



# CSI3105: Software Testing

## Module 4a: Domain Partitioning Introduction

Creative  
thinkers  
made here.

# Learning Objectives

- Equivalence class partitioning
  - Boundary value analysis
- } Essential black-box techniques for generating tests for **functional** testing.

# Learning Objectives

- Video 4b
  - Define Functional testing and test plans
- Video 4c
  - The test selection problem and exhaustive testing
- Video 4d
  - Equivalence class testing
- Video 4e
  - Boundary value analysis

# Learning Objectives

- Equivalence class partitioning
  - Boundary value analysis
- }
- Essential black-box techniques for generating tests for **functional** testing.

# Learning Objectives

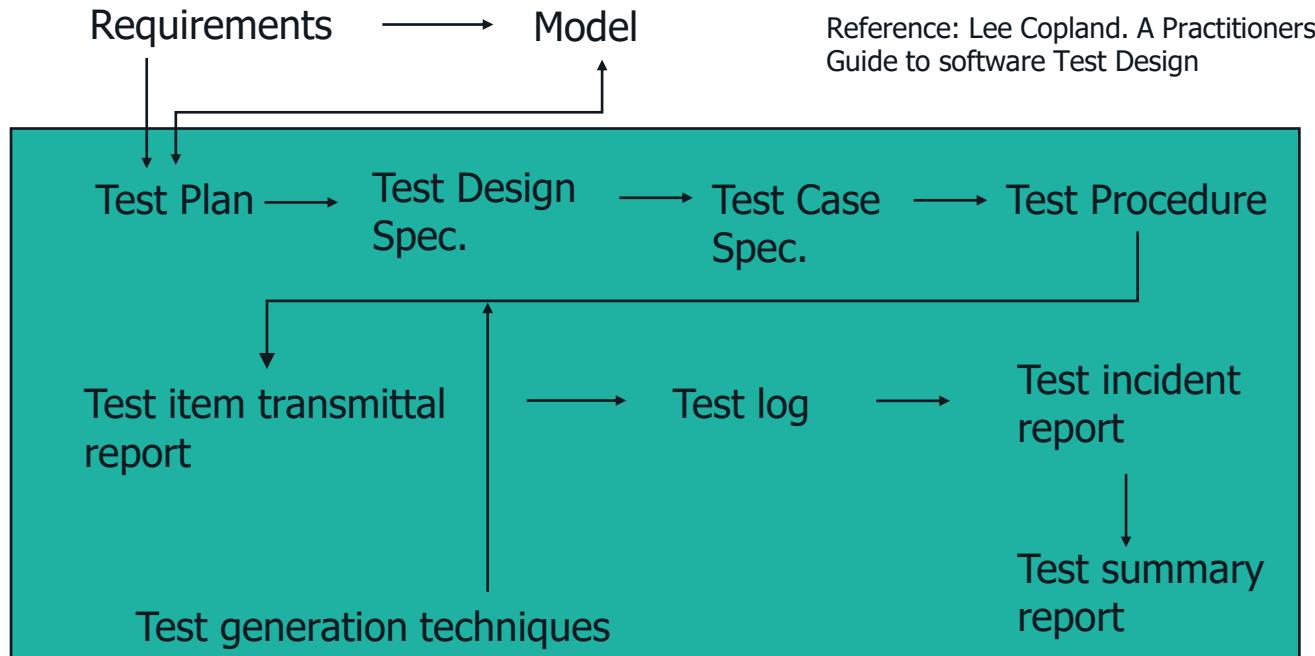
- Video 4b
  - Define Functional testing and test plans
- Video 4c
  - The test selection problem and exhaustive testing
- Video 4d
  - Equivalence class testing
- Video 4e
  - Boundary value analysis

# Applications of test generation techniques

Test generation techniques described in this chapter belong to the **black-box** testing category.

These techniques are useful during **functional testing** where the objective is to test whether or not an application, unit, system, or subsystem, correctly implements the **functionality** as per the given **requirements**

# Functional Testing: Test Documents (IEEE829 Standard)



# Functional Testing: Documents



**Test Plan:** Describe scope, approach, resources, test schedule, items to be tested, deliverables, responsibilities, approvals needed. Could be used at the system test level or at lower levels.

**Test design spec:** Identifies a subset of features to be tested and identifies the test cases to test the features in this subset.

**Test case spec:** Lists inputs, expected outputs, features to be tested by this test case, and any other special requirements e.g. setting of environment variables and test procedures. Dependencies with other test cases are specified here. Each test case has a unique ID for reference in other documents.

# Functional Testing: Documents (contd)



**Test procedure spec:** Describe the procedure for executing a test case.

**Test transmittal report:** Identifies the test items being provided for testing, e.g. a database.

**Test log:** A log observations during the execution of a test.

**Test incident report:** Document any special event that is recommended for further investigation.

**Test summary:** Summarize the results of testing activities and provide an evaluation.

# Test generation techniques in this chapter

Two techniques are considered: equivalence partitioning and boundary value analysis

Each of these test generation techniques is black-box and useful for generating test cases during functional testing.

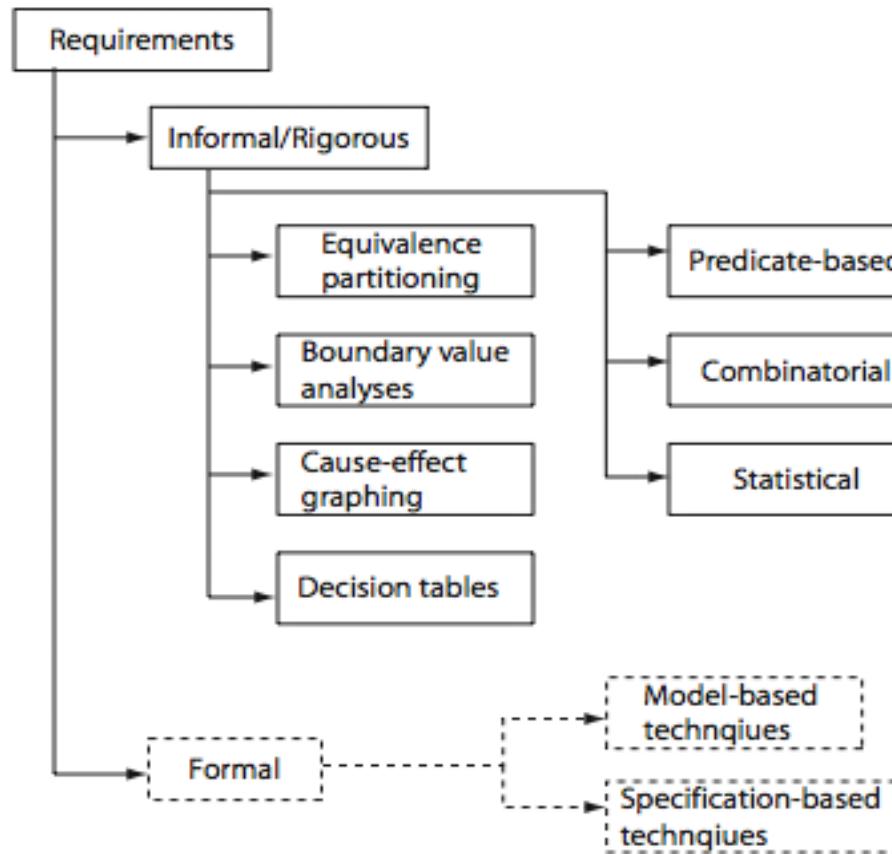
# Requirements and test generation

Requirements serve as the starting point for the generation of tests. During the initial phases of development, requirements may exist only in the minds of one or more people.

These requirements, more aptly ideas, are then specified rigorously using modeling elements such as use cases, sequence diagrams, and statecharts in UML.

Rigorously specified requirements are often transformed into formal requirements using requirements specification languages such as Z, S, and RSML.

# Test generation techniques



# Test selection problem

Let  $D$  denote the input domain of a program  $P$ . The test selection problem is to select a subset  $T$  of tests such that execution of  $P$  against each element of  $T$  will reveal all errors in  $P$ .

In general there does not exist any algorithm to construct such a test set. However, there are heuristics and model based methods that can be used to generate tests that will reveal certain type of faults.

# Test selection problem (contd.)

The challenge is to construct a test set  $T \subseteq D$  that will reveal as many errors in  $P$  as possible. The problem of test selection is difficult due primarily to the size and complexity of the input domain of  $P$ .

# Exhaustive testing

The large size of the input domain prevents a tester from exhaustively testing the program under test against all possible inputs. By ``exhaustive'' testing we mean testing the given program against every element in its input domain.

The complexity makes it harder to select individual tests.

# Large input domain

Consider program P that is required to sort a sequence of integers into ascending order. Assuming that P will be executed on a machine in which integers range from -32768 to 32767, the input domain of P consists of all possible sequences of integers in the range [-32768, 32767]. For a sequence of 3 ints =  $65536^3$

If there is no limit on the size of the sequence that can be input, then the input domain of P is infinitely large and P can never be tested exhaustively. If the size of the input sequence is limited to, say  $N_{\max} > 1$ , then the size of the input domain depends on the value of N.

# Complex input domain

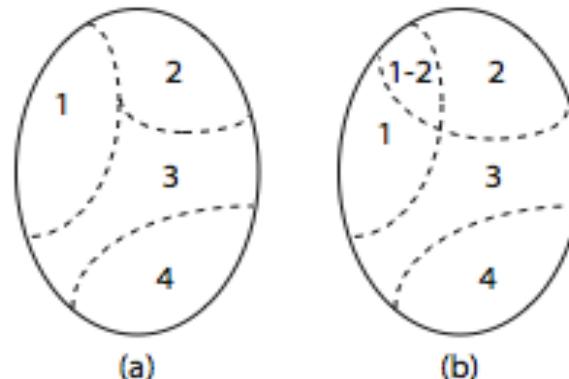
Consider a procedure P in a payroll processing system that takes an employee record as input and computes the weekly salary. For simplicity, assume that the employee record consists of the following items with their respective types and constraints:

ID: int;	ID is 3-digits long from 001 to 999.
name: string;	name is 20 characters long; each character belongs to the set of 26 letters and a space character.
rate: float;	rate varies from \$5 to \$10 per hour; rates are in multiples of a quarter.
hoursWorked: int;	hoursWorked varies from 0 to 60.

# Equivalence partitioning

Test selection using **equivalence partitioning** allows a tester to subdivide the input domain into a relatively small number of sub-domains, say  $N > 1$ , as shown (a).

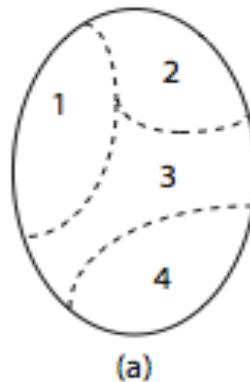
In strict mathematical terms, the sub-domains by definition are disjoint. The four subsets shown in (a) constitute a partition of the input domain while the subsets in (b) are not. Each subset is known as an **equivalence class**.



# Program behavior and equivalence classes

The equivalence classes are created assuming that the program under test exhibits the **same behavior** on all elements, i.e. tests, within a class.

This assumption allow the tester to select exactly one test from each equivalence class resulting in a test suite of exactly N tests.

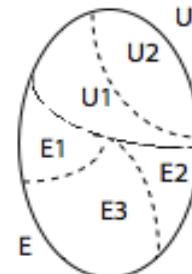


- E1:  $0 - 49 = f$
- E2:  $50 - 70 = c$
- E3:  $71 - 79 = b$
- E4:  $80 - 100 = a$

# Faults targeted

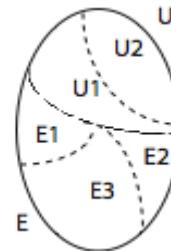
The entire set of inputs to any application can be divided into at least two subsets: one containing all the expected, or legal, inputs (E) and the other containing all unexpected, or illegal, inputs (U).

Each of the two subsets, can be further subdivided into subsets on which the application is required to behave differently (e.g. E1, E2, E3, and U1, U2).



# Faults targeted (contd.)

*Equivalence class partitioning selects tests that target any faults in the application that cause it to behave incorrectly when the input is in either of the two classes or their subsets.*



# Example 1

Consider an application A that takes an integer denoted by `age` as input. Let us suppose that the only legal values of `age` are in the range  $[1..120]$ . The set of input values is now divided into a set **E** containing all integers in the range  $[1..120]$  and a set **U** containing the remaining integers.



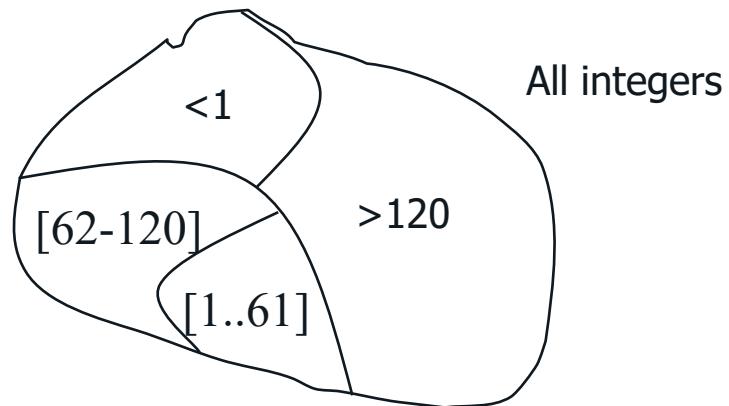
# Example 1 (contd.)

Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement **R1** and those in the range [62..120] according to requirement **R2**.

Thus, **E** is further subdivided into two regions depending on the expected behavior.

Similarly, it is expected that all invalid inputs less than 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of **U** into two categories.

# Example 1 (contd.)



# Example 1 (contd.)

Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e., two regions containing expected inputs and two regions containing the unexpected inputs.

It is expected that any single test selected from the range [1..61] will reveal any fault with respect to **R1**. Similarly, any test selected from the region [62..120] will reveal any fault with respect to **R2**. A similar expectation applies to the two regions containing the unexpected inputs.

# Example 2

We have an application that requires a username of length 6-12



# Effectiveness

The **effectiveness** of tests generated using equivalence partitioning for testing application A, is judged by the ratio of the number of faults these tests are able to expose to the total faults lurking in A.

As is the case with any test selection technique in software testing, the effectiveness of tests selected using equivalence partitioning is less than 1 for most practical applications. The effectiveness can be improved through an unambiguous and complete specification of the requirements and carefully selected tests using the equivalence partitioning technique described in the following sections.

# Equivalence classes based on program output

In some cases the equivalence classes are based on the **output** generated by the program. For example, suppose that a program outputs an integer.

It is worth asking: ``Does the program ever generate a 0? What are the maximum and minimum possible values of the output?''

These two questions lead to two the following equivalence classes based on outputs:

# Equivalence classes based on program output (contd.)

E1: Output value v is 0.

E2: Output value v is the maximum possible.

E3: Output value v is the minimum possible.

E4: All other output values.

Based on the **output equivalence classes** one may now derive equivalence classes for the inputs. Thus, each of the four classes given above might lead to one equivalence class consisting of inputs.

# Equivalence classes for variables: range

Eq. Classes	Example	
	Constraints	example
One class with values inside the range and two with values outside the range.	speed $\in [60..90]$	{50}U, {75}E, {92}U
	area: float	{{-1.0}U, {15.52}E}
	area $\geq 0.0$	
	age: int $0 < \text{age} < 120$	{-1}U, {56}E, {132}U

# Equivalence classes for variables: strings

Equivalence Classes	Example	
	Constraints	Classes
At least one containing all legal strings and one all illegal strings based on any constraints.	firstname: string	$\{\{Sue2\}U, \{Sue\}E, \{Loooong Name\}U\}$

# Equivalence classes for variables: enumeration

Equivalence Classes	Example	
	Constraints	Classes
Each value in a separate class	autocolor:{red, blue, green}	{ {red}E {blue}E, {green}E }
	up:boolean	{ {true}E, {false}E }

# Equivalence classes for variables: arrays

Equivalence Classes	Example	
	Constraints	Classes
One class containing all legal arrays, one containing the empty array, and one containing a larger than expected array.	int [ ] aName: new int[3];	{[ ]}U, {[-10, 20, 3]}E, {[-9, 0, 12, 15]}U

# Equivalence classes for variables: compound data type

Arrays in Java and records, or structures, in C++, are compound types. Such input types may arise while testing components of an application such as a function or an object.

While generating equivalence classes for such inputs, one must consider legal and illegal values for each component of the structure.

```
struct Account {  
    int account_number;  
    string account_name;  
    float balance;  
};
```

# uni-dimensional partitioning

One way to partition the input domain is to consider one input variable at a time. Thus, each input variable leads to a partition of the input domain. We refer to this style of partitioning as **uni-dimensional** equivalence partitioning or simply **uni-dimensional** partitioning.

*This type of partitioning is used commonly.*

# Partitioning Example

Consider an application that requires two integer inputs  $x$  and  $y$ . Each of these inputs is expected to lie in the following ranges:  $3 \leq x \leq 7$  and  $5 \leq y \leq 9$ .

For uni-dimensional partitioning we apply the partitioning guidelines to  $x$  and  $y$  individually. This leads to the following six equivalence classes.

# Partitioning Example (contd.)

E1:  $x < 3$

E2:  $3 \leq x \leq 7$

E3:  $x > 7$

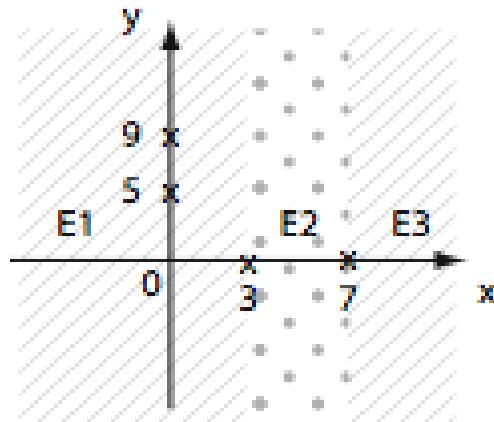
← y ignored.

E4:  $y < 5$

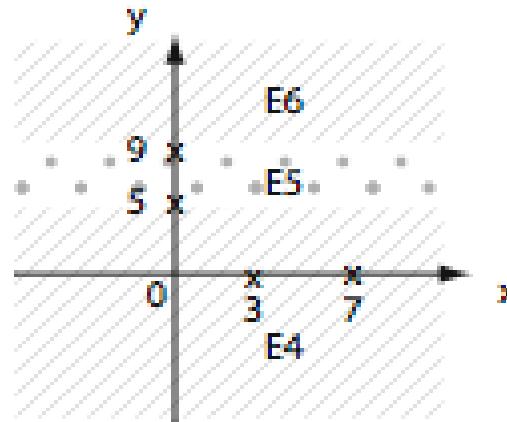
E5:  $5 \leq y \leq 9$

E6:  $y > 9$

← x ignored.



(a)



(b)

# Multidimensional partitioning

Another way is to consider the input domain  $I$  as the set product of the input variables and define a relation on  $I$ . This procedure creates one partition consisting of several equivalence classes. We refer to this method as **multidimensional equivalence partitioning** or simply **multidimensional** partitioning.

Multidimensional partitioning leads to a large number of equivalence classes that are difficult to manage manually. Many classes so created might be infeasible. Nevertheless, equivalence classes so created offer an increased variety of tests as is illustrated in the next section.

# Multidimensional partitioning

## Example

Once again lets consider an application that requires two integer inputs  $x$  and  $y$ . Each of these inputs is expected to lie in the following ranges:  $3 \leq x \leq 7$  and  $5 \leq y \leq 9$ .

# Multidimensional partitioning

## Example (contd.)

For multidimensional partitioning we consider the input domain to be the set product  $X \times Y$ . This leads to 9 equivalence classes.

$$E1: x < 3, y < 5$$

$$E2: x < 3, 5 \leq y \leq 9$$

$$E3: x < 3, y > 9$$

$$E4: 3 \leq x \leq 7, y < 5$$

$$E5: 3 \leq x \leq 7, 5 \leq y \leq 9$$

$$E6: 3 \leq x \leq 7, y > 9$$

$$E7: x > 7, y < 5$$

$$E8: x > 7, 5 \leq y \leq 9$$

$$E9: x > 7, y > 9$$

# Multidimensional partitioning

## Example (contd.)

E1:  $x < 3, y < 5$

E3:  $x < 3, y > 9$

E2:  $x < 3, 5 \leq y \leq 9$

E4:  $3 \leq x \leq 7, y < 5$

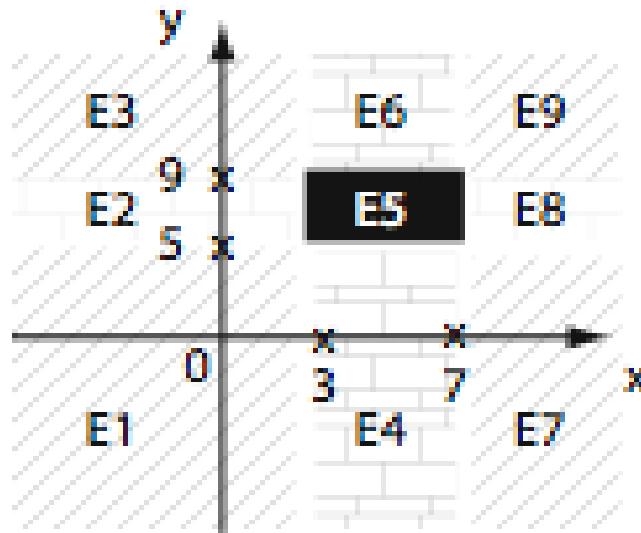
E5:  $3 \leq x \leq 7, 5 \leq y \leq 9$

E6:  $3 \leq x \leq 7, y > 9$

E7:  $> 7, y < 5$

E8:  $x > 7, 5 \leq y \leq 9$

E9:  $x > 7, y > 9$



9 equivalence classes:

# Partitioning Terminology

## Strong vs. Weak Equivalence Class Testing

### • Weak Equivalence Class Testing:

- In weak equivalence class testing, we **test one variable at a time**, selecting one representative value from each equivalence class while keeping other variables constant.
- This aligns with **uni-dimensional equivalence partitioning**, where each variable is considered independently.

### • Strong Equivalence Class Testing:

- In strong equivalence class testing, we **test combinations of values across multiple variables**, selecting a test case from each combination of equivalence classes.
- This aligns with **multidimensional equivalence partitioning**, where we consider interactions between variables and create test cases for each combination.

• Weak partitioning tests each variable separately (like uni-dimensional partitioning).

• Strong partitioning tests multiple variables together (like multidimensional partitioning).

• Strong equivalence testing results in more test cases but provides better coverage of input interactions.

# Systematic procedure for equivalence partitioning

1. **Identify the input domain:** Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use.

Environment variables, such as class variables used in the method under test and environment variables in Unix, Windows, and other operating systems, also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets.

# Systematic procedure for equivalence partitioning (contd.)

2. **Equivalence classing:** Partition the set of values of each variable into disjoint subsets.

Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. partitioning the input domain using values of one variable, is done based on the the expected behavior of the program.

Values for which the program is expected to behave in the ``same way" are grouped together. Note that ``same way" needs to be defined by the tester.

# Systematic procedure for equivalence partitioning (contd.)

**3. Combine equivalence classes:** This step is usually omitted and the equivalence classes defined for each variable are directly used to select test cases. However, by not combining the equivalence classes, one misses the opportunity to generate useful tests.

The equivalence classes are combined using the multidimensional partitioning approach described earlier.

# Systematic procedure for equivalence partitioning (contd.)

4. Identify infeasible equivalence classes: An infeasible equivalence class is one that contains a combination of input data that cannot be generated during test. Such an equivalence class might arise due to several reasons.

For example, suppose that an application is tested via its GUI, i.e. data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering a palette of valid inputs only. There might also be constraints in the requirements that render certain equivalence infeasible.

# Selecting test data

Given a set of equivalence classes that form a partition of the input domain, it is relatively straightforward to select tests. However, complications could arise in the presence of infeasible data and don't care values.

In the most general case, a tester simply selects one test that serves as a representative of each equivalence class.

# Weak/ Strong EQ Class Testing

- Test selection criteria can be
  - Weak
  - Strong
- These two criteria can then be sub classified as
  - Normal
  - Robust

# Weak/ Strong EQ Class Testing

## Date Function Problem

- Valid Equivalence Classes

$M1 = \{ \text{month} : 1 \leq \text{month} \leq 12 \}$

$D1 = \{ \text{day} : 1 \leq \text{day} \leq 31 \}$

$Y1 = \{ \text{year} : 1812 \leq \text{year} \leq 2012 \}$

- Invalid Equivalence Classes

$M2 = \{ \text{month} : \text{month} < 1 \}$

$M3 = \{ \text{month} : \text{month} > 12 \}$

$D2 = \{ \text{day} : \text{day} < 1 \}$

$D3 = \{ \text{day} : \text{day} > 31 \}$

$Y2 = \{ \text{year} : \text{year} < 1812 \}$

$Y3 = \{ \text{year} : \text{year} > 2012 \}$

# Weak/ Strong EQ Class Testing

- Normal tests only consider the valid EQ classes
- Valid Equivalence Classes**

$M1 = \{ \text{month} : 1 \leq \text{month} \leq 12 \}$

$D1 = \{ \text{day} : 1 \leq \text{day} \leq 31 \}$

$Y1 = \{ \text{year} : 1812 \leq \text{year} \leq 2012 \}$

- Invalid Equivalence Classes

$M2 = \{ \text{month} : \text{month} < 1 \}$

$M3 = \{ \text{month} : \text{month} > 12 \}$

$D2 = \{ \text{day} : \text{day} < 1 \}$

$D3 = \{ \text{day} : \text{day} > 31 \}$

$Y2 = \{ \text{year} : \text{year} < 1812 \}$

$Y3 = \{ \text{year} : \text{year} > 2012 \}$

Day	Month	Year	Expected Output
15	6	1912	16/6/1912

# Weak Robust EQ Class Testing

- Robust tests consider the **valid and invalid** EQ classes

- Valid Equivalence Classes

$$M1 = \{ \text{month} : 1 \leq \text{month} \leq 12 \}$$

$$D1 = \{ \text{day} : 1 \leq \text{day} \leq 31 \}$$

$$Y1 = \{ \text{year} : 1812 \leq \text{year} \leq 2012 \}$$

- Invalid Equivalence Classes

$$M2 = \{ \text{month} : \text{month} < 1 \}$$

$$M3 = \{ \text{month} : \text{month} > 12 \}$$

$$D2 = \{ \text{day} : \text{day} < 1 \}$$

$$D3 = \{ \text{day} : \text{day} > 31 \}$$

$$Y2 = \{ \text{year} : \text{year} < 1812 \}$$

$$Y3 = \{ \text{year} : \text{year} > 2012 \}$$

Day	Month	Year	Expected Output
15	6	1912	16/6/1912
-1	6	1912	day not in range
32	6	1912	day not in range
15	-1	1912	month not in range
15	13	1912	month not in range
15	6	1811	year not in range
15	6	2013	year not in range

# Weak Robust EQ Class Testing

- Robust tests consider the **valid and invalid** EQ classes

- Valid Equivalence Classes

$$M1 = \{ \text{month} : 1 \leq \text{month} \leq 12 \}$$

$$D1 = \{ \text{day} : 1 \leq \text{day} \leq 31 \}$$

$$Y1 = \{ \text{year} : 1812 \leq \text{year} \leq 2012 \}$$

- Invalid Equivalence Classes

$$M2 = \{ \text{month} : \text{month} < 1 \}$$

$$M3 = \{ \text{month} : \text{month} > 12 \}$$

$$D2 = \{ \text{day} : \text{day} < 1 \}$$

$$D3 = \{ \text{day} : \text{day} > 31 \}$$

$$Y2 = \{ \text{year} : \text{year} < 1812 \}$$

$$Y3 = \{ \text{year} : \text{year} > 2012 \}$$

Day	Month	Year	Expected Output
15	6	1912	16/6/1912
-1	6	1912	day not in range
32	6	1912	day not in range
15	-1	1912	month not in range
15	13	1912	month not in range
15	6	1811	year not in range
15	6	2013	year not in range

# Strong Robust EQ Class Testing

- Strong robust tests consider the **valid and invalid** EQ classes
- Cartesian product between all sets
- $M = 3, D = 3, Y = 3$
- $3 \times 3 \times 3 = 27$  tests

- Valid Equivalence Classes

$$M1 = \{ \text{month} : 1 \leq \text{month} \leq 12 \}$$

$$D1 = \{ \text{day} : 1 \leq \text{day} \leq 31 \}$$

$$Y1 = \{ \text{year} : 1812 \leq \text{year} \leq 2012 \}$$

- Invalid Equivalence Classes

$$M2 = \{ \text{month} : \text{month} < 1 \}$$

$$M3 = \{ \text{month} : \text{month} > 12 \}$$

$$D2 = \{ \text{day} : \text{day} < 1 \}$$

$$D3 = \{ \text{day} : \text{day} > 31 \}$$

$$Y2 = \{ \text{year} : \text{year} < 1812 \}$$

$$Y3 = \{ \text{year} : \text{year} > 2012 \}$$

# GUI design and equivalence classes

While designing equivalence classes for programs that obtain input exclusively from a keyboard, one must account for the possibility of errors in data entry. For example, the requirement for an application.

The application places a constraint on an input variable **X** such that it can assume integral values in the range 0..4. However, testing must account for the possibility that a user may inadvertently enter a value for **X** that is out of range.

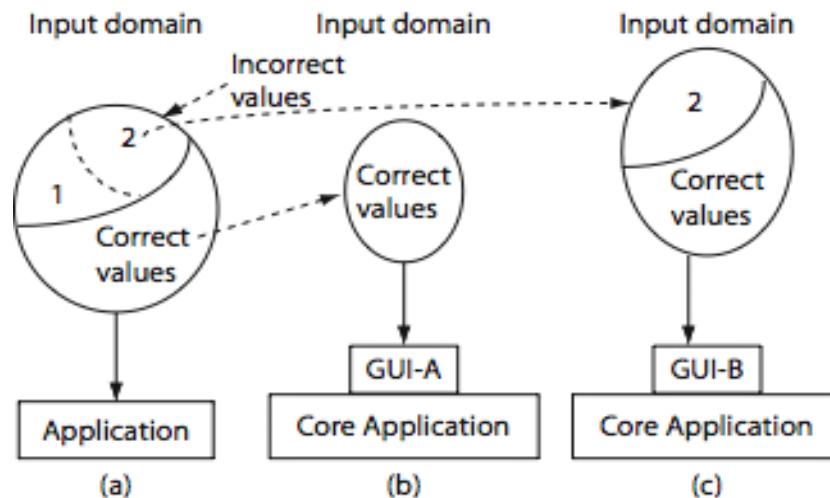
# GUI design and equivalence classes (contd.)



Suppose that all data entry to the application is via a GUI front end. Suppose also that the GUI offers exactly five correct choices to the user for **X**.

In such a situation it is impossible to test the application with a value of **X** that is out of range. Hence only the correct values of **X** will be input. See figure on the next slide.

# GUI design and equivalence classes (contd.)



# Equivalence classes goals

## The motivation

- *Completeness of test coverage*
  - *At least one test per partition of the input domain*
  - *Helps raise our confidence that we are testing the functionality and all areas of the input domain*
- *Reduces test case duplication*
  - *Better use of resources*
  - *Less redundant tests*

# Errors at the boundaries

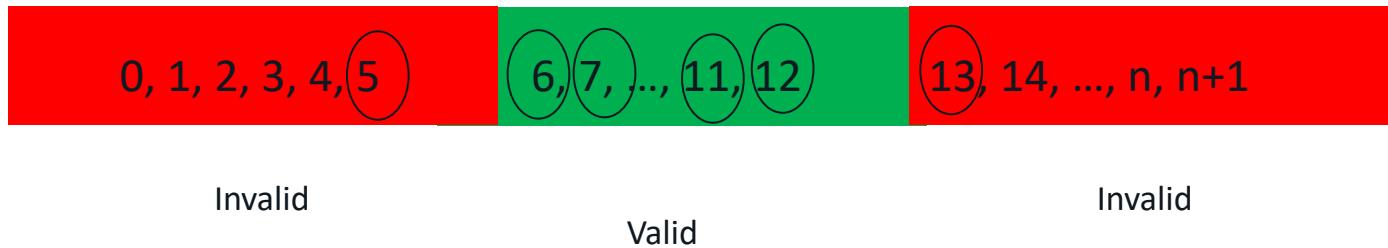
Experience indicates that programmers make mistakes in processing values at and near the **boundaries of equivalence classes**.

For example, suppose that method **M** is required to compute a function **f1** when  $x \leq 0$  is true and function **f2** otherwise. However, **M** has an error due to which it computes **f1** for  $x < 0$  and **f2** otherwise.

Obviously, this fault is revealed, though not necessarily, when **M** is tested against  $x=0$  but not if the input test set is, for example,  $\{-4, 7\}$  derived using equivalence partitioning. In this example, the value  $x=0$ , lies at the boundary of the equivalence classes  $x \leq 0$  and  $x > 0$ .

# Boundary value analysis (BVA)

We have an application that requires a username of length 6-12



# Boundary value analysis (BVA)

Boundary value analysis is a test selection technique that targets faults in applications at the boundaries of equivalence classes.

While equivalence partitioning selects tests from within equivalence classes, boundary value analysis focuses on tests at and near the boundaries of equivalence classes.

Certainly, tests derived using either of the two techniques may overlap.

# BVA: Procedure

- 1 Partition the input domain using uni-dimensional partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in this step.
- 2 Identify the boundaries for each partition. Boundaries may also be identified using special relationships amongst the inputs.
- 3 Select test data such that each boundary value occurs in at least one test input.

# BVA: Example: 1. Create equivalence classes

Assuming that an item **code** must be in the range 99...999 and **quantity** in the range 1...100,

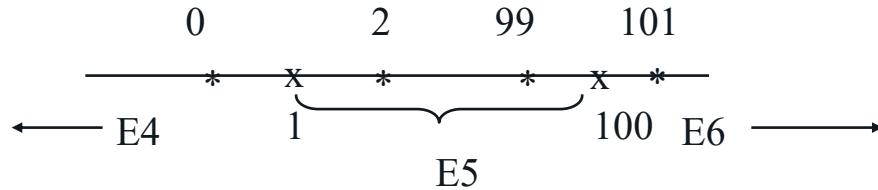
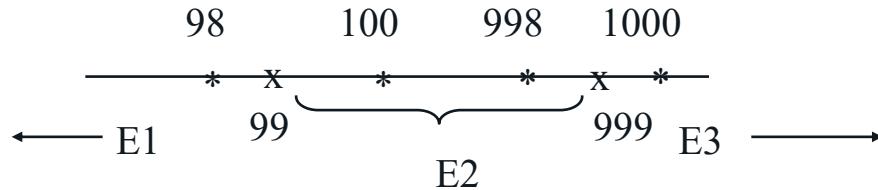
## Equivalence classes for code:

- E1: Values less than 99.
- E2: Values in the range.
- E3: Values greater than 999.

## Equivalence classes for qty:

- E4: Values less than 1.
- E5: Values in the range.
- E6: Values greater than 100.

# BVA: Example: 2. Identify boundaries



Equivalence classes and boundaries for `findPrice`. Boundaries are indicated with an x. Points near the boundary are marked \*.

## BVA: Example: 3. Construct test set

Test selection based on the boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary. Consider the following test set:

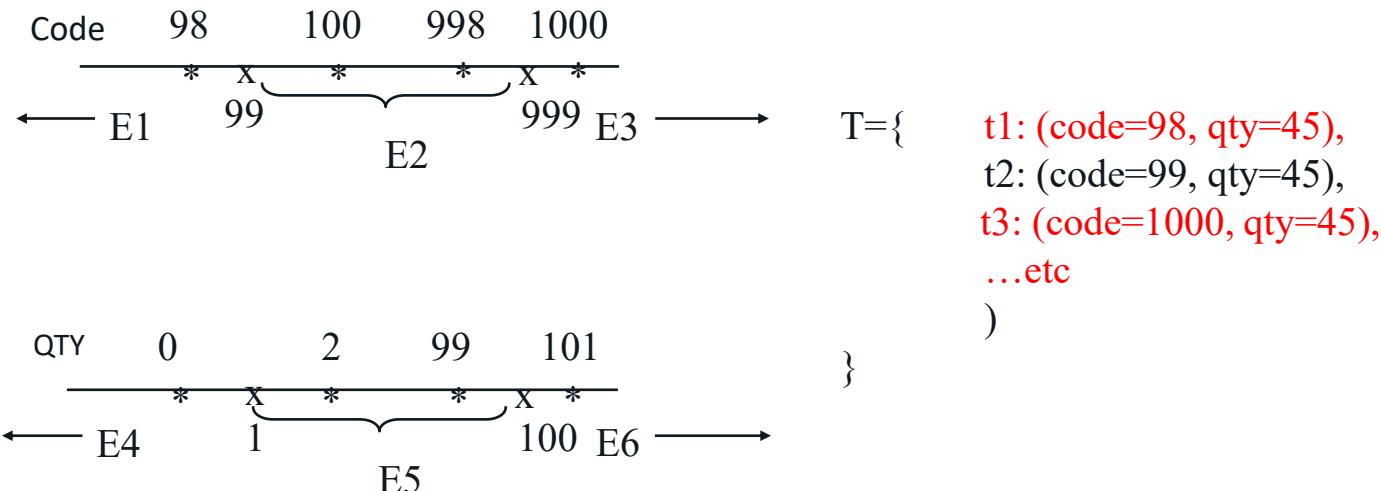
```
T={      t1: (code=98, qty=0),  
          t2: (code=99, qty=1),  
          t3: (code=100, qty=2),  
          t4: (code=998, qty=99),  
          t5: (code=999, qty=100),  
          t6: (code=1000, qty=101)  
    }
```



Illegal values of code and qty included.

# BVA: Example: 3. Construct test set

By separating the correct and incorrect values of different input variables, we increase the possibility of detecting errors



# BVA: Recommendations

Relationships amongst the input variables must be examined carefully while identifying boundaries along the input domain. This examination may lead to boundaries that are not evident from equivalence classes obtained from the input and output variables.

Additional tests may be obtained when using a partition of the input domain obtained by taking the product of equivalence classes created using individual variables.

# Summary

Equivalence partitioning and boundary value analysis are the most commonly used methods for test generation while doing functional testing.