Topic 8

Software Testing

Part 3

Functional Testing

- Structural testing – white box
  - test as much of path coverage as possible
  - can only test code that exists
    - may have missed several requirements
    - but can still achieve high coverage
- Functional testing – black box
  - Make sure the code meets the requirements

Functional Testing

- Must consider unstated requirements
  - like error handling
  - ease of use
- Ideally, we should go through all requirements systematically and develop tests for them
  - Standard methods of doing so exist

Functional Testing Example

- Triangle Analyzer Requirements
  - Program prompts user for input
  - User enters three real numbers, separated by commas
    - e.g. 2.5, 6, 6.5
  - Program responds with:
    - Equilateral: sides define equilateral triangle
    - Isosceles: likewise
    - Scalene: likewise
    - Not a triangle: no valid triangle with those side lengths
      - (e.g. 3, 4, 1000 since 3 + 4 >= 1000 [triangle inequality])

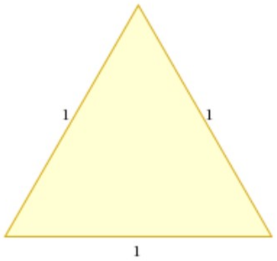
Functional Testing Example

- What tests cases will we use?
  - These will be functional (black-box) test cases, since we are working only from the requirements

Functional Testing Example

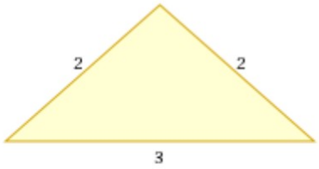
Test Case 1: Equilateral triangle

`1,1,1`



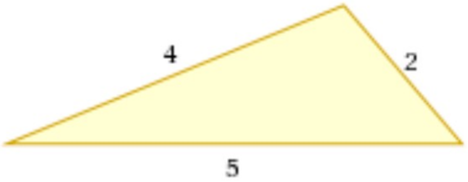
Test Case 2: Isosceles triangle

`3,2,2`



Test Case 3: Scalene triangle

`5,2,4`



Test Case 4: Not a triangle

`7,2,4`

Functional Testing Example

- Probably want to test with all possible combinations of each test as well
  - (2,4,5) (2,5,4) (4,2,5) (4,5,2) (5,2,4) (5,4,2)
- Need to consider unstated requirements as well:
  - Invalid input characters: 1, a, 2
  - Invalid number of inputs 1, 2
  - Invalid input format: 1 2 3
  - etc.

Functional Testing

- How to choose good functional test cases?
- We will look at each of the following :
  - Test all possible outputs
  - Test both valid and invalid inputs
  - Test around boundaries
  - Test extreme values
  - Test input syntax
  - Guess at possible errors

Functional Testing

- These techniques will generate lots of test cases
  - A test case may be one input, or a sequence of inputs (depending on the program)
  - Often more test cases for erroneous input than for valid input (unhappy paths)

Test all possible outputs

- For each possible output specified
  - Write a test case that will produce that kind of output

- Examples:
  - Triangle analyzer
    - One case for each of Equilateral, Isosceles, Scalene, and Not a Triangle
  - Parking garage simulator
    - A case for parking a car; one for retrieving a car
    - Should also have cases for garage full and no such car

Test valid and invalid inputs

- Often, an individual input x to a program has:
  - A valid range like x>=0, or 1<=x<=12
  - valid set of values
    - x is a string of alphanumeric characters
    - x $\in$ {red, green, blue}
    - Inputs outside these ranges/sets are invalid.

- For each input to the program:
  - Test at least one valid value
  - Test at least one invalid value
  - Test invalid values near boundaries of range (-1 and 13)

Test valid and invalid inputs

- Individual inputs to a program can include:
  - Things types in to a console or a GUI control
  - Command-line options
  - Values in configuration files
  - etc.
- Examples of invalid inputs:
  - Triangle analyzer: Test -1 or Z as the length of a side
  - Day planner program: Enter Jqx as the name of a month

Testing near boundaries

- Failures often occur close to boundaries
  - Boundaries between different kinds of output
  - Boundaries between valid and invalid inputs
- Such failures are often due to faults such as
  - Errors in arithmetic
  - Using <= instead of <
  - Not initializing a loop properly

We should test at and/or around boundaries

Test boundaries

- Triangle analyzer
  - Test case: 2, 2, 4.00001 (almost, but not a triangle)
  - Test case: 2, 2, 4 (right on boundary)

- Pop machine dispenser software:
  - If the user has exactly enough money to buy a can it should be dispensed
    - but does not require change
    - If the program erroneously contains a test like:
      - `if (balance > cost)`
      - this will not be allowed (since the test should be >=)
  - Here, a boundary test will find and test this situation

Test extreme values

- Software may not handle very large or very small values correctly due to things like
  - Buffer or arithmetic overflows
  - Mistaken assumptions that a string will be non-empty

- These can easily crash a program

Test extreme values

Examples:

- With just about any program that accepts user input:
  - Empty strings
  - Very long strings

Ex. Triangle analyzer:

4321432134, 543234344, 6566765888 (very large)

0.00000003, 0.00000008, 0.00000005 (very small)

Test input syntax

- Something omitted: 5, 12 13 (comma missing)
- Too few/many values: 5, 12 or 5, 12, 13, 20
- Invalid tokens: 5, 12, qwe

- In these situations
  - Program should not just crash
  - Give informative error message, and recover if possible
  - Do not just accept and process as if correct
    - Ex. for too many inputs: just process the first 3 valid ones (undesirable)
    - can be even worse than just crashing, as user does not know what is happening
  - Do not silently deny
    - let user know what is happening

Guess at faults

- Finally, use intuition to think of how a program might be wrong:
  - Might be better to get person A to think of possible faults in person B's code

- Example with the triangle analyzer:
  - To see whether a triangle is isosceles, the code must test all three distinct pairs amongst the three numbers for equality:
  - What if not all three pairs have been tested by the code?
  - Thus, test all of 2,2,3; 2,3,2; and 3,2,2

Functional vs. Structural Testing

- Functional (black-box) testing
  - Advantages
    - Ensures program meets requirements
    - Test boundaries, etc., explicitly
  - Disadvantages
    - Cannot test undocumented features
    - May not test hidden implementation details thoroughly

Functional vs. Structural Testing

- Structural (white-box) testing
  - Advantages
    - Tests all code and implementation details
    - Coverage metrics can give us an idea of the extent to which our code is tested
  - Disadvantages
    - Cannot test whether all desired features are implemented
    - We saw how 100% statement coverage essentially means nothing in relation to code quality / correctness