The magazine of the ACCU

www.accu.org

{cvu}

Volume 23 • Issue 1 • March 2011 • £3

{cvu}
Volume 23 Issue 1
March 2011
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Stephen Baynes, Alexander Demin, Pete Goodliffe, Paul Grenyer, Richard Harris, Jon Jagger, Frances Love, Chris Oldwood, Roger Orr, Nat Pryce, Matthew Wilson

ACCU Chair
Hubert Matthews
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
Mick Brooks
accumembership@accu.org

ACCU Treasurer
Stewart Brodie
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

What was, what is, and what may be

Jane was becoming increasingly confused. Since joining her team, she'd had a variety of little 'missions' to complete, all working towards the release of a replacement product, and (she had to admit) they'd even been *interesting* missions, technologically speaking. She'd even felt she'd had some influence on the design of the new system. Nevertheless, she was now finding it hard to escape a feeling of futility about it.

It seemed that the project was becoming bogged-down in constantly re-visiting details of design that she (and her colleagues, she felt sure) had already thought decided. No-one had a very clear idea of how the different components should communicate together, or even what those components should be. There was no clarity on some fundamental things like the domain types to be used, or the persistence mechanism for them. In short, a lack of concrete requirements. Oh sure, there was a broad agreement about what the system should 'do' – it must be be like the old system, but shinier and faster. And more extensible.

She suspected it was this last part that was causing all the trouble. Arequirements vacuum had inspired a universe of possibility in which all things were possible. Instead of 'just' replacing the existing system, it had to be able to support other output styles, manipulate new data types, received from as-yet-non-existent sources, all dynamically configurable (of course). The possibilities were endless.

Yes. Jane was **sure** that was the problem. Instead of actually asking people what they *required*, the project had become encumbered and paralysed by the dreams of *what might be*.

The question now was – what should she **do** about it?

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

The articles in this magazine have all been written by programmers, for programmers – and have been contributed free of charge.

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 23.2:	1st April 2011
C Vu 23.3:	1st June 2011
Overload 103:	1st May 2011
Overload 104:	1st July 2011

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS

The First Little Step into Test-Driven Development

Alexander Demin takes a good look at Google Test.

The software development world is changing rapidly – new versions of the operating systems, compilers, libraries are coming up faster and faster. It's actually great. Lots of options allow you to choose the development tools ideally fitting your personal requirements. Approaches to developing good quality software are also changing all the time. Nowadays the cool words in the programming world are object oriented design, functional programming, extreme programming and of course test-driven development (TDD).

Though I have more than ten years' experience of programming, and it covers various languages from machine code and assembler up to functional programming, I have discovered the test driven development world quite recently. Programmers are often very conservative (and quite lazy!) and they do not like to change their habits. I am a perfect example. But when I stepped over my laziness and started to use TDD I felt that my development became more predictable, more stable. I managed to split complex tasks into pieces, and manage code interdependencies significantly more easily and faster. More importantly: I have stopped repeating my coding mistakes, reintroducing already fixed bugs and now I am able to refactor my code anytime without any fear of breaking something important a day before the release. Why? All thanks to test driven development.

I would like to share my experiences on entering the wonderful world of TDD and hope to encourage somebody to join.

My main background is C and C++, so I will cover these languages, but all ideas mentioned are common for lots of modern languages (Java, C#, Python, Delphi etc).

Let's start from the beginning. Usually the first program written by a newbie is Hello World. Assume you have done it already and you want to do something more complex.

Let's assume you studied a lot of computer science and you know how to implement a very fast multiplication function. Listing 1 is what it might look like.

I want to warn the reader that this particular example is not ideal in terms of coding style and it's not clear in logic, it uses a lot of C/C++ 'cool short' expressions and so on. Also the function has some weird line with 920 and 847. This is intentional, and will be covered later.

Listing 1

```
// File: mult.h
#ifndef _MULT_H
#define _MULT_H
int mult(int a, int b);
#endif

// File: mult.cc
#include "mult.h"
int mult(int a, int b) {
  if (!a || !b) return 0;
  int r = 0;
  if (a == 920 && b == 847) r++;
  do {
    if (b & 1) r += a;
    a <<= 1;
  } while (b >>= 1);
  return r;
}
```

Listing 2

```
#include "mult.h"
#include <iostream>

int main(int argc, char* argv[]) {
  while(1) {
    std::cout << "enter a: ";
    int a;
    std::cin >> a;
    std::cout << "enter b: ";
    int b;
    std::cin >> b;
    std::cout << "a * b = " << mult(a, b)
       << std::endl;
  }
}
```

Now, you have done the code. You definitely know that it should work more reliably and faster because your computer science background tells you that. How can you make sure that it works correctly? The function code is quite 'non-understandable' and you cannot swear that it works correctly just by looking on the source. You have to try it on. The first and the most obvious way to create a simple example might be that shown in Listing 2.

Then you run it, play with it a bit, try a couple of examples and then come to the conclusion that it works. Later you add the `mult.cc` file to your project and probably delete the test example source because you do not need it anymore. You have linked the function into your application and you are almost happy.

Let's step back for a second now and imagine that unfortunately sometimes your application gives a wrong result or perhaps crashes and you suspect that the issue is your **`mult()`** function. You have to find your original test source or even write it again because you have lost it, then run it again under a debugger and try to find what the problem is. And now imagine you have hundreds or thousands of similar functions in your application and you have to re-test them all. It's a nightmare.

Well, let me show you another way – the test driven development way. We will use the excellent Google Test Framework 1.5.0 for that. You can download and unpack it in your working directory:

```
wget   http://googletest.googlecode.com/files/
gtest-1.5.0.tar.gz
gzip -dc gtest-1.5.0.tar.gz | tar xvf -
```

It will create `gtest-1.5.0` directory in your current folder. We will refer to this directory below so make sure that you use proper directory names in your compilation commands.

Then you create the unit test (`mult_unittest.cc`, in Listing 3).

ALEXANDER DEMIN

Alexander Demin is a software engineer with a PhD in Computer Science. Constantly exploring new technologies he is always ready to drill down into the code with a disassembler to prove that the bug is out there. He may be contacted at alexander@demin.ws.

Listing 3

```
#include <gtest/gtest.h>
#include "mult.h"

TEST(multTest, simple) {
  EXPECT_EQ(91, mult(7, 13));
}
```

Listing 4

```
#include <gtest/gtest.h>

int main(int argc, char **argv) {
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

This file contains the simple test case. The meaning of it is explained below.

Then the test main runner module (`runner.cc`, in Listing 4).

This runner will execute all declared tests in your test application. This piece of code can be almost the same for any of your unit test suites. It just parses the command line arguments and runs all tests.

Now let's compile it. If you are running Linux and have the GCC C++ compiler version 3 or later you can use the following command:

```
g++ -Igtest-1.5.0/include -Igtest-1.5.0 -o
mult_unittest gtest-1.5.0/src/gtest-all.cc
mult.cc mult_unittest.cc runner.cc
```

The mult_unittest executable should be generated. Let's run it:

```
./mult_unittest
```

It prints something like this:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from multTest
[ RUN      ] multTest.simple
[       OK ] multTest.simple
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[  PASSED  ] 1 test.
```

Let's go back and look at it in more detail now. We have created a test case named **`multTest.simple`** in the file `mult_unittest.cc` (**`multTest`** is the test suite name and the **`simple`** is the test name in the suite) which runs your function with 7 and 13 as the parameters and checks that result is 91. The macro for the test declaration is **`TEST(...)`**. The magic happens in the **`EXPECT_EQ (...)`**. This function call has two arguments: the first one is the expected value and the second is the real one. If they are equal the function passes through quietly but if they are different it reports an error message.

The Google Test Framework provides a bunch of similar functions to check various conditions with different argument types. The **`EXPECT_*`** function family does not abort the test run. It just prints the report about a test failure and keeps going to execute other tests. The **`ASSERT_*`** functions (for example, **`ASSERT_EQ()`**) stop the test suite run and terminate the runner. They are convenient when there is no reason to continue testing on a fatal error (for example, a database is not available).

But in our case the test runner reports a successful test execution – the test case has been executed and the result is correct. That's fine but this test case is so obvious and checks only one pair of numbers. You need more. Because the **`mult()`** function has some weird checking of the argument for zero at the beginning let's test it. You add one more test case – **`multTest.zero`** (File: `mult_unittest.cc`, Listing 5).

Let's compile with the same command and run **`mult_unittest`** executable again. It should print this:

```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from multTest
[ RUN      ] multTest.simple
[       OK ] multTest.simple
[ RUN      ] multTest.zero
[       OK ] multTest.zero
[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran.
[  PASSED  ] 2 tests.
```

The new test passes successfully as well and the **`mult()`** function seems to handle checking the parameter for zero correctly. But we still have an unsolved issue – your application using the function **`mult()`** fails and it means this function sometime returns wrong value. Let's add a stronger test to file `mult_unittest.cc` (Listing 6).

This test (**`multTest.all`**) checks all possible values of arguments from 0 to 999. Let's compile and run it again:

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from multTest
[ RUN      ] multTest.simple
[       OK ] multTest.simple
[ RUN      ] multTest.zero
[       OK ] multTest.zero
[ RUN      ] multTest.all
mult_unittest.cc:18: Failure
Value of: mult(a, b)
  Actual: 779241
Expected: a * b
Which is: 779240
[  FAILED  ] multTest.all
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran.
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] multTest.all

 1 FAILED TEST
```

Listing 5

```
#include <gtest/gtest.h>
#include "mult.h"

TEST(multTest, simple) {
  EXPECT_EQ(91, mult(7, 13));
}

TEST(multTest, zero) {
  EXPECT_EQ(0, mult(0, 7));
  EXPECT_EQ(0, mult(7, 0));
}
```

Listing 6

```
#include <gtest/gtest.h>
#include "mult.h"

TEST(multTest, simple) {
  EXPECT_EQ(91, mult(7, 13));
}

TEST(multTest, zero) {
  EXPECT_EQ(0, mult(0, 7));
  EXPECT_EQ(0, mult(7, 0));
}

TEST(multTest, all) {
  for (int a = 0; a < 1000; ++a)
    for (int b = 0; b < 1000; ++b)
      EXPECT_EQ(a * b, mult(a, b));
}
```

Listing 7

```
TEST(multTest, all) {
  for (int a = 0; a < 1000; ++a)
    for (int b = 0; b < 1000; ++b)
      EXPECT_EQ(a * b, mult(a, b))
        << "wrong result on a=" << a << " and
           b=" << b;
}
```

Wow! The test fails. It means we have found the problem. We see that in line 18 of `mult_unittest.cc` there is a test failure: the expected value is 779240 but the actual one is 779241. It's a great result, but we also need to know which exact parameters cause this error. So let's modify the test (Listing 7).

This code will also print the error message and the values of **a** and **b** on failure. The **EXPECT_EQ(...)** can be used the output stream similar to **std::cout**, for example, to print out the diagnostics on a test failure.

Compile and run it again. We should get the following result:

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from multTest
[ RUN      ] multTest.simple
[       OK ] multTest.simple
[ RUN      ] multTest.zero
[       OK ] multTest.zero
[ RUN      ] multTest.all
mult_unittest.cc:17: Failure
Value of: mult(a, b)
  Actual: 779241
Expected: a * b
Which is: 779240
wrong result on a=920 and b=847
[  FAILED  ] multTest.all
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran.
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] multTest.all

 1 FAILED TEST
```

Now we know exactly that the function fails when **a**=920 and **b**=847. This is the problem. And now we can fix the 'problem' by removing the line **if a == 920 && b == 847) r++;** from the `mult.cc` file. Listing 8 is an error free version of the `main.cc`.

Well, now compile it and run **mult_unittest** once again. Here is the output:

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from multTest
[ RUN      ] multTest.simple
[       OK ] multTest.simple
[ RUN      ] multTest.zero
[       OK ] multTest.zero
[ RUN      ] multTest.all
[       OK ] multTest.all
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran.
[  PASSED  ] 3 tests.
```

All tests work perfectly and now you are sure that your function **mult()** is fully error free.

Let's analyse what we've done. We have created the function **mult()** and also the tests which can be used any time to prove its proper functioning. At this point test driven development strongly recommends

Listing 8

```
#include "mult.h"

int mult(int a, int b) {
  if (!a || !b) return 0;
  int r = 0;
  do {
    if (b & 1) r += a;
    a <<= 1;
  } while (b >>= 1);
  return r;
}
```

you include the test build and execution into your project build. For example, this is the part of your myapp project makefile:

```
...
all: build

build:
  cc -o myapp main.cc mult.cc
```

You should add the test compilation and run into this makefile:

```
...
release: build test

build:
  g++ -o myapp main.cc mult.cc

test:
g++ -Igtest-1.5.0/include -Igtest-1.5.0 -o
mult_unittest gtest-1.5.0/src/gtest-all.cc
mult.cc mult_unittest.cc runner.cc

./mult_unittest
```

Why do you need this? You need this because each time you release the project (using release target) it will compile and run the test suite to make sure that the current implementation of the **mult()** function is ok and works as you expect.

Now imagine you want to check whether it is reasonable to use your own hacky implementation of the simple arithmetic operation as the multiplication. Let's run your test suite again using the command:

```
./mult_unittest --gtest_print_time
   --gtest_filter=multTest.all
```

We ask Google Test framework to print the test execution time and also we ask to run only one test using the filter by name.

The output:

```
Note: Google Test filter = multTest.all
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from multTest
[ RUN      ] multTest.all
[       OK ] multTest.all (1266 ms)
[----------] 1 test from multTest (1297 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1328
             ms total)
[  PASSED  ] 1 test.
```

It reports only one test run (**testMult.all**) and it takes 1279 ms on my Core 2 Duo laptop (timing on your machine may be different).

Now you want to try another fairly simple implementation for the **mult()** function (file `mult.cc`, in Listing 9).

Let's compile it using exactly the same command as we used for the first implementation:

Listing 9

```
#include "mult.h"
int mult(int a, int b) {
  return a * b;
}
```

```
g++ -Igtest-1.5.0/include -Igtest-1.5.0 -o
mult_unittest gtest-1.5.0/src/gtest-all.cc
mult.cc mult_unittest.cc runner.cc
```

and run it:

```
./mult_unittest --gtest_print_time
   --gtest_filter=multTest.all
```

The output should look like this:

```
Note: Google Test filter = multTest.all
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from multTest
[ RUN      ] multTest.all
[       OK ] multTest.all (1094 ms)
[----------] 1 test from multTest (1141 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1171
ms total)
[  PASSED  ] 1 test.
```

We see it takes only 1094ms on my laptop and it's faster than our original handmade implementation.

Now you know that the original implementation is not quite so good and may be optimized or replaced by a better one.

So what is that we have achieved by this entire exercise? What is the point of it?

Firstly, we have created a test mechanism for our function. This mechanism can be used at a later time to prove the function logic and it can be fully automated. Once created it can be re-used as many times as you want. You do not lose your efforts applied initially for creating the testing routine.

Secondly, we have included the test run into the project build. If the function logic is broken for some reason (you've changed the code accidentally or maybe the new version of the compiler has generated the wrong code) the test will automatically point you towards it by failing the build.

And thirdly, we tried two different implementations of the **mult()** function using the same test suite. This means you can easily refactor the code without any fear of breaking something. The tests will check the function results and the expectations from the function. You have determined the function behaviour via the test cases and from this point you can easily play with the function implementation. On top of this we have tested two different implementations for execution time and now we have enough information to choose the better one.

These are really awesome results – you have automated the error checking procedure for your project. You do not need to do any manual runs anymore, playing with parameters to make sure that everything works as expected after any recent changes. Let's imagine how just a little extra effort of writing a 5 minute test case (comparing to the original user interactive test application) gave us so much additional information and helped to create a better design for the application. It's definitely worth it.

There is probably an argument that in some cases testing can be tricky because real world applications are much more complex than this isolated example. That is 100% correct, however the answer to it is also very simple: you have to write testable code from the beginning. Every time a piece of code is done, ask yourself – how will I test it? And maybe you will write the code a bit more simply, a bit more split into simple sub-tasks, a bit more isolated from external dependencies and so on. Definitely writing testable code is a complicated issue and there are a lot of techniques for it: dependency injection, isolating the business logic from the object instantiation (**operator new**), using inheritance and polymorphism instead of overly complicated **if**/**switch** constructions and so on and so forth.

Listing 10

```
// File: mult.h
#ifndef _MULT_H
#define _MULT_H
int mult(int a, int b);
#endif

// File: mult.c (buggy version)
#include "mult.h"
int mult(int a, int b) {
  int r = 0;
  if (!a || !b) return 0;
  if (a == 920 && b == 847) r++;
  do {
    if (b & 1) r += a;
    a <<= 1;
  } while (b >>= 1);
  return r;
}
```

Of course I have referenced many things from the object oriented world which make it easier to use unit testing. Applications with object oriented design in most cases are quite easy to test but the classic procedural languages like C or Pascal, for example, are not out of the question either.

Let's see how to test a similar example written in ANSI C. Your sources are in Listing 10.

I will use another Google testing framework here – cmockery 0.1.2. This framework was designed to test C code and it's a very powerful framework. On top of the set of **assert_*** functions it can help to find memory leaks, and buffer under- and over-runs.

Let's get it:

```
wget http://cmockery.googlecode.com/files/
cmockery-0.1.2.tar.gz
gzip -dc cmockery-0.1.2.tar.gz | tar xvf -
```

This command will create the `cmockery-0.1.2` folder in your current directory. We will use it so do make sure you do all runs below with this as the current directory.

Let me show you the test suite with the same functionality but written in C (**mult_test.h** in Listing 11 and `mult_test.c` in Listing 12), and the runner (Listing 13).

Let's compile it with GCC version 3 or higher:

```
gcc -Icmockery-0.1.2/src/google -o mult_test
cmockery-0.1.2/src/cmockery.c mult.c mult_test.c
runner.c
```

If everything is correct you should test **mult_test** executable. Let's run it:

```
./mult_test
```

and it will print something like Listing 14.

Listing 11

```
#ifndef _MULT_TEST_H
#define _MULT_TEST_H
void mult_simple_test(void **state);
void mult_zero_test(void **state);
void mult_all_test(void **state);
#endif
```

Listing 12

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmockery.h>

void mult_simple_test(void **state) {
   assert_int_equal(91, mult(7, 13));
}

void mult_zero_test(void **state) {
   assert_int_equal(0, mult(0, 7));
   assert_int_equal(0, mult(7, 0));
}

void mult_all_test(void **state) {
  int a, b;
  for (a = 0; a < 1000; ++a)
    for (b = 0; b < 1000; ++b)
      assert_int_equal(a * b, mult(a, b));
}
```

Listing 13

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmockery.h>
#include "mult_test.h"

int main(int argc, char* argv[]) {
   const UnitTest tests[] = {
      unit_test(mult_simple_test),
      unit_test(mult_zero_test),
      unit_test(mult_all_test),
   };
   return run_tests(tests);
}
```

Listing 14

```
mult_simple_test: Starting test
mult_simple_test: Test completed successfully.
mult_zero_test: Starting test
mult_zero_test: Test completed successfully.
mult_all_test: Starting test
0xbe3e8 != 0xbe3e9
ERROR: mult_test.c:19 Failure!
mult_all_test: Test failed.
1 out of 3 tests failed!
    mult_all_test
```

The **`mult_all_test`** fails on line 19 and it reports that the expected value of multiplication is 0xBE3E8 (decimal 779240 = 920 * 847) but the actual one is 0xBE3E9 (decimal 779240). Now we *fix* the **`mult()`** function removing buggy line **`if (a == 920 && b == 847) r++;`**, giving Listing 15, an error-free version of `mult.c`., and run the test suite again.

Listing 15

```
#include "mult.h"

int mult(int a, int b) {
  int r = 0;
  if (!a || !b) return 0;
  do {
    if (b & 1) r += a;
    a <<= 1;
  } while (b >>= 1);
  return r;
}
```

Now it prints this:

```
mult_simple_test: Starting test
mult_simple_test: Test completed successfully.
mult_zero_test: Starting test
mult_zero_test: Test completed successfully.
mult_all_test: Starting test
mult_all_test: Test completed successfully.
All 3 tests passed
```

We see now all three tests work fine. Of course C-based unit testing is not as advanced and comfortable in terms of reporting or code organization. You have to declare your test cases in the header file and in the runner but this is a limitation of the C language. The cmockery framework from Google makes the most of what is technically possible for comfortable testing in C. But even if the reporting is not ideal you are always informed about which test fails and in which line.

Other languages have unit testing frameworks as well. jUnit for Java, pyUnit for Python and so on. The principles of unit testing are exactly the same – running small pieces of your application in isolation.

QA (Quality Assurance) testing and regression testing are separate big topic in themselves, and are handled differently. Good unit tests should be fast so they don't slow down the compilation process on the project. But sometimes you want to do stress testing for your code – maybe execute something millions of times, check memory allocation for leaks, create the test for a recently fixed bug to avoid its reintroduction later and so on. These kinds of tests can take a long time and it's not comfortable to run them on every project build. Here, QA and regression testing step onto the scene. It's also quite an interesting topic and I will try to cover it soon as well. ■

cqf.com

CERTIFICATE IN QUANTITATIVE FINANCE
CQF

Expand Your Mind and Career

Designed by quant expert Dr Paul Wilmott, the CQF is a practical six month-part time course that covers every gamut of quantitative finance, including derivatives, development, quantitative trading and risk management.

Find out more at **cqf.com**.

ENGINEERED FOR THE FINANCIAL MARKETS