CS 213 Spring 2016
Lecture 17: Mar 22
Black-box Unit Testing

Boundary Value Analysis

- Consider the method `charAt(i)` of the `String` class. This method implements a function that takes as input an integer position, and returns the character at that position in the string.

- The range of the input, i, is:

$$0 \leq i \leq s - 1$$

  where s is the length of the string.

- To test that the `charAt` method works correctly for *a given string of length s*, we would need to build the following input test cases (assuming $s > 4$):
  - Lower boundary value $i = 0$
  - Upper boundary value $i = s - 1$
  - A small increment on the lower boundary value, $i = 1$
  - A small decrement on the upper boundary value, $i = s - 2$
  - A *nominal* value of i, that is one that is somewhere in the range that is not any of the above. Say $i = s/2$.

- This is the basic **boundary value analysis** approach. In general, if a function has an integer input i, which is in the interval $l \leq i \leq h$, then the boundary value analysis of the function selects the test input values l, $l + \epsilon$, i_n, $h - \epsilon$, and h, where i_n is a nominal value of i somewhere in the range, and ϵ is a small value, typically 1 for an integer.

Robustness Testing

- A simple extension of the basic boundary value analysis includes two additional input values, $l - \epsilon$ and $h + \epsilon$.

- These values fall outside the range but are close to the end values. They should result in the function implementation returning gracefully, without computing a result.

- A Java method such as `charAt` would handle these inputs by throwing an exception. It is a graceful exit because it gives the client the choice to handle this situation as appropriate.

- The test cases with robust testing of `charAt` would be:

$$-1,\ 0,\ 1,\ s/2,\ s-2,\ s-1,\ s$$

- Robustness testing uses seven test cases on a single-variable input function.

- The main focus of robustness testing is on exception handling.

- Testing with $s = null$ would fall outside the scope of testing method `charAt`. Instead, this null test would be in the scope of testing the client of `charAt`.

Multi-variable Boundary Values

Suppose you wrote a function `addHour` that, given a day and a time in hours, adds one hour to the time and outputs the resulting day and time. The days are numbered 1 (Sunday) through 7 (Saturday), and the hours are numbered 0 (midnight) through 23 (11pm). So, for instance, `addHour(1,23)` results in `(2,0)`, `addHour(7,23)` results in `(1,0)`, and `addHour(3,12)` results in `(3,13)`. How would you test this function?

- The input variables here are integer d, for day, and integer h for hour, with the ranges $1 \leq d \leq 7$ and $0 \leq h \leq 23$.

- To create the test cases, we follow the procedure for single-variable boundary value analysis, except we repeat for both variables, and when we run through the gamut for one variable, we keep the value of the other fixed at the nominal value. We'll use d_n and h_n to denote nominal values for day and hour, respectively.

- Then the list of test cases, using (day,hour) tuples, are:

$$(d_n, 0), (d_n, 1), (d_n, h_n), (d_n, 22), (d_n, 23),$$

$$(1, h_n), (2, h_n), (d_n, h_n), (6, h_n), (7, h_n)$$

  One test case, (d_n, h_n) appears twice, so there are nine distinct test cases.

- Using $d_n = 4$ and $h_n = 12$, we have the following actual test cases:
  $(4, 0), (4, 1), (4, 12), (4, 22), (4, 23), (1, 12), (2, 12), (6, 12), (7, 12)$

- Robustness testing will need four more test cases:
  $(d_n, -1), (d_n, 24), (0, h_n), (8, h_n)$,
  i.e. $(4, -1), (4, 24), (0, 12), (8, 12)$.

Multi-variable Boundary Values

- However, this general approach does not capture the important test cases $(1, 23)$ and $(7, 23)$ mentioned at the beginning of this discussion.

- The reason is the general approach followed above only works if the variables are *independent*. That is, if there are n independent variables, then the test cases are built by varying *each independent variable* through all its test values, while holding *all the other variables* at their respective nominal values.

- If the variables are dependent on each other, then we need to use all value tuples obtained by combining the test values of the variables in all possible ways.

- For the `addHour` examples, this would mean all possible combinations of d boundary values with those of h boundary values.

- The test values for d are $1, 2, 4, 6, 7$ and those for h are $0, 1, 12, 22, 23$, so there are 25 combinations, each of which will be a test case.

- These combinations will include $1, 23$ and $7, 23$ among others.

- Robustness testing does not get affected due the inter-dependence unless there are particular legitimate individual values whose combination is not legitimate, i.e. the combination is an error-producing case.

Equivalence Classes

The boundary values approach results in a lot of test cases, many of which are repetitious/redundant:

Test	d	h
1	1	0
2	1	1
3	1	12
4	1	22
5	1	23
6	2	0
7	2	1
.	.	.
.	.	.
.	.	.
24	7	22
25	7	23

Consider each group of tests with the same d and different h's. In the first group of 5 tests, with $d = 1$, all tests except the last – (1,23) – result in the same kind of output, i.e. one in which the output h is one more than the input h, and the output d is the same as the input d.

Why not break up the tests into *categories* such that all tests in each category are similar in the kind of output that is produced, and the logical grouping of tests in a category is different from the grouping in any other?

Equivalence Classes

- **Category 1**: All tests for which the output d is the same as the input d.

$$C1 = \{(d, h) \mid h \neq 23\}$$

- **Category 2**: All tests for which the output d is different from the input d, but not 1.

$$C1 = \{(d, h) \mid d \neq 7,\ h = 23\}$$

- **Category 3**: All tests for which the output d is different from the input d, and is 1.

$$C1 = \{(7, 23)\}$$

For each category in which there is more than one test case, the boundary values approach can be followed to pick test cases:

- **Category 1**: $\{(4,0),(4,1),(4,12),(4,21),(4,22)\}$
- **Category 2**: $\{(1,23),(2,23),(4,23),(5,23),(6,23)\}$

Each category is called an *equivalence class*, and the categorization is called *equivalence partitioning*. Equivalence partitioning reduces the number of test cases compared to the boundary values approach, with comparable effectiveness in testing. (At times, strict boundary values are used to pick multiple test cases from each category.)

Equivalence partitioning can also be done for test cases that are expected to result in errors, for robustness testing.

Equivalence Classes

Say there is a function called `pointInRectangle`. Given a rectangle of width w, height h, upper left corner (x, y), and a point (p_x, p_y), this function returns 0 if the point is on the rectangle, -1 if it is outside, and 1, if it is inside. How will you test if this function works correctly? All inputs are integers. For now, assume that all inputs except w and h are non-negative, and w and h are greater than 0. (In this coordinate system, the origin is at the top-left corner, with the x-axis going rightward, and the y-axis going downward.)

- This function does not have a continuum of results. Instead, it partitions the result into three classes of *outcomes*.
- This then leads to partioning the input into equivalence classes.
- Each equivalence class would comprise all inputs that result in one of the possible outcomes, or force one type of error.
- Since we are assuming all inputs are legal coordinates, to start with, we will build equivalence classes for the possible correct outcomes.
- To do this, we need to look at all possible categories of where the point can be with respect to the rectangle for each of the possible outcomes.

Equivalence Classes

The classes of inputs for all possible outcomes are:

- The point can be on one of the four sides, but not at the corners.
- It can be at one of the four corners.
- It can be inside.
- It can be outside (left, right, above, below).

Here are the equivalance classes for these point position categories:

1. $\{(x, y, w, h, p_x, p_y) \mid x < p_x < x + w, p_y = y\}$
2. $\{(x, y, w, h, p_x, p_y) \mid x < p_x < x + w, p_y = y + h\}$
3. $\{(x, y, w, h, p_x, p_y) \mid p_x = x, y < p_y < y + h\}$
4. $\{(x, y, w, h, p_x, p_y) \mid p_x = x + w, y < p_y < y + h\}$
5. $\{(x, y, w, h, p_x, p_y) \mid p_x = x, p_y = y\}$
6. $\{(x, y, w, h, p_x, p_y) \mid p_x = x + w, p_y = y\}$
7. $\{(x, y, w, h, p_x, p_y) \mid p_x = x, p_y = y + h\}$
8. $\{(x, y, w, h, p_x, p_y) \mid p_x = x + w, p_y = y + h\}$
9. $\{(x, y, w, h, p_x, p_y) \mid x < p_x < x + w, y < p_y < y + h\}$
10. $\{(x, y, w, h, p_x, p_y) \mid p_x < x\}$
11. $\{(x, y, w, h, p_x, p_y) \mid p_x > x + w\}$
12. $\{(x, y, w, h, p_x, p_y) \mid p_y < y\}$
13. $\{(x, y, w, h, p_x, p_y) \mid p_y > y + h\}$

- Note that the class for when the point is inside the rectangle assumes that $w \geq 2$ and $h \geq 2$. (Without this assumption, this will fall into an equivalence class that exercises an error - we will consider errors after we finish with the error-free cases.)

Equivalence Classes - Valid situations

- For each equivalence class, we need to generate at least one test case.
- If an equivalence class has more than one possible case, choose cases using the boundary values approach.

Here is a sample set of test cases corresponding to the equivalence classes, taking $x = y = 3$ and $w = h = 5$ (Add cases for a full boundary value treatment for each class):

1. $(3, 3, 5, 5, 4, 3)$, $(3, 3, 5, 5, 7, 3)$
2. $(3, 3, 5, 5, 4, 8)$, $(3, 3, 5, 5, 7, 8)$
3. $(3, 3, 5, 5, 3, 4)$, $(3, 3, 5, 5, 3, 7)$
4. $(3, 3, 5, 5, 8, 4)$, $(3, 3, 5, 5, 8, 7)$
5. $(3, 3, 5, 5, 3, 3)$
6. $(3, 3, 5, 5, 8, 3)$
7. $(3, 3, 5, 5, 3, 8)$
8. $(3, 3, 5, 5, 8, 8)$
9. $(3, 3, 5, 5, 4, 4)$, $(3, 3, 5, 5, 7, 4)$, $(3, 3, 5, 5, 4, 7)$, $(3, 3, 5, 5, 7, 7)$
10. $(3, 3, 5, 5, 2, 3)$, $(3, 3, 5, 5, 2, 8)$
11. $(3, 3, 5, 5, 9, 3)$, $(3, 3, 5, 5, 9, 8)$
12. $(3, 3, 5, 5, 3, 2)$, $(3, 3, 5, 5, 8, 2)$
13. $(3, 3, 5, 5, 3, 9)$, $(3, 3, 5, 5, 8, 9)$

Equivalence Classes - Invalid situations

To the above valid situations, we need to add test cases for the invalid situations:

- One of x, y could be negative.
- One of w, h could be zero or less.
- One of p_x, p_y could be negative.

1. $\{(x, y, w, h, p_x, p_y) \mid x < 0, y, p_x, p_y \geq 0, w, h > 0\}$
2. $\{(x, y, w, h, p_x, p_y) \mid y < 0, x, p_x, p_y \geq 0, w, h > 0\}$
3. $\{(x, y, w, h, p_x, p_y) \mid w \leq 0, x, y, p_x, p_y \geq 0, h > 0\}$
4. $\{(x, y, w, h, p_x, p_y) \mid h \leq 0, x, y, p_x, p_y \geq 0, w > 0\}$
5. $\{(x, y, w, h, p_x, p_y) \mid p_x < 0, x, y, p_y \geq 0, w, h > 0\}$
6. $\{(x, y, w, h, p_x, p_y) \mid p_y < 0, x, y, p_x \geq 0, w, h > 0\}$

Test cases can be generated for each of these equivalence classes as we did for the valid input equivalence classes.

Equivalence Classes - Binary Search

How would you test a function that implements binary search on a sorted array, returning true if found, and false otherwise?

- Two outcomes, *true* or *false*. Assume no duplicates in array.
- Note that for the *true* outcome, there are exactly n test cases. However, for the *false* outcome, there are $n + 1$ equivalence classes, because each failure position (before first item, after last item, between any two items for each pair of successive items) is a collection of possible values.
- If the size of the array is n, then there are n cases in the equivalence class for outcome *true*, and $n + 1$ equivalence classes for outcome *false*.
- For *true*, there would be n test cases.
- For *false*, there would be at least one test case per equivalence class, using boundary values for each class. For example, if the first item is 10, and the second is 16, then for the corresponding *false* outcome, the equivalence class would be all values in the range $11 - -15$. Using boundary values would give the test cases 11 and 15 (boundaries), 13 (nominal), and 12 and 14 (one off the boundaries).

Can black-box testing (can't see the source code) validate whether the implementation is in fact *binary* search, and not some other kind that works correctly?

Equivalence Classes - Insertion Sort

How would you test a function that implements insertion sort on an array?

- If the size of the array is n, there are $n!$ equivalence classes (permutations of input set), assuming no duplicates. This is an impractical test if n is large. Which raises the question: is there a small set of small n's for which the implementation can be tested, and we can be confident that we do not need to test for other n's?

- For instance, say we test for $n = 3$ and all test cases show the implementation works correctly. Is there anything different about testing with $n > 3$? There should not be, for insertion sort.

- The small n cases form a kind of boundary where things may break. So at the top level, we can test for $n = 1$, $n = 2$, $n = 3$ and be done.

- For $n = 1$, there is only one equivalence class: $\{x\}$, i.e. any value x.

- For $n = 2$, denoting the values as x and y, there are two equivalence classes: $\{(x, y) \mid x < y\}$, and $\{(x, y) \mid y < x\}$.

- For $n = 3$, denoting the values as x, y, and z, there are six equivalence classes: $\{(x, y, z) \mid x < y < z\}$, $\{(x, y, z) \mid x < z < y\}$, $\{(x, y, z) \mid y < x < z\}$, $\{(x, y, z) \mid y < z < x\}$, $\{(x, y, z) \mid z < x < y\}$, $\{(x, y, z) \mid z < y < x\}$.

Equivalence Classes - Insertion Sort

- If we assume that duplicate keys are permitted, it will result in a larger number of equivalence classes. For instance, the original class

  $\{(x, y, z) \mid x < y < z\}$

  will now break up into the classes:

  $\{(x, y, z) \mid x = y = z\}$,
  $\{(x, y, z) \mid x = y < z\}$,
  $\{(x, y, z) \mid x < y = z\}$, and
  $\{(x, y, z) \mid x < y < z\}$.

- The other original equivalence classes will also break down into "component" classes, but some of these may be duplicates, and should be discarded. For instance, each original class will give

  $\{(x, y, z) \mid x = y = z\}$,

  and we only retain one of these copies.

Can black-box testing validate whether the implementation is in fact *insertion* sort, and not some other kind that works correctly?