Lecture 03:
Black-Box Testing (advanced - Part 1)

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Spring 2018
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
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)
Recap

Equivalence Class Partitioning
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
ECP – Simple Example

```java
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

> = 18
< 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 \([0, 150]\)

\[
\begin{align*}
\text{green area} &= \text{valid} \\
\text{white area} &= \text{invalid}
\end{align*}
\]

\( [18, 150] \)

\( [0, 18) \)

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).
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>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC2</td>
<td>x</td>
<td></td>
<td>x</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>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Input ➞ Output
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>x</td>
<td></td>
</tr>
<tr>
<td>TC3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

EC3 could be split up into several separate ECs; then we would need more TCs.
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)
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]` ➔ EC1
- `accept == false` ➔ not in `[18, 85]` ➔ EC2
- `accept == true/false` ➔ `(80, 85]` ➔ EC3

Var `gender`:
- `accept == true/false` ➔ `{male}` or `{female}` ➔ EC4, EC5

Var `accept`:
- `true or false` ➔ EC7, EC8
Example (cont.)

Input: Gender & Age | Output: accept/reject

**UI – Case A**

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

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: 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>In/Out</th>
<th>Valid EC</th>
<th>Invalid EC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Age</strong></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><strong>Gender</strong></td>
<td>C6: Male</td>
<td>C8: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C7: Female</td>
<td></td>
</tr>
<tr>
<td><strong>Result</strong></td>
<td>C10: ’acc’</td>
<td>C9: &lt;empty&gt;</td>
</tr>
<tr>
<td></td>
<td>C11: ‘reject’</td>
<td></td>
</tr>
<tr>
<td><strong>Message</strong></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 Gend. Result Mess.</td>
<td>em</td>
<td>M</td>
<td>em</td>
<td>mis</td>
<td>oth</td>
<td>em</td>
</tr>
<tr>
<td>Covers</td>
<td>C5</td>
<td></td>
<td>C6</td>
<td>C4</td>
<td>C8</td>
<td>C9</td>
</tr>
<tr>
<td></td>
<td>C10</td>
<td>C1</td>
<td>C6</td>
<td>C11</td>
<td>C12</td>
<td>C13</td>
</tr>
<tr>
<td></td>
<td>C12</td>
<td>C1</td>
<td>C7</td>
<td>C13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Minimum set of tests covering all ECs. All invalid input ECs covered individually.
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>

What if this valid input changes to ‘female’?
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)
Cause-Effect Graphing

1. The tester must decompose the specification into lower level units.

2. For each specification unit, the tester needs to identify causes and effects.
   - A cause is a distinct input condition (or equivalence class of input conditions).
   - An effect is an output condition (or equivalence class of output conditions) or a system transformation.
   - A table of causes and effects helps the tester to record the necessary details.
   - Then relationships between the causes and effects should be determined, ideally in the form of a set of rules (business rules).

3. From the set of rules, the boolean cause-effect graph is constructed, which in turn is converted into a decision table.

4. Test cases are then generated from the decision table.
Cause-Effect Graphing

1. The tester must decompose the specification into lower level units.

2. For each specification unit, the tester needs to identify causes and effects.
   - A cause is a distinct input condition (or equivalence class of input conditions).
   - An effect is an output condition (or equivalence class of output conditions) or a system transformation.
   - A table of causes and effects helps the tester to record the necessary details.
   - Then relationships between the causes and effects should be determined, ideally in the form of a set of rules (business rules).

3. From the set of rules, the boolean cause-effect graph is constructed, which in turn is converted into a decision table.

4. Test cases are then generated from the decision table.

Rules (simplified):

- If below or equal 80 years → accept
- If above 85 years → reject
- If in (80, 85] and ‘female’ → accept
- If in (80, 85] and ‘male’ → reject
Cause-Effect Graphing

1 and/or 3

2

‘3’ occurs if both/one of ‘1’ and ‘2’ are present

2

‘2’ occurs if ‘1’ occurs

1

‘2’ occurs if ‘1’ does not occur
Cause-Effect Graphing

One advantage of this method is that development of the rules and the graph from the specification allows for a thorough inspection (logically, semantically) of the specification.
## Decision Table Format

<table>
<thead>
<tr>
<th>Case</th>
<th>Female</th>
<th>Male</th>
<th>&lt;= 80</th>
<th>&gt;80 &amp; &lt;= 85</th>
<th>&gt; 85</th>
<th>Accept</th>
<th>Reject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rule 2</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Rule 3</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rule 4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Test cases would then be generated from this table, E.g.:

- TC1: male, 83 → reject
- TC2: male, 75 → accept (could also be female)
- TC3: female, 88 → reject (could also be male)
- TC4: female, 83 → accept
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)
Combinatorial Designs

- ECP and BVA define test cases per class
- Combinations of equivalence classes need to be handled
- Combinatorial explosion needs to be handled
Combinatorial Testing – Example 0

2 input variables

Age: 4 radio buttons + empty = 5 values
Gender: 2 radio buttons + empty = 3 values

Minimum coverage of all ECs:
6 test cases

All input combinations?

Result: <text>
Message: <text>

Result text in {<empty>, accept, reject}
Message text in {<empty>, missing input}
Combinatorial Testing – Example 0

2 input variables

Age: 4 radio buttons + empty = 5 values
Gender: 2 radio buttons + empty = 3 values

Minimum coverage of all ECs:
6 test cases

All input combinations?
5 x 3 = 15 test cases

Result text in {<empty>, accept, reject}
Message text in {<empty>, missing input}
How Do We Test This?

34 switches (boolean) => $2^{34} = 1.7 \times 10^{10}$ possible inputs => $1.7 \times 10^{10}$ tests
How Do We Test This?

34 switches = $2^{34}$
= $1.7 \times 10^{10}$ possible inputs
= $1.7 \times 10^{10}$ tests
(34-way interaction)

1-way interactions:
= each switch at least once ‘on’ and once ‘off’
= 2 tests:

0000000000000000000000000000000000
1111111111111111111111111111111111

... 

3-way interactions: need only 33 tests
4-way interactions: need only 85 tests

Problem: How do we know this?

This covers all 68 ECs, assuming two ECs (\{true\} and \{false\}) per variable.
How does it work?

2 switches = 2 boolean variables (‘0’ or ‘1’):

all combinations:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2 x 2 = 4 tests

3 switches:

all combinations:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2³ = 8 tests
How does it work?

2 switches = 2 boolean variables (‘0’ or ‘1’):

- **All combinations**: 2 x 2 = 4 tests

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3 switches:

- **All combinations**: $2^3 = 8$ tests

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3 switches:

- **All 2-way interactions**: $C(3, 2) \times 2^2 = 3 \times 4 = 12$

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X</th>
<th>Z</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
10 Booleans: 2-Way Interactions?

How many tests needed to cover all 2-way interactions?

0 = effect off
1 = effect on

$2^{10} = 1,024$ tests for all 10-way interactions
10 Booleans: 2-Way Interactions?

0 0 1 0 0 1 1 0 1 0
0 1 0 1 0 1 0 1 0 0
1 0 0 1 1 0 0 0 1 1
1 1 0 0 1 1 1 0 0 0
1 1 1 1 0 0 1 0 1 1
0 0 1 0 1 0 0 1 0 1
x x x x x 0 1 1 x 0

0 = effect off
1 = effect on

7 (6?) tests for all 2-way interactions

$2^{10} = 1,024$ tests for all 10-way interactions
10 Booleans: 3-Way Interactions?

How many tests needed to cover all 3-way interactions?

0 = effect off
1 = effect on

\[2^{10} = 1,024\] tests for all 10-way interactions
10 Booleans: 3-Way Interactions?

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

0 = effect off
1 = effect on

13 tests for all 3-way interactions

$2^{10} = 1,024$ tests for all 10-way interactions
10 Booleans: 3-Way Interactions?

All 3-way interactions between the first 3 variables

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

0 = effect off
1 = effect on

13 tests for all 3-way interactions

$2^{10} = 1,024$ tests for all 10-way interactions

MTAT.03.159 / Lecture 03 / © Dietmar Pfahl 2018
Combinatorial Testing

What is it?

- Methods for systematically testing t-way interaction effects of input (or configuration parameter) values.

Why do it?

- The interaction of specific combinations of input values may trigger failures that won’t be triggered if testing input values (or configurations) only in isolation.
Is Testing 2-Way Interactions Enough?

Analyses of failure-triggering conditions showed this:

- Medical device (dark blue)
- NASA distrib. DB (light blue)
- Browser (green)
- Web server (magenta)
- Network security (orange)
- TCAS* module (purple)

* Traffic Collision Avoiding System

Several studies have shown that Pair-wise Testing triggers between 50% and 90% of all failures.

More strengths than 6-way interaction has hardly ever shown to find more defects.
Two Scopes of Combinatorial Testing

Test Configurations

App 1: Pizza Service

<table>
<thead>
<tr>
<th>Test Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
</tr>
<tr>
<td>Sm</td>
</tr>
<tr>
<td>Sm</td>
</tr>
<tr>
<td>Med</td>
</tr>
<tr>
<td>Med</td>
</tr>
<tr>
<td>Lg</td>
</tr>
<tr>
<td>Lg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test case</th>
<th>OS</th>
<th>CPU</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Windows</td>
<td>Intel</td>
<td>IPv4</td>
</tr>
<tr>
<td>2</td>
<td>Windows</td>
<td>AMD</td>
<td>IPv6</td>
</tr>
<tr>
<td>3</td>
<td>Linux</td>
<td>Intel</td>
<td>IPv6</td>
</tr>
<tr>
<td>4</td>
<td>Linux</td>
<td>AMD</td>
<td>IPv4</td>
</tr>
</tbody>
</table>

Pizza Ordering
System under test (SUT)
Combinatorial Testing – Test Inputs

App 2: Travel Service

Many variables
Many values per variable
Need to abstract values (equivalence classes, boundary values)

Plan: flt, flt+hotel, flt+hotel+car
From: CONUS, HI, AK, Europe, Asia …
To: CONUS, HI, AK, Europe, Asia …
Compare: yes, no
Date-type: exact, 1to3, flex
Depart: today, tomorrow, 1yr, Sun, Mon …
Return: today, tomorrow, 1yr, Sun, Mon …
Adults: 1, 2, 3, 4, 5, 6
Minors: 0, 1, 2, 3, 4, 5
Seniors: 0, 1, 2, 3, 4, 5
Combinatorial Testing – Test Config

Platform configuration parameters:

- OS: Windows XP, Apple OS X, Red Hat Linux
- Browser: Internet Explorer, Firefox
- Protocol: IPv4, IPv6
- CPU: Intel, AMD
- DBMS: MySQL, Sybase, Oracle

Total number of combinations: $3 \times 2 \times 2 \times 2 \times 3 = 72$

Do we really need to test all 72 combinations?
Pair-Wise Testing (2-Way Interaction)

# of pairs:

\[
\binom{5}{2} = \frac{5!}{2!(5-2)!} = 10
\]

# of 2-way interactions is ?:

\[
40 = \binom{5}{2} 2^2 \leq N \leq \binom{5}{2} 3^2 = 90
\]

- a) OS [3]: Windows XP, Apple OS X, Red Hat Linux
- b) Browser [2]: Internet Explorer, Firefox
- c) Protocol [2]: IPv4, IPv6
- d) CPU [2]: Intel, AMD
- e) DBMS [3]: MySQL, Sybase, Oracle
Pair-Wise Testing (2-Way Interaction)

# of pairs:

\[ \binom{5}{2} = \frac{5!}{2!(5-2)!} = 10 \]

# of 2-way interactions is N=57:

\[ 40 = \binom{5}{2} 2^2 \leq N \leq \binom{5}{2} 3^2 = 90 \]

a:b 3 x 2 = 6
a:c 3 x 2 = 6
a:d 3 x 2 = 6
a:e 3 x 3 = 9
b:c 2 x 2 = 4
b:d 2 x 2 = 4
b:e 2 x 3 = 6
c:d 2 x 2 = 4
c:e 2 x 3 = 6
d:e 2 x 3 = 6

\[ 57 \]

a) OS [3]: Windows XP, Apple OS X, Red Hat Linux
b) Browser [2]: Internet Explorer, Firefox
c) Protocol [2]: IPv4, IPv6
d) CPU [2]: Intel, AMD
e) DBMS [3]: MySQL, Sybase, Oracle
Pair-Wise Testing (2-Way Interaction)

# of pairs:
\[ \binom{5}{2} = \frac{5!}{2!(5-2)!} = 10 \]

# of 2-way interactions is N=57:
\[ 40 = \binom{5}{2} 2^2 \leq N \leq \binom{5}{2} 3^2 = 90 \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a:b</td>
<td>3 x 2 = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a:c</td>
<td>3 x 2 = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a:d</td>
<td>3 x 2 = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a:e</td>
<td>3 x 3 = 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b:c</td>
<td>2 x 2 = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b:d</td>
<td>2 x 2 = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b:e</td>
<td>2 x 3 = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c:d</td>
<td>2 x 2 = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c:e</td>
<td>2 x 3 = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d:e</td>
<td>2 x 3 = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the lower bound for the number of test cases?

- a) OS [3]: Windows XP, Apple OS X, Red Hat Linux
- b) Browser [2]: Internet Explorer, Firefox
- c) Protocol [2]: IPv4, IPv6
- d) CPU [2]: Intel, AMD
- e) DBMS [3]: MySQL, Sybase, Oracle
Pair-Wise Testing (2-Way Interaction)

Only 9 tests needed, if we want to test all interactions of one parameter with one other parameter (pair-wise interaction)

Pair (a, e): 3 x 3 = 9 Combinations

<table>
<thead>
<tr>
<th>Test</th>
<th>OS (a)</th>
<th>Browser (b)</th>
<th>Protocol (c)</th>
<th>CPU (d)</th>
<th>DBMS (e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
</tbody>
</table>

NOTE: In this case, the lower bound is the same as what is actually needed for covering all 2-way interactions. But that is just a coincidence!
Pair-Wise Testing (2-Way Interaction)

Only 9 tests needed, if we want to test all interactions of one parameter with one other parameter (pair-wise interaction).

<table>
<thead>
<tr>
<th>Test</th>
<th>OS (a)</th>
<th>Browser (b)</th>
<th>Protocol (c)</th>
<th>CPU (d)</th>
<th>DBMS (e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
</tbody>
</table>

Pair (a, b):
3 x 2 = 6 Combinations
Pair-Wise Testing (2-Way Interaction)

Only 9 tests needed, if we want to test all interactions of one parameter with one other parameter (pair-wise interaction).

Pair (b, c):

<table>
<thead>
<tr>
<th>Test</th>
<th>OS (a)</th>
<th>Browser (b)</th>
<th>Protocol (c)</th>
<th>CPU (d)</th>
<th>DBMS (e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
</tbody>
</table>

2 x 2 = 4 Combinations
ACTS Tool (NIST & UT Arlington)
ACTS – Example 1

3 variables: VAR1, VAR2, VAR3
2 values per variable: 0, 1

⇒

4 TCs
ACTS – Example 2

<table>
<thead>
<tr>
<th>VAR1</th>
<th>VAR2</th>
<th>VAR3</th>
<th>VAR4</th>
<th>VAR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

5 variables: VAR1 … VAR 5
3 values per variable: 0, 1, 2
⇒ 15 TCs
ACTS – Example 3

5 variables
2 or 3 values per variable
⇒ 9 TCs
Defining Variables of SUT

12 variables
Two to many values per variable
Variable Interaction Strength

Define interaction strength for all variables or for subsets of variables.
Output: Covering Array

Variables and their values
Combinatorial Testing Links

http://cse.unl.edu/~citportal/

http://csrc.nist.gov/groups/SNS/acts/index.html
How to (Automatically) Generate Test Oracles?

Creating test (input) data is the (relatively) easy part!

How do we check that the code worked correctly on the test input?

• **Using existing test** sets (with known oracles) – easy if test sets exist

• **Crash testing** code to ensure it does not crash for randomly generated test input (‘fuzz testing’) – easy but of limited value

• **Embedded assertions** – incorporate assertions in code to check critical states at different points in the code – will they cover all possible incorrect states?

• **Model-checking** using a mathematical model of the system under test and a model checker to generate expected results for each input – expensive but tractable
Combinatorial Testing – Summary

Combinatorial testing makes sense where

More than ~8 variables and less than 300-400
Logical or numeric interaction of variables

New algorithms make large-scale combinatorial testing possible

Tool-support exists (research prototypes)
Model-checking facilitates automatic test oracle generation
Recommended Textbook Exercises

Chapter 4
1, 2, 3, 4, 5
8, 11, 12
Lab 3: Combinatorial Testing

Task 1

1. Create system of Booking app in ACTS tool
2. Generate covering arrays for all strengths
3. Run Booking app with test data and generate output files
4. Inspect output for crashes and other failures and create tables and graphs with data for each run (strengths 1 to n – using 2 algorithms)
Lab 3: Combinatorial Testing

Task 2

1. Create system of Booking app in ACTS tool; setting parameters such that covering arrays have not more than 250 tests; use information from Task 1 as well as the given rules
2. Generate covering arrays for all strengths
3. Run Booking app with test data and generate output files
4. Inspect output and count unique failures

Repeat until maximum amount of unique failures found.
Lab 3: Combinatorial Testing

Task 3

1. Analyze the output from Task 1 and try to find failures that are not reported through error messages from the application

   - look only at the output generated with maximum strength interaction (full combinatorial testing)
   - Use the rule set to spot incorrect outputs
Next Weeks

• Lab 3:
  – Combinatorial Testing

• Lecture 4:
  – White-Box Testing Techniques (advanced)

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