Input Space Partitioning

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

The Test Selection Problem

- The input domain of a program consists of all possible inputs that could be taken by the program.
- 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 errors as possible.
**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?

**Main Idea**

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

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

- **Step 2: Identify equivalence classes**
  - Partition the set of values of each variable into disjoint subsets, based on the expected behavior.

- **Step 3: Combine equivalence classes**
  - Use some well-defined strategies to avoid potential explosion

- **Step 4: Remove infeasible combinations of equivalence classes**

Input Space Partitioning

- Introduction
- **Equivalence Partitioning**
- Boundary-Value Analysis
- Summary
Input Parameter 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

Partition

- A partition defines a set of equivalent classes, or blocks
  - All the members in an equivalence class contribute to fault 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
Interface-Based IPM (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**: Not all information is reflected in the interface, and testing some functionality may require parameters in combination

Interface-Based IPM (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: $\{\epsilon\}, \{Sue\}, \{Sue2\}, \{Too long a name\}$.
**Interface-Based IPM (3)**

- **Enumeration**: Each value in a separate class
  - For example, consider `auto_color \in \{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 IPM (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 IPM (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.

- **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.

**Example (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**.

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

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

<table>
<thead>
<tr>
<th>Partition</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation of Side 1 to 0</td>
<td>&gt; 0</td>
<td>= 0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>Relation of Side 2 to 0</td>
<td>&gt; 0</td>
<td>= 0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>Relation of Side 3 to 0</td>
<td>&gt; 0</td>
<td>= 0</td>
<td>&lt; 0</td>
</tr>
</tbody>
</table>

Example (3)

- **Functionality-based IPM**: Consider the traditional geometric classification of triangles

<table>
<thead>
<tr>
<th>Partition</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric classification</td>
<td>Scalene</td>
<td>Isoceles</td>
<td>Equilateral</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
### Example (4)

<table>
<thead>
<tr>
<th>Partition</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric classification</td>
<td>Scalene</td>
<td>Isoceles, not equilateral</td>
<td>Equilateral</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Param</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle</td>
<td>(4, 5, 6)</td>
<td>(3, 3, 4)</td>
<td>(3, 3, 3)</td>
<td>(3, 4, 8)</td>
</tr>
</tbody>
</table>

### 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
1 2
Correct values

Application

Correct values
GUI-A
Core Application

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 $f_1$ when condition $x \leq 0$ and function $f_2$ otherwise. However, $M$ has an error such that it computes $f_1$ for $x < 0$ and $f_2$ otherwise.

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)

- Tests are selected to include, for each variable, values at and around the boundary.

- An example test set is $T = \{ t_1: (\text{code} = 98, \text{qty} = 0), t_2: (\text{code} = 99, \text{qty} = 1), t_3: (\text{code} = 100, \text{qty} = 2), t_4: (\text{code} = 998, \text{qty} = 99), t_5: (\text{code} = 999, \text{qty} = 100), t_6: (\text{code} = 1000, \text{qty} = 101) \}$
**Example (5)**

```java
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 sampling 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 IPM is easier to perform, but may miss some important semantic information; functionality-based IPM is more challenging, but can be very effective in many cases.
- Boundary analysis considers values both at and near boundaries.