Equivalence Partitioning

From S. Somé, A. Williams

1

Equivalence Class Partitioning

- Suppose that we were going to test a method that implements the absolute value function for integers.
- Definition

```
public int abs( int x )
```

- Exhaustive testing would require testing *every* possible value of the type `int`.
  - Leaving aside the issue of practicality, this would still be overkill in terms of the potential to find bugs.
- Instead, see if we can partition the input domain into equivalence classes, based on the similarity of input values.

Definition and Example

- A set or range of input domain values can be considered to be an equivalence class if they can *reasonably* be expected to cause "similar" responses from the implementation under test.
- Example: for the absolute value function
  - What would be different between -36 and -37 as input data?
    - Probably ... not much. The result is the negative of the input data. These two values are candidates to be in the same equivalence class.
  - On the other hand, -36 and +37 would react differently.
    | -36 | = 36, while | 37 | = 37.
    - In one case, the absolute value is the negative of the input, while in the other case, the output is the same as the input. These two values should definitely be in different equivalence classes.

Example set of classes

- A potential set of equivalence classes for the absolute value function, expressed in domain notation, could be:

```
[Integer.MIN_VALUE, -1] [0] [1,Integer.MAX_VALUE]
```

- Rationale:
  - negative numbers: output should be negative of input.
  - positive numbers: output should be the same as the input
  - zero: it could be in either of the above (what is -0 anyway...?), but since no other value has that property, it should be in its own equivalence class.

Choose test values

`[Integer.MIN_VALUE, -1]` `[0]` `[1,Integer.MAX_VALUE]`

- Strategy: choose a "representative" value from each equivalence class. Any value ought to be as good as any other

`[Integer.MIN_VALUE, -1]`:	Choose `-34`
`[0]`:	Choose `0`
`[1,Integer.MAX_VALUE]`:	Choose `+42`

Back to the equivalence classes

- An improved strategy for choosing test values from equivalence classes is:
  - Choose representative values as before.
  - Choose all values on a boundary.
  - Choose all values that are "one off" from a boundary.
    - Intuitive idea but does not always apply…

Add boundary values

`[Integer.MIN_VALUE, -1] [0] [1,Integer.MAX_VALUE]`

- With our additional criteria…
  - `[Integer.MIN_VALUE, -1]`: Choose `-34, -2, -1`
  - `[0]`: Choose `0`
  - `[1,Integer.MAX_VALUE]`: Choose `1, 2, +42`
- What about those other boundaries…?
  - `Integer.MIN_VALUE, Integer.MIN_VALUE + 1,`
  - `Integer.MAX_VALUE -1, Integer.MAX_VALUE`
  - Is there a risk of errors near those boundaries?

Valid and Invalid Classes: Ranges (1)

- If a specification includes input conditions, these can be used to derive equivalence classes:

1. If an input condition specifies a range of values, this defines three classes:
   - within range: a valid input equivalence class
   - too large: an invalid input equivalence class
   - too small: a invalid input equivalence class

Valid and Invalid Classes : Ranges (2)

2. If an input condition specifies a range of values, and there is reason to believe the values would be handled differently, this leads to the following classes:
   - One valid equivalence class for each set of values that would be handled similarly
     - This may result in one equivalence class per value, if each value is distinctive.
   - Two invalid equivalence classes: too large, too small

Valid and Invalid Classes: Enumerations

3. If an input condition specifies an enumerated set of values (e.g. "car", "truck", etc.):
   - One valid equivalence class for each value in the enumeration.
   - One invalid equivalence class: all values not in the enumerated set (i.e. everything else).

- Watch out for potential bugs related to implementation of enumerated types as integer code values, which has a larger domain (ie using an int that does NOT correspond to a value of the enum).
  - Example:

```
public static final int CAR = 1;
public static final int TRUCK = 2;
```

Valid and Invalid Classes: Presence / absence

4. If an input condition specifies a "must be", situation (e.g. "first character of the identifier must be a letter"), this leads to:
   - One valid equivalence class (e.g. the first character is a letter).
   - One invalid equivalence class (e.g. the first character is not a letter).

   Exercise: work out the equivalence classes for testing a postal code in Canada

Valid and Invalid Classes: When in doubt...

5. Finally, if there is any reason to believe that elements in an equivalence class are not handled in an identical manner by the implementation software, split the equivalence class into smaller classes.
   - e.g., 1-800 numbers are treated differently than other other 1-<3 digit area code> telephone numbers.

Equivalence Class Partitioning

- Also, consider creating an equivalence partition that handles the default, empty, blank, null, zero, or none conditions.
  - Default: no value supplied, and some value is assumed to be used instead.
  - Empty: value exists, but has no contents.
    - e.g. Empty string ""
  - Blank: value exists, and has content.
    - e.g. String containing a space character " "
  - Null: value does not exist or is not allocated.
    - E.g. object that has not been created.
  - Zero: numeric value
  - None: when selecting from a list, make no selection.

Equivalence Class Table

External condition	Valid equivalence classes	Invalid equivalence classes
• integer value between 1 and 10 • one of X, Y, or Z	V1: [1,10] V2: [X] V3: [Y] V4: [Z]	I1: [-∞,0] I2: [11,+∞] I3: [non-integer] I4: [not X, Y, or Z]

Test Case Strategy

- Once the set of equivalence classes has been identified, here is how to derive test cases:
  1. Assign a unique identifier to each equivalence class.
  2. Until all valid equivalence classes have been covered by at least one test case, write a new test case covering as many of the valid equivalence classes as possible.
  3. Until all invalid equivalence classes have been covered, write a test case that covers one, and only one, of the uncovered invalid equivalence classes.
  - For each test case, annotate it with the equivalence class identifiers that it covers.

Equivalence Classes Partitioning – Triangle Example (1)

- Specification
  - Input is three integers (sides of a triangle: a, b, c)
  - Each side must be a positive number less or equal to 20.
  - Output type of the triangle:
    - Equilateral: if a = b = c
    - Isosceles: if 2 pairs of sides are equals
    - Scalene if no pair of sides is equal
    - Invalid: if $a \geq b + c$, $b \geq a + c$, or $c \geq a + b$

Equivalence Classes Partitioning – Triangle Example (2)

- According to heuristic #1

Input condition	Valid EC	Invalid EC
Sides (a,b,c)	V1: all are (0,20]	I1: a > 20 I2: b > 20 I3: c > 20 I4: a ≤ 0 I5: b ≤ 0 I6: c ≤ 0

Equivalence Classes Partitioning – Triangle Example (3)

- Class V1 too broad, and can be subdivided (heuristic #5)
- Based on the treatment to data - handling of data
  - V1: a, b, c such that the triangle is equilateral
  - V2. a, b, c such that the triangle is isosceles
  - V3. a, b, c such that the triangle is scalene
  - V4. a, b, c such that it's not a triangle (yet valid inputs…)
- Based on input (driven by intuition?)
  - V5. a = b = c
  - V6. a = b, a ≠ c
  - V7. a = c, a ≠ b
  - V8. b = c, a ≠ b
  - V9. a ≠ b, a ≠ c, b ≠ c
- Based on triangle property (in fact breaks down v4)
  - V10. a, b, c such that a >= b + c
  - V11. a, b, c such that b >= a + c
  - V12. a, b, c such that c >= a + b

Equivalence Classes Partitioning – Triangle Example (4)

Equivalence classes	A	B	C	Response	ID
V1 V5	3	3	3	Equilateral	T1
V2 V6	2	2	3	Isosceles	T2
V2 V7	2	3	2	Isosceles	T3
V2 V8	3	2	2	Isosceles	T4
V3 V9	2	3	4	Scalene	T5
V4, V8, V10	20	2	2	Not a triangle	T6
V4, V7, V11	2	5	2	Not a triangle	T7
V4, V6, V12	2	2	5	Not a triangle	T8

Equivalence Classes Partitioning – Triangle Example (5)

I1 V2 V8	25	19	19	Error	T9
I2 V2 V7	19	25	19	Error	T10
I3 V2 V6	19	19	25	Error	T11
I4 V2 V8	-1	5	5	Error	T12
I5 V2 V7	5	-1	5	Error	T13
I6 V2 V6	5	5	-1	Error	T14

Equivalence Classes Partitioning - Problems

- Specification doesn't always define expected output for invalid test-cases.
- Strongly typed languages eliminate the need for the consideration of some invalid inputs.
- Brute-force approach of defining a test case for every combination of the inputs ECs
  - Provides good coverage, but…
  - …is impractical when number of inputs and associated classes is large