Scientific Unit Testing

Dave Steffen, Ph.D.
Software Lead
dsteffen@scitec.com

in Physics
which is relevant

SCiTEC
Science & Technology Innovation

www.scitec.com

If I have seen further than others, it is by standing upon the shoulders of giants.

-- Isaac Newton

Earlier this week: Phil Nash, Clare Macrae, Ben Saks

- T. Winters and H. Wright, *All Your Tests Are Terrible...* CppCon 2015
- Fedor Pikus, *Back to Basics: Test-driven Development* CppCon 2019

Kevlin Henney:

- "Structure and Interpretation of Test Cases" NDC Conferences 2019
- "Test Smells and Fragrances" DevWeek 2014

Properties of Good Tests

0. Existence!

1. Correctness
2. Completeness
3. Readability
4. Demonstrability
5. Resilience

- Easy to run
- Fast to run
- TDD
- Deterministic
- Code coverage / regulatory requirement
- …

Remember, bad tests are almost always better than no tests!

Unit Testing OO Code

Test using only the public interface

("Black Box" testing)

- Forces better design ("design for testability")
- Avoids tight coupling to implementation
- Unit tests are also examples and documentation

Object Oriented testing

Let's unit test this class

```
1  // Cup.h
2
3  class Cup {
4    public:
5      Cup();                // creates an empty cup
6
7      bool IsEmpty();
8
9      bool Fill();          // if empty, fills completely, returns true
10                           // otherwise leaves cup full, return false
11
12     bool Drink();         // If full, empties the cup and returns true
13                           // otherwise leaves cup empty and returns false
14   private:
15     ...
16 }
```

- Filling a full cup fails (wastes beer)
- Drinking from an empty cup is just dissapointing
- You've already written the implementation in your head.

Testing Classes

The Old Way

```
1  // Cup.h
2
3  class Cup {
4    public:
5      Cup();
6      bool IsEmpty();
7      bool Fill();
8      bool Drink();
9    private:
10     ...
11 }
```

```
TEST_CASE( "Cup::Cup"     ) {...};
TEST_CASE( "Cup::IsEmpty") {...};
TEST_CASE( "Cup::Fill"    ) {...};
TEST_CASE( "Cup::Drink"   ) {...};
```

The "old" approach:
Test each member function individually.

Test each member function. OK...

```
// test the constructor
TEST_CASE( "Cup::Cup" )
{
  Cup cup;                    // new cups are supposed to be empty
  REQUIRE( cup.IsEmpty() );   // test this.
};
```

```
// test isEmpty
TEST_CASE( "Cup::IsEmpty" )
{
  Cup cup;                      // empty by construction (see above)
  REQUIRE( cup.IsEmpty() );      // test this.
};
```

Using only the public interface,

- Both tests read the same...
- because we need each member function to test the other.

The "Black Box Conundrum"

Fundamentally if we only test via the public interface, we have a circular-logic problem:

you have to use the interface to test the interface, so at some point you have to trust something which you haven't tested.

You never *really* know if a member function is correct.

Maybe the whole thing is a bunch of bugs that hide each other!

Common Solutions

- Ignore the problem
- Declare one member function "correct by inspection" and start from there
- Abandon Black Box testing as impractical

```
#define private public

#include "Cup.h"

// test constructor
TEST_CASE( "Cup::Cup" ) {
  Cup cup;
  REQUIRE( cup.m_isEmpty );
};
```

Please don't do this!

Test state of internals

"White Box" Testing:

Open up the class (somehow) and examine its internal state

White box testing

(done right)

If we are going to break encapsulation, do it correctly.

Option 2: Give access to a trusted friend

```
// Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Fill();
    bool Drink();
  private:
    friend struct CupTester
    ...
};
```

```
// TestCup.h

struct CupTester {
  bool& is_empty;
  CupTester(Cup& c) : is_empty(c.is_empty);
};

TEST_CASE( "ctor" ) {
  Cup cup;
  CupTester tester(cup);
  // directly test internal state
  REQUIRE( cup_tester.is_empty );
};
```

Pros:

no UB

Cons:

Changes the source code

but not in any way that matters?

Behavior Driven Development

```
1  // Cup.h
2
3  class Cup {
4    public:
5      Cup();
6      bool IsEmpty();
7      bool Fill();
8      bool Drink();
9    private:
10     ...
11 }
```

```
TEST_CASE( "A new cup is empty"           ) = []{...};
TEST_CASE( "An empty cup can be filled"   ) = []{...};
TEST_CASE( "A filled cup is full"         ) = []{...};
TEST_CASE( "Drinking epties a filled cup" ) = []{...};
                        ...
```

Don't test the member functions individually, test the class' behavior as a whole

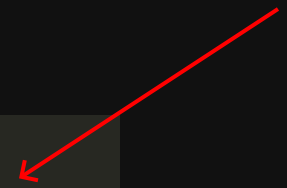
- Test names now read like the design spec

How does it work?

```
 1 // Cup.h
 2
 3 class Cup {
 4   public:
 5     Cup();
 6     bool IsEmpty();
 7     bool Fill();
 8     bool Drink();
 9   private:
10     ...
11 }
```

```
TEST_CASE( "A new cup is empty" ) {
  Cup cup;
  REQUIRE( cup.IsEmpty() );
};
```

```
TEST_CASE( "An empty cup can be filled" ) {
  Cup cup;    // empty by construction
  auto success = cup.Fill();
  expect( success );
};
```



But wait -- this is the same code we started with

Why does changing the name make this OK?

How does Behavior-Driven Testing solve the Black Box Conundrum?

Behavior, not Implementation

```
1  // Cup.h
2
3  class Cup {
4    public:
5      Cup();
6      bool IsEmpty();
7      bool Fill();
8      bool Drink();
9    private:
10     ...
11 }
```

```
TEST_CASE( "A new cup is empty" ) {
  Cup cup;
  REQUIRE( cup.IsEmpty() );
};
```

- We are now testing consistency of interface, not correctness of implementation Repeat this to yourself a few times

- If the constructor is wrong, and isEmpty is *also* wrong in exactly the right way to hide it, *and all other behaviors follow suit*, the observable behavior is correct!

A bug that cannot be observed under *any* conditions
is not a bug

Modern Philosophy of Science:

Popper's Falsifiability Criterion:

- Science would like to
  1. make statements that can be proven true or false
  2. Prove them true or false
- But generally you *can't* prove them true
- You *can* do the next best thing:
  1. make statements that can be proven false
  2. Try to prove them false
     - ... *and fail*

Confidence tracks the thoroughness of the tests

Popper's Falsifiability Criterion applied to Unit Tests:

If a class exhibits correct behaviour *everywhere*, we must declare its implementation to be correct.

Even if the implementation is entirely self-cancelling bugs.

Make a falsifiable hypothesis:

"This code has a bug"

Then write the test to observe it
If you can't observe it, it's not there

Confidence in correctness

tracks

completeness of the testing

Empirical science:

Reverse-engineering the Universe by writing unit tests against its observable interface

Scientists want to white-box test reality but can't

```
#define private public

#include <reality.h>

TEST_CASE( "Quantum Mechanics" ) {
  ...
};
```


We can't do this

We have ~350 years of experience with the logic, procedures, and epistemology of black-box testing.

Behavior-driven black-box testing

is on sound philosophical foundations

Are there any practical results we can take away?

Your software is a system to be studied, poked, and prodded for the presence of bugs

Unit tests (and other tests) are the experimental apparatus to detect them.

Maybe experimental science can give us a few pointers...

Basic lab procedure: weighing a sample

Question: do you believe we have 30 grams of coffee?

Basic lab procedure: weighing a sample

Zero the scale *with* the container to duplicate the exact conditions of the experiment.

Basic lab procedure: weighing a sample

Only after calibrating our equipment for *exactly* the situation to be measured can we trust the result.

This is *exactly* TDD's procedure for a bugfix

- TDD:
  1. Write a (failing) unit test to demonstrate the defect
  2. Fix the defect
  3. Run the test. If it passes, then done

"Test Driven Development: it's not so much about what it does to your code, it's about what it does to your mind"

-- Fedor Pikus

Add calibration?

- To be really sure, zero the scale, then put on a known quantity to make sure the scale reads correctly.

Unit test equivalent: intentionally introduce bug into code, make sure the unit test fails *the right way* with *the right error message*

What is "right"

```
1 double pi()
2 {
3     // float version of acos
4     return acosf(-1);
5 }
```

```
TEST_CASE( "pi accuracy" )
{
  expect(pi() == 3.14159274101257);
  //             ^ ^^^^^^ accurate digits
}
```

π = 3.14159265358979

pi() = 3.14159274101257

This is probably close enough... or is it?

- Error in Earth equatorial circumference ~ 1.12 m
- R_e = 6378137 m (7 significant figures)

What is "right"

```
1 double pi()
2 {
3     return acos (-1);
4 }
```

```
TEST_CASE( "pi accuracy" )
{
  expect(pi() == 3.14159274101257);
  //             ^ ^^^^^^ accurate digits
}
```

double precision! (π == M_PI)

This just broke

- Incorrect "right" answer creates brittle test
- Unnecessarily "pins" to specific algorithm

This is just error propagation

(the bane of all freshman physics labs)

Qualities of experiments

1. Precision
2. Reproducibility
3. Accuracy

Precision

Precision maximizes information content

- Use a test framework with good error messages
  - and use it!
- Limit tests to the code in question

```
1 TEST_CASE( "widget" ) {
2   ...
3   std::vector<Widget> wvec;
4   REQUIRE( wvec.empty() );   // really?
5
6   wvec.push_back(Widget{5});
7   REQUIRE( wvec.size() == 1); // really?
8 };
```

- Make sure the error shows up in the right place with the right context (no red herrings!)

Reproducibility

(handling nondeterministic results)

Most scientific experiments have measurement error or noise.

- Interference from the environment: noisy signals
- Inherent in complex systems with too many variables
- Inherent in quantum mechanics

Solutions:

- Isolate
- Subtract
- Detect and eliminate erroneous results
- Statistical analysis

Reproducibility

(handling unreliable results)

"Tests should fail because the code under test fails, and for no other reason" -- Titus Winters

This is about handling interactions between your test and the environment

- Unreliable or "flaky" tests:
  - depend on timing or external state (test servers, file systems) that leads to test failures.

First, *verify the nondeterminism is due to external influences*

- That is: verify that these are false alarms

Option 1: Isolation

Isolate tests from the environment

easy ... unless the code in question is explicitly supposed to interact with the environment

Unit Testing:

- Mock the external thing
- Create the external thing as part of the test (forking a process to open a socket)
- Dedicated Test servers / databases / filesystems
- Dedicated hardware instances for embedded testing
- *These can be expensive*

Eliminate noise from test environment by insulation and separation

Option 2: Subtraction

Measure the unwanted environmental effects

Adjust tests to compensate

Example:

Client must connect to server within X seconds

- Measure network latency Y
- Cue test framework to allow X + Y seconds

Adapt test parameters to current test conditions

Option 3: Detect and Respond

1. Develop independent sensors to detect invalid situations
2. Develop tools and processes to handle invalid results

Options:

- If the environment invalidates the test
  - Mark test as "pending" and run later
  - Mark test as "not run" (and have a software dev process to handle this)
- Elevate "lack of test resources" to management

Option 4: Statistics

As a last resort, intrinsic noise can be handled by statistics

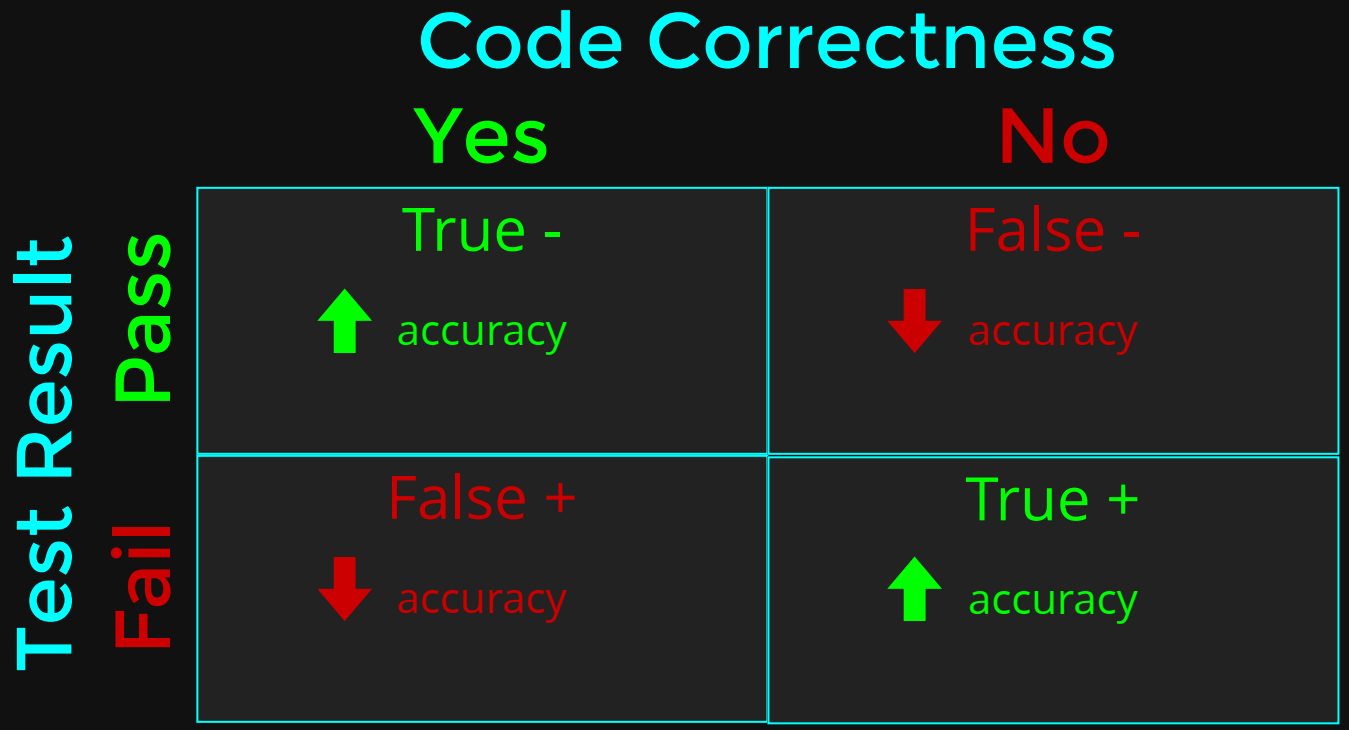
- Collect many samples, find the expected rate of false failue
  - Rig test framework to run the test repeatedly; report failure only if failure rate exceeds threshold
- Develop heuristics or statistics to detect anomalous results and either justify throwing them out, or select for closer investigation
  - If it fails every Thursday night, rig test framework to know this

Automate the "ignore" of flaky tests to avoid desensitizing engineers to failures

Accuracy

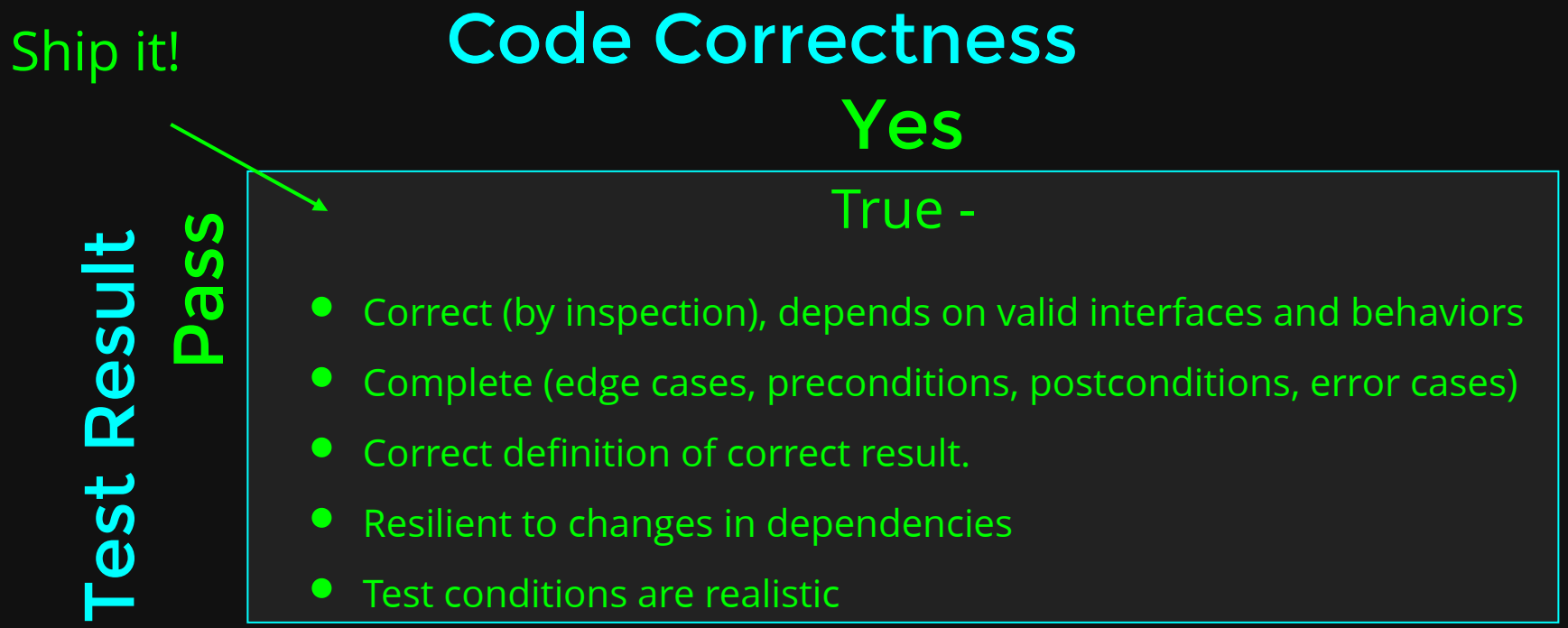
Accuracy means the results match reality

+ Positive == bug - Negative == no bug



Accuracy

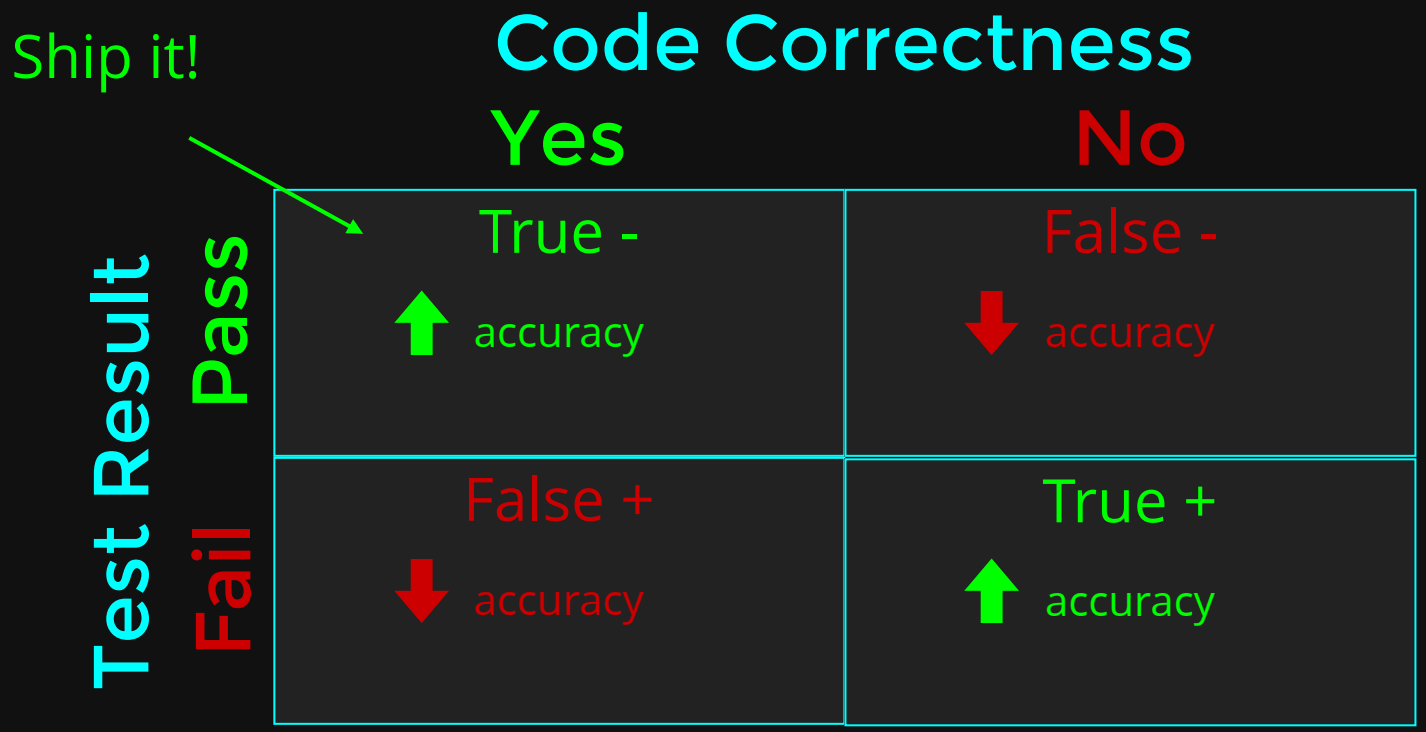
Accuracy means the results match reality



Accuracy

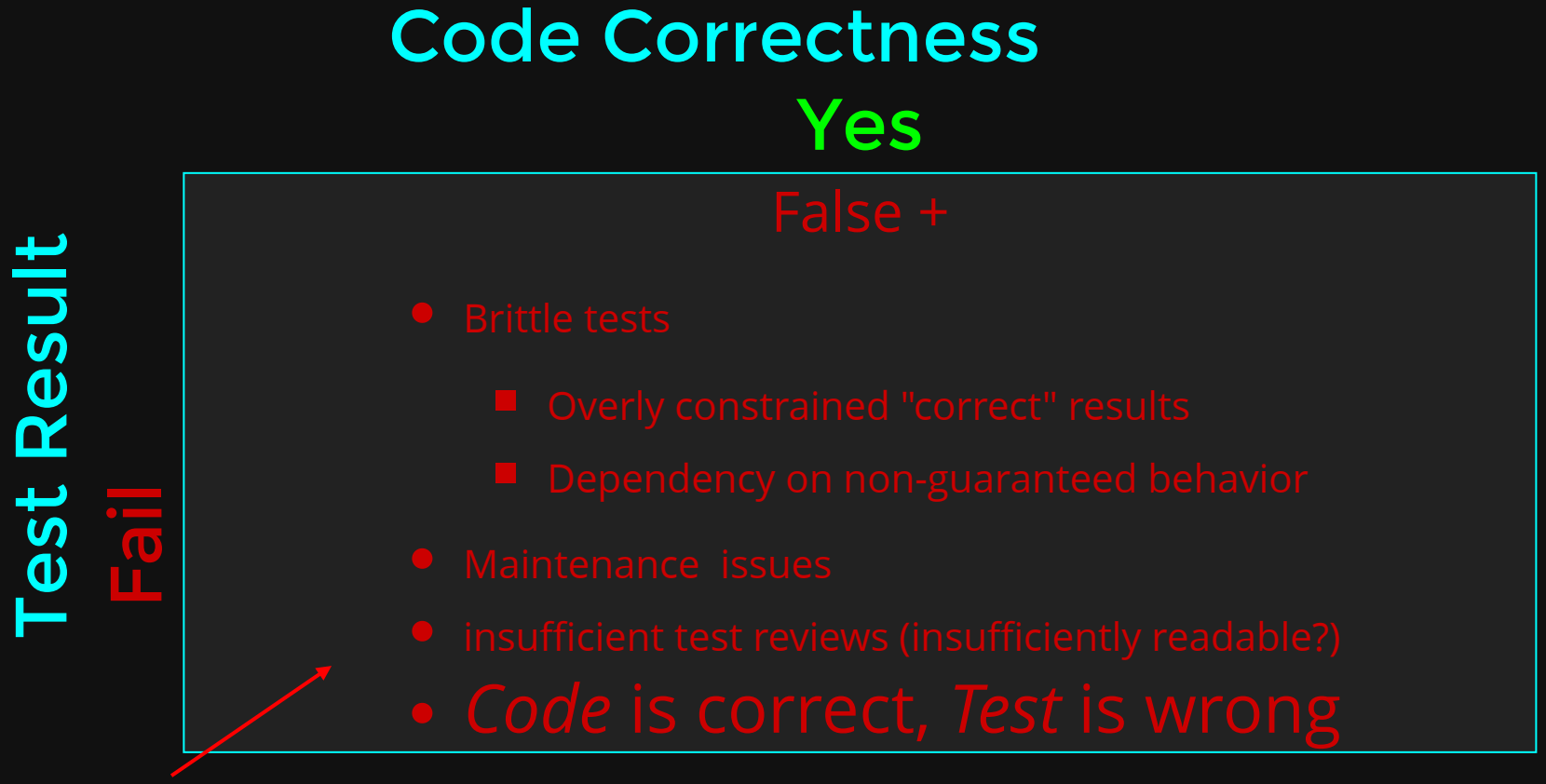
Accuracy means the results match reality

\+ Positive == bug - Negative == no bug



Accuracy

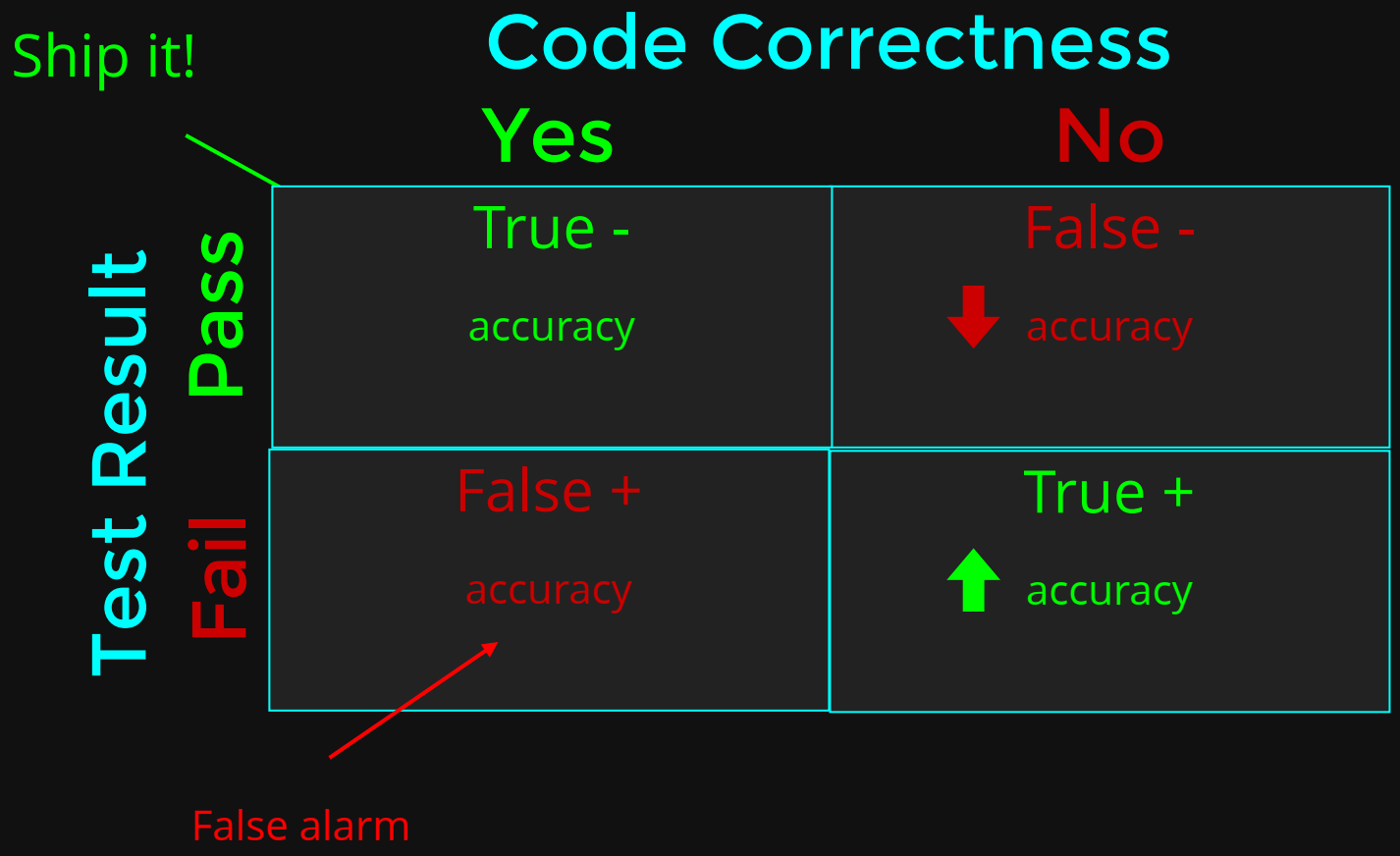
Accuracy means the results match reality



Accuracy

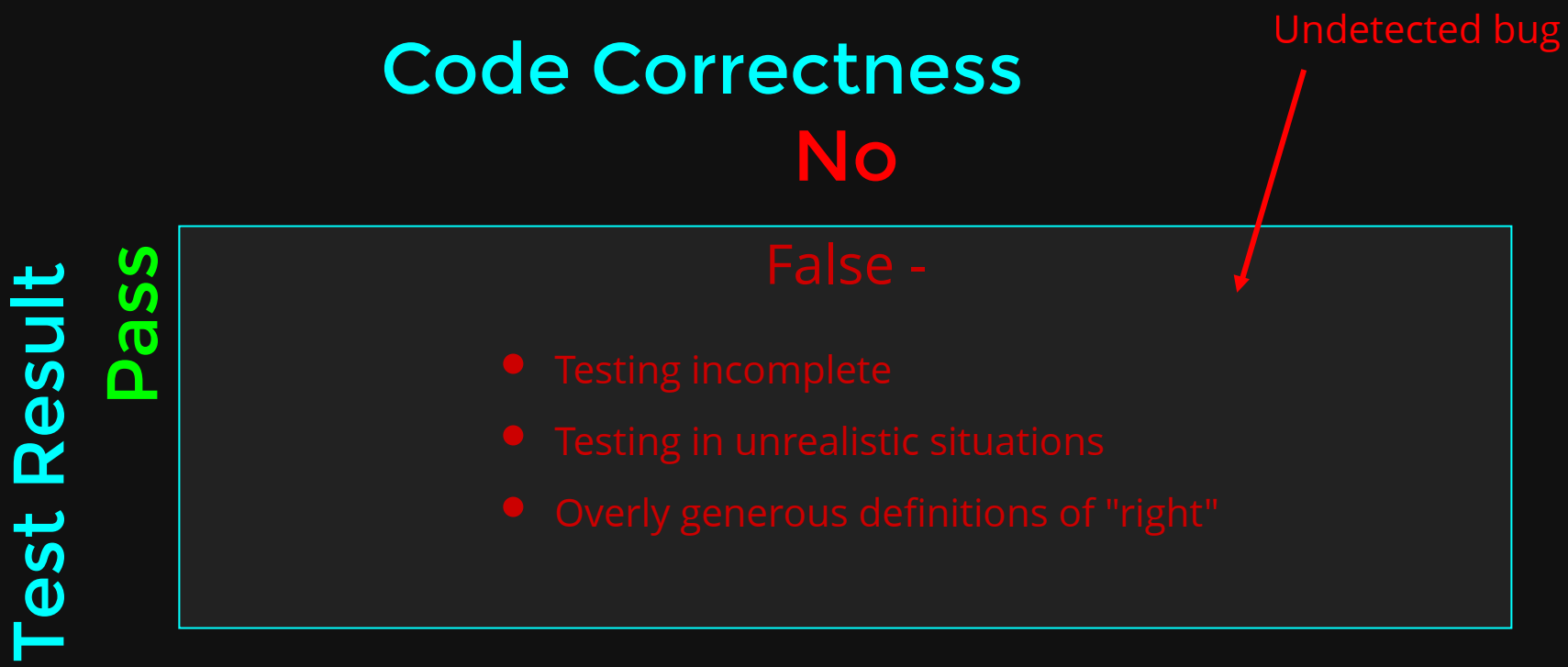
Accuracy means the results match reality

\+ Positive == bug - Negative == no bug



Accuracy

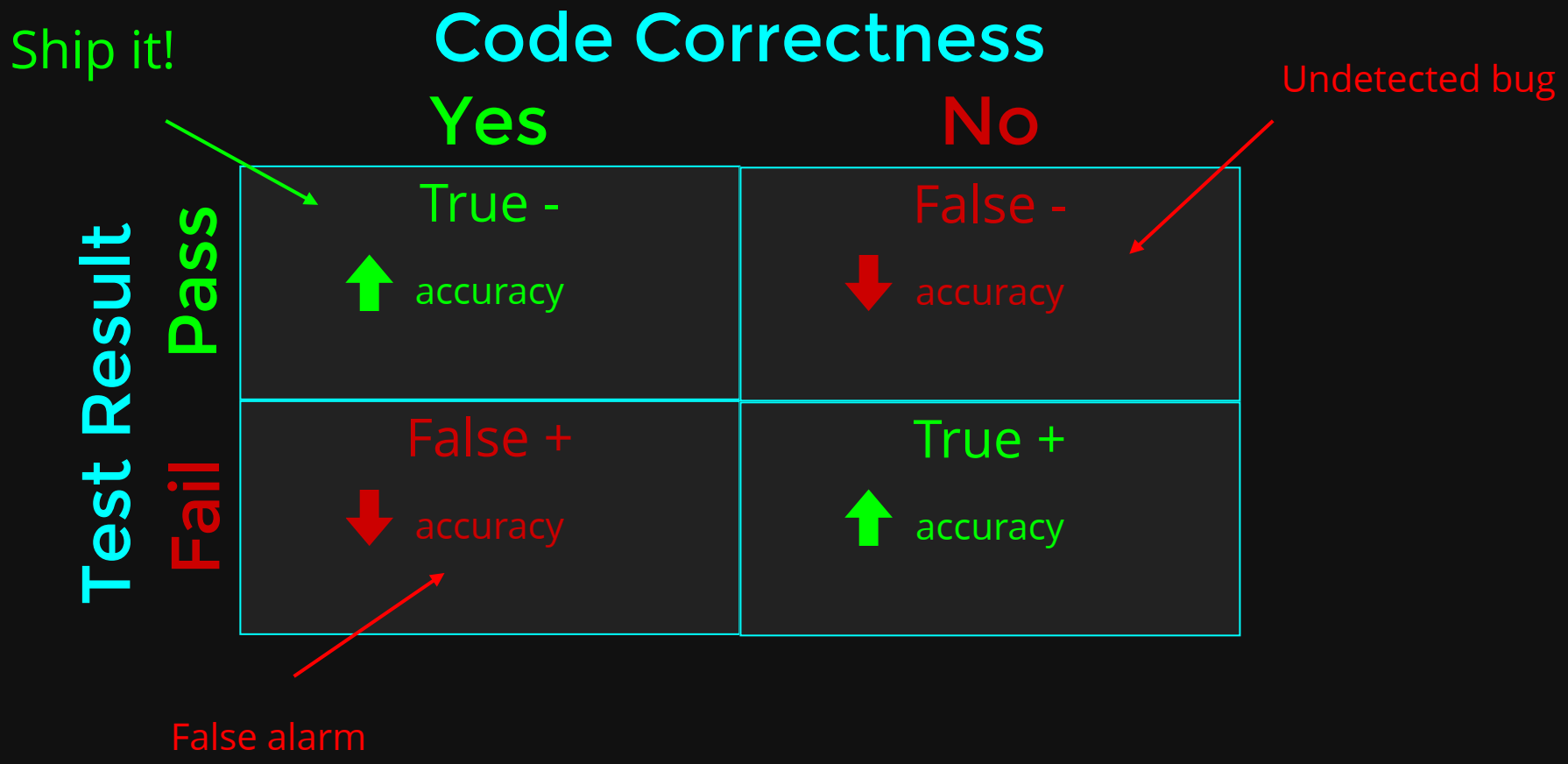
Accuracy means the results match reality



Accuracy

Accuracy means the results match reality

\+ Positive == bug - Negative == no bug



Accuracy

Accuracy means the results match reality

Code Correctness

No

Test Result

Fail

True +

Tests find a bug. Success!

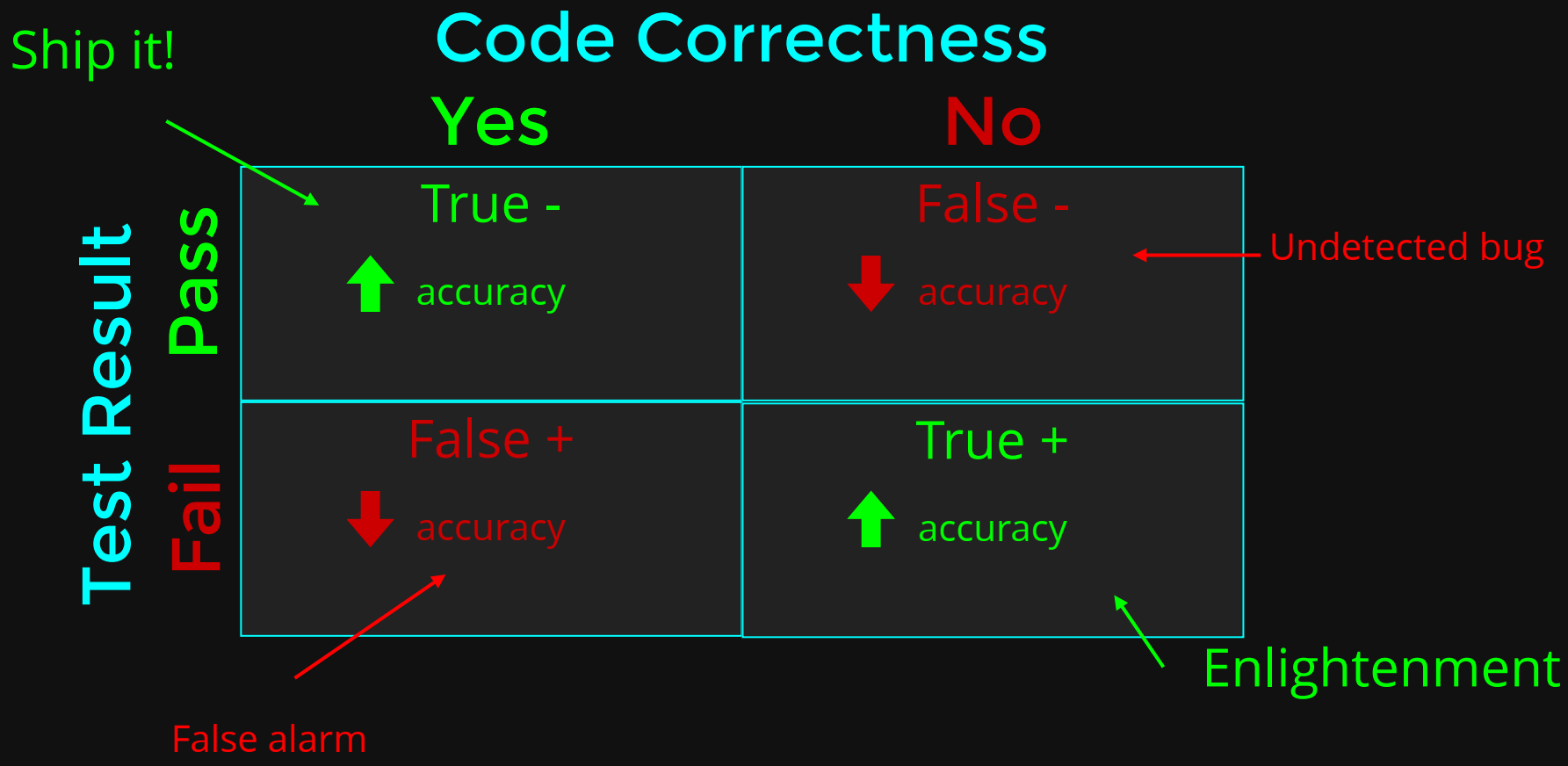
You don't use science to show you're right,
you use science to become right
- Randall Munroe

Enlightenment

Accuracy

Accuracy means the results match reality

\+ Positive == bug - Negative == no bug



Unit Testing is Science

Unit tests (and other tests) are experimental apparatus to detect bugs in your code.

- Hypothesis *C* : this code is correct
- Unit tests attempt to show *!C*
- Confidence in *C* tracks thoroughness of tests

A.K.A "The Scientific Method"

Go forth

write good tests

and do good science

Acknowledgements

Neil Sexon, SciTec

for peer reviews and good ideas

Kris Jusiak, Quantlab Financial

Check out his C++20 macro-free unit test framework at github.com/boost-experimental/ut

Various SciTec developers

for suffering through early versions of this talk

References

All talks available on YouTube

- T. Winters and H. Wright, *All Your Tests Are Terrible...*

  CppCon 2015 https://youtu.be/u5senBJUkPc
- Fedor Pikus, *Back to Basics: Test-driven Development*

  CppCon 2019 https://youtu.be/RoYljVOj2H8
- Phil Nash, *Modern C++ Testing with Catch2*

  CppCon 2018 https://youtu.be/Ob5_XZrFQH0 (And see any number of other talks by Phil on the subject)
- Kevlin Henney: *Structure and Interpretation of Test Cases*

  NDC Conferences 2019 https://youtu.be/tWn8RA_DEic
- Kevlin Henney: *Test Smells and Fragrances*

  DevWeek 2014 https://youtu.be/wCx_6kOo99M

References

- Phil Nash "Test Driven C++ With Catch"
- Phil Nash "Modern C++ Testing with Catch 2"
- [Boost].µt

1