Tests goals – properties – good principles

General idea and overview

There is slightly different advice around, pick your favourite:

Write F.I.R.S.T tests first

Fast:	To be run frequently
Independent:	To make failures meaningful
Repeatable:	To be run everywhere with same outcome
Self-validating:	To be automatised
Timely:	To make production code testable

General idea and overview

There is slightly different advice around, pick your favourite:

Google's perspective

Readable:	Tests are correct by inspection
Correct:	Do not rely on known bugs and test real scenarios
Complete:	Test most edge cases
Documenting:	Demonstration of how the API works
Resilient:	Repeatable, independent, hard to break, hermetic, etc.

General idea and overview

There is one not-negotiable **requirement of tests**, though:

General idea and overview

There is one not-negotiable **requirement of tests**, though:

A small interlude: Unit test frameworks

Essential for self-validation and automation!

1 Boost Test Library

Documentation

```
#define BOOST_TEST_MODULE My Test
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_CASE(first_test)
{
    int i = 1;
    BOOST_TEST(i);
    BOOST_TEST(i == 2);
}
```

```
Running 1 test case...
test_file.cpp(8): error: in "first_test": check i == 2 has
    ↪ failed [1 != 2]

*** 1 failure is detected in the test module "My Test"
```

A small interlude: Unit test frameworks

Essential for self-validation and automation!

1. Boost Test Library — Documentation
2. GoogleTest — User's guide

```
#include <gtest/gtest.h>
#include "src/factorial.h"

TEST(FactorialTest, HandlesZeroInput) {
    EXPECT_EQ(factorial(0), 1);
}

TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(factorial(1), 1);
}

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

A small interlude: Unit test frameworks

Essential for self-validation and automation!

1. Boost Test Library — Documentation
2. GoogleTest — User's guide
3. Vir's Unit Test Framework — GitHub repository

```
#include <vir/test.h>

TEST(test_name) {
    int test = 3;
    COMPARE(test, 2) << "more " << "details";
    //COMPARE(test, 2).on_failure("more ", "details");
}
```

```
FAIL: ┌ at tests/testfile.cpp:5 (0x40451f):
FAIL: │ test (3) == 2 (2) -> false more details
FAIL: └ test_name

Testing done. 0 tests passed. 1 tests failed.
```

A small interlude: Unit test frameworks

Essential for self-validation and automation!

1. Boost Test Library — Documentation
2. GoogleTest — User's guide
3. Vir's Unit Test Framework — GitHub repository
4. ...and so many mores! — List on Wikipedia

Not all have the same weight and functionality

Pick your favourite depending on your needs and project:

1. Natural choice if you already depend on Boost
2. More complex but also more powerful
3. Very simple and lightweight

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs

Having tests passing is simple...

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs

Having tests passing is simple...

```
int cube(int x) {
    //TODO: To be implemented
    return 0;
}

TEST(cube_test) {
    VERIFY(0 == cube(2));
    VERIFY(0 == cube(3));
    VERIFY(0 == cube(42));
}
```

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs

...better to have them failing!

```
int cube(int x) {
    //TODO: To be implemented
    return 0;
}

TEST(cube_test) {
    VERIFY(8 == cube(2));
    VERIFY(27 == cube(3));
    VERIFY(74088 == cube(42));
}
```

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs
- Never mock the code that you are testing

This is probably useless

```
class MockCoffeeMachine : public CoffeeMachine {
    //For simplicity we assume always in temperature
    double warmUp() override {
        return constants::brewingT;
    }
}

BOOST_AUTO_TEST_CASE(warmUp_test) {
    MockCoffeeMachine machine{};
    BOOST_REQUIRE_CLOSE(machine.warmUp(),
                        constants::brewingT, 0.1);
}
```

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs
- Never mock the code that you are testing

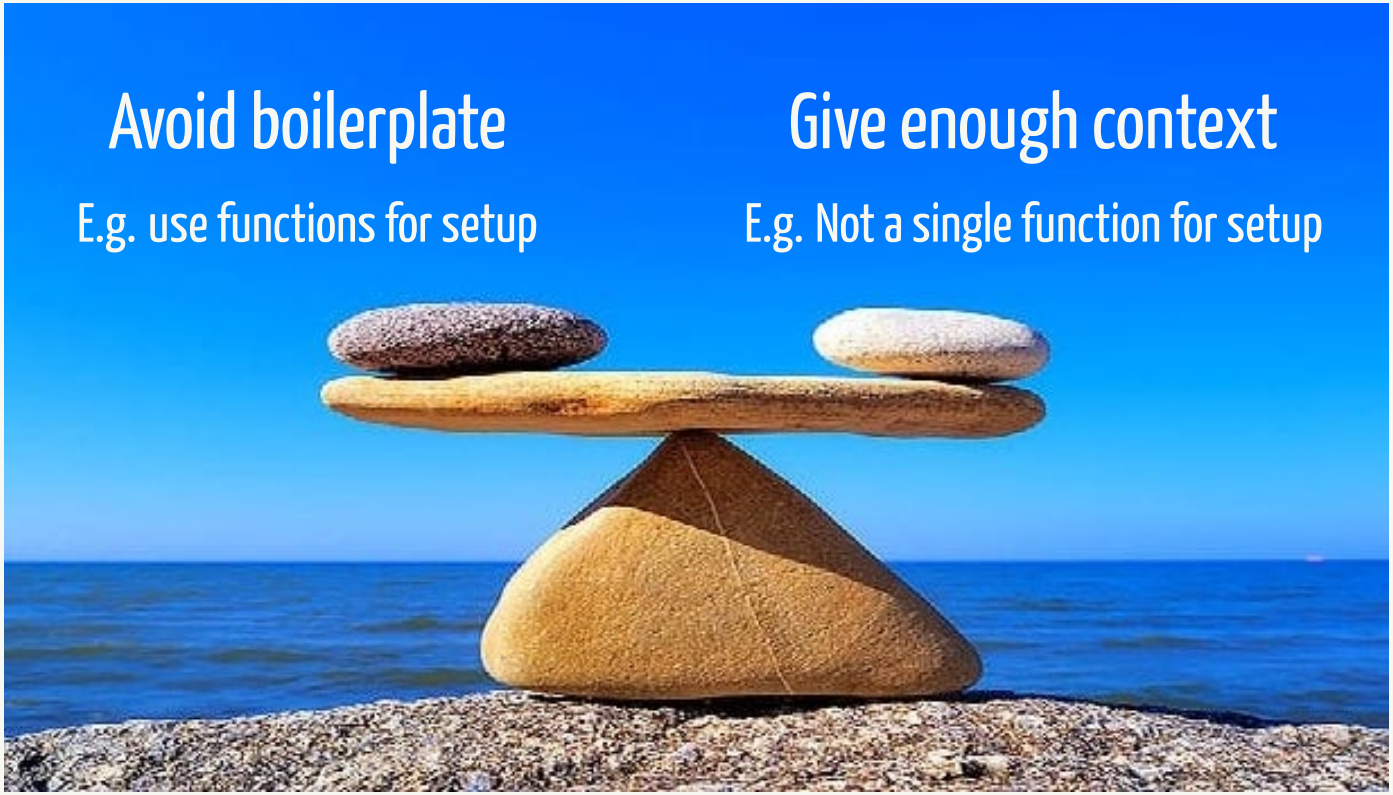
Think of a clear reaction procedure in your team

What should happen in your project when a test fails?
...and when a bug that was not caught by tests is found?

↓

Never miss any chance to improve your tests!

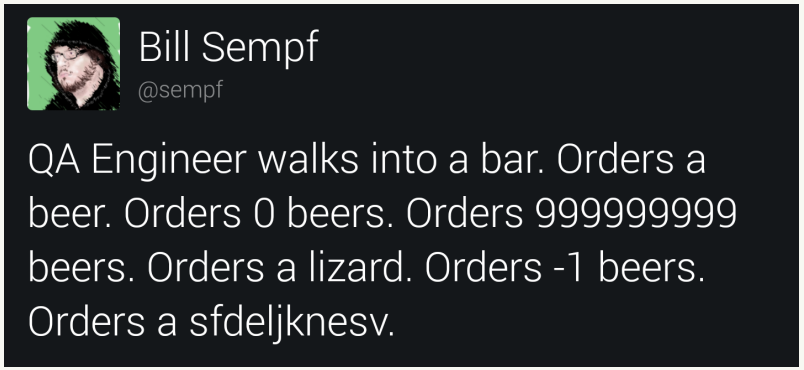
Tests readability



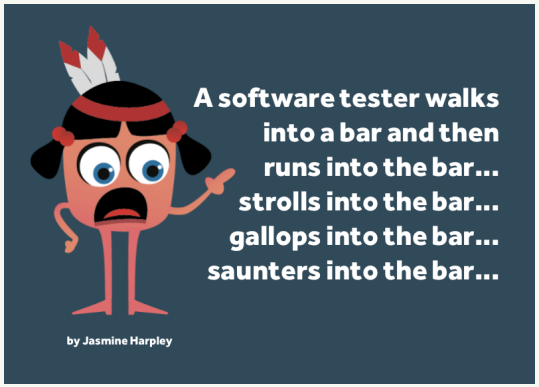
Remember: Tests are correct by inspection!

Tests completeness: What should I test and how?

- Values



- Actions



...and anything that makes sense!

Tests completeness: What should I test and how?

A very nice example from Titus Winter

```
TEST(FactorialTest, BasicTests) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(120, Factorial(5));
}
```

Tests completeness: What should I test and how?

A very nice example from Titus Winter

```
int Factorial(int n) {
    if (n == 1) return 1;
    if (n == 5) return 120;
    return -1; // TODO(goofus): figure this out.
}

TEST(FactorialTest, BasicTests) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(120, Factorial(5));
}
```

Tests completeness: What should I test and how?

A very nice example from Titus Winter

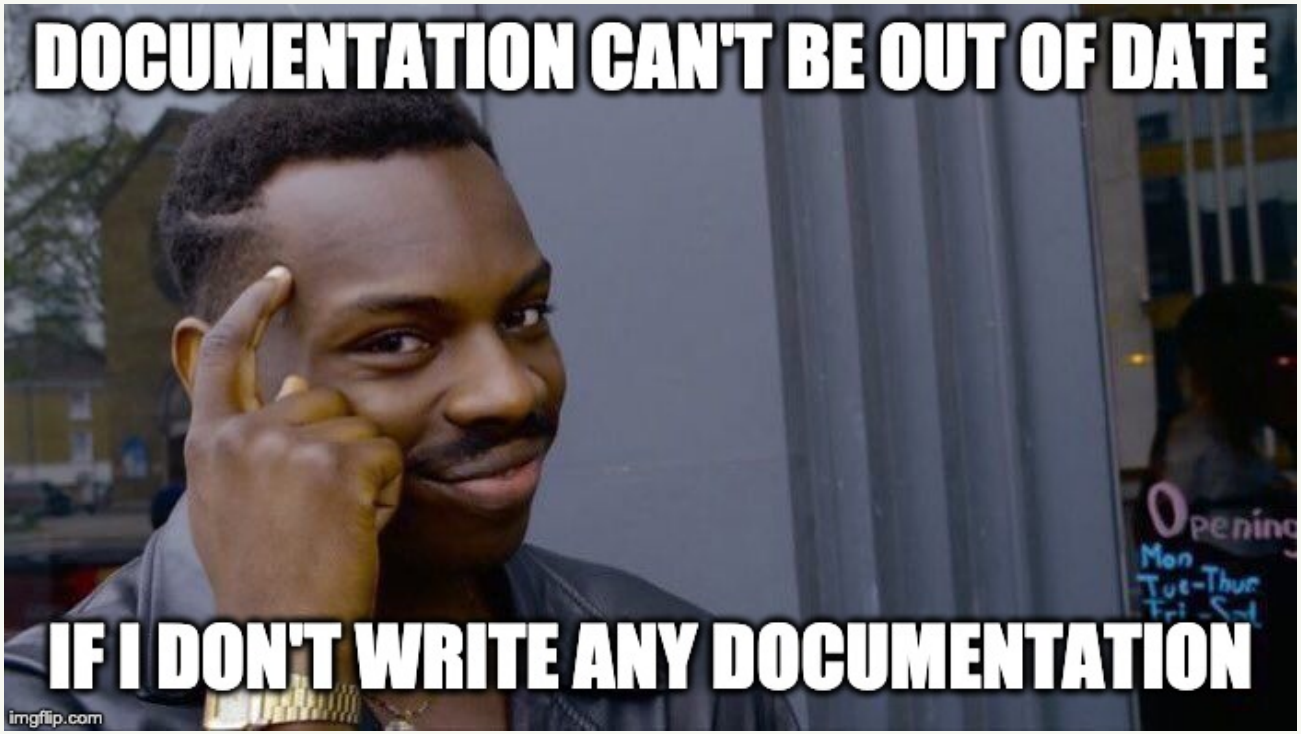
```
int Factorial(int n) {
    if (n == 1) return 1;
    if (n == 5) return 120;
    return -1; // TODO(goofus): figure this out.
}

TEST(FactorialTest, BasicTests) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(120, Factorial(5));
    EXPECT_EQ(1, Factorial(0));
    EXPECT_EQ(479001600, Factorial(12));
    EXPECT_EQ(std::numeric_limits::max<int>() , Factorial(13));
    EXPECT_EQ(1, Factorial(0));
    EXPECT_EQ(std::numeric_limits::max<int>() , Factorial(-10));
}
```

This will pretty much naturally emerge when doing TDD

The hidden value of tests

Few words about tests as API usage examples

Of course, a meme, do not take it seriously, but...



...good tests should show how to use your code!

In this sense, it should be avoided to use implementation details in tests (cf. white box testing downsides).

Tests resilience: Google advice

- No flaky tests → Tests should always give the same outcome
- No brittle tests → One failure should not trigger many failures
- Tests reference values should not come from the system under test!
- Changing tests order should never change the outcome
- Tests should be as hermetic as possible → No I/O, no network, etc.
- No deep dependence → Avoid any implicit assumption

Let's see some examples!

This list and all GoogleTest example are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

A sneaky flaky test

```
TEST(UpdaterTest, RunsFast) {
    Updater updater;
    updater.UpdateAsync();
    SleepFor(Seconds(.5)); // Half a second should be *plenty*.
    EXPECT_TRUE(updater.Updated());
}
```


This list and all GoogleTest example are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

A sneaky flaky test

```
TEST(UpdaterTest, RunsFast) {
    Updater updater;
    updater.UpdateAsync();
    SleepFor(Seconds(.5)); // Half a second should be *plenty*.
    EXPECT_TRUE(updater.Updated());
}
```

An infamous brittle pattern

```
TEST(Logger, LogWasCalled) {
    StartLogCapture();
    EXPECT_TRUE(Frobber::Start());
    EXPECT_THAT(
      Logs(), Contains("file.cc:421: Opened file frobber.config" )
    );
}
```


This list and all GoogleTest example are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

Where are the reference values coming from?

```
BOOST_AUTO_TEST_CASE(Dirac_M)
{
    const hmc_float refs[4] =
        { 2610.3804893063798, 4356.332327032359,
          2614.2685771909237, 4364.1408252701831 };
    test_fermionmatrix<physics::fermionmatrix::Dslash>(refs);
}
```


This list and all GoogleTest example are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

Where are the reference values coming from?

```
BOOST_AUTO_TEST_CASE(Dirac_M)
{
    const hmc_float refs[4] =
        { 2610.3804893063798, 4356.332327032359,
          2614.2685771909237, 4364.1408252701831 };
    test_fermionmatrix<physics::fermionmatrix::Dslash>(refs);
}
```

A non-hermetic test: What if run twice in parallel?

```
TEST(Server, StorageTest) {
    StorageServer* server = GetStorageServerHandle();
    auto my_val = rand();
    server->Store("testkey", my_val);
    EXPECT_EQ(my_val, server->Load("testkey"));
}
```


This list and all GoogleTest example are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

Deep dependence example

```
class File {
  public:
    // ...
    virtual bool StatWithOptions(Stat_t* stat, StatOptions opts) {
        return this->Stat(stat); // In base class ignore options
    }
};
TEST(Filesystem, FSUsage) {
    // ... and call to StatWithOptions
    EXPECT_CALL(file, Stat(_)).Times(1);
}  // This relies on a call to base StatWithOptions!!
```

The law of implicit interfaces (AKA Hyrum's law)

Given enough users of your public interface, soon or later someone will start implicitly relying on your implementation details (speed, memory consumption, etc.).

Hyrum Wright

This list and all GoogleTest example are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

White and black box testing

BDD in its original idea

Test driven development as discipline

What to do, now?