MTAT.03.159: Software Testing

Lecture 02: Basic Black-Box and White-Box Testing Techniques
(Textbook Ch. 4 & 5)

Spring 2018

Dietmar Pfahl
email: dietmar.pfahl@ut.ee
Structure of Lecture 2

- Black-Box vs. White-Box Testing
- Basic Black-Box Testing Techniques
- Basic White-Box Testing Techniques
- Lab 2
Testing is difficult

Assume a ’magic’ Function $M$

$M (x, y) \rightarrow z$
with $x, y: \text{int (32 bit)}$

Exhaustive testing:

How many test cases,
If only valid input (=int) used?

Black Box

$M (x, y) = ?$
Testing is difficult

Assume a ’magic’ Function M

\[ M(x, y) \rightarrow z \]

with \( x, y: \text{int (32 bit)} \)

Exhaustive pos. testing:

\[ 2^{32} \times 2^{32} \]

\[ = 2^{64} \sim 1.8 \times 10^{19} \text{ test cases (input data + expected output)} \]
Testing is difficult

Assume a 'magic' Function \( M \)

\[ M(x, y) \rightarrow z \]

with \( x, y: \text{int (32 bit)} \)

Possible approaches:
- ???

White Box

```c
... if ( x - 100 <= 0 ) {
    if ( y - 100 <= 0 ) {
        if ( x + y - 200 == 0 ) {
            z = x / (y - 100);
        }
    }
}
...```

MTAT.03.159 / Lecture 02 / © Dietmar Pfahl 2018
Testing is difficult

Assume a ‘magic’ Function $M$

$M(x, y) \rightarrow z$

with $x, y$: int (32 bit)

Possible approaches:
- Execute each statement
- Read (review) code

White Box

```java
... 
if ( x - 100 <= 0 ) {
    if ( y - 100 <= 0 ) {
        if ( x + y - 200 == 0 ) {
            z = x / (y - 100);
        }
    }
}
... 
```

How?
Testing is difficult

Assume a ’magic’ Function M

\[ M(x, y) \rightarrow z \]

with \( x, y: \text{int (32 bit)} \)

1st if = true: \( x \leq 100 \)
2nd if = true: \( y \leq 100 \)
3rd if = true: \( x + y = 200 \)

\[ M(100, 100) \rightarrow \text{crash} \]

White Box

```java
...
if ( x - 100 <= 0 ) {
    if ( y - 100 <= 0 ) {
        if ( x + y - 200 == 0 ) {
            z = x / (y - 100);
        }
    }
}
...```

Black-Box vs. White-Box

External/user view:
- Check conformance with specification -> function coverage

Abstraction from details:
- Source code not needed

Scales up:
- Different techniques at different levels of granularity

USE BOTH!

Internal/developer view:
- Allows tester to be confident about code coverage

Based on control and data flow:
- Easier debugging

Does not scale up:
- Most useful at unit & integration testing levels, as well as regression testing
Black-Box vs. White-Box

Gray-Box Testing

External/user view:
Check conformance with specification -> function coverage
Abstraction from details:
Source code not needed
Scales up:
Different techniques at different levels of granularity

Internal/developer view:
Allows tester to be confident about code coverage
Based on control or data flow:
Easier debugging
Does not scale up:
Most useful at unit & integration testing levels, as well as regression testing

USE BOTH!

Gray-Box Testing combines black-box and white-box testing; typically, the focus is on input/output testing (black-box view) which is informed by structural information of the code (white-box view).

Example: The tester knows that certain constraints on the input are checked by the unit under test.

Application, e.g., in regression testing: apply (or update) black-box test cases only where code has been changed;
Black-Box vs. White-Box
Black-Box vs. White-Box

Specification-based Testing:
Test against specification

Goal of BBT: Tries to check whether specified functionality is available and working correctly
Black-Box vs. White-Box

Specification-based Testing:
Test against specification

System

Specification

Implementation

Goal of BBT: Tries to check whether specified functionality is available and working correctly

Unexpected functionality: Cannot be (directly) revealed by black-box techniques
Black-Box vs. White-Box

System

Specification

Implementation

Structural Testing: Test against implementation

Goal of WBT: Tries to check, whether the Implementation is working correctly (there is no dead code, it’s maintainable, etc.); useful for debugging;
Black-Box vs. White-Box

System

Specification

Implementation

Goal of WBT: Tries to check, whether the Implementation is working correctly (there is no dead code, it’s maintainable, etc.); useful for debugging;

Structural Testing: Test against implementation

Missing functionality: Cannot be (directly) revealed by white-box techniques
Black-Box vs. White-Box

Specification-based Testing: Test against specification

Structural Testing: Test against implementation

System

Specification

Implementation

Missing functionality: Cannot be (directly) revealed by white-box techniques

Unexpected functionality: Cannot be (directly) revealed by black-box techniques
How do Black-Box and White-Box Testing relate to one another?

1. Develop an initial Test suite using BB techniques
2. Apply BB coverage criteria to enhance it
3. Analyze the parts of the code uncovered by BB test suite
4. Enhance the Test suite using WB techniques
5. Apply WB coverage criteria to enhance it
Structure of Lecture 2

• Black-Box vs. White-Box Testing
• Basic Black-Box Testing Techniques
• Basic White-Box Testing Techniques
• Lab 2
Black-Box Testing Techniques

- Equivalence class partitioning (ECP)
- Boundary value analysis (BVA)
- Cause-effect graphing
- Combinatorial testing
- State transition testing (State-based testing)
- Exploratory testing
- Usability testing
- A/B testing (UX)
Equivalence Class Partitioning (ECP)

- Split input space into classes which the software handles equivalently with regards to the output produced

- Select test cases to cover each class

\[\text{green area} = \text{valid}\]
\[\text{white area} = \text{invalid}\]
ECP – Simple Example

public static boolean adultFunction(int age) {
    boolean adult;
    if (age >= 18)
        adult = true;
    else
        adult = false;
    return adult;
}

What are the ECs?

green area = valid
white area = invalid
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

What are the ECs?

green area = valid
white area = invalid
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Note that this spec is rather vague:

• it is unclear at what age one is an adult
• It is unclear what happens, if invalid input is entered
• It is unclear whether certain plausibility checks about feasible ages are made, e.g.:
  • Can a person be older than 150 years?
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Note that this spec is rather vague:

- it is unclear at what age one is an adult
- It is unclear what happens, if invalid input is entered
- It is unclear whether certain plausibility checks about feasible ages are made, e.g.:
  - Can a person be older than 150 years?

Use own domain knowledge (adult age starts with 18)
Talk to developers and ask for clarification
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Output 1 = ‘adult’ \( \Rightarrow \) age \( \geq 18 \)
Output 2 = ‘not adult’ \( \Rightarrow \) age < 18
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Output 1 = ‘adult’ \(\Rightarrow\) age in \([18, 150]\)
Output 2 = ‘not adult’ \(\Rightarrow\) age in \([0, 18)\)
Output 3 = ‘invalid input’ \(\Rightarrow\) age not in \([0, 150]\)

green area = valid
white area = invalid
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Output 1 = ‘adult’ \(\Rightarrow\) age in \([18, 150]\)
Output 2 = ‘not adult’ \(\Rightarrow\) age in \([0, 18)\)
Output 3 = ‘invalid input’ \(\Rightarrow\) age not in \([0, 150]\)

\[\text{green area} = \text{valid}\]
\[\text{white area} = \text{invalid}\]

Could be refined into:
- age < 0
- age > 150
- age not an int
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Output 1 = ‘adult’ \( \Rightarrow \) age in \([18, 150)\)
Output 2 = ‘not adult’ \( \Rightarrow \) age in \([0, 18)\)
Output 3 = ‘invalid input’ \( \Rightarrow \) age not in \([0, 150]\)

Output 3 was not mentioned in the specification but it’s good practice to think about this possibility (programmers hopefully do!). Also the maximum age was not mentioned in the spec; the tester would have to talk to the developers to find out whether there is an age limit implemented (e.g., as plausibility check).
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in \([0, 18)\)
EC2: integer in \([18, 150]\)
EC3: integer not in \([0, 150]\) or not an integer

Output variable ‘adult’:
EC4: true
EC5: false

Output variable ‘error’:
EC6: ‘invalid input’

This is a variable derived based on reasoning of the tester
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in \([0, 18)\)
EC2: integer in \([18, 150]\)
EC3: integer not in \([0, 150]\) or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

These are ECs derived based on reasoning of the tester
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (minimum):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in \([0, 18)\)
EC2: integer in \([18, 150]\)
EC3: integer not in \([0, 150]\) or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (minimum):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (minimum):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>;
    error = ‘invalid input’
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (minimum):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in \([0, 18)\)
EC2: integer in \([18, 150]\)
EC3: integer not in \([0, 150]\) or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Coverage of ECs:

<table>
<thead>
<tr>
<th></th>
<th>EC1</th>
<th>EC2</th>
<th>EC3</th>
<th>EC4</th>
<th>EC5</th>
<th>EC6</th>
<th>EC7</th>
<th>EC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>TC2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>TC3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

EC3 could be split up into several separate ECs; then we would need more TCs

Coverage of ECs:

<table>
<thead>
<tr>
<th></th>
<th>EC1</th>
<th>EC2</th>
<th>EC3</th>
<th>EC4</th>
<th>EC5</th>
<th>EC6</th>
<th>EC7</th>
<th>EC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>TC2</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>TC3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
ECP Guidelines [Myers 79/04]

If input class:

- is a range, e.g., \( x = [0, 9] \) or
- is an ordered list of values, e.g., owner = \( <1, 2, 3, 4> \)
  \( \rightarrow \) one in-range/list and two out-of-range/list classes are defined
- is a set, e.g., vehicle is in \( \{\text{car, motorcycle, truck}\} \) or
- is a “must be” condition (boolean)
  \( \rightarrow \) one in-set and one out-of-set class are defined
- is anything else (e.g., invalid)
  \( \rightarrow \) partition further

Union set of all ECs should cover complete in/output space. ECs must not overlap.

Have enough test cases to cover all valid input classes
Have one test case for each invalid input class
ECP in case of more than one input

This figure is metaphor for the union set of ECs of all input variables.
Example – Insurance System

Specification Statement:

- System shall reject over-age insurance applicants

Specification Item:

- Reject male insurance applicants, if over the age of 80 years on day of application
- Reject female insurance applicants, if over the age of 85 years on day of application
Example –
Insurance System

Basic ECs:

Var age: accept == true \(\Rightarrow\) [18, 80] or (80, 85]
to avoid overlap \(\Rightarrow\) [18, 80] or (80, 85] \(\Rightarrow\) EC1, EC2
accept == false \(\Rightarrow\) not in [18, 85] \(\Rightarrow\) EC3

Var gender: accept == true \(\Rightarrow\) male or female \(\Rightarrow\) EC4, EC5
accept == false \(\Rightarrow\) not in {male, female} \(\Rightarrow\) EC6

Var accept: true or false \(\Rightarrow\) EC7, EC8
Example (cont.)

Input: Gender & Age | Output: accept/reject

UI – Case A

| Age:     | in [18, 80]? | O |
|         | in (80, 85]? | O |
|         | in (85, 99]? | O |
|         | other?       | O |

Gender: male O
female O

Enter

Result: <text>
Message: <text>

Result text in {<empty>, accept, reject}
Message text in {<empty>, missing input}

UI – Case B

Please enter gender (m, f):
<Message text>
Please enter age (integer>0):
<Message text>
Result: <text>

Message text in {
<empty>,
invalid input – retry or quit with Ctrl^D}
Result text in {
accept,
reject}
### Example – UI Case A

**Input:** Gender & Age | **Output:** accept/reject

#### Classes

<table>
<thead>
<tr>
<th>C1:</th>
<th>???</th>
</tr>
</thead>
</table>

**Class C1:**

<table>
<thead>
<tr>
<th>Age: in [18, 80]</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>in (80, 85)</td>
<td>O</td>
</tr>
<tr>
<td>in (85, 99)</td>
<td>O</td>
</tr>
<tr>
<td>other</td>
<td>O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gender:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>male</td>
<td>O</td>
</tr>
<tr>
<td>female</td>
<td>O</td>
</tr>
</tbody>
</table>

**Enter**

**Result:** <text>

**Message:** <text>

Result text in \{<empty>, accept, reject\}
Message text in \{<empty>, missing input\}
Example – UI Case A

Input: Gender & Age | Output: accept/reject

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>InputAge: [18, 80]</td>
</tr>
<tr>
<td>C2</td>
<td>InputAge: (80, 85]</td>
</tr>
<tr>
<td>C3</td>
<td>InputAge: (85, 99]</td>
</tr>
<tr>
<td>C4</td>
<td>InputAge: other</td>
</tr>
<tr>
<td>C5</td>
<td>InputAge: &lt;empty&gt;</td>
</tr>
<tr>
<td>C6</td>
<td>InputGender: Male</td>
</tr>
<tr>
<td>C7</td>
<td>InputGender: Female</td>
</tr>
<tr>
<td>C8</td>
<td>InputGender: &lt;empty&gt;</td>
</tr>
<tr>
<td>C9</td>
<td>OutputResult: &lt;empty&gt;</td>
</tr>
<tr>
<td>C10</td>
<td>OutputResult: ‘accept’</td>
</tr>
<tr>
<td>C11</td>
<td>OutputResult: ‘reject’</td>
</tr>
<tr>
<td>C12</td>
<td>OutputMsg: &lt;empty&gt;</td>
</tr>
<tr>
<td>C13</td>
<td>OutputMsg: ’missing input’</td>
</tr>
</tbody>
</table>

What do you say about: C1, C3, and C4?
Example – UI Case A

Input: Gender & Age | Output: accept/reject

Classes

C1: InputAge: [18, 80]
C2: InputAge: (80, 85]
C3: InputAge: (85, 99]
C4: InputAge: other
C5: InputAge: <empty>
C6: InputGender: Male
C7: InputGender: Female
C8: InputGender: <empty>
C9: OutputResult: <empty>
C10: OutputResult: ‘accept’
C11: OutputResult: ‘reject’
C12: OutputMsg: <empty>
C13: OutputMsg: ‘missing input’

Test Cases

Data: age, gender, result, message

How many test cases to cover all classes?
Example – UI Case A

Input: Gender & Age | Output: accept/reject

Classes

C1: InputAge: [18, 80]
C2: InputAge: (80, 85]
C3: InputAge: (85, 99]
C4: InputAge: other
C5: InputAge: <empty>
C6: InputGender: Male
C7: InputGender: Female
C8: InputGender: <empty>
C9: OutputResult: <empty>
C10: OutputResult: ‘accept’
C11: OutputResult: ‘reject’
C12: OutputMsg: <empty>
C13: OutputMsg: ’missing input’

Test Cases

Data: age, gender, result, message

TC1: <empty>, <empty>, <empty>, ’missing input’
TC2: 56, male, ’accept’, <empty>
TC3: 83, male, ’reject’, <empty>
TC4: 88, female, ’reject’, <empty>
TC5: other, female, ’reject’, <empty>

minimal,
TCs cover all classes
Example – UI Case A

Input: Gender & Age | Output: accept/reject

Classes

C1: InputAge: [18, 80]
C2: InputAge: (80, 85]
C3: InputAge: (85, 99]
C4: InputAge: other
C5: InputAge: <empty>
C6: InputGender: Male
C7: InputGender: Female
C8: InputGender: <empty>
C9: OutputResult: <empty>
C10: OutputResult: ‘accept’
C11: OutputResult: ‘reject’
C12: OutputMsg: <empty>
C13: OutputMsg: ‘missing input’

Test Cases

TC1: <empty>, male, <empty>, ’missing input’
TC2: other, <empty>, <empty>, ’missing input’
TC3: 56, male, ’accept’, <empty>
TC4: 83, male, ’reject’, <empty>
TC5: 88, female, ’reject’, <empty>
TC6: other, female, ’reject’, <empty>

If we consider ’missing input’ to be an error message caused by invalid input (<empty>), then it’s good practice to check for the effect of each invalid input class independently.
### Example – UI Case A

Input: Gender & Age | Output: accept/reject

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid EC</th>
<th>Invalid EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>C1: [18, 80]</td>
<td>C5: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C2: (80, 85)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3: (85, 99)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C4: other</td>
<td></td>
</tr>
<tr>
<td>Gender</td>
<td>C6: Male</td>
<td>C8: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C7: Female</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case (TC)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age Gend.</td>
<td>em M</td>
<td>oth em</td>
<td>56 M</td>
<td>83 M</td>
<td>88 F</td>
<td>oth F</td>
</tr>
<tr>
<td>Tests</td>
<td>C5</td>
<td>C8</td>
<td>C1 C6</td>
<td>C2 C6</td>
<td>C3 C7</td>
<td>C4 C7</td>
</tr>
</tbody>
</table>
Example – UI Case A
Input: Gender & Age | Output: accept/reject

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid EC</th>
<th>Invalid EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>C1: [18, 80]</td>
<td>C5: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C2: (80, 85]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3: (85, 99]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C4: other</td>
<td></td>
</tr>
<tr>
<td>Gender</td>
<td>C6: Male</td>
<td>C8: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C7: Female</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TC</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age Gend.</td>
<td>em M</td>
<td>oth em</td>
<td>56 M</td>
<td>83 M</td>
<td>88 F</td>
<td>oth F</td>
</tr>
<tr>
<td>Tests</td>
<td>C5</td>
<td>C8</td>
<td>C1 C6</td>
<td>C2 C6</td>
<td>C3 C7</td>
<td>C4 C7</td>
</tr>
</tbody>
</table>

What is missing?
Example – UI Case A

Input: Gender & Age | Output: accept/reject

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid EC</th>
<th>Invalid EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>C1: [18, 80]</td>
<td>C5: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C2: (80, 85)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3: (85, 99)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C4: other</td>
<td></td>
</tr>
<tr>
<td>Gender</td>
<td>C6: Male</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C7: Female</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C8: &lt;empty&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TC</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age Gend.</td>
<td>em M</td>
<td>oth</td>
<td>56 M</td>
<td>83 M</td>
<td>88 F</td>
<td>oth F</td>
</tr>
<tr>
<td>Tests</td>
<td>C5</td>
<td>C8</td>
<td>C1</td>
<td>C6</td>
<td>C2</td>
<td>C7</td>
</tr>
</tbody>
</table>

Must also check coverage of output ECs!
Example – UI Case A
Input: Gender & Age | Output: accept/reject

### Classes

| C1 | InputAge: [18, 80] |
| C2 | InputAge: (80, 85] |
| C3 | InputAge: (85, 99] |
| C4 | InputAge: other |
| C5 | InputAge: <empty> |
| C6 | InputGender: Male |
| C7 | InputGender: Female |
| C8 | InputGender: <empty> |
| C9 | OutputResult: <empty> |
| C10| OutputResult: ‘accept’ |
| C11| OutputResult: ‘reject’ |
| C12| OutputMsg: <empty> |
| C13| OutputMsg: ’missing input’ |

### Test Cases

| TC1 | <empty>, male, <empty>, ’missing input’ |
| TC2 | other, <empty>, <empty>, ’missing input’ |
| TC3 | 56, male, ’accept’, <empty> |
| TC4 | 83, male, ’reject’, <empty> |
| TC4* | 83, female, ’accept’, <empty> |
| TC5 | 88, female, ’reject’, <empty> |
| TC6 | other, female, ’reject’, <empty> |

Now, we have covered all cause-effect relationships (→ Cause-Effect Graphing)
Example – UI Case B
Input: Gender & Age | Output: accept/reject

Classes
C1: ???

Please enter gender (m, f):
<Message text>
Please enter age (integer>0):
<Message text>
Result: <text>

Message text in {
<empty>,
invalid input – retry or quit with Ctrl^D}
Result text in {
accept,
reject}
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input ...’
C13: OutputResult: accept
C14: OutputResult: reject

Please enter gender (m, f):
<Message text>
Please enter age (integer>0):
<Message text>
Result: <text>

Message text in {
    <empty>,
    invalid input – retry or quit with Ctrl^D}
Result text in {
    accept,
    reject}
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input ...’
C13: OutputResult: accept
C14: OutputResult: reject

Test Cases

TC1: Ctrl^D
TC2: not(m, f), ’invalid’, Ctrl^D
TC3: m, <empty>, Ctrl^D
TC4: m, <empty>, other, ’invalid’, Ctrl^D
TC5: m, <empty>, [18, 80], <empty>, accept
TC6: m, <empty>, (80, 85], <empty>, reject
TC7: f, <empty>, <18, <empty>, reject
TC8: f, <empty>, >85, <empty>, reject

...
Example – UI Case B
Input: Gender & Age | Output: accept/reject

Classes
C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input …’
C13: OutputResult: accept
C14: OutputResult: reject

Test Cases
TC1: Ctrl^D
TC2: g, ’invalid’, Ctrl^D
TC3: m, <empty>, Ctrl^D
TC4: m, <empty>, 3.5, ’invalid’, Ctrl^D
TC5: m, <empty>, 56, <empty>, accept
TC6: m, <empty>, 83, <empty>, reject
TC7: f, <empty>, 5, <empty>, reject
TC8: f, <empty>, 103, <empty>, reject
...
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: ’invalid input ...’
C13: OutputResult: accept
C14: OutputResult: reject

Test Cases

Every path from the root to a leaf is (at least) one test case
Boundary Value Analysis

• Adds to the equivalence partitioning method
• Select test cases to represent each side of the class boundaries
Boundary Value Analysis Guidelines

- Range $a..b \Rightarrow a$, $b$, just above $a$, just below $b$
- List of values $\Rightarrow$ max, min, just below min, just above max

- Boundaries of externally visible data structures shall be checked (e.g. ordered sets, arrays)
- Output bounds should be checked
Example – UI Case B

Input: Gender & Age | Output: accept/reject

Classes

C1: InputGender: m
C2: InputGender: f
C3: InputGender: not(m, f)
C4: InputGender: Ctrl^D
C5: InputAge: integer in [18, 80]
C6: InputAge: integer in (80, 85]
C7: InputAge: integer <18
C8: InputAge: integer >85
C9: InputAge: Ctrl^D
C10: InputAge: other than C5-C9
C11: OutputMsg: <empty>
C12: OutputMsg: 'invalid input ...'
C13: OutputResult: accept
C14: OutputResult: reject

Test Cases

TC1: Ctrl^D
TC2: g, 'invalid’, Ctrl^D
TC3: m, <empty>, Ctrl^D
TC4: m, <empty>, 3.5, ’invalid’, Ctrl^D
TC5: m, <empty>, 56, <empty>, accept
TC5L: m, <empty>, 18, <empty>, accept
TC5U: m, <empty>, 80, <empty>, accept
TC6: m, <empty>, 83, <empty>, reject
... 
TC7: f, <empty>, 5, <empty>, reject
TC8: f, <empty>, 103, <empty>, reject
...
Combinatorial Designs

- ECP and BVA define test cases per equivalence class.
- In ECP testing, each EC needs to be covered once.
- In Combinatorial Testing all possible combinations of ECs of the input variables need to be covered.
Structure of Lecture 2

• Black-Box vs. White-Box Testing
• Basic Black-Box Testing Techniques
• Basic White-Box Testing Techniques
• Lab 2
White-Box Testing Techniques

- Control-Flow Testing
- Symbolic Execution
- Data-Flow Testing
- Mutation Testing
- Static Code Analysis
- Reviews

Lectures 4 & 6
Testing Strategies

Black Box Testing

White Box Testing

requirements

input

events

output
White-Box Testing

There are many possible paths! $5^{20} \sim 10^{14}$ different paths

Selective Testing
Selective Testing

- a selected path

2 Major Strategies

- Control flow testing
- Data flow testing
Code Coverage (Test Coverage)

Definition:

Measures the extent to which certain code items related to a defined test adequacy criterion have been executed (covered) by running a set of test cases (= test suite)

Goal:

Define test suites such that they cover as many (disjoint) code items as possible
Main Classes of Test Adequacy Criteria

Control Flow Criteria:

- Statement, decision (branch), condition, and path coverage are examples of control flow criteria.
- They rely on syntactic characteristics of the program (ignoring the semantics of the program computation).

Data Flow Criteria:

- Require the execution of path segments that connect parts of the code that are intimately connected by the flow of data (-> ‘annotated control flow graph’).
Code Coverage Measure – Example

Statement Coverage ($CV_s$)

Portion of the statements tested by at least one test case.

\[
CV_s = \left( \frac{S_t}{S_p} \right) \times 100\%
\]

$S_t$ : number of statements tested

$S_p$ : total number of statements
Code Coverage Measure – Tools

For Java:
IntelliJ code coverage
Emma
JaCoCo
Clover
etc.

http://www.eclemma.org/index.html

Note: EclEmma requires Eclipse
Code Coverage Measure – EclEmma

Branch coverage

Line coverage

public boolean addAll(int index, Collection c) {
    if (c.isEmpty()) {
        return false;
    } else if (_size == index || _size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while (it.hasNext()) {
            pred = insertListable(pred, succ, it.next());
        }
    }
    return true;
}
Code Coverage Measure – IntelliJ Code Coverage Tool

View coverage results:

In the Project tool window:

- src (33% classes, 13% lines covered)
- MetersToInchesConverter (50% methods, 16% lines covered)
- OddChecker (0% methods, 0% lines covered)

In the dedicated Coverage tool window:

- Coverage Summary for 'all classes in scope': 33% classes, 13% lines covered
- Main (0% methods, 0% lines covered)
- MetersToInchesConverter (100% methods, 16% lines covered)
- OddChecker (0% methods, 0% lines covered)
Code Coverage Measure – IntelliJ Code Coverage Tool

Use the color indicators in the left gutter to detect the uncovered lines of code.

To find out how many times a line has been hit, click the line in the gutter area.

The pop-up window that opens shows the statistic for the line at caret. For lines with conditions, the pop-up window also provides statistic.
Code Coverage Measure – IntelliJ Code Coverage Tool

Use the color indicators in the left gutter to detect the uncovered lines of code.

To find out how many times a line has been hit, click the line in the gutter area.

The pop-up window that opens shows the statistic for the line at caret. For lines with conditions, the pop-up window also provides statistic.

MTAT.03.159 / Lecture 02 / © Dietmar Pfahl 2018
Code Coverage Measure – IntelliJ Code Coverage Tool

Use the color indicators in the left gutter to detect the uncovered lines of code.

To find out how many times a line has been hit, click the line in the gutter area.

The pop-up window that opens shows the statistic for the line at caret. For lines with conditions, the pop-up window also provides statistic.
Control Flow Graph (CFG)

Program

```
x = z-2;
y = 2*z;
if (c) {
    x = x+1;
y = y+1;
}
else {
    x = x-1;
y = y-1;
}
z = x+y;
```

Control Flow Graph

- **B₁**: \( x = z-2; \) \( y = 2*z; \)
- **B₂**: \( x = x+1; \) \( y = y+1; \)
- **B₃**: \( x = x-1; \) \( y = y-1; \)
- **B₄**: \( z = x+y; \)

Edges:
- **B₁** → **B₂** if \( c = T \)
- **B₁** → **B₃** if \( c = F \)
- **B₂** → **B₄**
Control Flow Graph (CFG)

Program

```c
x = z-2;
y = 2*z;
if (c) {
    x = x+1;
y = y+1;
}
else {
    x = x-1;
y = y-1;
}
```

Control Flow Graph

- Blocks (=Nodes): 4
- Edges: 4

Blocks:
- B₁: x = z-2; y = 2*z;
- B₂: x = x+1; y = y+1;
- B₃: x = x-1; y = y-1;
- B₄: empty

Condition:
- c = T
- c = F
Control Flow Graph (CFG)

- **Blocks**: 4
- **Edges**: 4

**Nodes**: 8
- **Edges**: 8

- $d_1$ is a 'dummy node'
- **Entry and exit nodes are 'dummy nodes'**

**Control Flow Graph**

**Blocks**
- $B_1$: $x = z-2$; $y = 2*z$
- $B_2$: $x = x+1$; $y = y+1$
- $B_3$: $x = x-1$; $y = y-1$
- $B_4$: empty

**Cycles**
- $c = T$: $c = F$
Control Flow – Example

If (d1) then {
    if (d2) then {s1}
    s2
    while (d3) do {s3}
}
else {
    if (d4) then {
        repeat {s4} until (d5)
    }
}
Control Flow – Example

If (d1) then {
  if (d2) then {s1}
  s2
} else {
  if (d4) then {
    repeat {s4} until (c5)
  }
}

CFG(f)

If (d1) then {
  CFG(d1=true)
} else {
  CFG(d1=false)
}
Control Flow – Example

If (d1) then {
  if (d2) then {s1}
  s2
  while (d3) do {s3}
} else {
  if (d4) then {
    repeat {s4} until (d5)
  }
}
Control Flow – Example

If (d1) then {
  if (d2) then {s1}
  s2
  while (d3) do {s3}
} else {
  if (d4) then {
    repeat {s4} until (c5)
  }
}

CFG(if)

s2

CFG(while)

s2

CFG(repeat)

if (d4) then {
  CFG(repeat)
}

else {
  if (d4) then {
    CFG(while)
  }
}
Control Flow – Example

If (d1) then {
  if (d2) then {s1}
  s2
  while (d3) do {s3}
} else {
  if (d4) then {
    repeat {s4} until (d5)
  }
} else {
  if (d4) then {
    repeat {s4} until (d5)
  }
}
Control Flow – Example

If (d1) then {
    if (d2) then {s1}
    s2
    while (d3) do {s3}
}
else {
    if (d4) then {
        repeat {s4} until (d5)
    }
}
Overview of Control Flow Criteria

Statement (or Block) Coverage – all nodes
Decision (or Branch) Coverage – all edges
Condition Coverage
Condition/Decision Coverage
Multiple Condition Coverage
Modified Condition Decision Coverage (MC/DC)
Linearly Independent Paths
Loop Testing
...

MTAT.03.159 / Lecture 02 / © Dietmar Pfahl 2018
Control-Flow Graph for Exception Handling

```java
void tryCatchTestMethod(int b, int c, int t) {
    try {
        mightThrowAnException(b);
    } catch (Exception e) {
        b = 3;
    } finally {
        t = b*3;
        c = 3;
    }
}
```
Statement Coverage

Execute each statement at least once
Use tools to monitor execution
More practice in Lab 2

A possible concern may be:
Dead code
Life Insurance Example

```java
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)  
   return (TRUE);
2: if (gender == male & age < 80)    
   return (TRUE);
3: return (FALSE);
```

In the following assume that the following pre-conditions have been checked:
- Parameter 'gender' is in {female, male}
- Parameter 'age' is integer and >= 18
Statement Coverage /1

```java
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85) return TRUE;
2: return TRUE;
3: if (gender == male & age < 80) return TRUE;
4: return FALSE;
```

Test:

```
0 %
```
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)
2:     return(TRUE);
3: if (gender == male & age < 80)
4:     return(TRUE);
5: return(FALSE);

Test:
AccClient(83, female)->true

40 %
Statement Coverage /3

```java
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)
  2:   return (TRUE);
3: if (gender == male & age < 80)
  4:   return (TRUE);
5: return (FALSE);
```

Test:
AccClient(83, female)->true
AccClient(83, male)->false

80 %
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)
2: return(TRUE);
3: if (gender == male & age < 80)
4: return(TRUE);
5: return(FALSE);

Test:
AccClient(83, female)->true
AccClient(83, male)->false
AccClient(25, male)->true

100 %
Same Test Suite but Incorrect Code in Life Insurance Example (1)

```
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 80)  
   return(TRUE);
2:   return(TRUE);
3:   return(FALSE);
```

Test:
AccClient(83, female)->false
AccClient(83, male)->false
AccClient(25, male)->true

Where is the bug?

80 %
Same Test Suite but Incorrect Code in Life Insurance Example (1)

```java
boolean AccClient(int age; gtype gender) {
    if (gender == female & age < 80)
        return (TRUE);
    if (gender == male & age < 80)
        return (TRUE);
    return (FALSE);
}
```

Test:
- AccClient(83, female)->false
- AccClient(83, male)->false
- AccClient(25, male)->true

80 %
Same Test Suite but Incorrect Code in Life Insurance Example (2)

```java
boolean AccClient(int age; gtype gender)

1: if (gender == female & age > 85) return (TRUE);
2: return (TRUE);
3: if (gender == male & age < 80) return (TRUE);
4: return (FALSE);
```

1 fault

1 fault triggers
1 failure

Test:
AccClient(83, female)->false
AccClient(83, male)->false
AccClient(25, male)->true

80 %
Same Test Suite but Incorrect Code in Life Insurance Example (2)

```java
boolean AccClient(int age; gtype gender)

1: if (gender == female & age > 80)
2:     return (TRUE);
3: if (gender == male & age < 80)
4:     return (TRUE);
5: return (FALSE);
```

Test:
AccClient(83, female)->true
AccClient(83, male)->false
AccClient(25, male)->true
Statement Coverage : Dead Code?

```java
boolean AccClient(int age; gtype gender)

1: if (gender == female){
2:    if (age < 85)
3:        return(TRUE);
4:    return(FALSE);}
5: if (gender == male){
6:    if (age < 80)
7:        return(TRUE);
8:    return(FALSE);}
9: return(FALSE);
```

Test:
AccClient(83, female)->true
AccClient(83, male)->false
AccClient(25, male)->true

78 %
Statement Coverage: Dead Code?

```java
boolean AccClient(int age; gtype gender)

1: if (gender == female){
2:     if (age < 85)
3:         return(TRUE);
4:     return(FALSE);} 
5: if (gender == male){
6:     if (age < 80)
7:         return(TRUE);
8:     return(FALSE);} 
9: return(FALSE);
```

Test:
AccClient(83, female)->true
AccClient(83, male)->false
AccClient(25, male)->true

78%
Statement Coverage : Dead Code?

```java
boolean AccClient(int age; gtype gender)
1: if (gender == female){
2:    if (age < 85)
3:        return(TRUE);
4:    return(FALSE);
5: if (gender == male){
6:    if (age < 80)
7:        return(TRUE);
8:    return(FALSE);
9: return(FALSE);
```

Test:
- AccClient(83, female)->true
- AccClient(83, male)->false
- AccClient(25, male)->true

100 %
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)
   return (TRUE);
2:   return (TRUE);
3: if (gender == male & age < 80)
4:   return (TRUE);
5: return (FALSE);

Test:

0 %
Decision (Branch) Coverage /2

boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)
2: return(TRUE);
3: if (gender == male & age < 80)
4: return(TRUE);
5: return(FALSE);

Test:
AccClient(83, female)->true

25 %
boolean AccClient(int age, gtype gender)

1: if (gender == female & age < 85)
2:   return (TRUE);
3: if (gender == male & age < 80)
4:   return (TRUE);
5: return (FALSE);

Test:
AccClient(83, female)->true
AccClient(83, male)->false

75 %
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)  
2:     return(TRUE);  
3: if (gender == male & age < 80)  
4:     return(TRUE);  
5: return(FALSE);

Test:
AccClient(83, female)->true  
AccClient(83, male)->false  
AccClient(25, male)->true

100 %
Condition Coverage

• Test all conditions (in all predicate nodes):
  • Minimum: Each condition must evaluate at least once
  • Simple: Each condition must evaluate at least \textit{once to ‘true’ and once to ‘false’}

• Example of a decision (predicate) with two conditions:
  \begin{verbatim}
  If (A==female & B<85) then ...
  \end{verbatim}

• A predicate may contain several conditions connected via Boolean operators
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)  
2:    return (TRUE);
3: if (gender == male & age < 80)  
4:    return (TRUE);
5: return (FALSE);

Test:

0 %
Condition Coverage /2

boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)
2:    return(TRUE);
3: if (gender == male & age < 80)
4:    return(TRUE);
5: return(FALSE);

Test:
AccClient(83, female)->true

50 % or 25 %
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85)
2:    return(TRUE);
3: else if (gender == male & age < 80)
4:    return(TRUE);
5: else return(FALSE);

Test:
AccClient(83, female)->true
AccClient(83, male)->false

100 % or 62.5 %
**Condition Coverage /4**

```java
boolean AccClient(int age; gtype gender)

1: if (gender == female & age < 85) return(TRUE);
2:    return(TRUE);
3: if (gender == male & age < 80) return(TRUE);
4:    return(FALSE);
5: return(FALSE);
```

Test:
- AccClient(83, female)->true
- AccClient(83, male)->false
- AccClient(25, male)->true

100 % or 75 %
Advanced Condition Coverage

Condition/Decision Coverage (C/DC)
- as DC plus: every condition in each decision is tested in each possible outcome

Modified Condition/Decision coverage (MC/DC)
- as above plus, every condition shown to independently affect a decision outcome (by varying that condition only)
  
  Def: A condition independently affects a decision when, by flipping that condition’s outcome and holding all the others fixed, the decision outcome changes
- this criterion was created at Boeing and is required for aviation software according to RCTA/DO-178B

Multiple-Condition Coverage (M-CC)
- all possible combinations of condition outcomes within each decision is checked
CC, DC, C/DC, M-CC, MC/DC Examples

If (A==fem & B<85) ...

Minimum and Simple Condition (CC):
(TF) A = fem; B = 200 (D: False)
[(FT) A = male; B = 80 (D: False)]

Decision (DC):
(TT) A = fem; B = 80 (D: True)
(FT) A = male; B = 80 (D: False)

Condition/Decision (C/DC):
(TT) A = fem; B = 80 (D: True)
(FF) A = male; B = 200 (D: False)

Multiple Condition (M-CC):
(TT) A = fem; B = 80 (D: True)
(FT) A = male; B = 80 (D: False)
(TF) A = fem; B = 200 (D: False)
(FF) A = male; B = 200 (D: False)

Modified Condition/Decision (MC/DC):
(TT) A = fem; B = 80 (D: True)
(FT) A = male; B = 80 (D: False)
(TF) A = fem; B = 200 (D: False)
Modified Condition/Decision (MC/DC)

If \( A = \text{fem} \) and \( B < 85 \) then ...

<table>
<thead>
<tr>
<th>TC</th>
<th>A</th>
<th>B</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T (fem)</td>
<td>T (80)</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F (male)</td>
<td>T (80)</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>T (fem)</td>
<td>F (200)</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F (male)</td>
<td>F (200)</td>
<td>F</td>
</tr>
</tbody>
</table>

Multiple Condition:
- \((TT) A = \text{fem}; B = 80 \ (D: \text{True})\)
- \((FT) A = \text{male}; B = 80 \ (D: \text{False})\)
- \((TF) A = \text{fem}; B = 200 \ (D: \text{False})\)
- \((FF) A = \text{male}; B = 200 \ (D: \text{False})\)

TC1+TC2: change in A -> Dec changed
TC1+TC3: change in B -> Dec changed

All other TC combinations in which only one condition outcome changes don’t have an effect on the decision outcome.

Result: only TC1, TC2, and TC3 needed
Paths Coverage

• **Path Coverage Criterion**: Select a test set $T$ such that, by executing $P$ for each test case $t$ in $T$, all paths leading from the initial to the final node of $P$'s control flow graph are traversed.

• In practice, however, the number of paths is too large, if not infinite (e.g., when we have loops).

• Some paths are infeasible (e.g., not practical given the system’s business logic).

• It may be important to determine “critical paths”, leading to more system load, security intrusions, etc.
Independent Path Coverage

- McCabe cyclomatic complexity estimates number of test cases needed
- The number of independent paths needed to cover all simple paths at least once in a program
  - Visualize by drawing a CFG
  - $CC = \#(edges) - \#(nodes) + 2$
  - $CC = \#(decisions) + 1$
Independent Paths Coverage – Example

- Independent Paths Coverage
  - Requires that a minimum set of linearly independent paths through the control flow-graph be executed

- This test strategy is the rationale for McCabe’s cyclomatic number (McCabe 1976) …
  - … which is equal to the number of test cases required to satisfy the strategy.

Cyclomatic Complexity = 5 + 1 = 6
Independent Paths Coverage – Example

Edges: 1-2-3-4-5-6-7-8-9-10-11-12-13-14

Path1: 1-0-0-1-0-1-0-1-0-1-0---0---0---0---0
Path2: 1-0-1-0-1-1-1-1-1-1-0---0---0---0---0
Path3: 1-0-0-1-0-1-1-1-1-1-0---0---0---0---0
Path4: 0-1-0-0-0-0-0-0-0-0-1---0---0---1---0
Path5: 0-1-0-0-0-0-0-0-0-0-1---0---0---0---1
Path6: 0-1-0-0-0-0-0-0-0-0-1---0---0---0---0
Independent Paths Coverage – Example

Edges: 1-2-3-4-5-6-7-8-9-10-11-12-13-14

Why no need to cover Path7 ???

Path7: 1-0-1-0-1-0-0-1-0---0---0---0---0

Path1: 1-0-0-1-0-1-0-0-1-0---0---0---0---0

Path2: 1-0-1-0-1-1-1-1-1-0---0---0---0---0

Path3: 1-0-0-1-0-1-1-1-1-0---0---0---0---0

-P3: 1-0-1-0-1-1-0-0-1-0---0---0---0---0
Independent Paths Coverage – Example

Edges: 1-2-3-4-5-6-7-8-9-10-11-12-13-14

Why no need to cover Path7 ???

Path7: 1-0-1-0-1-1-0-0-1-0---0---0---0---0

Because it equals Path1+Path2-Path3 !!!

Path1: 1-0-0-1-0-1-0-0-1-0---0---0---0---0

Path2: 1-0-1-0-1-1-1-1-1-0---0---0---0---0

P1+P2: 2-0-1-1-1-2-1-1-2-0---0---0---0---0

Path3: 1-0-0-1-0-1-1-1-1-0---0---0---0---0

-P3: 1-0-1-0-1-1-0-0-1-0---0---0---0---0
Control-Flow Coverage Relationships

**Subsumption:**

A criterion C1 subsumes another criterion C2, if any test set \( \{T\} \) that satisfies C1 also satisfies C2.
Loop Testing

- Simple loop
- Nested loops
- Concatenated loops
- Unstructured loops
Loop Testing: Simple Loops

Minimum conditions - simple loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop m < n
5. set loop counter to (n-1), n and (n+1): passes twice through the loop and once not

… where n is the maximum number of allowable passes
Nested Loops

Extend simple loop testing

Reduce the number of tests:

- start at the innermost loop; set all other loops to minimum values
- conduct simple loop test; add out of range or excluded values
- work outwards while keeping inner nested loops to typical values
- continue until all loops have been tested
Structure of Lecture 2

• Black-Box vs. White-Box Testing
• Basic Black-Box Testing Techniques
• Basic White-Box Testing Techniques
• Lab 2
Lab 2: Part 1 – Black-Box Testing

Lab 2 (week 27: Mar 06 - Mar 07) – Black-Box Testing (5%)  

Submission Deadlines:  
  • Tuesday Labs: Monday, 13 Mar, 23:59  
  • Wednesday Labs: Tuesday, 14 Mar, 23:59  

• Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late
Lab 2: Part 1 – Black-Box Testing (cont’d)

Test Cases:
Input & Exp. Output
1 3 4 → triangle type x
2 5 5 → triangle type y
0 1 2 → no triangle
...

Strategies:
- Equivalence Class Partitioning
- Boundary Value Analysis

Test Report:
TC1 → pass
TC2 → pass
TC3 → pass
TC4 → fail → defect
...

Program
Lab 2: Part 2  – White-Box Testing

Lab 2 (week 22: Mar 06 - Mar 07) - White-Box Testing (5%)

WBT Instructions
WBT Sample Code

Submission Deadlines:
- Tuesday Labs: Monday, 13 Mar, 23:59
- Wednesday Labs: Tuesday, 14 Mar, 23:59

• Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late
Lab 2: Part 2 – White-Box Testing (cont’d)

Instructions

Control-Flow Graph
Set of 10+ Test Cases 1
Set of 15+ Test Cases 2

Coverage Criteria:
- Instruction/Statement (Line)
- Branch (Decision)
Tool: IntelliJ IDEA or Eclipse Plugin

Test Report 2 & Test Coverage 2a + 2b
Recommended Textbook Exercises

Chapter 4
   1, 2, 3, 4, 5
   8, 11, 12

Chapter 5
   2, 5, 6, 9, 10,
   11, 14
Next two weeks …

• Lab 2:
  – Black-Box and White-Box Testing

• Lecture 3:
  – Advanced Black-Box Testing Techniques

• In addition to do:
  – Read textbook chapters 4 & 5 (available via OIS)