Black-Box Testing Techniques

Partition Testing

- Also known as *input space partitioning* and *equivalence partitioning*.

- 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 is "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*** (corresponding to inputs deemed valid from the specification) and ***invalid*** (corresponding to inputs deemed erroneous from the specification)

Partition Testing (cont'd)

- However, it is sometimes possible and convenient to identify *multivariate classes* which partition an input space directly.
- Consider the following specification…

Partition Testing Example

- Program Specification:

  An ordered pair of numbers, (x, y), are input and a message is output stating whether they 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) | x<y } (V)

  { (x, y) | x>y } (V)

  { (x, y) | x=y } (V)

  { input is other than an ordered pair of numbers } (I)

Valid classes

Invalid class

Sample Program Design

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

```
 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

EQUIVALENCE CLASSES	TEST CASES			
	1	2	3	4
{ (x, y) | x>y } (V)	V			
{ (x, y) | x<y } (V)		V		
{ (x, y) | x=y } (V)			V	
{ other } (I)				I

Dealing with Complex Multiple-Input Situations (cont'd)

- the most critical issue in partition testing is the choice of an "equivalence relation" that defines the classes used.

Some Simple Heuristics for Identifying Equivalence 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 Equivalence Classes (cont'd)

- "Range of Values" Situations
  - "Input HOURS will range in value from 0 to 40."
  - Identify one valid and two invalid classes:

    { HOURS $\in$ [0,40] } (V),
    { HOURS < 0 } (I),
    { HOURS > 40 } (I)

Some Simple Heuristics for Identifying Equivalence Classes (cont'd)

- "Possible Differences" Situations
  - "For HOURS $\leq$ 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. { HOURS $\in$ [0,40] } (V) becomes:

    { HOURS $\in$ [0,20]} (V),

    { HOURS $\in$ (20,40] } (V)

Another 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…

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%.

Equivalence Classes for City Tax Specification 1:

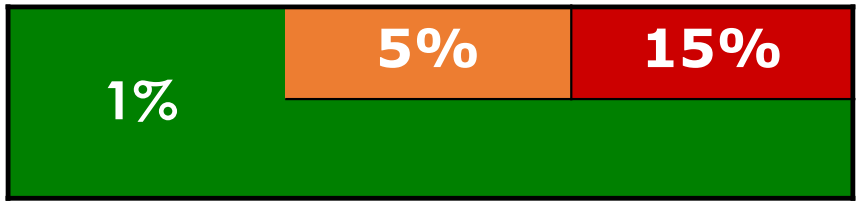
- ***Res?***
  - { yes } (V)
  - { no } (V)
  - { other } (I)
- ***Gross_Pay***
  - [0, 30K] (V)
  - (30K,50K] (V)
  - (50K, MAX] (V)
  - < 0 (I)
  - > MAX (I)

Two-Dimensional (Valid) Input Space

		Gross_Pay		
		≤ 30K	(30K, 50K]	> 50K
Res?	yes	1%	5%	15%
	no	1%	1%	1%

If we partition the space comprised of these classes based solely on the specified outputs, what would the result be?

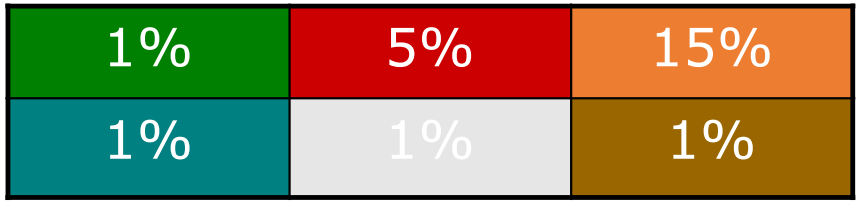
Partitioning Based on Specified Output



Would you be comfortable with the degree of coverage afforded by choosing ONE test case from each of these 3 partitions? Why or why not?

The “Brute-Force” Approach

We could “hedge our bet” by associating a sep-arate partition with ***every (feasible) combi-nation*** of classes from the sets:



The “Brute-Force” Approach (cont’d)

- 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 variable) is sometimes referred to as *“Strong Equivalence Class Testing.”*
- The term *“Weak Equivalence Class Testing”* refers to specifying the *minimum* number of test cases required to cover all classes. (This will always be the largest number of disjoint classes associated with a single variable.)

Some conclusions

- In general, the connection between partition testing and exhaustive testing is tenuous, at best.
- Partitioning a complex, multi-dimensional input space based solely on differences in specified output behavior can be risky.

Some conclusions (cont'd)

- The "brute-force" approach of associating a separate partition with every (feasible) combination of classes from the sets provides excellent black-box coverage, but is impractical when the number of inputs and associated sets of classes is large.

Some conclusions (cont'd)

- Choosing an equivalence relation by "second-guessing" the most likely implementation based on how the specification is written is more in keeping with the idea that, *according to the specification,* every element of a given partition would be "handled" (e.g., mapped to an output) "in the same manner."