Førsteårsprojekt (F2010)

Claus Brabrand
[brabrand@itu.dk]
IT University of Copenhagen
Black-Box Testing

("FÅP": First-year Project Course, ITU, Denmark)

Claus Brabrand
[ brabrand@itu.dk ]
Outline

- Warm-up Exercise:
  - Introductory exercise to Black-Box Testing

- Black-Box Testing the Implementation:
  - Dynamic black-box testing

- "Equivalence Partitioning"
  - Equivalence Relations and Equivalence Partitioning

- Black-Box Testing the Specification:
  - Static black-box testing

- Static Analysis:
  - Undecidability, Approximation, and Static Analysis
White-box vs. Black-box Test

- **White-box Testing:**
  - (aka., “structural testing”)
  - (aka., “internal testing”)

- **Test focus:**
  - `source code`

```
int in();
int out();

int n = in();
if (odd(n))
  n = n/2;
else
  n = 3*n+1;
out(n);
```

- **Black-box Testing:**
  - (aka., “behavioral testing”)
  - (aka., “external testing”)
  - (aka., “input-output testing”)

- **Test focus:**
  - `specification` (or intention)

```
? ~ ~
program spec
```

Complementary Approaches!!!
Black-Box Testing

The goal of black-box testing is:

- Make sure *impl* solves problem it is supposed to:
  - i.e., *impl ~ spec* (relation between impl & spec)

Point of departure: *spec, not impl*!

- not a particular program which ”claims” to solve problem
  (but all reasonable possible implementations)
- testing w/o insights of code

**Static**
(test *spec*):

**Dynamic**
(test *impl*):

Claus Brabrand, ITU, Denmark

BLACK-BOX TESTING

[ 5 ]

Feb 09, 2010
Exercise: Triangle Test Test

- **Equilateral triangle (T-3):**
  - All three sides have equal length

- **Isosceles triangle (T-2):**
  - Two sides have equal length

- **Scalene triangle (T-1):**
  - All sides have different length

```c
enum Triangle { EQUILATERAL, ISOSCELES, SCALENE }
Triangle isTriangle(int a, int b, int c);
```

The program reads **three integer values** from an input dialog. (The three values represent the lengths of the sides of a triangle.) The program displays a message that states whether the triangle is:
- equilateral, isosceles, or scalene.

**Q:** Which **test cases** should we use?

- E.g., (1,1,1), (2,2,2), (3,3,3), (4,4,4), (5,5,5), ...?
Appropriate Test Cases?

- **Representativeness?**

  **SYSTEMATIC TESTING!**

  **Advice:** Avoid making lots of ad-hoc tests "just to be on the safe side".

  **Advice:** Cases relative to how we *might* conceive solving the problem & how it *might* be wrong. (This involves "guessing" how implementations might work)

  **Advice:** Carefully chose cases such that:
  - same case => same error; *and*
  - same error => same case.

  **Advice:** Test *typical* and *extreme* cases.
A Triangle Test (cont’d)

<table>
<thead>
<tr>
<th>#</th>
<th>Property</th>
<th>Input</th>
<th>Example</th>
<th>Expected Output</th>
<th>Actual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/</td>
<td>Valid T-1</td>
<td>(2,3,4)</td>
<td>Scalene</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>2/</td>
<td>Valid T-3</td>
<td>(5,5,5)</td>
<td>Equilateral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3/</td>
<td>Valid T-2</td>
<td>(6,6,7)</td>
<td>Isosceles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4/</td>
<td>Variants of 3/</td>
<td>(6,7,6)</td>
<td>Isosceles</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(7,6,6)</td>
<td>Isosceles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5/</td>
<td>Zero value</td>
<td>(8,0,9)</td>
<td>Invalid input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6/</td>
<td>Negative value</td>
<td>(2,3,-4)</td>
<td>Invalid input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7/</td>
<td>Line (not triangle)</td>
<td>(1,2,3)</td>
<td>Not triangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8/</td>
<td>Variants of 7/</td>
<td>(1,3,2)</td>
<td>Not triangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3,1,2)</td>
<td>Not triangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9/</td>
<td>One line too long</td>
<td>(1,2,8)</td>
<td>Not triangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10/</td>
<td>Variants of 9/</td>
<td>(1,8,2)</td>
<td>Not triangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(8,1,2)</td>
<td>Not triangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11/</td>
<td>Dot (all zero)</td>
<td>(0,0,0)</td>
<td>Not triangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12/</td>
<td>Non-integers</td>
<td>(1,\frac{1}{2},5)</td>
<td>Invalid input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13/</td>
<td>Wrong # arguments</td>
<td>(7,8)</td>
<td>Invalid input</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1,2,3,4)</td>
<td>Invalid input</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NB: Did you remember to also specify the EXPECTED OUTPUT?,
Outline

- **Warm-up Exercise:**
  - Introductory exercise to Black-Box Testing

- **Black-Box Testing the Implementation:**
  - Dynamic black-box testing

- **"Equivalence Partitioning"**
  - Equivalence Relations and Equivalence Partitioning

- **Black-Box Testing the Specification:**
  - Static black-box testing

- **Static Analysis:**
  - Undecidability, Approximation, and Static Analysis
Black-Box Testing

...of the "min-max program"

| Specification: "Min-Max"
| The program receives some non-negative numbers as arguments; finds the minimum and maximum among those, and prints the results |

Advice: Avoid making lots of ad-hoc tests "just to be on the safe side".

Advice: Test typical and extreme cases.

Advice: Cases relative to how we might conceive solving the problem & how it might be wrong. (This involves "guessing").

<table>
<thead>
<tr>
<th>Input Property</th>
<th>Example Input</th>
<th>Expected Output</th>
<th>Actual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>No numbers</td>
<td>[]</td>
<td>Error</td>
<td>...</td>
</tr>
<tr>
<td>One number</td>
<td>[2]</td>
<td>(2,2)</td>
<td></td>
</tr>
<tr>
<td>Two nums, increasing</td>
<td>[3,4]</td>
<td>(3,4)</td>
<td></td>
</tr>
<tr>
<td>Two nums, decreasing</td>
<td>[6,5]</td>
<td>(5,6)</td>
<td></td>
</tr>
<tr>
<td>Two nums, equal</td>
<td>[7,7]</td>
<td>(7,7)</td>
<td></td>
</tr>
<tr>
<td>Many nums, increasing</td>
<td>[11,12,13]</td>
<td>(11,13)</td>
<td></td>
</tr>
<tr>
<td>Many nums, decreasing</td>
<td>[26,25,24]</td>
<td>(24,26)</td>
<td></td>
</tr>
<tr>
<td>Many nums, mid great</td>
<td>[32,39,35]</td>
<td>(32,39)</td>
<td></td>
</tr>
<tr>
<td>Many nums, mid smallest</td>
<td>[45,41,48]</td>
<td>(41,48)</td>
<td></td>
</tr>
<tr>
<td>Negative numbers</td>
<td>[-1,-2]</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>Fractionals</td>
<td>[(\frac{1}{2}, \frac{3}{4})]</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>Not numbers</td>
<td>[foo,#$!]</td>
<td>Error</td>
<td></td>
</tr>
</tbody>
</table>
## Test Exercise

- **Black-Box Test the following program:**

  The program takes a list of pos nums, sorts them, and prints the result.

<table>
<thead>
<tr>
<th>Specification: ”Sorting”</th>
</tr>
</thead>
</table>

*The program takes a list of pos nums, sorts them, and prints the result.*

<table>
<thead>
<tr>
<th>Input Property</th>
<th>Example Input</th>
<th>Expected Output</th>
<th>Actual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>No numbers</td>
<td>[]</td>
<td>[]</td>
<td>...</td>
</tr>
<tr>
<td>One number</td>
<td>[2]</td>
<td>[2]</td>
<td></td>
</tr>
<tr>
<td>Two num, increasing</td>
<td>[3,4]</td>
<td>[3,4]</td>
<td></td>
</tr>
<tr>
<td>Two num, decreasing</td>
<td>[6,5]</td>
<td>[5,6]</td>
<td></td>
</tr>
<tr>
<td>Two num, equal</td>
<td>[7,7]</td>
<td>[7,7]</td>
<td></td>
</tr>
<tr>
<td>Many num, increasing</td>
<td>[11,12,13]</td>
<td>[11,12,13]</td>
<td></td>
</tr>
<tr>
<td>Many num, decreasing</td>
<td>[26,25,24]</td>
<td>[24,25,26]</td>
<td></td>
</tr>
<tr>
<td>Many num, mid great</td>
<td>[32,39,35]</td>
<td>[32,35,39]</td>
<td></td>
</tr>
<tr>
<td>Many num, mid smallest</td>
<td>[45,41,48]</td>
<td>[41,45,48]</td>
<td></td>
</tr>
<tr>
<td>Non-positive numbers</td>
<td>[-1,-2]</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>Fractionals</td>
<td>[(\frac{1}{2}, \frac{3}{4})]</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>Not numbers</td>
<td>[foo,#$!]</td>
<td>Error</td>
<td></td>
</tr>
</tbody>
</table>
Test Exercise

Black-Box Test the following program:

<table>
<thead>
<tr>
<th>Specification: ”Insert-into-sorted-list”</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>The method takes a positive integer ’x’ and a list of sorted positive ints ’L’ and inserts the ’x’ into ’L’, yielding another sorted list.</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input Property</th>
<th>Example Input</th>
<th>Expected Output</th>
<th>Actual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>(7, [])</em></td>
<td><em>7</em></td>
<td><em>…</em></td>
</tr>
<tr>
<td></td>
<td><em>(8, [2])</em></td>
<td><em>[2,8]</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>(9, [3, 4])</em></td>
<td><em>[3,4,9]</em></td>
<td></td>
</tr>
<tr>
<td>’x’ least</td>
<td><em>(1, [5, 6])</em></td>
<td><em>[1,5,6]</em></td>
<td></td>
</tr>
<tr>
<td>’x’ greatest</td>
<td><em>(9, [7, 8])</em></td>
<td><em>[7,8,9]</em></td>
<td></td>
</tr>
<tr>
<td>’x’ middle</td>
<td><em>(4, [2, 5])</em></td>
<td><em>[2,4,5]</em></td>
<td></td>
</tr>
<tr>
<td>’x’ zero</td>
<td><em>(0, [5, 9])</em></td>
<td><em>Error</em></td>
<td></td>
</tr>
<tr>
<td>’x’ negative</td>
<td><em>(-2, [3, 7])</em></td>
<td><em>Error</em></td>
<td></td>
</tr>
<tr>
<td>L unsorted</td>
<td><em>(5, [8, 4])</em></td>
<td><em>Error</em></td>
<td></td>
</tr>
<tr>
<td>L contains zero</td>
<td><em>(3, [0, 4])</em></td>
<td><em>Error</em></td>
<td></td>
</tr>
<tr>
<td>L contains negative</td>
<td><em>(4, [-5, 6])</em></td>
<td><em>Error</em></td>
<td></td>
</tr>
</tbody>
</table>
### Specification: "fromRoman"

Consider a method `int fromRoman(String r)` that is supposed to **convert** a **Roman Numeral** to the corresponding integer, using the symbols:


*The following rules determine the Roman Numeral corresponding to a positive number:*

- In general, the symbols of a Roman Numeral are added together from left to right (e.g., ”II” = 2, ”XX” = 20, ”XXXI” = 31, and ”MMVIII” = 2008);
- The symbols ‘I’, ‘X’, and ‘C’ may appear up to 3 times in a row; the symbol ‘M’ may appear any number of times; and the symbols ‘V’, ‘L’, and ‘D’ cannot be repeated;
- When a lesser symbol appears before a greater one, the lesser symbol is subtracted, not added (e.g., ”IV” = 4, ”IX” = 9, ”XL” = 40, and ”CM” = 900).
- The symbol ‘I’ may appear once before ‘V’ and ‘X’; the symbol ‘X’ may appear once before ‘L’ and ‘C’; the symbol ‘C’ may appear once before ‘D’ and ‘M’; and the symbols ‘V’, ‘L’, and ‘D’ cannot appear before a greater symbol (e.g., 45 is written ”XLV”, not ”VL”; and 49 is written ”XLIX”, not ”IL”; and 1998 is written ”MCMXCVIII”, not ”MIM”).*
Programs are vulnerable "around the edges":

- e.g. testing legal inputs (time, in hours):

<table>
<thead>
<tr>
<th>Property</th>
<th>Input</th>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-1</td>
<td>-1</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>0</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Typical</td>
<td>15 (e.g.)</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>23</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Maximum+1</td>
<td>24</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>

- e.g. testing legal inputs (dates, in April):

<table>
<thead>
<tr>
<th>Property</th>
<th>Input</th>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-1</td>
<td>00/4</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>01/4</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Typical</td>
<td>17/4 (e.g.)</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>30/4</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Maximum+1</td>
<td>31/4</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>

BVA = "Boundary-Value Analysis"
Test "Powers-of-Two"

- Programs vulnerable "around powers-of-two":
  - e.g. years of age (assume held in a byte):

<table>
<thead>
<tr>
<th>Property</th>
<th>Input</th>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-1</td>
<td>-1</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>0</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Typical</td>
<td>27 (e.g.)</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>255</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Maximum+1</td>
<td>256</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>

- e.g. #game-spectators (assume held in a 16-bit word):

<table>
<thead>
<tr>
<th>Property</th>
<th>Input</th>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-1</td>
<td>-1</td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>0</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Typical</td>
<td>12345 (e.g.)</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>65535</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>Maximum+1</td>
<td>65536</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>
Test "Empty Input"

- Default / empty / blank / null / zero / none / $\varepsilon$:
  - e.g., 'any program':

<table>
<thead>
<tr>
<th>Property</th>
<th>Input</th>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>No input</td>
<td>$\varepsilon$</td>
<td>Error message</td>
<td></td>
</tr>
</tbody>
</table>
Test "Invalid Input"

- Invalid / illegal / wrong / garbage / bogus data:
  - e.g., calculator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Input</th>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid input</td>
<td>+*31</td>
<td>Error message</td>
<td></td>
</tr>
<tr>
<td>Bogus data!!!</td>
<td>#$+~´?!=</td>
<td>Error message</td>
<td></td>
</tr>
</tbody>
</table>

Claus Brabrand, ITU, Denmark

BLACK-BOX TESTING

Feb 09, 2010
Outline

- **Warm-up Exercise:**
  - Introductory exercise to Black-Box Testing

- **Black-Box Testing the Implementation:**
  - Dynamic black-box testing

- **"Equivalence Partitioning"**
  - Equivalence Relations and Equivalence Partitioning

- **Black-Box Testing the Specification:**
  - Static black-box testing

- **Static Analysis:**
  - Undecidability, Approximation, and Static Analysis
Testing: Infinite process

- Recall: “testing is an incomplete process”
  - (i.e., “testing can’t prove absence of bugs”)

- There are *infinitely* many possible inputs:
  - (hence, testing will take an *infinite* amount of time)

```
(0,0)  (1,0)  (0,1)  (-1,0)  (0,-1)  (1,1)  (-1,-1)
  ↓    ↓    ↓    ↓    ↓    ↓    ↓
sum(x,y)  sum(x,y)  sum(x,y)  sum(x,y)  sum(x,y)  sum(x,y)  sum(x,y)
  ↓    ↓    ↓    ↓    ↓    ↓    ↓
  0    1    1   -1    -1    2   -2
```

Crash course on **Relations**

**Relations**

**Equivalence Relations**
**Crossproduct:** '×'

- **Crossproduct** (binary operator on sets):
  - Given sets:
    - \( A = \{ 0, 1 \} \)
    - \( B = \{ \text{true, false} \} \)
  - \( A \times B = \{ (0, \text{true}), (0, \text{false}), (1, \text{true}), (1, \text{false}) \} \)
  - i.e., *creates sets of pairs*

**Exercise:**

- \( A \times A = \{ (0,0), (0,1), (1,0), (1,1) \} \)
- \( Z \times Z = \{ (0,0), (0,1), (0,1), \ldots, (1,0), (1,1), \ldots, (42,87), \ldots \} \)
- \( (A \times A) \times B = \{ ((0,0), \text{true}), ((0,1), \text{true}), \ldots, ((1,1), \text{false}) \} \)
Relations

- **Example 1**: “even” relation: \([\text{even} \subseteq \mathbb{Z}]\)
  - Written as: \([\text{even} \ 4] \) as a short-hand for: \([\text{even} \ 4] \)
  - … and as: \([\nexists \text{even} \ 5] \) as a short-hand for: \([\nexists \text{even} \ 5] \)

- **Example 2**: “equals” relation: \(\{=\} \subseteq \mathbb{Z} \times \mathbb{Z}\)
  - Written as: \([2 = 2] \) as a short-hand for: \([2 = 2] \)
  - … and as: \([2 \neq 3] \) as a short-hand for: \([2 \neq 3] \)

- **Example 3**: “dist-btwn” relation: \(\{\rightarrow\} \subseteq \text{CITY} \times \mathbb{Z} \times \text{CITY}\)
  - Written as: \((\text{Aarhus}, 310, \text{Copenhagen}) \)
  - as short-hand for: \((\text{Aarhus}, 310, \text{Copenhagen}) \)
Equivalence Relation

Let ‘∼’ be a binary relation over set A:

- ‘∼’ ⊆ A × A
- (x, y) ∈ ‘∼’ abbreviated as: x ∼ y // "x and y are related"

~ is an equivalence relation iff:

- Reflexive:
  - ∀x ∈ A: x ∼ x
- Symmetric:
  - ∀x, y ∈ A: x ∼ y ⇔ y ∼ x
- Transitive:
  - ∀x, y, z ∈ A: x ∼ y ∧ y ∼ z ⇒ x ∼ z
Exercise: Eq. Rel.

Which relations are *equivalence relations*: …and which are not (and why not)?:

- **a)** The "*less-than-or-equal-to*" relation: '≤'
  ```plaintext
  \{ (n,m) \mid n,m \in \mathbb{Z}, \ n \leq m \}
  ```

- **b)** The "*almost-total-relation-on-integers*", (relating all numbers except 42, but relating 42 with 42):
  ```plaintext
  \{ (n,m) \mid n,m \in (\mathbb{Z}\setminus\{42\}) \} \cup \{ (42,42) \}
  ```

- **c)** The "*is-congruent-modulo-three*" relation:
  ```plaintext
  \{ (n,m) \mid n,m \in \mathbb{Z}, \ (n \mod 3) = (m \mod 3) \}
  ```

- **d)** The "*have-talked-together*" relation:
  ```plaintext
  \{ (p,q) \mid n,m \in PEOPLE, \ p \text{ and } q \text{ have talked together} \}
  ```

- **e)** The "*is-in-the-same-group-as*" relation:
  ```plaintext
  \{ (s,t) \mid s,t \in ITU-BACH-09, \ s \text{ and } t \text{ are in same KF04 group} \}
  ```
Eq. Rel. $\iff$ Partition

- **Equivalence relation ’$\sim$’**
  
  \[
  \{ (A,A), (B,B), (A,B), (B,A), \\
  (P,P), (X,X), (Y,Y), (Z,Z), \\
  (X,Y), (Y,X), (X,Z), (Z,X), \\
  (Y,Z), (Z,Y) \}
  \]

- **E.g.:**
  
  - $A \sim B$, $P \sim P$, $X \sim X$, $X \sim Z$
  - $A \not\sim P$, $B \not\sim Y$, $P \not\sim Z$

- **Partition ’$\sim$’:**

  - *Partition Image* (not shown)

  - *Canonical representatives*:
    
    \[
    [A] = [B] = \{ A, B \} \\
    [P] = \{ P \} \\
    [X] = [Y] = [Z] = \{ X, Y, Z \}
    \]

  *Capture the same information; i.e., notion of "equivalence"*
Testing: Infinite process

- Recall: "testing is an incomplete process"
  - (i.e., "testing can’t prove absence of bugs")

- There are *infinitely* many possible inputs:
  - (hence, testing will take an *infinite* amount of time)

\[
\begin{array}{cccccccc}
(0,0) & (1,0) & (0,1) & (-1,0) & (0,-1) & (1,1) & (-1,-1) \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\text{sum}(x,y) & \text{sum}(x,y) & \text{sum}(x,y) & \text{sum}(x,y) & \text{sum}(x,y) & \text{sum}(x,y) & \text{sum}(x,y) \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
0 & 1 & 1 & -1 & -1 & 2 & -2 \\
\end{array}
\]
Equivalence Partitioning

- Partition input:

  ![Diagram showing partition input with categories neg, zero, pos]

- Finitary partition:
  - If finite # categories (aka. "equivalence classes")
    - Here 3x: \{ "zero", "pos", "neg" \}

- We can now test all equivalence classes
  - Using representative elements from each category
Test Sum (cont’d)

- We can now test all equivalence classes
  - Using representative input from each category

Sum (testing all equivalence classes):

<table>
<thead>
<tr>
<th>Property</th>
<th>Input</th>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos, Pos</td>
<td>(1,2)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Neg, Pos</td>
<td>(-3,4)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Zero, Pos</td>
<td>(0,5)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Pos, Neg</td>
<td>(6,-7)</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>Neg, Neg</td>
<td>(-8,-9)</td>
<td>-17</td>
<td></td>
</tr>
<tr>
<td>Zero, Neg</td>
<td>(0,-10)</td>
<td>-10</td>
<td></td>
</tr>
<tr>
<td>Pos, Zero</td>
<td>(11,0)</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Neg, Zero</td>
<td>(-12,0)</td>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>Zero, Zero</td>
<td>(0,0)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Frequent Partitions for Testing

- **Numbers:**
  - Positive, Negative, Zero
  - Zero, One, (Two), Many (aka. ”Greenlandic Numbers”)

- **Lists:**
  - Length-0, Length-1, Length-2, Length-3+
  - Ascending-elements, Descending-elements, UnSorted

---

**Advice:**
Consider how problem *might* be solved
Partition into *qualitatively different* categories such that:
- ”same case ⇒ same error”; and
- ”same error ⇒ same case”.
Test Sequencing

Question: which is "best" (…and why)?

A) white-box testing ; black-box testing
   (i.e., white-box testing first)

...or...

B) black-box testing ; white-box testing
   (i.e., black-box testing first)

Answer: (usually) 'B'

Settle overall problems first: impl + spec
Before zooming in on details of the impl
Outline

■ Warm-up Exercise:
  ■ Introductory exercise to Black-Box Testing

■ Black-Box Testing the Implementation:
  ■ Dynamic black-box testing

■ "Equivalence Partitioning"
  ■ Equivalence Relations and Equivalence Partitioning

■ Black-Box Testing the Specification:
  ■ Static black-box testing

■ Static Analysis:
  ■ Undecidability, Approximation, and Static Analysis
Causes of Bugs

- **NB:** The number one "cause" is: *spec*!

Often due to:
- *complexity*
- *tight schedule*
- *under-specification*
- *under-documentation*
Black-Box Testing

- The goal of black-box testing is:
  - Make sure $\text{impl}$ solves problem it is supposed to:
    - i.e., $\text{impl} \sim \text{spec}$

- Point of departure:
  - $\text{spec}$, not $\text{impl}$
    - not a particular program which "claims" to solve problem
    - testing w/o insights of code

**Static** (test $\text{spec}$):

**Dynamic** (test $\text{impl}$):
## Spec ”Warning Words” (I/III)

### Unconditionals (always):
- ’Always’, ’for every’, ’for each’, ’for all’, …
- Try to violate (i.e., find exemptions from rule)!

### Unconditionals (never):
- ’None’, ’never’, ’not under any circumstances’, …
- Try to violate (i.e., find exemptions from rule)!

### Unsubstantiated claims (trivially):
- Check assumptions (that nothing’s swept under the rug)!

[cf. ”Software Testing”, R.Patton, p.61]
Spec "Warning Words" (II/III)

- Unspecified conditionals:
  - 'Some(-times)', 'often', 'usually', 'ordinarilly', 'mostly', ...
  - Unclear spec (under which circumstances)?

- Continuations:
  - 'Etcetera', 'and so forth', 'and so on', ...
  - Check that spec is comprehensively unambiguous?

- Examples:
  - 'E.g.', 'for example', 'such as', ...
  - Is example representative (what about other examples)?
Spec “Warning Words” (III/III)

- **Positive adjectives:**
  - ’*Good’*, ’*fast’*, ’*efficient’*, ’*small’*, ’*reliable’*, ’*stable’*, …
  - Subjective (needs objectification if to be used for testing)!

- **Alegedly completed:**
  - ’*Handled’*, ’*processed’*, ’*taken care of’*, ’*eliminated’*, …
  - Is something hidden?

- **Incompleted:**
  - ’*Skipped’*, ’*unnecessary’*, ’*superfluous’*, ’*rejected’*, …
  - Is something forgotten?
Finally, watch out for:

- "If … Then" (with missing "Else"):
  - Check what happens in the "Else-case"

- IF …
- THEN …
- ELSE … ?!
Outline

- Warm-up Exercise:
  -Introductory exercise to Black-Box Testing

- Black-Box Testing the Implementation:
  -Dynamic black-box testing

- ”Equivalence Partitioning”
  -Equivalence Relations and Equivalence Partitioning

- Black-Box Testing the Specification:
  -Static black-box testing

- Static Analysis:
  -Undecidability, Approximation, and Static Analysis
Rice’s Theorem (1953)

“Any interesting problem about the runtime behavior program* is undecidable”

-- Rice’s Theorem [paraphrased] (1953)

Examples:

- does program ’P’ always halt?
- is the value of integer variable ’x’ always positive?
- does variable ’x’ always have the same value?
- which variables can pointer ’p’ point to?
- does expression ’E’ always evaluate to true?
- what are the possible outputs of program ’P’?
- …
Safe Approximations

- Most interesting properties are **undecidable**:
  - e.g., "does program 'P' always halt"?
    - we can never "decide" this line automatically (in all cases)

- Same goes for 'errors':
  - e.g., "does 'P' ever divide by zero"?
Soundness & Completeness

- Compilers use **safe approximations**
  - computed via ”static analyses” such that:

<table>
<thead>
<tr>
<th><strong>Soundness:</strong></th>
<th><strong>Completeness:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="ok-error" alt="Sound analysis" /></td>
<td><img src="ok-error" alt="Complete analysis" /></td>
</tr>
<tr>
<td>Analysis reports no errors</td>
<td>Analysis reports an error</td>
</tr>
<tr>
<td>⇒ Really are no errors</td>
<td>⇒ Really is an error</td>
</tr>
</tbody>
</table>

- ![Okay!](ok-error) | ![Dunno?](ok-error) | ![Dunno?](ok-error) | ![Error!](ok-error)
Undecidability (self-referentiality)

Consider "The Book-of-all-Books":

- This book contains the titles of all books that do not have a self-reference (i.e. don't contain their title inside)
- Finitely many books; i.e.:
  - We can sit down & figure out whether to include or not...
- Q: What about "The Book-of-all-Books";
  - Should it be included or not?

"Self-referential paradox" (many guises):

- e.g. "This sentence is false"
Termination Undecidable!

- **Assume** termination is *decidable* (in Java);
- i.e. ∃ some program, halts: PROG → BOOL

```
bool halts(Program p) { ... }
```

```
-- P0.java --

bool halts(Program p) { ... }
Program p0 = read_program("P0.java");
if (halts(p0)) loop();
else halt();
```

- **Q:** Does $P_0$ loop or terminate...? :)  

- **Hence:** "*Termination is undecidable*
- ...for Java, C, C++, Pascal, $\lambda$-Calculus, Haskell, ...
Example: Type Checking

Will this program have type error (when run)?

```java
void f() {
    var b;
    if (<EXP>) {  
        b = 42;
    } else {  
        b = true;
    } /* some code */
    if (b) ...;
}
```

Type error $\iff$ `<EXP>` evaluates to `true`

**Undecidable (in all cases)**

i.e., what `<EXP>` evaluates to (when run)
Example: Type Checking (cont’d)

- Hence, languages use static requirements:

```c
void f() {
    bool b;  // instead of "var b;"
    /* some code */
    if (<EXP>) {
        b = 42;
    } else {
        b = true;
    }
    /* some more code */
}
```

- All variables must be declared
- And have constant type (throughout the program)
SYSTEMATIC TESTING !
Afl.Opg.: Roman Numerals

Afl.Opg. (to ”kasv”, deadline: Feb 16, 2010):

**Specification: ”fromRoman”**

Consider a method “`int fromRoman(String rn)`” that is supposed to **convert** a **Roman Numeral** to the corresponding integer, using the symbols (with values):

*The following rules determine the Roman Numeral corresponding to a positive number:*
- In general, the symbols of a Roman Numeral are added together from left to right (e.g., ”II” = 2, ”XX” = 20, ”XXXI” = 31, and ”MMVIII” = 2008);
- The symbols ‘I’, ‘X’, and ‘C’ may appear up to 3 times in a row; the symbol ’M’ may appear any number of times; and the symbols ’V’, ’L’, and ’D’ cannot be repeated;
- When a lesser symbol appears before a greater one, the lesser symbol is subtracted, not added (e.g., ”IV” = 4, ”IX” = 9, ”XL” = 40, and ”CM” = 900).
- The symbol ‘I’ may appear once before ’V’ and ’X’; the symbol ’X’ may appear once before ’L’ and ’C’; the symbol ’C’ may appear once before ’D’ and ’M’; and the symbols ’V’, ’L’, and ’D’ cannot appear before a greater symbol (e.g., 45 is written ”XLV”, not ”VL”; and 49 is written ”XLIX”, not ”IL”; and 1998 is written ”MCMXCVIII”, not ”IIMM”).

[P.Sestoft’08]

[NB: you may also criticize the spec (if you find it ambiguous); please do not use other specs of Roman Numerals! ]
Myers’ 10 Testing Principles

1) **expected output** is part of a test case
2) avoid testing own program
3) avoid testing own (organization’s) program
4) thoroughly inspect **result** of each test case
5) also write "invalid / unexpected" test cases
6) also check: **doesn’t do what not supposed to**
7) avoid **throw-away** test cases
8) never assume no errors when making test cases
9) Prob(more errors) ~ Prob(#errors already found)
10) Testing is creative+intellectually challenging task