Unit Testing in SU2: Methodology and Philosophy

Clark Pederson

11 June, 2020

Department of Mechanical Engineering
The University of Texas at Austin

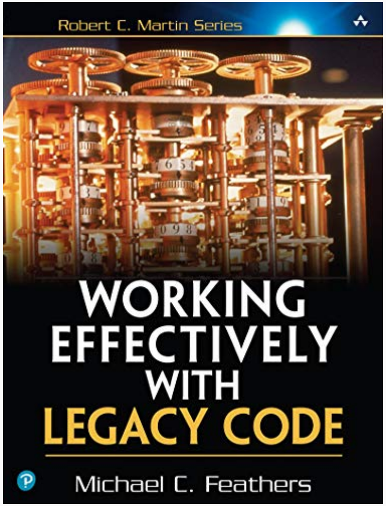
The University of Texas at Austin
Walker Department of Mechanical Engineering
Cockrell School of Engineering

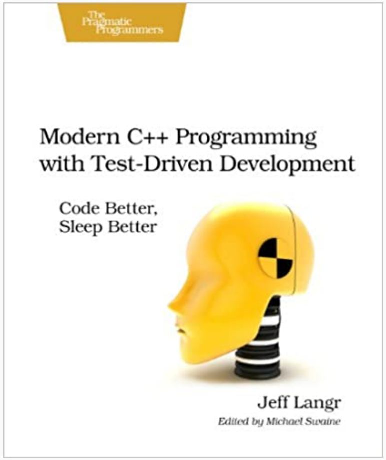
Why Unit Testing?

Introductory Questions

- What is a unit test?
- How do unit tests and validation tests differ?
- Aren’t validation tests sufficient?

Foundational References




The Pragmatic Programmers
Modern C++ Programming with Test-Driven Development
Code Better, Sleep Better
Jeff Langr
Edited by Michael Swaine

Why Unit Tests?

"Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse."

– Michael Feathers, *Working Effectively with Legacy Code*

“Edit and Pray”

- Default way of programming
- We study the code, make sure we understand the behavior, and then make careful changes.
- We run the validation tests and pray everything works.

`https://www.caranddriver.com/features/a27438340/cost-to-paint-car/`

Problems with “Edit and Pray”

- It would take a very long time to fully understand a codebase as big as SU2.
- Little, unintentional mistakes happen (such as forgetting a negative sign).
- Not all mistakes will immediately break the code.
- You spend a lot of time trying to understand the code, but future programmers must do re-work to understand the code *you* wrote.

“Cover and Modify”

- We cover the relevant code with tests.
- We ensure that existing behavior isn’t broken.
- We ensure that new behavior is correct.

`https://www.caranddriver.com/features/a27438340/cost-to-paint-car/`

Why Aren’t Validation Tests Sufficient?

Example 1

You’re developing a new feature and you want to test it to see if it works. You could do a full simulation, but that takes a lot of time and computing power. You want to check if your new code behaves correctly before you throw a lot of resources at it.

Why Aren’t Validation Tests Sufficient?

Example 2

You submit a PR and discover that one of the regression tests has failed. But...why? You know that something is broken, but its hard to track down what broke. You want more granular test coverage that can demonstrate what broke.

Why Aren’t Validation Tests Sufficient?

Example 3

You are fixing a very small bug. You know that you should prove that your bug fix worked, but it doesn’t seem logical to dedicate an entire validation case to one small bug fix. You want to write a small test for a small fix.

But I don't have time to write tests...

- Hypothetical question: You just finished writing your code. If you had to choose between finding a bug now and finding a bug in a year, which would you choose?
- One study[1] found that test-driven development increased development time by 30%, but also decreased bugs by 21%.

[1] Williams, L., Kudrjavets, G., & Nagappan, N. (2009, November). On the effectiveness of unit test automation at Microsoft. In 2009 20th International Symposium on Software Reliability Engineering (pp. 81-89). IEEE.

How Do the Tests Work?

What is a unit-testing framework?

```cpp
#include <iostream>

using std::cout;
using std::cin;

int Adder(int a, int b) {
  return a + b;
}

int main() {
  if (Adder(2, 2) != 4) {
    cout << "Error: Test 'Addition is correct' failed!";
    cout << endl;
    cout << "Expected: " << 4 << endl;
    cout << "Calculated: " << Adder(2, 2) << endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

```cpp
#include "catch.hpp"

int Adder(int a, int b) {
  return a + b;
}

TEST_CASE("Addition is correct", "[arithmetic]") {
  REQUIRE(Adder(2, 2) == 4);
}
```

Design Overview

- Catch2, the unit-testing framework, is included as a header file in the externals folder.
- All the tests are placed in a top-level directory named `UnitTests`
- A single test executable is compiled and run.
- When `ninja test` or `meson test` is run, only a single success or failure is shown. If the test driver ”failed”, then one or more unit tests failed.
- If more detail is needed, you can look at the logs or run the test driver manually.

Anatomy of a test

The header `catch.hpp` contains the macros used for unit tests.

```
#include "catch.hpp"

int Adder(int a, int b) {
  return a + b;
}

TEST_CASE("Addition is correct", "[arithmetic]") {
  REQUIRE(Adder(2, 2) == 4);
}
```

REQUIRE() is similar to assert(). It checks that the contained logical statement is true.

- `TEST_CASE` is a macro used to define the test
- "Addition is correct" is the name
- "[arithmetic]" is a tag

Floating point arithmetic

```
#include "catch.hpp"
#include <math.h>    // Defines atan, M_PI

TEST_CASE("Addition is correct", "[arithmetic]") {
  float pi = atan(1.0)*4.0;
  REQUIRE(pi == Approx(M_PI));
}
```

You can also customize the behavior of `Approx`:

- `Approx(M_PI).epsilon(0.01)` : A relative error of 1%
- `Approx(M_PI).margin(0.01)` : An absolute error of 0.01

Minimal Working Example

Test File

Make a new file: `SU2/UnitTests/tutorial.cpp`

```
#include "catch.hpp"

int Adder(int a, int b) {
  return a + b;
}

TEST_CASE("Addition is correct", "[arithmetic]") {
  REQUIRE(Adder(2, 2) == 4);
}
```

Adding to Build

Open `SU2/UnitTests/meson.build` and add your new test to `su2_cfd_tests`

```
# Add any new test files here
# ----------------------------------------------------------------------
# Begin unit test listings
# ----------------------------------------------------------------------

# Direct-mode tests:
su2_cfd_tests = files(['Common/geometry/primal_grid/CPrimalGrid_tests.cpp',
                       'Common/geometry/dual_grid/CDualGrid_tests.cpp',
                       'SU2_CFD/numerics/CNumerics_tests.cpp',
                       'tutorial.cpp'])
```

Compiling Unit Tests

Add the `-Denable-tests=true` flag to your meson configure call:

```
mkdir <builddir>
meson --prefix=<builddir> <builddir> -Denable-tests=true
ninja -C <builddir>
```

where `builddir` is the directory where you want to install SU2.

Running Unit Tests: Method #1

Input:

```
meson test -C <builddir>
```

Output:

```
ninja: Entering directory <builddir>
ninja: no work to do.
1/1 Catch2 test driver                      OK        0.72 s

Ok:                     1
Expected Fail:          0
Fail:                   0
Unexpected Pass:        0
Skipped:                0
Timeout:                0

Full log written to <builddir>/meson-logs/testlog.txt
```

Running Unit Tests: Method #2

Input:

```
ninja test -C <builddir>
```

Output:

```
ninja: Entering directory `/workspace/code/SU2/tutorial'
[0/1] Running all tests.
1/1 Catch2 test driver                      OK       0.27 s

Ok:                     1
Expected Fail:          0
Fail:                   0
Unexpected Pass:        0
Skipped:                0
Timeout:                0

Full log written to /workspace/code/SU2/tutorial/meson-logs/testlog.txt
```

Running Unit Tests: Method #3

Input:

```
<builddir>/UnitTests/test_driver
```

Output:

```
===============================================================================
All tests passed (12 assertions in 4 test cases)
```

Which method to use?

The first two methods are simple.

The third method gives you the most options to control the test driver.

Options: -s

Input:

```
<builddir>/UnitTests/test_driver -s
```

Output:

```
-------------------------------------------------------------------------------
Addition is correct
-------------------------------------------------------------------------------
../UnitTests/tutorial.cpp:7
...............................................................................

../UnitTests/tutorial.cpp:8: PASSED:
  REQUIRE( Adder(2, 2) == 4 )
with expansion:
  4 == 4

===============================================================================
All tests passed (12 assertions in 4 test cases)
```

Options: –list-tests

Input:

```
<builddir>/UnitTests/test_driver --list-tests
```

Output:

```
All available test cases:
  Center of gravity computation
      [Primal Grid]
  Volume Computation
      [Dual Grid]
  NTS blending has a minimum of 0.05
      [Upwind/central blending]
  Addition is correct
      [arithmetic]
4 test cases
```

Options: Test Name

Input:

```
<builddir>/UnitTests/test_driver "Addition is correct"
```

Output:

```
Filters: Addition is correct
===============================================================================
All tests passed (1 assertion in 1 test case)
```

Options: Test Tag

Input:

```
<builddir>/UnitTests/test_driver "[arithmetic]"
```

Output:

```
Filters: [arithmetic]
===============================================================================
All tests passed (1 assertion in 1 test case)
```

Now lets make the test fail

Edit the file `SU2/UnitTests/tutorial.cpp` and replace 4 with 3.

```
#include "catch.hpp"

int Adder(int a, int b) {
  return a + b;
}

TEST_CASE("Addition is correct", "[arithmetic]") {
  REQUIRE(Adder(2, 2) == 3);
}
```

Recompile and fail

Input:

```
ninja -C <builddir> test
```

Output:

```
1/1 Catch2 test driver                      FAIL     0.27 s (exit status 1)

Ok:                     0
Expected Fail:          0
Fail:                   1
Unexpected Pass:        0
Skipped:                0
Timeout:                0


The output from the failed tests:

1/1 Catch2 test driver                      FAIL     0.27 s (exit status 1)
```

Running failing test driver

Input:

```
<builddir>/UnitTests/test_driver
```

Output:

```
Addition is correct
-------------------------------------------------------------------------------
../UnitTests/tutorial.cpp:7
...............................................................................

../UnitTests/tutorial.cpp:8: FAILED:
  REQUIRE( Adder(2, 2) == 3 )
with expansion:
  4 == 3

===============================================================================
test cases:  4 |  3 passed | 1 failed
assertions: 12 | 11 passed | 1 failed
```

Lots More is Possible with Catch2

- Grouping tests into sections with similar setup or teardown
- Parameterized tests
- Logging context to report alongside failures
- Tests that are expected to throw exceptions
- Hiding tests from the default list
- Launching GDB post-mortem on failing tests
- Custom matchers

See the official Catch2 documentation for more details.

Where to find more help

https://su2code.github.io/docs_v7/Running-Unit-Tests/

https://su2code.github.io/docs_v7/Writing-Unit-Tests/

https://github.com/catchorg/Catch2

Conclusions

“Remember, code is your house, and you have to live in it.”

– Michael Feathers, *Working Effectively with Legacy Code*