Software Testing

Data Selection

Exhaustive Testing

- Consider the case of testing the addition of two integers.
- To be sure that every case works, we need to try every combination of numbers.
- If we are working with 32 bit integers, each integer can have 4,294,967,296 values.
- Combining these in all possible ways will require 18,446,744,073,709,551,616 tests.
- This is **exhaustive testing**.

Smart Testing

- The problem with exhaustive testing is the majority of these tests are duplicates.
- Smart testing says:
  - select the test data to eliminate the duplication and
  - reduce the number of tests down to a manageable number.

Black Box Testing

- Black box test data should
  - pick values that are not duplicates of other tests and
  - test the areas where errors commonly occur.
  - Testing 3 + 3 differs little from 7 + 7 since they are both the same values and they are both odd values.
- To test different values we would try:
  - adding two identical even values (eg. 12 + 12),
  - adding an even and an odd value (eg. 14 + 19),
  - adding an odd and an even value (eg. 23 + 12).

Black Box Testing

- We then need to test special values
- Special values are those where errors frequently occur such as:
  - transition points from one area of data to another
    - (moving from positive to negative values)
  - end points of the data values
    - (the maximum and minimum integer values),
  - special values (0 being a special value for addition),
  - values immediately adjacent other special values.

Black Box Testing

- For our example of adding two numbers we could add:
  - adding 0 to a number (12 + 0, -13 + 0)
  - adding 1 to a number (12 + 1, -13 + 1)
  - adding -1 to a number (12 + -1, -13 + -1)
  - adding MAXINT to a number (12 + MAXINT, -13 + MAXINT)
  - adding MININT to a number (12 + MININT, -13 + MININT)
  - adding MAXINT-1 to a number (12 + (MAXINT-1), -13 + (MAXINT-1))
  - adding MININT+1 to a number (12 + (MININT+1), -13 + (MININT+1))

Black Box Testing

- If we add these all up, we now have a total of 17 different tests.
- If the program can add all of these correctly, we have a lot of confidence that the software works as expected.
- This is a huge reduction in the number of tests compared to exhaustive testing

String Concatenation

- Check the edge conditions for empty strings
  - concat("", ""),
  - concat("", "a"),
  - concat("", "abc"),
  - concat ("a", ""),
  - concat("abc", "")
- concatenate routine values with a single character in one string
  - concat("a", "a")
  - concat("a", "abc")
  - concat("abc", "x")

String Concatenation

- routine longer strings that do not exceed the limit of the allocated array size of 16
  - concat("abcdefg", "hijk"),
  - concat("abcdefg", "hijklmn")
- a string that is the longest possible
  - concat("012345678901234", "x")
- a string that is longer than the allocated storage to prove it will fail
  - concat("012345678901234", "xy")

White Box Testing

- White box tests which are added to the test suite after examining the code.
- Reading the code, we also see that it can handle either of the parameters being NULL.
- To test these cases we need to add:
  - concat(NULL, "a")
  - concat(tmp, NULL)

Splitting text into words

```
int split(char line[], char words[][MAX_WORD_LEN + 1], int maxWords)
{
        char ch;
        char buf[MAX_WORD_LEN + 1] = { 0 };
        int lp = 0, bp = 0, wp = 0, result = 0;

        ch = line[lp];
        while( ch != '\0' && result >= 0)
        {
                while (ch != '\0' && (ch == ' ' || ch == '\t')) ch = line[++lp];

                bp = 0;
                while (ch != '\0' && !(ch == ' ' || ch == '\t'))
                {
                        buf[bp++]  = ch;
                        ch = line[++lp];
                }
                if (bp > 0)
                {
                        buf[bp]  = '\0';
                        if (wp >= maxWords)
                        {
                                result = -1;
                        }
                        else
                        {
                                strcpy(words[wp++],  buf);
                        }
                }
        }
        result = (result < 0) ? result : wp;
        return result;
}
```

Splitting text into words

- We test with
  - `char line[] = { " The quick\t brown fox " };`
- The test string tests for:
  - blanks at the begging and ending of the string,
  - single blanks between words,
  - multiple blanks between words,
  - a combination of blanks and tabs between words.
- This is not comprehensive testing but provides some confidence that the function works correctly.

Splitting text into words

- When we read the code, we will find that:
- there is a situation where it does not store a word into the word list if the variable **bp** is zero.
- this can only happen if there is the last letter in a word, immediately followed by the string terminator.
- This means, to test for this case, we need to include a test where there is no space or tab after the last word in the string.
- Examining the code further, we see that it checks to see if the number of words in the words array is less than the maximum size of the array before it adds the word into the array.
- To make sure this code works correctly, we need to add a test which will exceed the side of the array to make sure it handles this situation correctly.

Splitting text into words

- White box testing adds new test cases that would be missed if the code had not been examined.
- White box tests usually test the situations that rarely happen.
- Without these tests, we would have no idea whether this code works