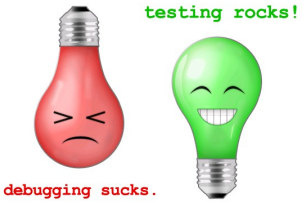

Testing on the Toilet
Data-Driven Traps!

September 4, 2008

When writing a unit test, it is tempting to exercise many scenarios by writing a data-driven test. For example, to test the function `IsWord`, you could write (`ARRAYSIZE` is a macro that returns the number of elements in an array):

```
const struct {const char* word; bool is_word;} test_data[] = {
  {"milk", true}, {"centre", false}, {"jklm", false}, };

TEST(IsWordTest, TestEverything) {
  for (int i = 0; i < ARRAYSIZE(test_data); i++)
    EXPECT_EQ(test_data[i].is_word, IsWord(test_data[i].word));
}
```

This keeps the test code short and makes it easy to add new tests but makes it hard to identify a failing test assertion (and to get the debugger to stop in the right place). As your code grows the test data tends to grow **faster than linearly**. For example, if you add a parameter called `locale` to `IsWord`, the test could become:

```
Locale LOCALES[] = { Word::US, Word::UK, Word::France, ... };
const struct {const char* word; bool is_word[NUM_LOCALES];} test_data[] = {
  {"milk", {true, true, false, ...}},  // one bool per language
  {"centre", {false, true, true, ...}},
  {"jklm", {false, false, false, ...}}
};

TEST(IsWordTest, TestEverything) {
  for (int i = 0; i < ARRAYSIZE(test_data); i++)
    for (int j = 0; j < ARRAYSIZE(LOCALES); j++)
      EXPECT_EQ(test_data[i].is_word[j], IsWord(test_data[i].word, LOCALES[i]));
}
```

The change was relatively easy to make: change the data structure, fill in the boolean values for other locales and add a loop to the test code. But even this small changed has made the test harder to read and slower as it repeats potentially unnecessary checks. In addition, both the test AND the code have changed. How do you know the test is not broken? (Actually, it is broken. Can you see the bug?) By contrast, **keeping the data in the test** gives us:

```
TEST(IsWordTest, IsWordInMultipleLocales) {
  EXPECT_TRUE(IsWord("milk", Word::UK));
  EXPECT_TRUE(IsWord("milk", Word::US));
  EXPECT_FALSE(IsWord("milk", Word::France));
}

TEST(IsWordTest, IsWordWithNonExistentWord) {   // 'jklm' test is not repeated
  EXPECT_FALSE(IsWord("jklm", Word::US));       // as it uses the same code path
}
```

The difference between these two code snippets is minor but real-life data-driven tests quickly become **unmanageable**. A complete example would not fit on this page but if you look at your code base, you will find a few specimens lurking in some (not-so) forgotten test classes. They can be identified by their large size, vague names and the fact that they provide little to no information about why they fail.

More information, discussion, and archives:
http://googletesting.blogspot.com