Black-Box Testing Techniques I

Software Testing and Verification

Lecture 4

Prepared by
Stephen M. Thebaut, Ph.D.

University of Florida
Definition of Black-Box Testing

- Testing based solely on analysis of requirements (specification, user documentation, etc.).
- Also know as *functional* testing.
- Black-box testing concerns techniques for *designing* tests; it is *not* a “level” of testing.
- Black-box techniques apply to *all* levels of testing (e.g., unit, component, product, and system).
Design Techniques Considered

- Partition testing
- Combinatorial Approaches
- Boundary Value Analysis
- Intuition & Experience
Partition Testing

- Also known as *input space partitioning* and *equivalence partitioning*.
- Can be thought of as “exhaustive testing *Las Vegas style*”...
- Idea is to *partition* the program’s *input space* based on a (small) number of *equivalence classes* such that, *according to the specification*, every element of a given partition would be “handled (e.g., mapped to an output) in the same manner.”
Partition Testing (cont’d)

• **If** the program happens to be implemented in such a way that being “handled in the same manner” means that either
  - every element of a partition is mapped to a correct output, or
  - every element of a partition is mapped to an incorrect output,

  **then** testing the program with just one element from each partition would be tantamount to *exhaustive testing*. 
Partition Testing (cont’d)

- Two types of equivalence classes are identified: *valid* and *invalid*.
- *Valid* classes correspond to inputs deemed nominal from the specification.
- *Invalid* classes correspond to inputs deemed erroneous from the specification.
Partition Testing Example

• Program Specification:

An ordered pair of numbers, \((x,y)\), is input and a message is output stating whether the numbers are in ascending order, descending order, or equal. If the input is other than an ordered pair of numbers, an error message is output.
Partition Testing Example (cont’d)

- Equivalence Classes:
  \[ \{ (x,y) \mid x < y \} \ (V) \]
  \[ \{ (x,y) \mid x > y \} \ (V) \quad \text{Valid classes} \]
  \[ \{ (x,y) \mid x = y \} \ (V) \]
Partition Testing Example (cont’d)

• Equivalence Classes:
  \{ (x,y) \mid x<y \} \ (V)
  \{ (x,y) \mid x>y \} \ (V) \quad \text{Valid classes}
  \{ (x,y) \mid x=y \} \ (V)
  \{ \text{input is other than an ordered pair of numbers} \} \ (I) \quad \text{Invalid class}
Valid \((x, y)\) Input Space
In this example, the specification implies that each input would be “handled” (i.e., mapped to an output) according to the numeric relationship between x and y, so equivalence classes are naturally defined in terms of that relationship.
Sample Program Design

- Conceptually, would the underlying assumption of partition testing hold for these classes if the following program design was employed?
Sample Program Design

- Conceptually, would the underlying assumption of partition testing hold for these classes if the following program design was employed?

```plaintext
if (input is other than an ordered pair of numbers)
    then output(“invalid input”)
else
    if x<y then output(“ascending order”)
    else
        if x>y then output(“ascending order”)
        else
            output(“equal”)
```
Identifying and Representing Test Cases

- Test case design requirements are often represented (documented) using a “test case COVERAGE MATRIX.”
- Columns in the matrix represent templates for test case inputs (and in some cases expected results).
- Rows represent the design rationale for each test case.
A Test Case Coverage Matrix

<table>
<thead>
<tr>
<th>EQUIVALENCE CLASSES</th>
<th>TEST CASES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>{ (x,y) \mid x&gt;y } (V)</td>
<td>V</td>
</tr>
<tr>
<td>{ (x,y) \mid x&lt;y } (V)</td>
<td></td>
</tr>
<tr>
<td>{ (x,y) \mid x=y } (V)</td>
<td></td>
</tr>
<tr>
<td>{ other } (I)</td>
<td></td>
</tr>
</tbody>
</table>

† Rule of thumb: include test cases that cover invalid classes one at a time to avoid “masking of errors.”
Dealing with More Complex Multiple-Input Program Specifications

- In the example above, \((x,y)\) input pairs were considered as a unit, yielding a set of disjoint classes partitioning the two-dimensional \(x,y\) (valid) input space.

- For more complex multiple-input program specifications, classes are often identified according to the attributes of INDIVIDUAL inputs (a.k.a. “input conditions”), and these sometimes overlap. For example...
Part of a More Complex Program Specification:

Three numbers, x, y, and z, are input. If x is a whole number and less than 40, and if y is non-negative, the output is \( z + \frac{y}{x} \). If x is greater than or equal to 40, or if y is positive, or if z is odd and at least as large as x, then the output is...
Dealing with More Complex Multiple-Input Program Specifications (cont’d)

- Some (valid) input attribute classes:
  - \{ x \mid x \text{ is a whole number} \} (V)
  - \{ x \mid x < 40 \} (V), \{ x \mid x \geq 40 \} (V)
  - \{ y \mid y = 0 \} (V), \{ y \mid y > 0 \} (V)
  - \{ z \mid z \text{ is odd} \} (V)
  - \{ (x, z) \mid z \geq x \} (V)
  
  ...
Dealing with More Complex Multiple-Input Program Specifications (cont’d)

- In such cases, COMBINATIONS of input conditions (attribute classes) that partition the input space must be inferred from the specification.

- In partition testing terms, this amounts to identifying a potentially complex, multidimensional “equivalence relation”.

- Cause-Effect Analysis, which we consider later, facilitates this by explicitly modeling the relationships between input “Causes” and output “Effects”.
Some Simple Heuristics for Identifying Input Attribute Classes

• “Must Be” Situations
  - “First character must be a letter.”
  - Identify one valid and one invalid class:
    \{1st char letter \} (V),
    \{1st char not letter\} (I)
Some Simple Heuristics for Identifying Input Attribute Classes (cont’d)

• “Range of Values” Situations
  – “Input HOURS will range in value from 0 to 40.”
  – Identify one valid and two invalid classes:
    \[ \{ \text{HOURS} \in [0,40] \} \ (V), \]
    \[ \{ \text{HOURS} < 0 \} \ (I), \]
    \[ \{ \text{HOURS} > 40 \} \ (I) \]
Some Simple Heuristics for Identifying Input Attribute Classes (cont’d)

• “Possible Differences” Situations
  – “For HOURS ≤ 20, output ‘Low’; for HOURS > 20, output ‘HIGH’.”
  – If the specification suggests that values in a class may be handled differently, divide the class accordingly. \{ \text{HOURS} \in [0,40] \} (V) becomes:

    \{ \text{HOURS} \in [0,20] \} (V),
    \{ \text{HOURS} \in (20,40] \} (V)
Another Relatively Simple (but Multiple-Input) Partition Testing Example

- Identify disjoint sets of classes for each input variable associated with the following program specification fragment.
- You may detect some “incompleteness” problems with the specification...
The first input is a yes/no response to the question “Do you reside within the city?” The second input is gross pay for the year in question.

A non-resident will pay 1% of the gross pay in city tax.

Residents pay on the following scale:
- If gross pay is no more than $30,000, the tax is 1%.
- If gross pay is more than $30,000, but no more than $50,000, the tax is 5%.
- If gross pay is more than $50,000, the tax is 15%.
City Tax Specification 1

The first input is a yes/no response to the question “Do you reside within the city?” The second input is gross pay for the year in question.

A non-resident will pay 1% of the gross pay in city tax.

Residents pay on the following scale:
- If gross pay is no more than $30,000, the tax is 1%.
- If gross pay is more than $30,000, but no more than $50,000, the tax is 5%.
- If gross pay is more than $50,000, the tax is 15%.
(Valid) Classes for City Tax Specification 1

- Res?
- Gross_Pay
(Valid) Classes for City Tax Specification 1

- **Res?**
  - { yes } (V)
  - { no } (V)
  - { other } (I)

- **Gross_Pay**
(Valid) Classes for City Tax Specification 1

- **Res?**
  - { yes } (V)
  - { no } (V)
  - { other } (I)

- **Gross_Pay**
  - [0, 30K] (V)
  - (30K, 50K] (V)
  - (50K, MAX] (V)
  - < 0 (I)
  - > MAX (I)
(Valid) Classes for City Tax Specification 1

- **Res?**
  - { yes } (V)
  - { no } (V)
  - { other } (I)

- **Gross_Pay**
  - [0, 30K] (V)
  - (30K, 50K] (V)
  - (50K, MAX] (V)
  - < 0 (I)
  - > MAX (I)

*Note that program behaviors associated with invalid inputs are not specified!*
### Two-Dimensional (Valid) Input Space

<table>
<thead>
<tr>
<th>Res?</th>
<th>( \leq 30K )</th>
<th>((30K, 50K])</th>
<th>( &gt; 50K )</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

What *equivalence relation* should be used to partition this input space for testing purposes?

Recall the idea: *according to the specification*, every element of a given partition would be “handled (e.g., mapped to an output) in the same manner.”
Some Possible *Equivalence Relations*

1. Partition the input space elements according to the 3 outputs to which they would be mapped.

2. Partition elements on the conservative assumption that each of the 6 combinations of the classes for the 2 inputs could be “handled” (i.e., mapped to outputs) differently.

3. Partition elements in accordance with the logic used to explain (but presumably not to dictate implementation requirements for) the mapping of classes or combinations of classes to outputs.†

† This approach is based on the availability of an *operational specification*, as explained later.
Consider option (1): If we partition the space comprised of these classes *based solely on the three specified outputs*, what would the result be?

Two-Dimensional (Valid) Input Space

<table>
<thead>
<tr>
<th>Gross_Pay</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Would you be comfortable with the degree of coverage afforded by choosing ONE test case for each of these 3 partitions? Why or why not?
Partitioning Based on Specified Output

**Gross_Pay**

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that there are FOUR possible ways to cover these 3 partitions!

Would you be comfortable with the degree of coverage afforded by choosing ONE test case for each of these 3 partitions? Why or why not?
Partitioning Based on Specified Output

**Gross_Pay**

<table>
<thead>
<tr>
<th>Res?</th>
<th>$\leq 30K$</th>
<th>$(30K, 50K]$</th>
<th>$&gt; 50K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>★ 1%</td>
<td>★ 5%</td>
<td>★ 15%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*(Does not cover Res? “no” class.)*

Would you be comfortable with the degree of coverage afforded by choosing ONE test case for each of these 3 partitions? Why or why not?
Partitioning Based on Specified Output

**Gross_Pay**

<table>
<thead>
<tr>
<th></th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*(Does not cover Gross_Pay? “≤ 30K” class.)*

Would you be comfortable with the degree of coverage afforded by choosing ONE test case for each of these 3 partitions? Why or why not?
Would you be comfortable with the degree of coverage afforded by choosing ONE test case for each of these 3 partitions? Why or why not?
Partitioning Based on Specified Output

Gross_Pay

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>☆ 5%</td>
<td>☆ 15%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Covers ALL attribute classes for each input, but NOT all combinations of attribute classes.)

Would you be comfortable with the degree of coverage afforded by choosing ONE test case for each of these 3 partitions? Why or why not?
**Two-Dimensional (Valid) Input Space**

<table>
<thead>
<tr>
<th>Res?</th>
<th>Gross_Pay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\leq 30K$</td>
</tr>
<tr>
<td>yes</td>
<td>1%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
</tr>
</tbody>
</table>

Consider option (2): Here we “hedge our bet” by assuming that each of the 6 combinations of the classes may be “handled” (i.e., mapped to specified outputs) differently...
Partitioning Based on the Conservative “All Class Combinations” Assumption

<table>
<thead>
<tr>
<th>Res?</th>
<th>Gross_Pay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>≤ 30K</td>
</tr>
<tr>
<td>yes</td>
<td>1%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
</tr>
</tbody>
</table>
Partitioning Based on the Conservative “All Class Combinations” Assumption

**Gross_Pay**

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

*(Covers ALL COMBINATIONS of attribute classes.)*

What are the pros and cons of this approach?
A brief divagation...

• Let’s pause for a moment to introduce some strength-of-coverage terminology associated with partition testing when the input space is multi-dimensional...
“Strong and Weak Equivalence Class Testing”

• The (“brute-force”) approach of specifying a test case for each (feasible) combination of classes (i.e., for each element in the Cartesian product of the classes associated with each input) is sometimes referred to as “Strong Equivalence Class Testing.”

• In contrast, “Weak Equivalence Class Testing” requires only that all classes be covered. (The minimum number of cases needed will always be the largest number of disjoint classes associated with any single input.)
Examples of “Strong and Weak Equivalence Class Testing”

<table>
<thead>
<tr>
<th>Strong: Gross_Pay</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res? yes</td>
<td>★ 1%</td>
<td>★ 5%</td>
<td>★ 15%</td>
</tr>
<tr>
<td>Res? no</td>
<td>★ 1%</td>
<td>★ 1%</td>
<td>★ 1%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Weak: Gross_Pay</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res? yes</td>
<td>★ 1%</td>
<td>5%</td>
<td>★ 15%</td>
</tr>
<tr>
<td>Res? no</td>
<td>1%</td>
<td>★ 1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
A brief divagation... (cont’d)

- Note that partition testing based on the conservative assumption that all input class combinations could be “handled” (i.e., mapped to outputs) differently (Option 2) results in strong equivalence class testing.

- But as previously demonstrated, partition testing based on specified output (Option 1) will not necessarily result in either strong or weak equivalence class testing.
Consider option (3): If we partition the space in accordance with the logic used to explain the mapping of input classes to outputs, what would the result be?
Partitioning Based on Operational Specification

- *Operational specifications* elucidate program requirements via procedural or logical explanations of *how those requirements could be satisfied*.

- However, the explanations are *not* intended to imply that any particular design or implementation approach must be used by developers.

- Thus, option (3) amounts to partitioning input space elements in accordance with a non-binding, explanatory description of how classes *could* be mapped to outputs.

- Recall from our specification...
Described Mapping of Inputs to Outputs

A non-resident will pay 1% of the gross pay in city tax.

Residents pay on the following scale:
- If gross pay is no more than $30,000, the tax is 1%.
- If gross pay is more than $30,000, but no more than $50,000, the tax is 5%.
- If gross pay is more than $50,000, the tax is 15%.
A non-resident will pay 1% of the gross pay in city tax.

Residents pay on the following scale:
- If gross pay is no more than $30,000, the tax is 1%.
- If gross pay is more than $30,000, but no more than $50,000, the tax is 5%.
- If gross pay is more than $50,000, the tax is 15%.
### Partitioning Based on Operational Specification

#### Gross_Pay

<table>
<thead>
<tr>
<th></th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Res?
### Partitioning Based on Operational Specification

#### Gross_Pay

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
</tbody>
</table>
A non-resident will pay 1% of the gross pay in city tax.

Residents pay on the following scale:
- If gross pay is no more than $30,000, the tax is 1%.
- If gross pay is more than $30,000, but no more than $50,000, the tax is 5%.
- If gross pay is more than $50,000, the tax is 15%.
## Partitioning Based on Operational Specification

### Gross_Pay

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td>1%</td>
<td></td>
</tr>
</tbody>
</table>
### Partitioning Based on Operational Specification

#### Gross Pay

<table>
<thead>
<tr>
<th>Res?</th>
<th>( \leq 30K )</th>
<th>((30K, 50K])</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implied Mapping of Inputs to Outputs

A non-resident will pay 1% of the gross pay in city tax.

Residents pay on the following scale:
- If gross pay is no more than $30,000, the tax is 1%.
- If gross pay is more than $30,000, but no more than $50,000, the tax is 5%.
- If gross pay is more than $50,000, the tax is 15%.
### Partitioning Based on Operational Specification

#### Gross\_Pay

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partitioning Based on Operational Specification

**Res?**

<table>
<thead>
<tr>
<th>Gross_Pay</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
</tbody>
</table>
Implied Mapping of Inputs to Outputs

A non-resident will pay 1% of the gross pay in city tax.

Residents pay on the following scale:
- If gross pay is no more than $30,000, the tax is 1%.
- If gross pay is more than $30,000, but no more than $50,000, the tax is 5%.
- If gross pay is more than $50,000, the tax is 15%.
### Partitioning Based on Operational Specification

<table>
<thead>
<tr>
<th>Gross_Pay</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Res?</strong></td>
<td>yes</td>
<td>1%</td>
<td>5%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partitioning Based on Operational Specification

**Gross_Pay**

<table>
<thead>
<tr>
<th>Res?</th>
<th>( \leq 30K )</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partitioning Based on Operational Specification

**Gross_Pay**

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>★ 1%</td>
<td>★ 5%</td>
<td>★ 15%</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
<td>★ 1%</td>
</tr>
</tbody>
</table>

What are the pros and cons of this approach?
Partitioning Based on Operational Specification

- Consider a modification of City Tax Specification 1 that offers an alternative explanation of how input classes could be mapped to outputs...

City Tax Specification 2

...If gross pay is no more than $30,000, the tax is 1%. If gross pay is more than $30,000, but no more than $50,000 the tax is 1% for non-residents and 5% for residents. If gross pay is more than $50,000, the tax is 1% for nonresidents and 15% for residents.
Partitioning Based on Operational Specification

• Consider a modification of City Tax Specification 1 that offers an alternative explanation of how input classes could be mapped to outputs...

City Tax Specification 2

...If gross pay is no more than $30,000, the tax is 1%.

If gross pay is more than $30,000, but no more than $50,000 the tax is 1% for non-residents and 5% for residents.

If gross pay is more than $50,000, the tax is 1% for nonresidents and 15% for residents.
## Partitioning Based on Operational Specification

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Partitioning Based on Operational Specification

- Consider a modification of City Tax Specification 1 that offers an alternative explanation of how input classes could be mapped to outputs...

City Tax Specification 2

...If gross pay is no more than $30,000, the tax is 1%.
If gross pay is more than $30,000, but no more than $50,000 the tax is 1% for non-residents and 5% for residents.
If gross pay is more than $50,000, the tax is 1% for nonresidents and 15% for residents.
### Partitioning Based on Operational Specification

**Gross_Pay**

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Partitioning Based on Operational Specification

- Consider a modification of City Tax Specification 1 that offers an alternative explanation of how input classes could be mapped to outputs...

City Tax Specification 2

...If gross pay is no more than $30,000, the tax is 1%.

If gross pay is more than $30,000, but no more than $50,000 the tax is 1% for non-residents and 5% for residents.

If gross pay is more than $50,000, the tax is 1% for nonresidents and 15% for residents.
### Partitioning Based on Operational Specification

#### Gross_Pay

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Partitioning Based on Operational Specification

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

**Gross_Pay**
Comparing the results for Specifications 1 & 2

**Gross_Pay**

<table>
<thead>
<tr>
<th></th>
<th>≤ 30K</th>
<th>(30K, 50K]</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res? yes</td>
<td>⭐ 1%</td>
<td>⭐ 5%</td>
<td>⭐ 15%</td>
</tr>
<tr>
<td>Res? no</td>
<td>⭐ 1%</td>
<td>⭐ 1%</td>
<td>⭐ 1%</td>
</tr>
</tbody>
</table>
Summarizing all four partitionings

Specified output:

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K)</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Spec. #1 I/O mapping:

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K)</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td></td>
</tr>
</tbody>
</table>

Spec. #2 I/O mapping:

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K)</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

All class combinations:

<table>
<thead>
<tr>
<th>Res?</th>
<th>≤ 30K</th>
<th>(30K, 50K)</th>
<th>&gt; 50K</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>no</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Some conclusions

• In general, the connection between partition testing and exhaustive testing is tenuous, at best. *(Are you Feeling Lucky?)*

• Partitioning a multi-dimensional input space based solely on differences in specified output behavior can yield relatively weak black-box coverage.

• The conservative approach of associating a separate partition with every (feasible) combination of classes yields relatively strong black-box coverage, but may be impractical when the number of inputs and associated sets of classes is large.
Some conclusions (cont’d)

• Partitioning an input space in accordance with an explanation of how inputs map to outputs seems in keeping with:

  ...according to the specification, every element of a given partition would be “handled” (e.g., mapped to an output) in the same manner"

but as previously noted, such descriptions are not normally intended to dictate HOW a program should be designed. Rather, they are more often used to merely elucidate WHAT is required.
Some conclusions (cont’d)

• The “brute-force” coverage strategy of specifying a test case for each (feasible) combination of classes is sometimes referred to as “Strong Equivalence Class Testing.”

• In contrast, “Weak Equivalence Class Testing” only requires that all classes be covered - not all (feasible) combinations of classes.
Coming Up Next...

- The approach considered next, *Cause-Effect Analysis*, extends the idea of partitioning a multi-dimensional input space by providing a systematic means for generating test case templates to cover different combinations of input “Causes” resulting in output “Effects.”