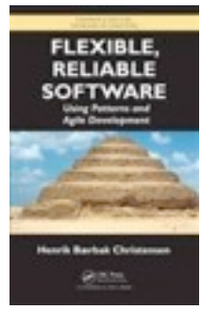
Boundary Analysis



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

- Some common examples:
  - iteration over an 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

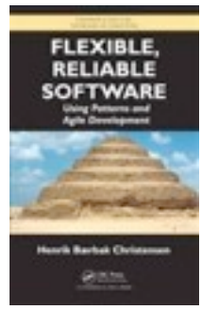
Boundary Value



- **Boundary Value**
  A boundary value is an element that lies right on or next to the edge of an equivalence class.
  - Example: chess board position has strong boundary, ‘a’, ‘h’, 1 and 8.

    It is thus very interesting to test row = 1, row = 8, column = ‘a’, column = ‘h’ as boundary.
  - complements EC analysis

Key Points



- **Observe unit preconditions**
  Do not generate ECs and test cases for conditions that a unit specifically cannot or should not handle.
- **Systematic testing assumes competent programmers**
  Equivalence partitioning and other testing techniques rely on honest and competent programmers that are using standard techniques
- **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.

Reading Material:

Pair-wise Testing

Ragnhild Van Der Straeten - ULB - Software Engineering and Project Management - 2012/2013