Lecture 02: Basic Black-Box Techniques

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Spring 2019
Lectures (J. Liivi 2-111)

- Lecture 1 (14.02) – Introduction to Software Testing
- Lecture 2 (21.02) – Basic Black-Box and White-Box Testing Techniques (overview)
- Lecture 3 (28.02) – BBT advanced: Combinatorial Testing
- Lecture 4 (07.03) – WBT advanced: Control-Flow and Data-Flow Coverage Criteria
- Lecture 5 (14.03) – Test Lifecycle, Test Levels, Test Tools
- Lecture 6 (21.03) – BBT advanced: State-Transition Testing
- Lecture 7 (28.03) – Behavioural Testing / GUI Testing / Visual Testing
- Lecture 9 (11.04) – Test-Suite Effectiveness / Mutation Testing
- Lecture 11 (25.04) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
- 02.05 - no lecture
- Lecture 12 (09.05) – Industry Guest Lecture (to be announced)
- Lecture 13 (16.05) – Exam Preparation
Lecture >> Lab 2

- Textbook Chapter 4: Test case design
  - Black-box testing techniques (→ Lab 2)
Structure of Lecture 2

- Black-Box vs. White-Box Testing
- Basic Black-Box Testing Techniques
  - Equivalence Class Partitioning (ECP)
  - Boundary Value Analysis (BVA)
- Lab 2
Testing is difficult

Assume a ‘magic’ Function $M$

$M(x, y) \rightarrow z$

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

Exhaustive pos. testing:

\[2^{32} \times 2^{32} = 2^{64} \approx 1.8 \times 10^{19}\] test cases (input data + expected output)
Testing is difficult

Assume a 'magic' Function M

\[ M(x, y) \rightarrow z = \text{SUM}(x, y) \]

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 $M$

$$M(x, y) \rightarrow z = \text{SUM}(x, y)$$

with $x, y$: 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);
        }
    }
}
```
Testing is difficult

Assume a ’magic’ Function M

\[ M(x, y) \rightarrow z = \text{SUM}(x, y) \]

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

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

White Box

```c
... 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

\[ M(x, y) \rightarrow z = \text{SUM}(x, y) \]
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) -> 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

## 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

**USE BOTH!**
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

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;
Different techniques at different levels of granularity

USE BOTH!
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

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

Unexpected functionality: Cannot be (directly) revealed by black-box techniques
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

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

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
Black-Box vs. White-Box

Specification-based Testing:
Test against specification

System

Specification

Implementation

Structural Testing:
Test against 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
  • Equivalence Class Partitioning (ECP)
  • Boundary Value Analysis (BVA)
• 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

  green area = valid
  white area = invalid

- Select test cases to cover each class
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’ → age ≥ 18
Output 2 = ‘not adult’ → age < 18

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]

green area = valid
white area = invalid

[18, 150]
[0, 18)
not in [0, 150]
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

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).
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’
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></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>x</td>
<td></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’

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></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></td>
</tr>
<tr>
<td>TC3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ECP Guidelines

Possible inputs/outputs must be known from specification or derived from exploration of the system.

If input is a range or an ordered list of values

→ one in-range/list and two out-of-range/list classes are defined

→ Example: \( x \) in \([0, 9]\) → EC1: \([0, 9]\), EC2: (-\(\infty\), 0), EC3: (9, +\(\infty\))

If input is a set or a “must be” condition (boolean)

→ one in-set and one out-of-set class are defined

→ Example: vehicle is in \{car, motorcycle, truck\} → EC1: \{car\}, EC2: \{motorcycle\}, EC3: \{truck\} → if additional inputs are possible, then additional ECs must be defined

If analysis of spec indicates that elements of input classes result in specific output classes (i.e., are treated equivalently) then additional classes may be defined.

If there is anything else (e.g., invalid inputs) → partition further
ECP Guidelines

Possible inputs/outputs must be known from specification or derived from exploration of the system.

If input is a range or an ordered list of values

→ one in-range/list and two out-of-range/list classes are defined

→ Example: x in [0, 9] → EC1: [0, 9], EC2: (-infinity, 0), EC3: (9, +infinity)

If input s a set or a “must be” condition (boolean)

→ one in-set and one out-of-set class are defined

→ Example: vehicle is in {car, motorcycle, truck} → EC1: {car}, EC2: {motorcycle}, EC3: {truck} → if additional inputs are possible, then additional ECs must be defined

If analysis of spec indicates that elements of input classes result in specific output classes (i.e., are treated equivalently) then additional classes may be defined.

If there is anything else (e.g., invalid inputs) → partition further
ECP in case of more than one input

This figure is metaphor for the union set of ECs of all input variables
From ECP to Test Cases
From ECP to Test Cases

Remember:
We must cover all ECs with Test Cases (TCs). We try to do this with as few TCs as possible!
From ECP to Test Cases

Step 1: Number each EC

Step 2: Apply the following rules ->

• Rule 1 – valid ECs: Combine as many valid ECs in one test case as possible

• Rule 2 – invalid ECs: Pick one invalid input EC and take a value from it; then combine it exclusively with values from valid input ECs

  • Reason: for each invalid EC there should be some dedicated error handling; this must be checked!

Remember: We must cover all ECs with Test Cases (TCs). We try to do this with as few TCs as possible!
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 ➞ [18, 80] or [18, 85]
  to avoid overlap ➞ [18, 80] or (80, 85] ➞ EC1, EC2
accept == false ➞ not in [18, 85] ➞ EC3

Var gender: accept == true ➞ male or female ➞ EC4, EC5
accept == false ➞ not in {male, female} ➞ EC6

Var accept: true or false ➞ 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
C1: ???

---

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

Gender:
- male O
- female O

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

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’

What do you say about: C1 to C5?
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>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>Other</td>
<td>56 M</td>
<td>83 M</td>
<td>88 F</td>
<td>Other</td>
</tr>
<tr>
<td>tests</td>
<td>C5</td>
<td>C6</td>
<td>C4</td>
<td>C8</td>
<td>C1</td>
<td>C6</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>C6</td>
<td>C4</td>
<td>C8</td>
<td>C1</td>
<td>C2</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></td>
<td></td>
</tr>
<tr>
<td>C1: [18, 80]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2: (80, 85]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3: (85, 99]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4: other</td>
<td></td>
<td>C5: &lt;empty&gt;</td>
</tr>
<tr>
<td>Gender</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6: Male</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7: Female</td>
<td></td>
<td>C8: &lt;empty&gt;</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</td>
<td>oth</td>
<td>56</td>
<td>83</td>
<td>88</td>
<td>oth</td>
</tr>
<tr>
<td>Tests</td>
<td>C5</td>
<td>C4</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>In/Out</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>
<tr>
<td>Result</td>
<td>C10: ‘acc’</td>
<td>C9: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C11: ‘reject’</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>C12: &lt;emp&gt;</td>
<td>C13: &lt;miss. In&gt;</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</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gend.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>em</td>
<td>em</td>
<td>mis</td>
<td>em</td>
<td>em</td>
<td>mis</td>
</tr>
<tr>
<td>Mess.</td>
<td>other</td>
<td>Male</td>
<td>acc</td>
<td>83</td>
<td>88</td>
<td>rej</td>
</tr>
<tr>
<td>covers</td>
<td>C5</td>
<td>C6</td>
<td>C9</td>
<td>C13</td>
<td>C6</td>
<td>C7</td>
</tr>
</tbody>
</table>

LTAT.05.006 / Lecture 02 / © Dietmar Pfahl 2019
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: ???

...
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
...

LTAT.05.006 / Lecture 02 / © Dietmar Pfahl 2019
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 (STOP) 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 $a..b \Rightarrow a, b, \text{just above } a, \text{just below } b$
- List of ordinal values $\Rightarrow \text{max, min, just below min, just above max}$

- 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
  • Equivalence Class Partitioning (ECP)
  • Boundary Value Analysis (BVA)
• Lab 2
Lab 2: Black-Box Testing

Lab 2 (week 26: Feb 26 & 27) – Black-Box Testing (10 pts)

BBT Instructions
BBT Documentation
BBT Application

Submission Deadlines:
- Tuesday Labs: Monday, 04 Mar, 23:59
- Wednesday Labs: Tuesday, 05 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: Black-Box Testing

– Triangle Program

Inputs:
Triangle side-1
Triangle side-2
Triangle side-3

Outputs:
Triangle perimeter
Triangle area
Triangle type

isosceles
right-angle
equilateral
scalene

At least 15 ECs
At least 20 test cases covering all defined ECs
Lab 2: Black-Box Testing (cont’d)

Test Cases:
Input & Exp. Output
1 3 4 → triangle type x, area, perimeter
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
...

Documentation

Program
To Do & Next Week

• Quiz 2 (in Moodle!):
  – Opens tomorrow morning – closes on Monday at 11:30am!

• Lab 2:
  – Basic Black-Box-Testing

• Lecture 3:
  – Black-Box Testing (advanced): Combinatorial Testing

• In addition to do:
  – Read Burnstein textbook chapter 4
Recommended Textbook Exercises

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