**Test Data Selection**

You can test hard or you can test smart. Testing hard is known as *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.

The problem with exhaustive testing is the vast majority of these tests are just duplicates of other tests. Smart testing says we should select the test data more carefully to eliminate the duplication and reduce the number of tests down to a manageable number.

This type of test data selection is for black box testing. White box testing will add onto the black box test values to ensure that every path through the software has been properly tested. This will add test values that drive the code through parts of the code that would not be executed by the black box testing data.

**Black Box Data Selection**[**​**](https://software-testing.sdds.ca/B-DataSelection2/data-selection-1#black-box-data-selection)

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),
* adding two identical odd values (eg. 23 + 23).

This tests adding routine values but does nothing to test the 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 to other special values.

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))

If we add these all up, we now have a total of 18 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 but provides the same level of confidence that the software works.

Now, let's consider the case of concatenating two strings. We have written our own concatenation function and want to test it to make sure it works before we use it in our programs. Take a look at the program:

void concat(char dest[], const char src[])  
 {  
 int dp = 0, sp = 0;  
 if (dest != NULL)  
 {  
 for (dp = 0; dest[dp] != '\0'; dp++);  
 if (src != NULL)  
 {  
 for (sp = 0; src[sp] != '\0'; sp++)  
 {  
 dest[dp++] = src[sp];  
 }  
 dest[dp] = '\0';  
 }  
 }  
 }

Copy

This code is designed to concatenate the src string onto the end of the dest string and work in every situation. To test this we need to:

* 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")
* 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")

Those tests cover all of the black box cases. Reading the code, we also see that it can handle either of the parameters being NULL. These are special cases that need to be tested. If the handling of NULL had not been in the specification for the funtion, we would only find it by inspecting the code and it would be a white box test. To do this test, we add:

* {concat(NULL, "a"),
* concat(tmp, NULL) }.

In the actual tests I created, I ended up with 15 tests for string concatenation. This gives high confidence the function works and is much shorter than exhaustive testing.

**White Box Testing**[**​**](https://software-testing.sdds.ca/B-DataSelection2/data-selection-1#white-box-testing)

The use of the NULL values in the example above is an example of *white box testing*. White box tests are those that are not stated in the requirements of the software but are found by reading the code. Often, programmers write code to cover unusual situations that were not placed in the requirements. In the case above, the programmer handled the cases of NULL being passed but this was likely not in the original requirements. Test cases for these situations can only be found by reading the code.

White box testing describes tests which are added to the test suite after examining the code. They are called *white box* because the code is no longer treated as a black box but needs to be opened up to see how it works inside. In the next example, we will also see some test cases that could be missed unless the code was examined carefully.

#define \_CRT\_SECURE\_NO\_WARNINGS  
#include <stdio.h>  
#include <string.h>  
  
#define MAX\_WORD\_LEN 20  
  
  
  
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;  
}  
  
int main(void)  
{  
 char line[] = { " The quick\t brown fox " };  
 char words[10][MAX\_WORD\_LEN + 1] = { 0 };  
 int nwords, i;  
  
 nwords = split(line, words, 10);  
  
 for (i = 0; i < nwords; i++)  
 {  
 printf("%s\n", words[i]);  
 }  
  
 return 0;  
}

Copy

This code splits a line of test into words and then returns the words as an array of strings. The function returns either a -1 if the maximum number of words has been exceeded or the number of words that were placed in the array.

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 does provide some confidence that the function works correctly. 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. **bp** is the counter for the number of letters in the word and we do not save the word unless there are some letters in it. While this makes sense, it raises the question of how we can have a word with no letters in it. After giving this some thought, you realize this can only happen if the last letter in a word is 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 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.

White box testing adds new test cases that would often 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 and might find that it fails in a production system when the rare condition actually happens.