Input Space Partitioning

Instructor : Ali Sharifara

CSE 5321/4321

Summer 2017

Input Space Partitioning

- Introduction
- Equivalence Partitioning
- Boundary-Value Analysis
- Summary

Agenda

- Quick Review
- Input Space Partitioning

Introduction

- Testing is about choosing elements from input domain

- The *input domain* of a program consists of all possible inputs that could be taken by the program
  - Easy to get started, based on description of the inputs

Input Domain

- For even small programs, the input domain is so large that it might be infinite. (e.g. gcd(int x, int y))
- *Input parameters* define the scope of the *input domain*:
  - Parameters to a method
  - Data read from a file
  - Global variable
  - User level inputs
  - etc
- Domain for each input parameter is partitioned into regions
- At least one value is chosen from each region

Quick Review

- What is the test selection problem ?
- What is the main idea of input space partitioning ?

Test Selection Problem

- Ideally, the test selection problem is to select a subset T of the input domain such that the execution of T will reveal all errors.

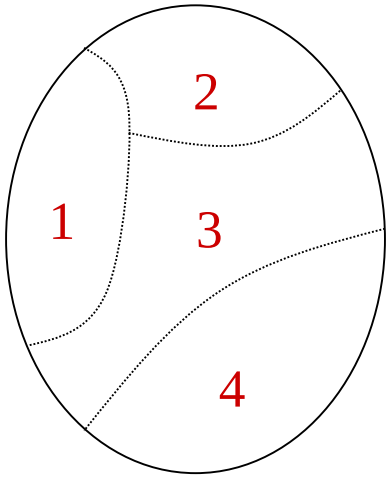
- In practice, the test selection problem is to select a subset of T within budget such that it reveals as many error as possible.

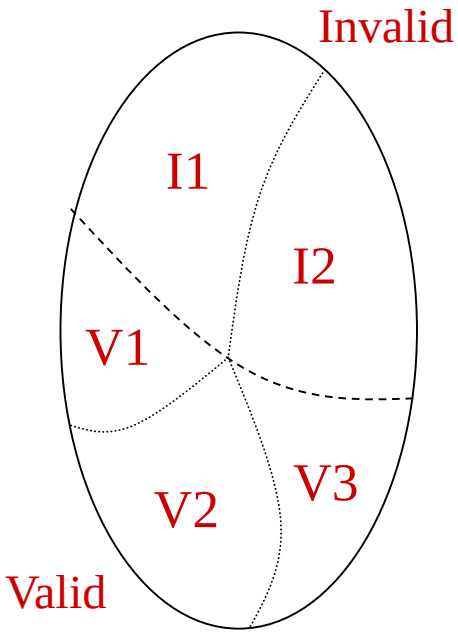
Partitioning

- Partition the input domain into a relatively small number of groups, and then select one representative from each group.

Based on two properties:

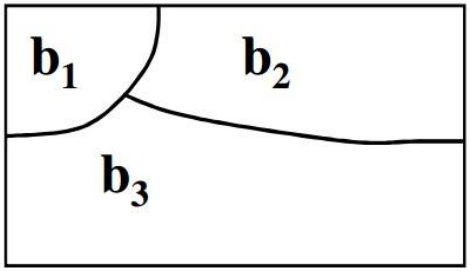
1. Completeness: The partition must cover the entire domain.
2. Disjointness: The blocks must not overlap.





Partitioning Cont.

- Domain D
- Partition scheme q of D
- The partition q defines a set of blocks, $B_q = b_1, b_2, \ldots b_q$
- The Partition must satisfy two properties :
  - The partition must cover the entire domain (completeness)
  - The blocks must not overlap (disjointness)



$$b_i \cap b_j = \Phi, \forall\, i \neq j, b_i, b_j \in B_q$$

$$\bigcup_{b \in Bq} b = D$$

Partitioning of input domain D into three blocks

Using Partitions - Assumptions

- Choose a value from each partition
- Each value is assumed to be equally useful for testing
  - Find characteristics in the input : parameters, semantic, description, …
  - Partition each characteristics
  - Choose tests by combining values from characteristics
- Some possible characteristic examples:
  - Input X is null
  - Order of the input file F (sorted, inverse sorted, arbitrary)
  - Input device (DVD, CD, VCR, computer, …)

Each characteristic C allows the tester to define a partition

Choosing Partitions

- Choosing (or defining) partitions seems easy, but is easy to get wrong
- Consider the "order of file F"

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

but … something's fishy …

What if the file is of length 1?

The file will be in all three blocks …

That is, disjointness is not satisfied

Solution:

Each characteristic should address just one property

File F sorted ascending

- **b1 = true**
- **b2 = false**

File F sorted descending

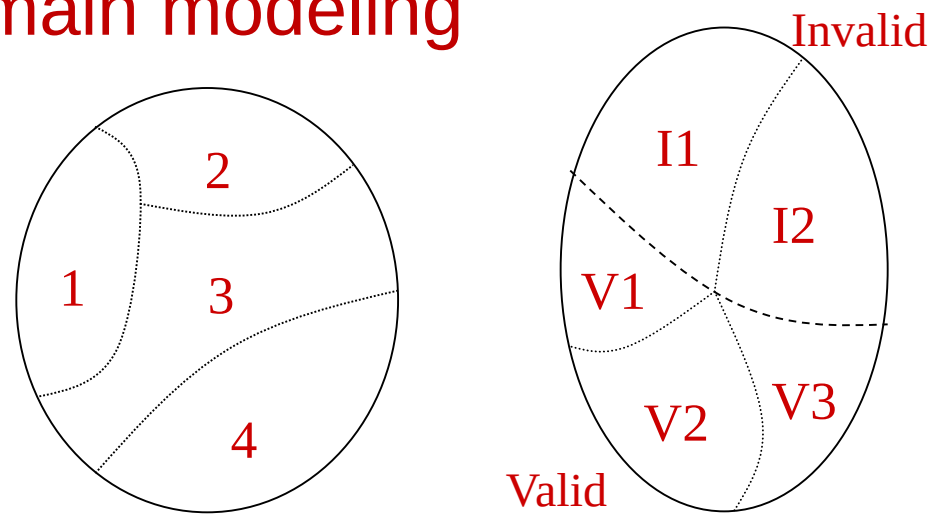
- **b1 = true**
- **b2 = false**

Properties of Partitions

- If the partitions are not complete or disjoint, that means the partitions have NOT been considered carefully enough
- They should be reviewed carefully, like any design attempt
- Different alternatives should be considered

Example

- Consider a program that is designed to sort a sequence of integers into the ascending order.
- What is the input domain of this program?
  Input domain modeling



Input Space Partitioning

- Introduction
- Equivalence Partitioning
- Boundary-Value Analysis
- Summary

Partition (review!)

- A partition defines a set of equivalent classes, or blocks
  - All the members in an equivalence class contribute to error detection in the same way
- A partition must satisfy two properties:
  - Completeness: A partition must cover the entire domain
  - Disjoint: The blocks must not overlap
- A partition is usually based on certain characteristic
  - e.g., whether a list of integer is sorted or not, whether a list allows duplicates or not

Input Domain Modeling

- Step 1: Identify testable components, which could be a method, a use case, or the entire system
- Step 2: Identify all of the parameters that can affect the behavior of a given testable component
  - Input parameters, environment configurations, state variables.
  - For example, insert(obj) typically behaves differently depending on whether the object is already in a list or not.
- Step 3: Identify characteristics, and create partitions for each characteristic
- Step 4: Select values from each partition, and combine them to create tests

Exercise

A tester defined three characteristics based on the input parameter car : Where Made, Energy Source, and Size. The following partitionings for these characteristics have at least two mistakes. Correct them.

Where Made	Energy Source	Size
North America	Gas	2-door
Europe	Electric	4-door
Asia	Hybrid	Hatch-back

Different Approaches to Input Domain Modeling (IDM)

- Interface-Based IDM
- Functionality-Based IDM

Interface-Based Input Domain Modeling (1)

- The main idea is to identify parameters and values, typically in isolation, based on the interface of the component under test.
- Advantage: Relatively easy to identify characteristics
- Disadvantage:
  - IDM may be incomplete and hence additional characteristics are needed
  - Not all information is reflected in the interface, and testing some functionality may require parameters in combination

Interface-Based Input Domain Modeling (2)

- Range: one class with values inside the range, and two with values outside the range
  - For example, let speed $\in$ [60 .. 90]. Then, we generate three classes {{50}, {75}, {92}}.
- String: at least one containing all legal strings and one containing all illegal strings.
  - For example, let fname: string be a variable to denote a first name. Then, we could generate the following classes: {{ε}, {Sue}, {Sue2}, {Too long a name}}.

Interface-Based Input Domain Modeling (3)

- Enumeration: Each value in a separate class
  - For example, consider auto_color ∈ {red, blue, green}. The following classes are generated, {{red}, {blue}, {green}}
- Array: One class containing all legal arrays, one containing only the empty array, and one containing arrays larger than the expected size
  - For example, consider int[] aName = new int [3]. The following classes are generated: {{[]}, {[-10, 20]}, {[-9, 0, 12, 15]}.

Functionality-Based Input Domain Modeling (1)

- The main idea is to identify characteristics that correspond to the intended functionality of the component under test
- Advantage: Includes more semantic information, and does not have to wait for the interface to be designed
- Disadvantage: Hard to identify characteristics, parameter values, and tests

Functionality-Based Input Domain Modeling (2)

- Preconditions explicitly separate normal behavior from exceptional behavior
  - For example, a method requires a parameter to be non-null.
- Postconditions indicate what kind of outputs may be produced
  - For example, if a method produces two types of outputs, then we want to select inputs so that both types of outputs are tested.
- Relationship of variables with special values (zero, null, blank,..)
- Relationships between different parameters can also be used to identify characteristics
  - For example, if a method takes two object parameters x and y, we may want to check what happens if x and y point to the same object or to logically equal objects

Identify Characteristics

- The interface-based approach *develops characteristics* directly from input parameters
- The functionality-based approach *develops characteristics* from functional or behavioral view

Example.1

Public Boolean findElement(List list, Element element)

//If list or element is null throw NullPointerException else returns true if element is in the list , false otherwise

List Characteristics

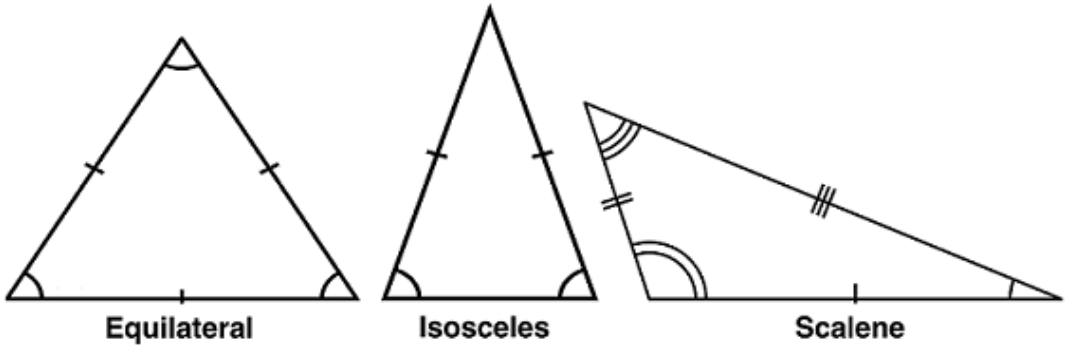
Interface-based	
Characteristics	**Blocks and Values**
List is null	b1= true b2= false
List is empty	b1 = true b2= false

Functionality-based	
Characteristics	**Blocks and Values**
Number of occurrences of element in list	b1= 0 b2= 1 b3= more than 1
Element occurs first in list	b1 = true B2 = false

Example.2 (1)

- Consider a triangle classification program which inputs three integers representing the lengths of the three sides of a triangle, and outputs the type of the triangle.
- The possible types of a triangle include scalene, equilateral, isosceles, and invalid.

```
int classify (int side1, int side2, int side3)
// 0: scalene, 1: equilateral, 2: isosceles; -1: invalid
```



Example.2 (2)

- Interface-based IDM: Consider the relation of the length of each side to some special value such as zero

Partition	b1	b2	b3
q_1 = Relation of Side 1 to 0	> 0	= 0	< 0
q_2 = Relation of Side 2 to 0	> 0	= 0	< 0
q_3 = Relation of Side 3 to 0	> 0	= 0	< 0

Example.2 (3)

- Functionality-based IDM: Consider the traditional geometric partitioning of triangles

Partition	b1	b2	b3	b4
Geometric classification	Scalene	Isoceles	Equilateral	Invalid

Oops … something's fishy … equilateral is also isosceles !
We need to refine the example to make characteristics valid

Example.2 (4)

Partition	b1	b2	b3	b4
q1= Geometric classification	Scalene	*Isoceles, not equilateral*	Equilateral	Invalid

Param	b1	b2	b3	b4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Perimeter of triangle P = a+b+c, if a + b > c, otherwise is invalid

Side (a), base (b), Side (c)

Functionality-Based IDM—triangle()

- A different approach would be to break the geometric characterization into four separate characteristics

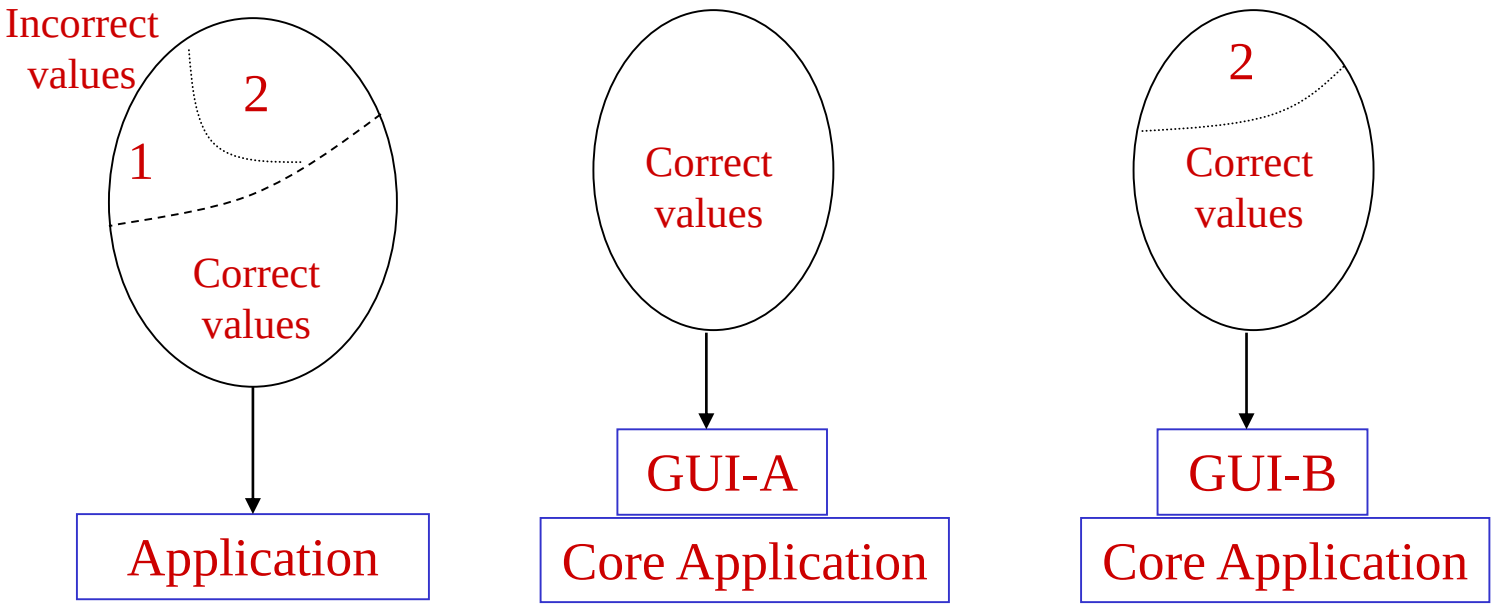
Characteristic	b_1	b_2
q_1 = "Scalene"	True	False
q_2 = "Isosceles"	True	False
q_3 = "Equilateral"	True	False
q_4 = "Valid"	True	False

- Use constraints to ensure that
  - Equilateral = True implies Isosceles = True
  - Valid = False implies Scalene = Isosceles = Equilateral = False

GUI Design (1)

- Suppose that an application has a constraint on an input variable X such that it can only assume integer values in the range 0 .. 4.
- Without GUI, the application must check for out-of-range values.
- With GUI, the user may be able to select a valid value from a list, or may be able to enter a value in a text field.

GUI Design (2)


Incorrect values
2
1
Correct values
Application
Correct values
GUI-A
Core Application
2
Correct values
GUI-B
Core Application

Input Space Partitioning

- Introduction
- Equivalence Partitioning
- Boundary-Value Analysis
- Summary

Motivation

- Programmers often make mistakes in processing values at and near the boundaries of equivalence classes.
- For example, a method M is supposed to compute a function f1 when condition x <= 0 and function f2 otherwise. However, M has a fault such that it computes f1 for x < 0 and f2 otherwise.
- Can you find an example that shows why a value near a boundary needs to be tested?

Boundary-Value Analysis

- A test selection technique that targets faults in applications at the boundaries of equivalence classes.
  - Partition the input domain
  - Identify the boundaries for each partition
  - Select test data such that each boundary value occurs in at least one test input

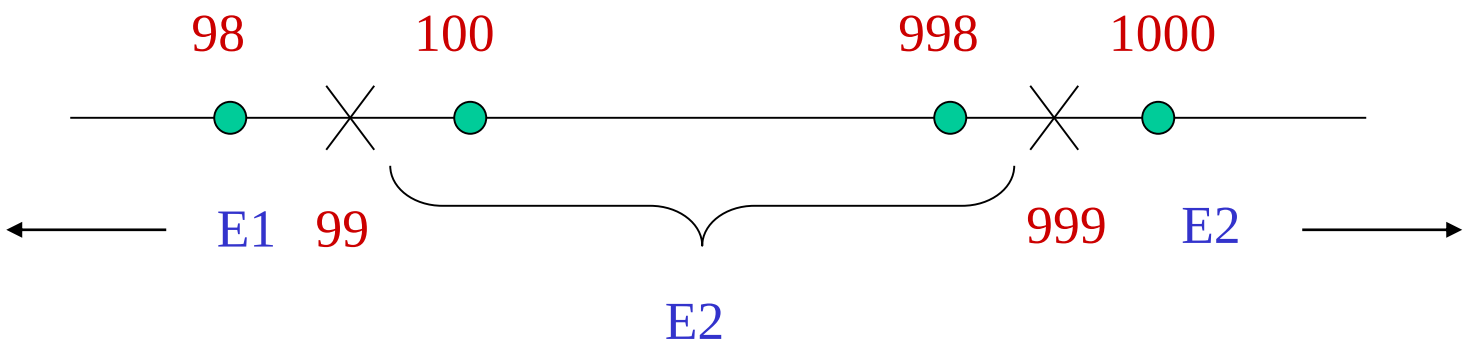
Example

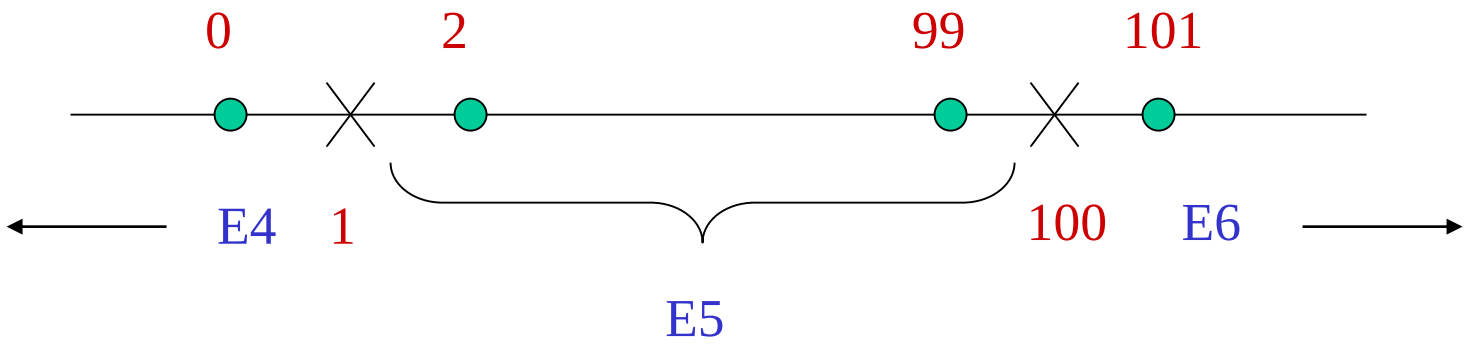
- Consider a method findPrices that takes two inputs, item code (99 .. 999) and quantity (1 .. 100).
- The method accesses a database to find and display the unit price, the description, and the total price, if the code and quantity are valid.
- Otherwise, the method displays an error message and return.

Example (2)

- Equivalence classes for code:
  - E1: Values less than 99
  - E2: Values in the range
  - E3: Values greater than 999
- Equivalence classes for quantity:
  - E4: Values less than 1
  - E5: Values in the range
  - E6: Values greater than 100

Example (3)





Example (4)

- Tests are selected to include, for each variable, values at and around the boundary
- An example test set is 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) }

Example (5)

```
public void findPrice (int code, int qty)
{
    if (code < 99 or code > 999) {
        display_error (“Invalid code”); return;
    }
   // begin processing
}
```

Example (6)

- One way to fix the problem is to replace t1 and t6 with the following four tests:

t7 = (code = 98, qty = 45),

t8 = (code = 1000, qty = 45),

t9 = (code = 250, qty = 0),

t10 = (code = 250, qty = 101).

Input Space Partitioning

- Introduction
- Equivalence Partitioning
- Boundary-Value Analysis
- Summary

Summary

- Test selection is about Partitioning the input space in a cost-effective manner.
- The notions of equivalence partitioning and boundary analysis are so common that sometimes we apply them without realizing it.
- Interface-based IDM is easier to perform, but may miss some important semantic information; functionality-based IDM is more challenging, but can be very effective in many cases.
- Boundary analysis considers values both at and near boundaries.