Back To Basics
Unit Testing

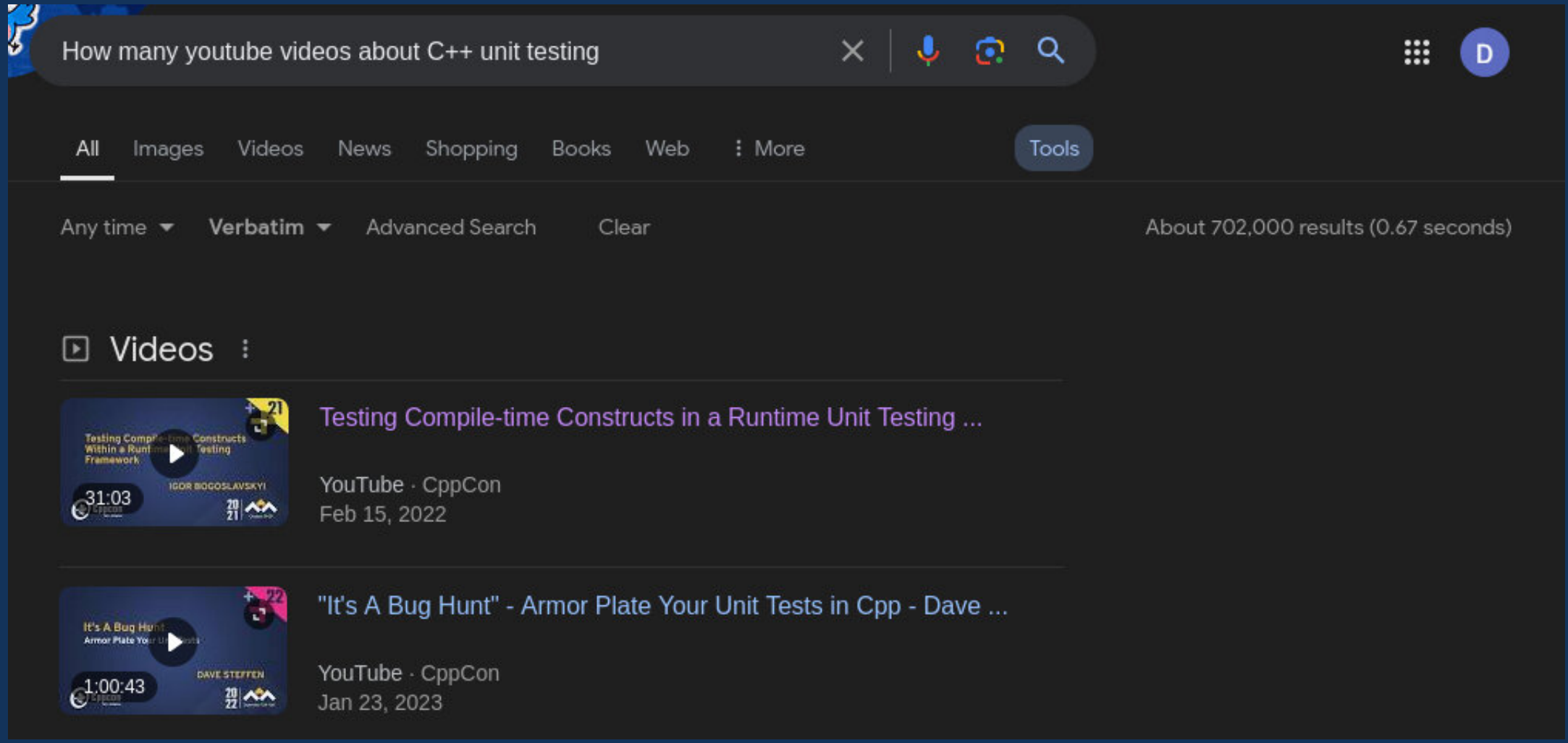
DAVE STEFFEN

Cppcon
The C++ Conference

20
24

September 15 - 20

Unit Testing is a Big Topic



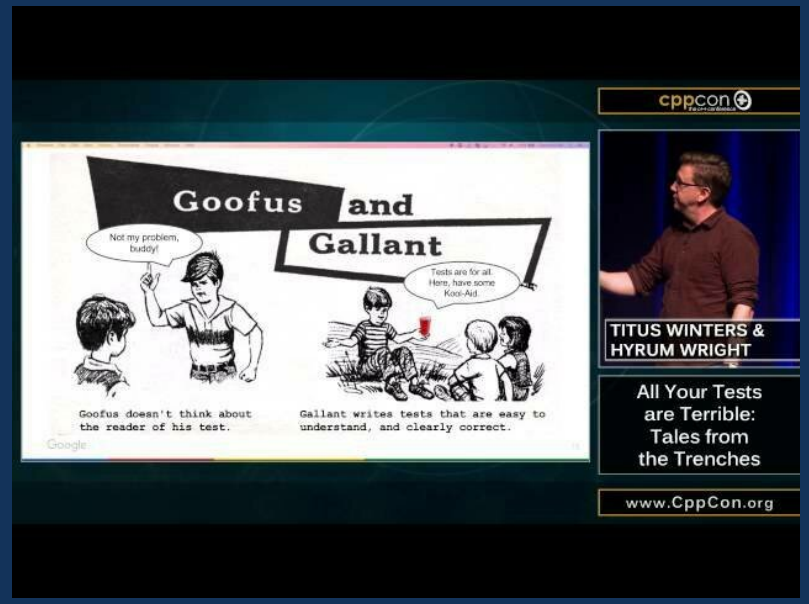
Unit Testing is a Big Topic

- Jody Hagins (today): The Most Important Design Guideline is Testability
- Greg Law and Mike Shah (tomorrow): Back To Basics, Debugging and Testing
- Chip Hogg (Wednesday), Making Hard Tests Easy (Robotics Track)
- Xiaofan Sun (Thursday): Mix Assertion, Logging, Unit Testing and Fuzzing
- Pete Muldoon (Wednesday) Dependency Injection in C++

- "Accelerated TDD" by Phil Nash just finished
- "C++ Testing like a Ninja for Novice Testers" next weekend: Jorge D. Ortiz-Fuentes and Rishabh Bisht

Learning Unit Testing

CppCon 2015: T. Winters & H. Wright “All Your Tests are Terrible..."



Rule 0

Write unit tests

Part 0: Basics of the Basics

Unit test*ing* is the act of testing the correctness of your code at the smallest possible unit: the function.

Unit test*s* are small, automated, stand alone executables that perform unit test*ing* on your code.

```
auto abs(int x) -> int {
  if (x >= 0) return x;
  else return -x;
}
```

"Code Under Test" (CUT)
"System Under Test" (SUT)

```
#include <cassert>

int main()
{
    assert( abs( 5) == 5 );
    assert( abs(-5) == 5 );
}
```

```cpp
auto abs(int x) -> int {
  if (x >= 0) return -x;
  else return -x;
}
```


OOPS

```cpp
#include <cassert>

int main()
{
    assert( abs( 5) == 5 );
    assert( abs(-5) == 5 );
}
```

```
program returned: 139
Program stderr

output.s: /app/example.cpp:5: int main(): Assertion `abs(5) == 5' failed.
Program terminated with signal: SIGSEGV
```

Unit Test Frameworks

Catch2

https://github.com/catchorg/Catch2

Phil Nash

Modern C++ Testing with Catch2

math.hpp

```
auto abs(int x) -> int;
```

math.cpp

```
1 auto abs(int x) -> int {
2   if (x >= 0) return x;
3   else return -x;
4 }
```

test_math.cpp

```
1 #include <catch2/catch_test_macros.hpp>
2 #include "math.hpp"
3
4 TEST_CASE("Absolute value tests"){
5     CHECK( abs( 5) == 5);
6     CHECK( abs(-5) == 5);
7 }
```

A test case tests one "thing" about the CUT

All the test cases for a function (class, header file) are the "Test Suite" for that thing

math.hpp

```
auto abs(int x) -> int;
```

math.cpp

```
1 auto abs(int x) -> int {
2   if (x >= 0) return x;
3   else return -x;
4 }
```

test_math.cpp

```
1 #include <catch2/catch_test_macros.hpp>
2 #include "math.hpp"
3
4 TEST_CASE("Absolute value tests"){
5     CHECK( abs( 5) == 5);
6     CHECK( abs(-5) == 5);
7 }
```

```
Program returned: 0
===============================================================
All tests passed (2 assertions in 1 test case)
```

```cpp
auto abs(int x) -> int {
  if (x >= 0) return -x;
  else return -x;
}
```


OOPS

```
Randomness seeded to: 3662634875

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
output.s is a Catch2 v3.7.0 host application.
Run with -? for options

-------------------------------------------------------------------------------
Absolute value tests
-------------------------------------------------------------------------------
/app/example.cpp:4
.......................................................

/app/example.cpp:5: FAILED:
  CHECK( abs( 5) == 5 )
with expansion:
  -5 == 5

===============================================================================
test cases: 1 | 1 failed
assertions: 2 | 1 passed | 1 failed
```

Test case name

File and line number of the test case

File and line number of the failure

Assertion that failed

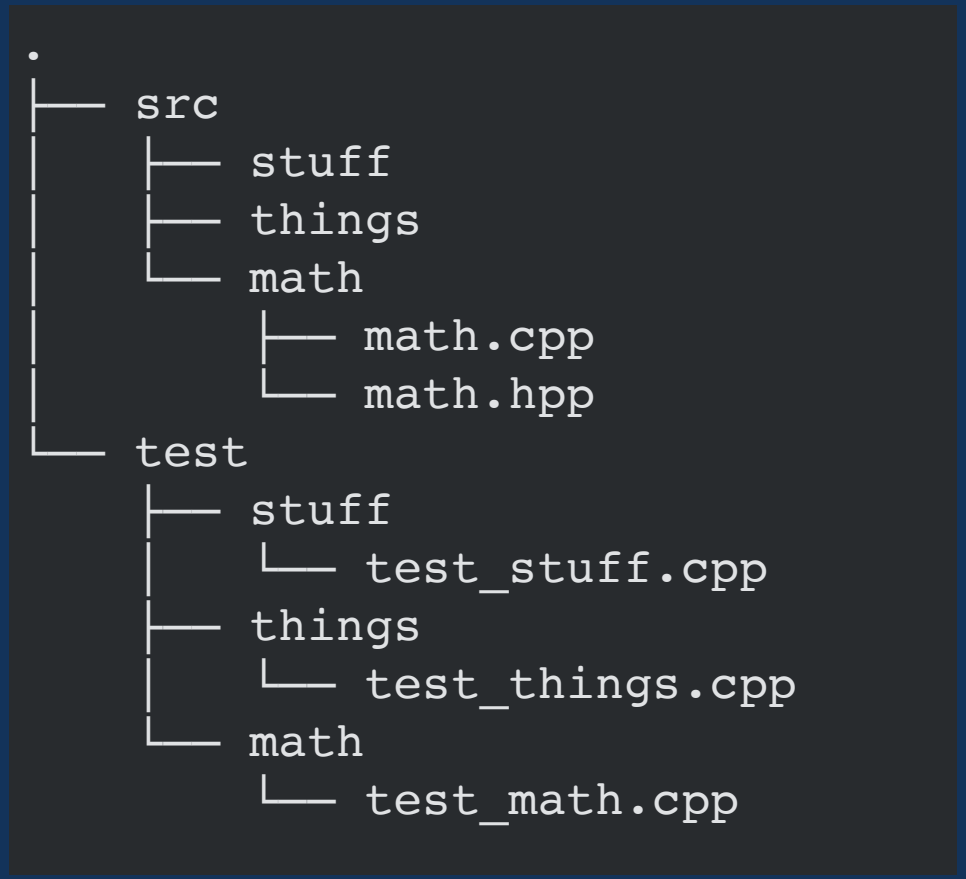
Values that caused the faiure

Test suite overall status

Use a good unit test framework.

(And learn how to use it)

Part 2: Care and Feeding

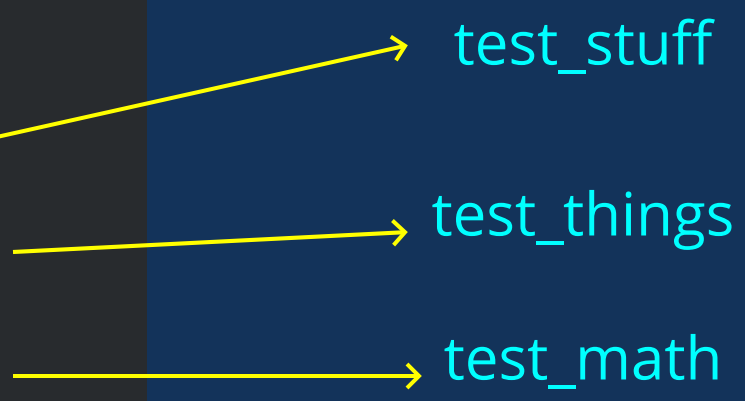


Part 2: Anatomy

```
.
├── src
│   ├── stuff
│   ├── things
│   └── math
│       ├── math.cpp
│       └── math.hpp
└── test
    ├── stuff
    │   └── test_stuff.cpp
    ├── things
    │   └── test_things.cpp
    └── math
        └── test_math.cpp
```

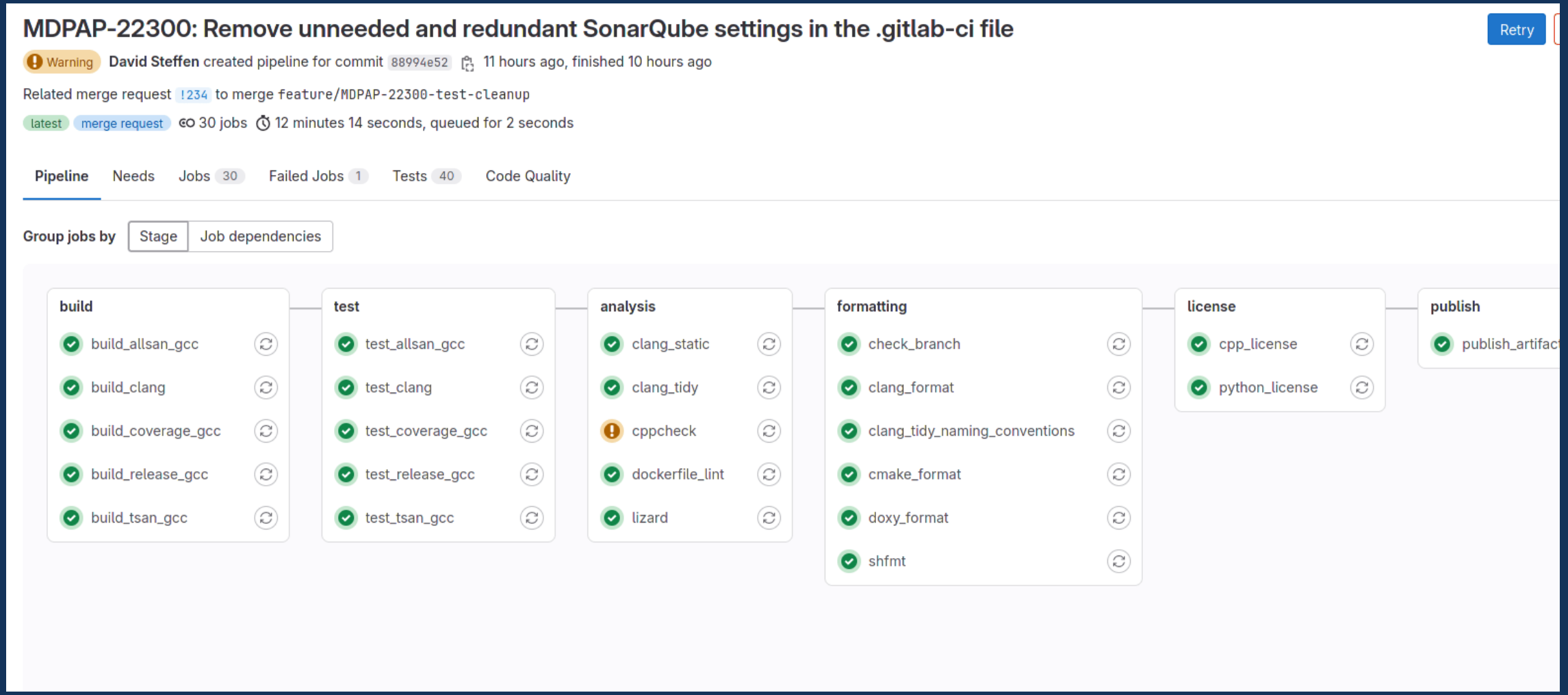
Part 2: Anatomy

```
.
├── src
│   ├── stuff
│   ├── things
│   └── math
│       ├── math.cpp
│       └── math.hpp
└── test
    ├── stuff
    │   └── test_stuff.cpp
    ├── things
    │   └── test_things.cpp
    └── math
        └── test_math.cpp
```



- Fast to build
- Fast to run

CI Pipelines



Titus Winters, Tradeoffs in the Software Workflow, ACCU 2022
https://youtu.be/l6Q7XaTleyI

Part 0 summary

Unit test*ing* is the act of testing the correctness of your code at the smallest possible unit: the function.

Unit test*s* are small, automated, stand alone executables that perform unit test*ing* on your code.

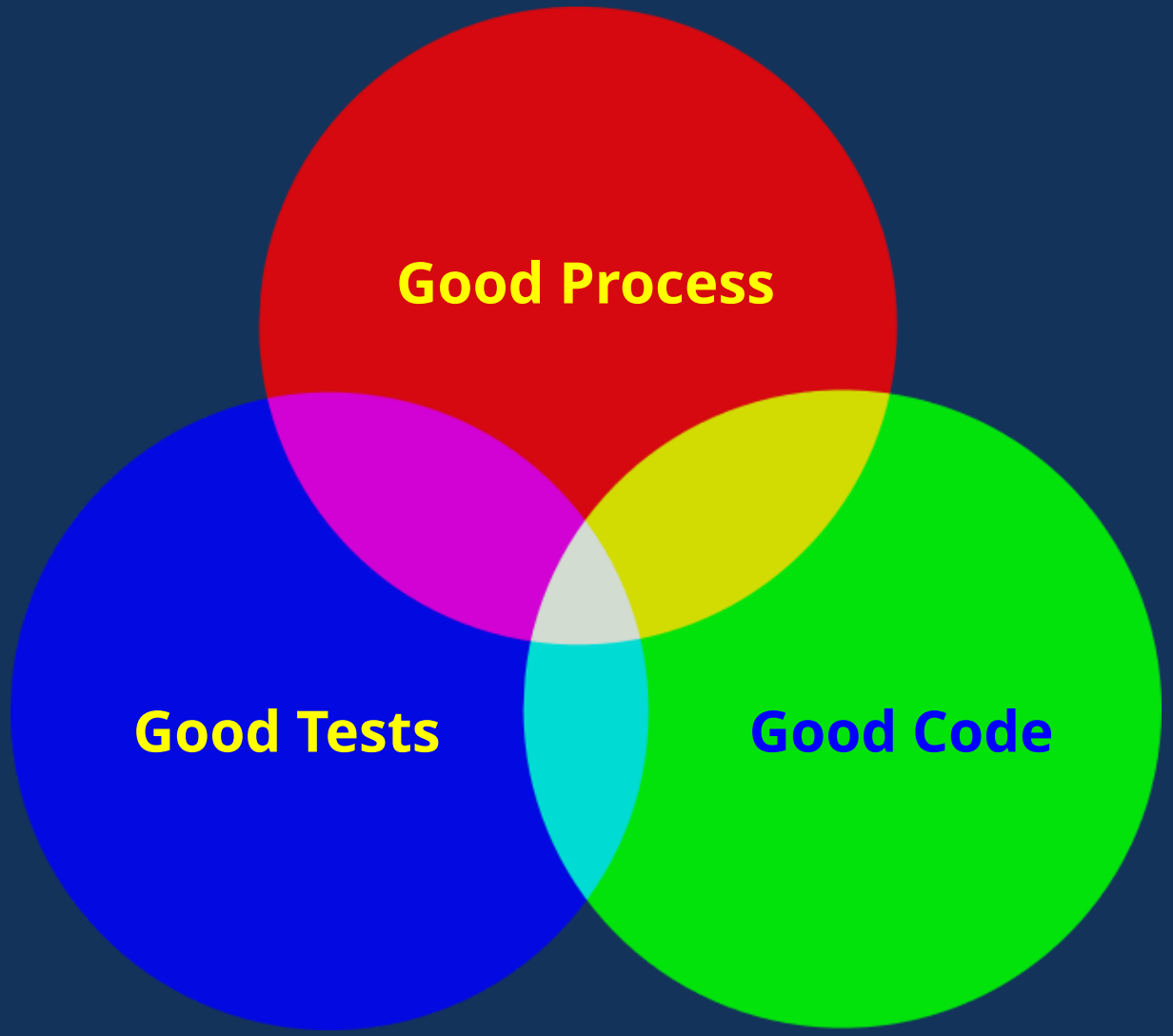
Use a Unit Test Framework

- Unit tests are the first line of defense (but not the only line)
- Unit tests are an integral part of the code base

Rule 0

Write unit tests

DOMAINS



Part 1: Good Tests

Falsifiability

Karl Popper's Falsifiability Criterion:

We cannot generally prove scientific theories true. We can only prove them false.

Program testing can be used to show the presence of bugs, but never to show their absence

-Edsger Dijkstra, *Notes on Structured Programming* 1970

We cannot prove our programs correct.

We *can* attempt to prove them *incorrect* and fail

Confidence tracks the quality and thoroughness of testing

We must always make falsifiable hypotheses

Take some good advice from science

A good experiment is:

- Repeatable
- Replicable
- Accurate
- Precise
- Reliable
- Hermetic
- Calibrated
- …

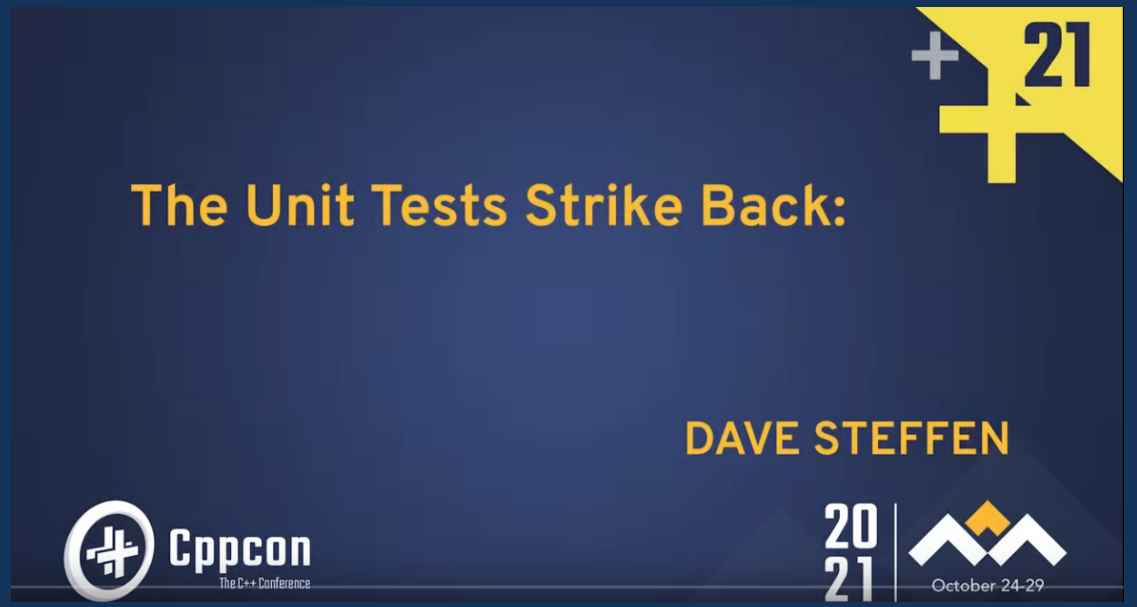
Repeatable and Replicable

Repeatable: you get the same answer every time

Replicable: your colleages get the same answer you do

Dealing with non-repeatable tests?

See my talk from three years ago:



Hermeticity

Hermeticity: Sealed off from environmental effects

Hermetic: Airtight, impervious to external influence. (Merriam Webster)

Hermetic tests: tests run against a test environment (i.e., application servers and resources) that is entirely self-contained (i.e., no external dependencies like production backends). (https://abseil.io/resources/swe-book/html/ch23.html)

Any time bits enter or leave your unit test, your test results also depend on things you're not trying to test and don't control.

Solution: "Mocks" (and test doubles)

Precison and Accuracy

- Accuracy: measurements are "right"
- Precision: measurements are "informative"

Dave Steffen

"It's a Bug Hunt" - Armor Plate Your Unit Tests

Precision for Unit Tests

How fast can we move from

"we know there's a problem"

to

"we know where the problem is

and can start handling it"

Precision

- Clarity: output should be complete and easy to understand
  - Good output when a test fails
    - File name and line number; test case name, assertion, values, etc
    - All good unit test frameworks do this. *Use Them*
- Organization: A failure should show up in the tests for that code, and as few other places as possible
  - Don't mix tests; a given test case should only have asserts for the behavior in question

Confusing or ambiguous results aren't a problem most of the time;
they are a big problem under pressure

Mocks:
A(nother) Tale of Two Cities

Test Doubles (Mocks, Fakes, Stubs) are a technique for writing unit tests mainly to improve *hermeticity* of the test

There are two competing schools of TDD:

- Detroit / Classicist
- London / Mockist

Adrian Booth, *Test Driven Development Wars: Detroit vs London, Classicist vs Mockist*

https://medium.com/@adrianbooth/test-driven-development-wars-detroit-vs-london-classicist-vs-mockist-9956c78ae95f

Maciej Falski, *Detroit and London Schools of Test-Driven Developement*

https://blog.devgenius.io/detroit-and-london-schools-of-test-driven-development-3d2f8dca71e5

Test Doubles are accuracy / precision tradeoffs

Using Test Doubles improves precision:

- Your class or test uses a Thing
  - If Thing fails, its tests fail *and your tests fail* *(less precision)*

- Use a MockThing in your tests
  - If Thing fails, its tests fail *and nothing else* *(more precision)*

London / Mockist tests are precise;
no red herrings, no extraneous
signals to sort through

Test Doubles are accuracy / precision tradeoffs

Using Test Doubles *reduces* accuracy:

- Your class or test uses a Thing
  - Tests use real Things (*more accurate*)
- Use a MockThing in your tests
  - Tests use non-real MockThings (*less accurate*)

Every difference between the Test Double and the Real Thing is a gap the bugs can come through.

Are your mocks changing to match changes in the real thing?

Plus, have you tested your Mocks?

Beware circular reasoning!

Accuracy in science

- Accuracy: measurements are close to the "right" answer
  - Make a falsifiable hypothesis: "Our code has no bugs"
  - Finding a bug is a positive result

True Positive P_T : Test fails and there is a bug

True Negative N_T : Test passes because there are no bugs

False Positive P_F : Test fails but *there is no bug*

False Negative N_F : Test passes but *there are bugs*

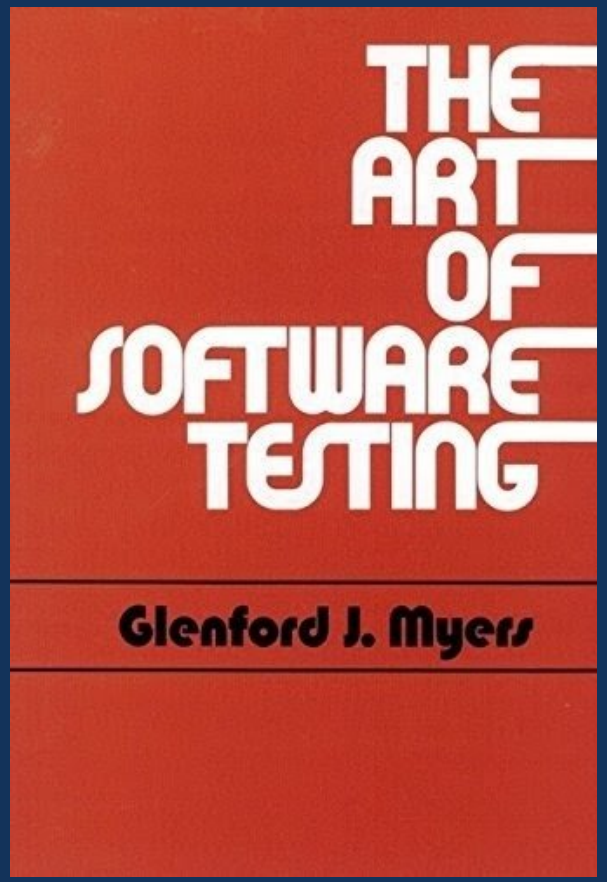
Accuracy

The result of your test matches reality of code

- Test completely : give bugs no place to hide
- Test Correctly : correct statement of "right" answer
- Test Validity: no circular logic or unfalsifiable tests

Completeness

The full story:

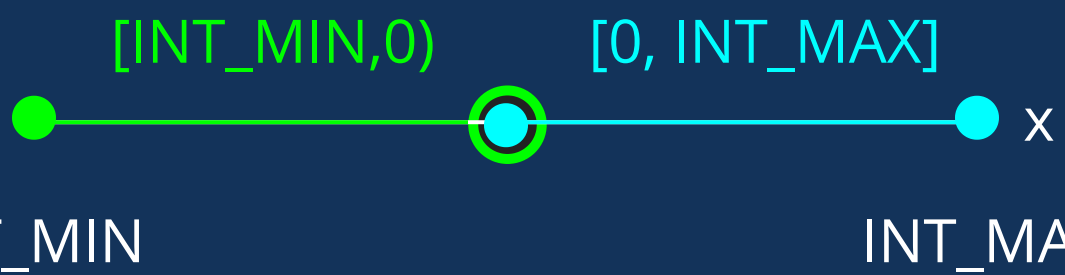


Wiley & Sons 1979
ISBN 0-471-04328-1

(We'll do the simple version!)

Step 1: Identify execution paths

```
1 auto abs(int x) -> int {
2   if (x >= 0) return x;
3   else return -x;
4 }
```



Domain:
[INT_MIN, INT_MAX]

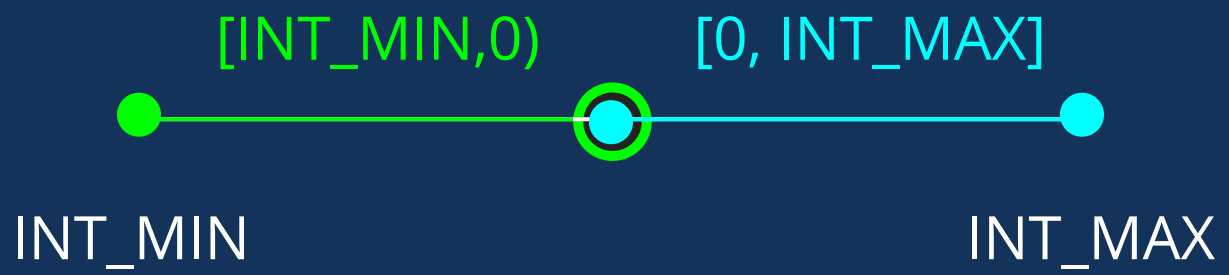
```
#include <catch2/catch_test_macros.hpp>
#include "math.hpp"

TEST_CASE("Absolute value tests"){
    CHECK( abs(5)  == 5);
    CHECK( abs(-5) == 5)
```

Step 2: Boundary Conditions

```
1 auto abs(int x) -> int {
2   if (x >= 0) return x;
3   else return -x;
4 }
```

- Check the boundaries for additional test cases
  - Places where behavior changes
  - Places where easy mistakes live (< vs <=)



- INT_MAX : nothing new
- 0 : nothing new
- -1 : nothing new
- INT_MIN: whoops....

INT_MAX = +2147483647

INT_MIN = -2147483648

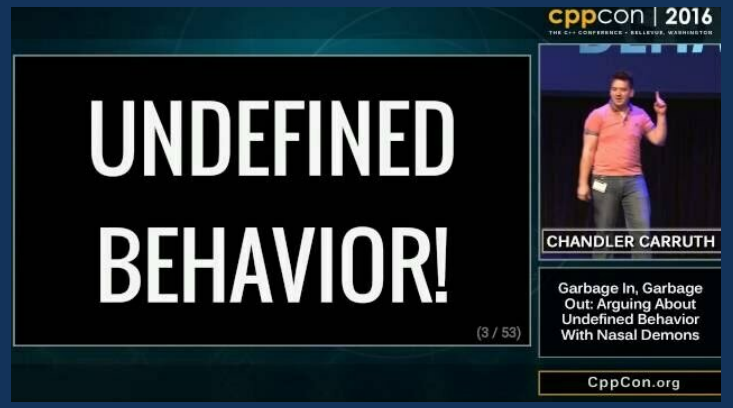
UNDEFINED BEHAVIOR

Ansel Sermersheim and Barbara Geller, CppCon 2021

Back to Basics: Undefined Behavior

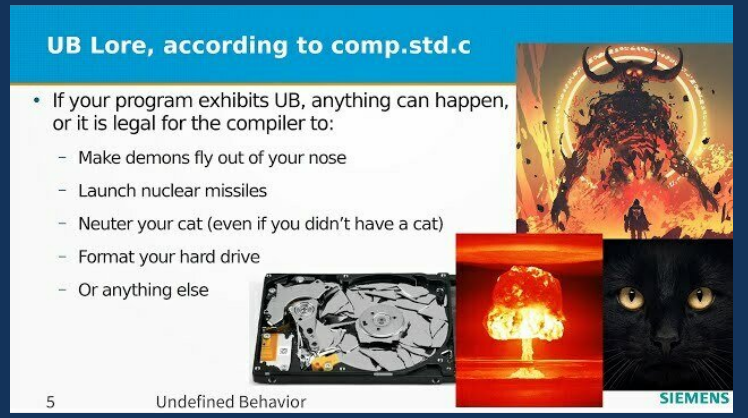
Chandler Carruth, CppCon 2016

Garbage In, Garbage Out: Arguing about Undefined Behavior...



Fedor Pikus CppCon 2023

Undefined Behavior in C++: What Every Programmer Should Know And Fear



Part 1: Testing	Precision	Completeness

```
1 auto abs(int x) -> int {
2   if (x >= 0) return x;
3   else return -x;
4 }
```

- Non-negative integer: [0, INT_MAX]
- *Valid* Negative integers: (INT_MIN, 0)
- Invalid integer: INT_MIN

as written, INT_MIN is *untestable*.

Don't test invalid input

(there's no valid result!)

What if we rewrite to check for errors?

```
1 auto abs(int x) -> int
2 {
3   if (x == std::numeric_limits<int>::min())
4     throw std::domain_error("Can't take abs of INT_MIN")
5
6   if (x >= 0) return x;
7   else return -x;
8 }
```

abs(INT_MIN) is now defined behavior

Now, the error case is a valid equivalence class we should test.

```cpp
auto abs(int x) -> int
{
  if (x == std::numeric_limits<int>::min())
    throw std::domain_error("Can't take abs of INT_MIN")

  if (x >= 0) return x;
  else return -x;
}
```

```cpp
#include <catch2/catch_test_macros.hpp>
#include "math.hpp"

TEST_CASE("Absolute value tests"){
    CHECK( abs(5)  == 5);
    CHECK( abs(-5) == 5)
}

TEST_CASE("Abs of INT_MIN is an error") {
    CHECK_THROWS_AS(
       abs(INT_MIN),
       std::domain_error);
}
```

Test Correctly

How do you know the result you're expecting is the right result?

```
float  compute_pi() { return std::acos(-1); }

TEST_CASE("Compute Pi") {
    float correct_value = ???
    CHECK( compute_pi() == correct_value );
}
```

How do we know what goes here?

The "Test Oracle" problem: from where do we get our "correct" results?

- External Authority (Archimedes, ~250 BC
- Simple cases that can be worked out
- Known properties of the result

Think about what the correct result is and how you know!

Alternatives

"Property Testing" : something that ought to be true of the output

- abs(x) : all results are positive
- Random number generator: results are not correlated

Can be used with large or automatically generated inputs

These kinds of tests are less "accurate" but can be very useful when "correct" is hard to characterize

Test Validly

```
float  compute_pi() { return std::acos(-1); }

TEST_CASE("Compute Pi") {
   float correct_value = what_I_got_by_running_it;
   CHECK( compute_pi() == correct_value );
}
```

Don't do this!

If your Test Oracle is your own code you do not have a falsifiable test.

We have just proven that "The code we wrote is the code we wrote" **

** Dave Farley, "The 3 Types of Unit Test in TDD" https://youtu.be/W40mpZP9xQQ
YouTube channel: https://www.youtube.com/c/ContinuousDelivery

Test Validly

```
float  compute_pi() { return std::acos(-1); }

TEST_CASE("Compute Pi") {
   float correct_value = std::acos(-1);
   CHECK( compute_pi() == correct_value );
}
```

Don't do this either

We have just proven that "The co[illegible]ve wrote is the code we wrote" **

REMOVE???

[illegible]s causes other problems too)

** Dave Farley, "The 3 Types of Unit Test in TDD" https://youtu.be/W40mpZP9xQQ

YouTube channel: https://www.youtube.com/c/ContinuousDelivery

Test Validity

```
float  compute_pi() { return std::acos(-1); }

TEST_CASE("Compute Pi") {
   float correct_value = what_I_got_by_running_it;
   CHECK( compute_pi() == correct_value );
}
```

Don't do this!

Acceptance Test (in this context):
The code does what it did last time

Extremely valuable when dealing with legacy or troublesome code that you don't understand

Clare Macrae: https://claremacrae.co.uk/

Quickly Testing Legacy C++ Code with Approval Tests

Part 1: Good Tests Summary

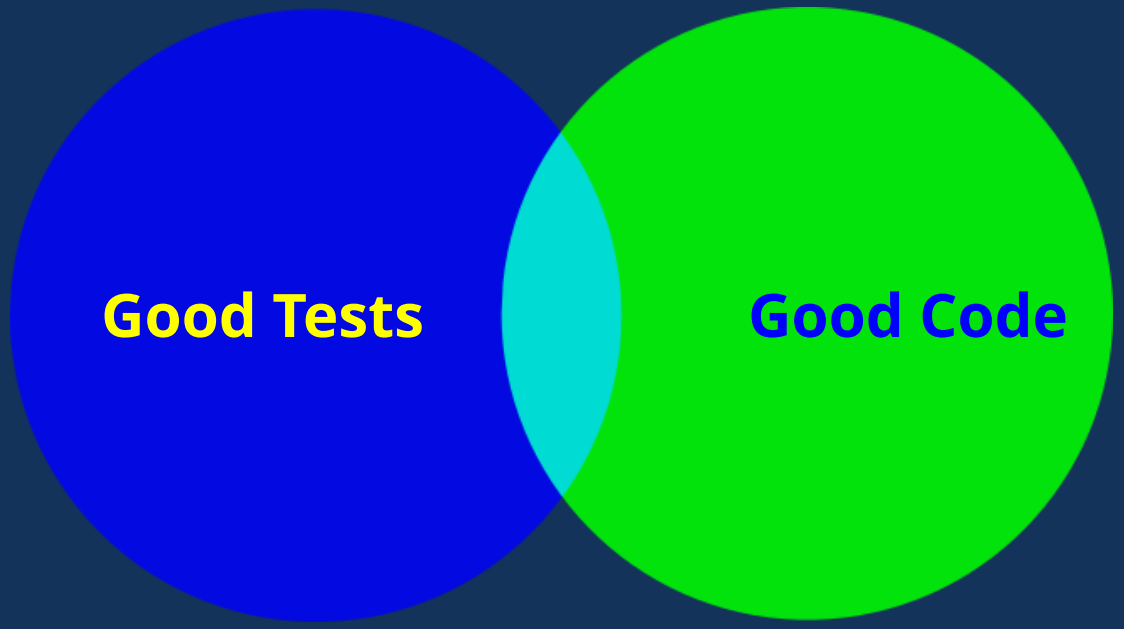
Tests as science:

- Falsifiable Hypotheses
  - Tests are here to detect bugs
- Repeatable
- Replicable
- Accurate
- Precise

Rule 0

Write unit tests

Part 2: Good Code



Three Aspects:

- Readable
- Maintainable
- Documentary

1) Readability

What does "good code" mean for unit tests?

Unit Tests must be correct by inspection

- Therefore, easily to inspect.

Simplicity!

"Simplicity is prerequisite for reliability."
— Edsger W. Dijkstra

Jon Jagger: Unit tests should have a cyclomatic complexity of 1

- Unit Tests should have no *logic* other than testing
- Conditionals should be rare (other than the assertions)
- Loops should be even more rare.

Aspirational Goal,

Not Rule

Readability

```
float  compute_pi() { return std::acos(-1); }

TEST_CASE("Compute Pi") {
   float correct_value = std::acos(-1);
   CHECK( compute_pi() == correct_value );
}
```

pretend this is 30 lines of complex math

SKIP

We have just proven that "The code we wrote is the code we wrote" **

It also makes your unit tests just as hard to understand as your code

If your unit test are *not* simpler and easier to read than the code they're testing, there's a problem

Readability

```
TEST_CASE("complexity") {
    XXXX XXX X XXXXXXXXXXXXXXXXXXXXXXXXXXXXX

    XXXX XXXX X XXXXXXXXXXXXX

    XXXXXXXXXXXXXX XXXXXXX
    XXXXXXXXXXXXXXXXXXXXXXXXX
    XXXXXXXXXXXX XX XXX
    XXXXXXXXXXXX XX XXXXXXX

    XX XXXX XXXXXXXX XXXXXXXXXXX
    XXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXXXXXXXXXXXXXX
    XXXXXXXXXXXX XX XXX
    XXXXXXXXXXXX XX XXXXXXX
}
```

Structure for Readability

The "AAA" pattern:

- Arrange
- Act
- Assert

Behavior-Driven Development:

- Given
- When
- Then

REMOVE ???

Wright and Winters: "Tests should read like a novel"

- Three acts: setup, confrontation, resolution
  - Set up the situation
  - Execute the test
  - Check the results
- Each should flow from the previous act with inevitability

The ultimate unit test readability goal:

Reviewers can tell your unit tests are correct
even if they haven't read the code under test
even if they don't know what it's supposed to do

Step 3: Interesting Conditions

"Interesting" values derive from studying the algorithm and the code.

Interesting is decided by you. You wrote the code, you know what's interesting.

- 0 is an interesting value. *Make it an EC if you want to!*
- "Interesting" is in the eye of the beholder.
  - Consider testing values that *look* important to reassure readers that this particular case is handled.

Detour: Testing Classes

A class in C++ is a user-defined type or data structure ... that has data and functions (also called member variables and member functions) as its members whose access is governed by the three access specifiers *private*, *protected* or *public*.

-- Wikipedia

From our original definition of Unit Testing,
we therefore test classes by testing their
member functions.

- Kevlin Henney "Structure and Interpretation of Test Cases"
- Kevlin Henney "Test Smells and Fragrances"

Let's unit test this class

```
// Cup.h

class Cup {
  public:
    Cup();                  // creates an empty cup

    bool IsEmpty();

    bool Fill();            // if empty, fills completely, returns true
                            // otherwise leaves cup full, return false

    bool Drink();           // If full, empties the cup and returns true
                            // otherwise leaves cup empty and returns false
  private:
    ...
}
```

The usual approach

```
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 "usual" approach:
Test each member function individually.

```cpp
// Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Fill();
    bool Drink();
  private:
    bool empty_;
}
```

```cpp
#include "Cup.h"

// test the constructor
TEST_CASE( "Cup::Cup" )
{
  Cup cup; // new cups are supposed to be empty
  CHECK( cup.empty_ == true );
}
```

Test state of internals

Which doesn't compile!

Two options: looks technical

actually deeply philosophical

White box testing DONE WRONG

```
#define private public
```

- This is a horrible idea: formally, undefined behavior
- It actually works reliably on most compilers

Don't do this.

White Box /Black Box

White Box:

- Unit Test by reaching in and accessing internal state

(the question is, how)

Black Box:

- Unit Test by strictly though public member functions

(the question is, how)

In C++, Black Box seems mandatory

In other languages it's optional (Python)

... but highly recommended.

White box requires breaking encapsuation

Give access to a trusted friend

```cpp
class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Fill();
    bool Drink();

  private:
    bool empty_;
    friend struct CupTester
};
```

```cpp
struct CupTester {
  bool& is_empty;
  CupTester(Cup& c) : empty(c.empty_);
};

TEST_CASE( "Cup::Cup" )
{
  Cup cup; // new cups are supposed to be empty
  CupTester tester(cup);

  CHECK( cup_tester.is_empty );
};
```

Deeper discussion:

The Unit Tests Strike Back: Testing the Hard Parts

White box testing

Commonly used, sometimes recommended

- An obvious approach (friendly for beginners?)
- Makes testing easy
  - Direct access to internal state
  - Easy to arrange for specific situations
  - Easy to hit edge and error condition
- Sometimes it's the only option
  - Particularly for legacy code

White box Maintenance

White Box Tests are very tightly coupled to the code

```
1  // Cup.h
2
3  class Cup {
4    public:
5      Cup();
6      bool IsEmpty();
7      bool Fill();
8      bool Drink();
9    private:
10     bool empty_;
11
12     friend struct CupTester
13     ...
14 };
```

```
1  struct CupTester {
2    bool& is_empty;
3    CupTester(Cup& c) : empty(c.empty_);
4  };
5
6  TEST_CASE( "Cup::Cup" )
7  {
8    Cup cup; // new cups are supposed to be empty
9    CupTester tester(cup);
10
11   CHECK( cup_tester.is_empty );
12 };
```

The names of your member variables are referenced in your tests!

White Box testing tends to create big maintainability problems

Black Box

What's the alternative?
Test each member function. OK...

```
// Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Fill();
    bool Drink();
};
```

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

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

```
// Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Fill();
    bool Drink();
};
```

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

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

```
1 Cup::Cup() : empty_{false} {}
2
3 bool Cup::IsEmpty() const { return !empty_; }
```

Each mistake

hides 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.

Black Box: Behaviors

```
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 empties a filled cup" ) = []{...};
                        ...
```

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

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;
  CHECK( cup.IsEmpty() );
};
```

```
TEST_CASE( "An empty cup can be filled" ) {
  Cup cup;
  bool success = cup.fill();
  CHECK( success );
};
```

- Mutually consistent mistakes revealed by further test cases
- 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!

Back to 2nd Realm: Unit Tests as Good Code

The ideal:

The unit test is correct by inspection, *even for someone who hasn't read the code and doesn't know what it's supposed to do*

```
// A fried of the Cup class
struct CupTester {

  bool& is_empty;

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

TEST_CASE("Constructor")
{
   Cup c;
   CupTester tester(c)
   CHECK(tester.is_empty);
}
```

```
TEST_CASE("A new cup is empty")
{
   Cup c;
   CHECK( c.isEmpty () );
}
```

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( "Drinking from a full cup makes it empty" ) {
  // GIVEN: a full cup
  Cup cup;
  cup.fill(); // previously tested

  // WHEN: we drink (successfully)
  auto successful_drink = cup.drink() == true );
  CHECK( successful_drink );

  // THEN: the cup is empty
  CHECK( cup.isEmpty() );
};
```

Note what Black box + BDD gives us:

- Test case name describes design spec ("what we want it to do")
- Test code *shows how to do it*

Unit tests can document proper and intended use

Downsides of BB

Black Box testing is only possible
IF the public interface is
well designed.

Revisit definitions

White Box Testing

Unit test*ing* is the act of testing the correctness of your code at the smallest possible unit: the function.

Black Box Testing

Unit test*ing* is the act of testing the correctness of your code at the smallest possible unit: *behaviors*

of functions or classes

```
TEST_CASE("Absolute value tests"){
    CHECK( abs( 5) == 5);
    CHECK( abs(-5) == 5)
}
```

Prefer: a test case tests one behavior

Unit Tests as Good Code

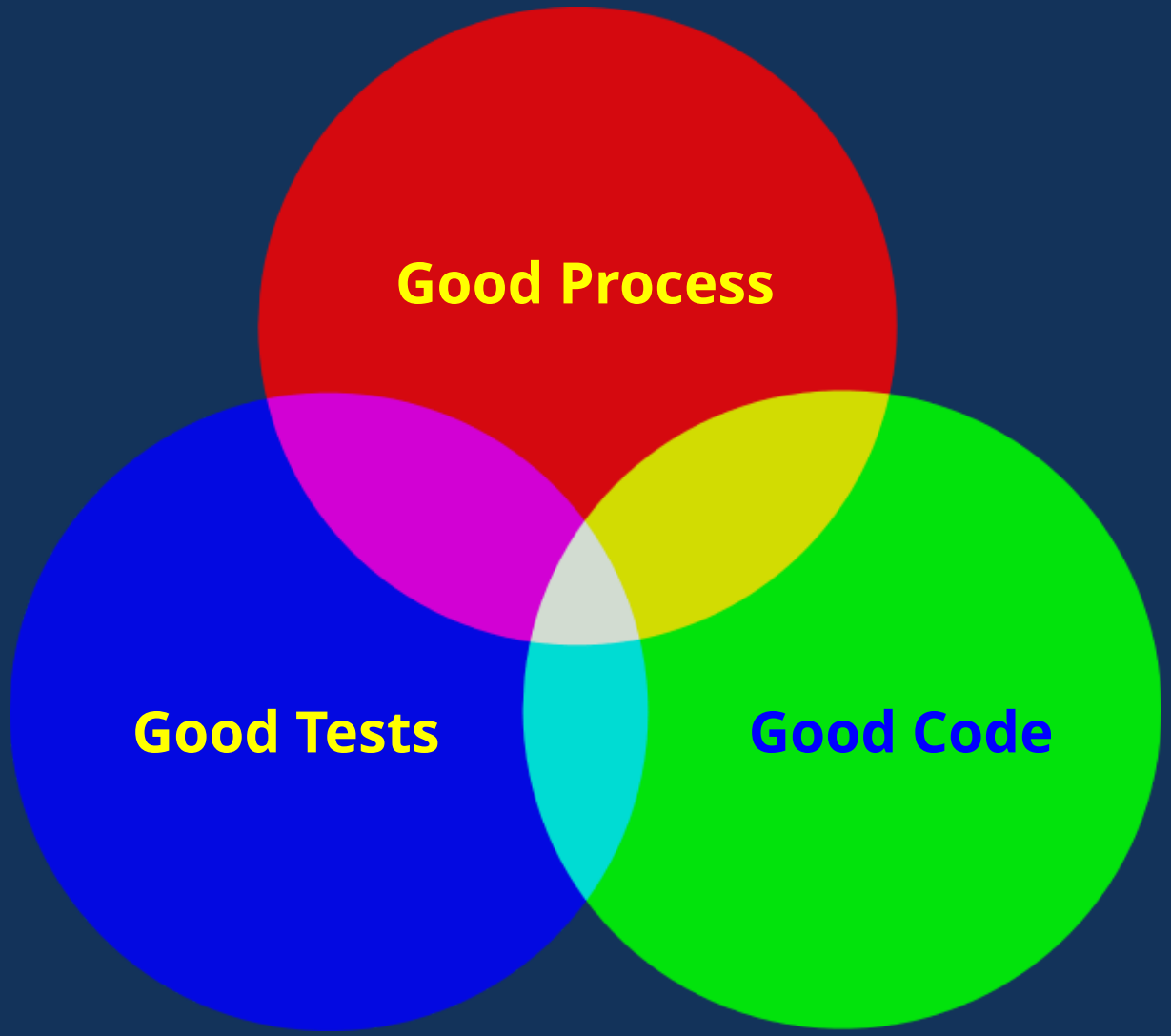
Summary:

- Readable
  - Simple (CC == 1)
  - Tell a story (three-act)
  - Correct-by-inspection (even if the reader doesn't know what it does)
- Maintainable
  - Minimize changes to UT code under maintenance (BB wins)
- Documentation (BB)
  - Max value of tests as docs.

Rule 0

Write unit tests

Part 3: Process



- Design
- TDD

This advice is of a very different nature:

We're not talking about code any more

Third Kingdom: Process: Workflow

- Unit tests are part of the CI/CD pipeline
- Unit tests are part of your development cycle (code, compile, run tests, code, compile, run tests...)

Therefore, unit tests need to run fast.

(usually, in seconds. Not minutes)

SUMMARIZE & REMOVE

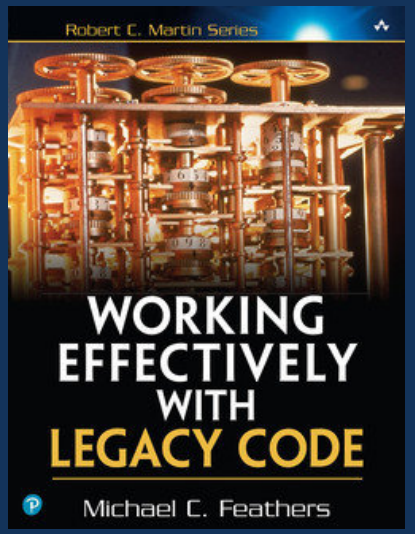
Remember "small, automated, standalone"

Also, therefore, unit tests must

- very very easy to set up
- very easy to run
- very easy to interpret (automatically)

Michael C. Feathers

https://www.artima.com/weblogs/index.jsp?blogger=mfeathers



Process: Design

Good Design (old)

- Simplicity (small functions, Single Responsibility Rule)
- Encapsulation and loose coupling
- SOLID
- Testability
- DRY
- ...

Process: Design

Good Design (new)

- Simplicity (small functions, Single Responsibility Rule)
- Encapsulation and loose coupling
- SOLID
- DRY
- ...

Testability

Third Kingdom: Process: Design

Test Early.

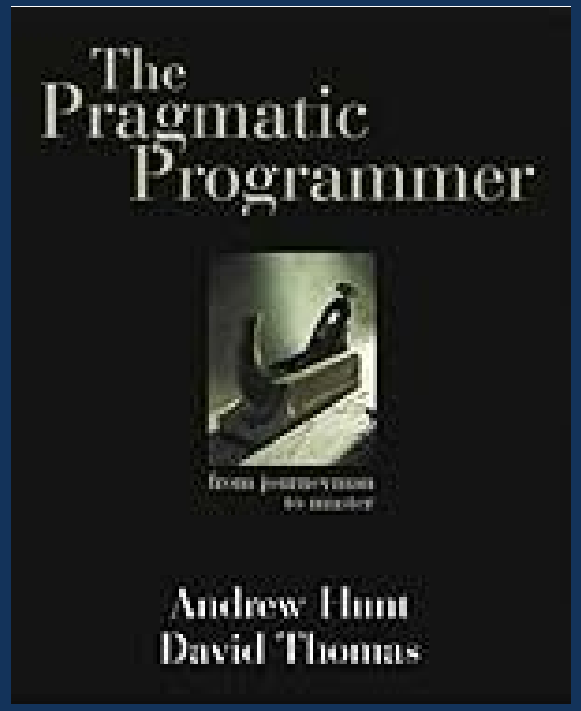
Test Often.

Test Automatically.

-- Andrew Hunt & David Thomas

The Pragmatic Programmer

Addison-Wesley 1999



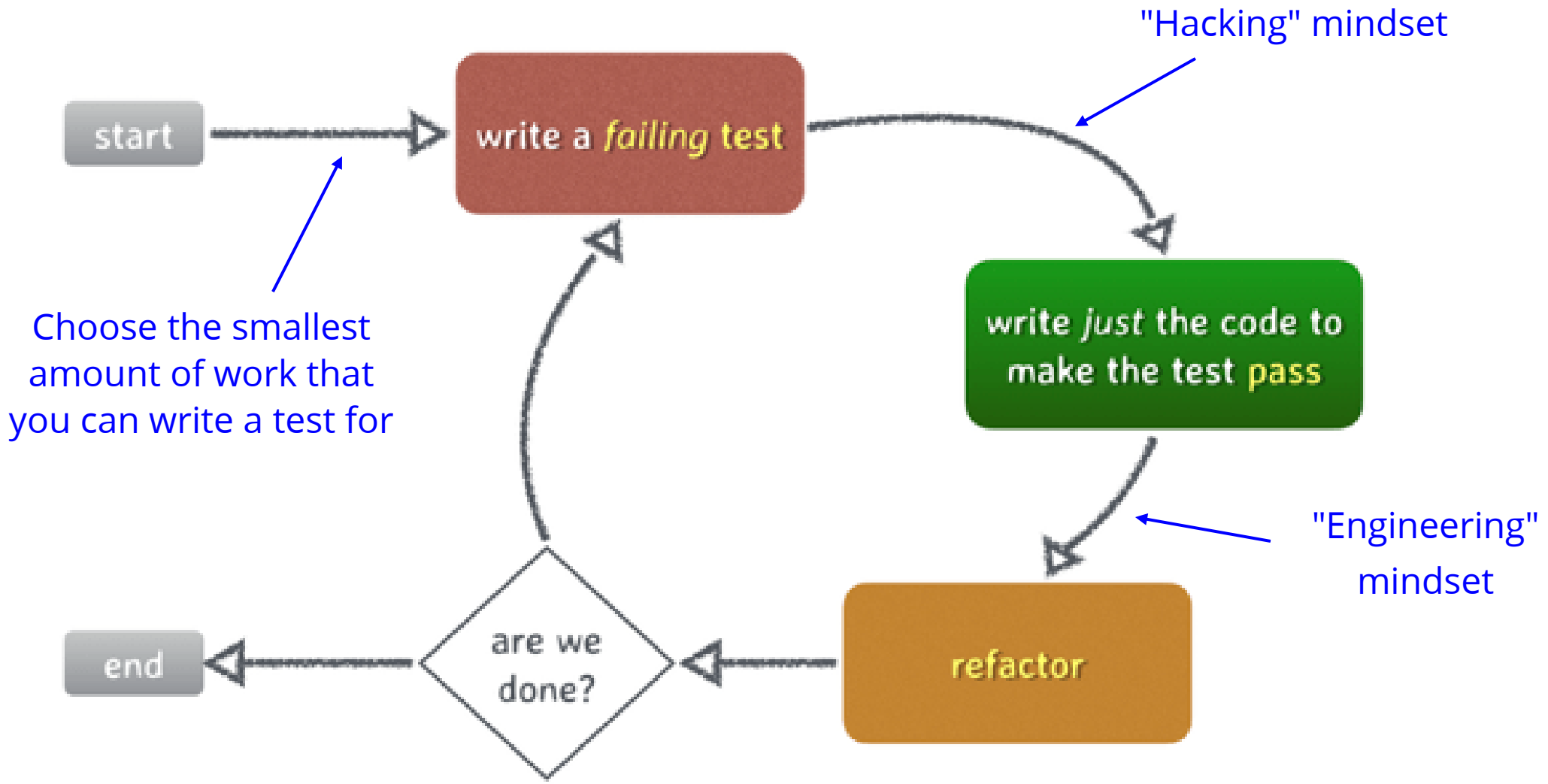
Don't leave testing for the end -- test as you go, let the tests improve your design as well as your code

Test Driven Development

Kent Beck developed TDD in the late 1990s as part of "Extreme Programming"

2002. *Test-Driven Development by Example*.
Addison-Wesley.(ISBN 978-0321146533)

TDD is complicated...



Courtesy of Phil Nash

TDD is difficult...

- The cycle should be *fast*
  - CI/CD piplines, unit tests, compilation... all have to be fast
- Constant switching between mental modes
- Has failure modes that need to be recognized and dealt with

  - Takes dedicated practice
  - Requires a new set of habits

TDD is worth it

“If debugging is the process of removing
software bugs, then programming must be
the process of putting them in.”
— Edsger W. Dijkstra

"You're not *writing the test* first -- you're *specifying what you want* first"
-- Kevlin Henney

"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

TDD references

Test Driven C++ - Phil Nash - CppCon 2020
https://youtu.be/N2gTxeIHMP0

Back to Basics: Test-driven Development - Fedor Pikus - CppCon 2019
https://youtu.be/RoYljVOj2H8

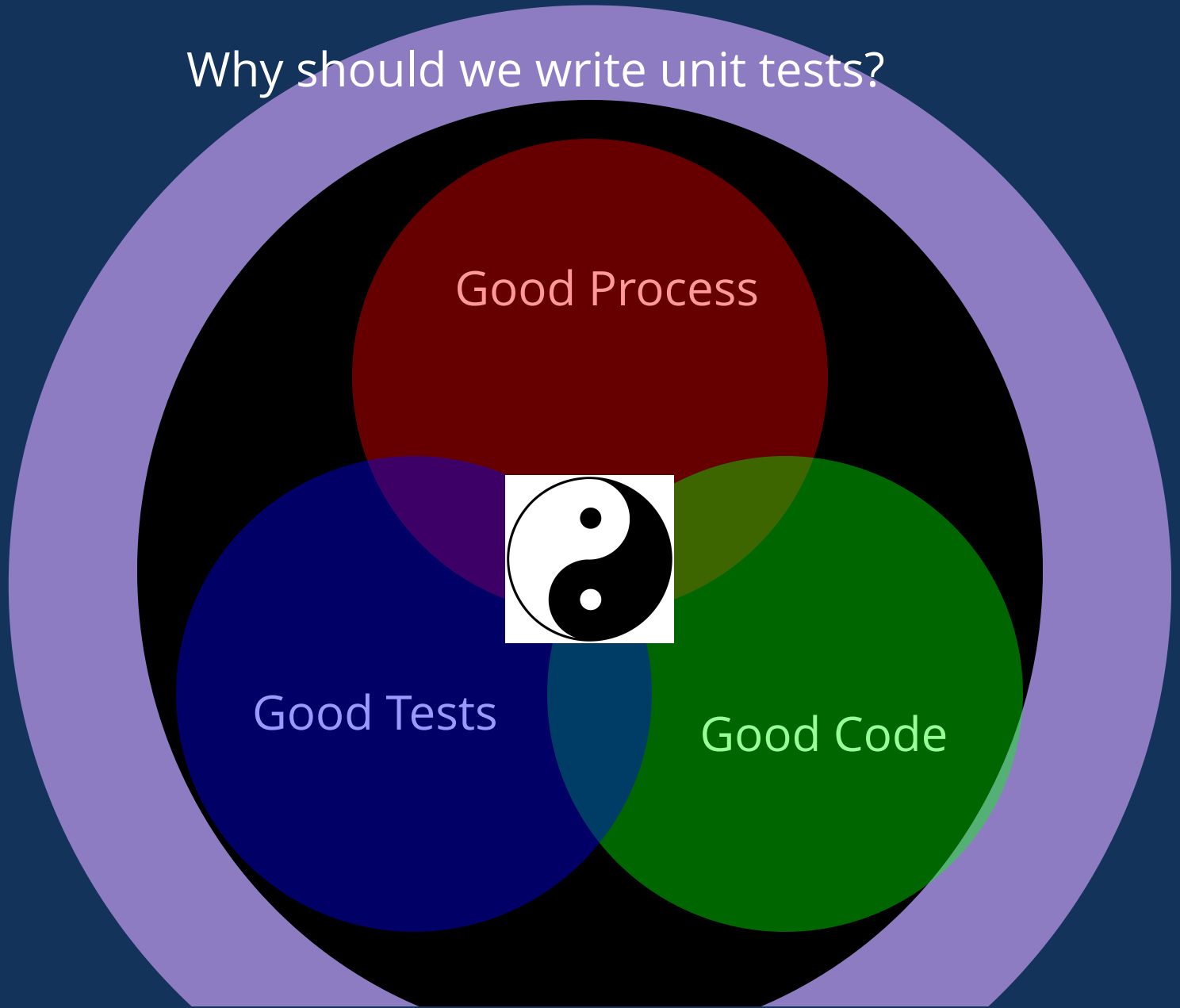
Kent Beck's Blog:
https://tidyfirst.substack.com/p/canon-tdd

∴ and about a zillion others

Part 3: Good Process

- Workflow
  - Running unit tests is part of the writing code cycle; they need to run fast
- Design
  - Good designs are testable, testable designs are good.
  - If your code is hard to test, revisit your design
- Test Driven Development
  - Red/Green cycle (writing the tests first is just a part of this)
  - Has a number of surprising benefits
  - Needs practice, and training. Take classes, read carefully, get a mentor

"Fourth Domain: WHY



- This is a 'category' of advice you will find; it has a very different nature
- It's not advice for *us.* It's advice for us to give to others: *managers*, new hires...

Part N: Why?

Why should we write unit tests?

Everything we just talked about, plus two important points:

- Ethics
- Psychology

Ethics

What are our responsibilities,

how to Unit Tests help?

- Future maintainers: tests and documentation
  - Future maintainers probably includes us
- Company / Employers / Project / Team
  - We need to deliver correct code
  - We need to deliver it efficiently
  - We need to show it's correct
- Customers
  - Software defects cause real problems to real humans

Up to Code - David Sankel - CppCon 2021
https://youtu.be/r_U9YFPWxEE

Ethics

If it's worth building, it's worth testing.

If it's not worth testing, why are you wasting your time working on it?

Scott Ambler

http://agiledata.org/essays/tdd.html

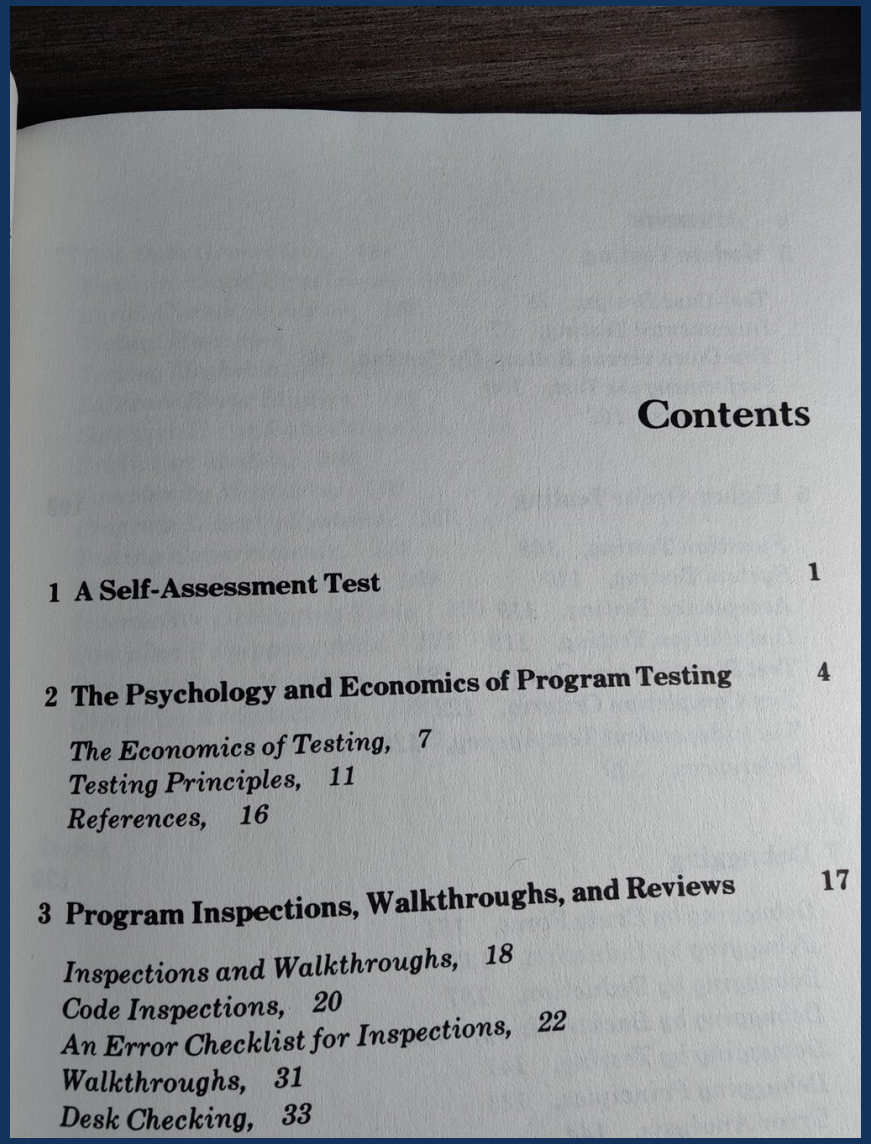
from Kevlin Henney -- What we talk about when we talk about unit testing https://youtu.be/-WWIeXmm4ec

Up to Code - David Sankel - CppCon 2021
https://youtu.be/r_U9YFPWxEE

Psychology

Glenford J Myers, "The Art of Software Testing" 1970:

"... such considerations as the feasibility of "completely" testing a program, knowing who should test a program, and adopting the appropriate frame of mind toward testing appear to contribute more toward successful testing than do the purely technological considerations."



Contents

Third Kingdom: Psychology

Myers: "Testing is the process of executing a program with the intent of finding errors"

- Humans are goal-oriented.
- "If our goal is to demonstrate that a program has no errors, then we shall subconsciously be steered toward this goal; ... we shall tend to select test data that have a low probability of causing the program to fail"
- Testing is a *destructive* process

REMOVE

Note, this matches the philosophy of modern science discussed earlier.

- The first thing you do with your pet [theory | code] is to hit it with a sledge hammer to see what breaks.
- This is not a normal thing for humans; this takes training and practice.

"The first principle is not to fool yourself - and you are the easiest person to fool"

-- Richard Feynman

Third Kingdom: Psychology

Psychological safety

Are you afraid of your code

Kate Gregory "Emotional Code" https://youtu.be/uloVXmSHiSo

- Checking what doesn't need to be checked

```
1 if (pPolicy){ delete pPolicy; }
2
3 auto int_vals = std::vector<int>();
4 assert (int_vals.empty());
```

- "I can't be sure I'm right"
- "I can't be sure I'll be looked after"
- "That was a team-mate's code, I don't know what they wanted"
  - or, "I don't know if they did it right"
- "I've been hurt before"

Title Text

FEAR

"You can't bully someone into not being afraid"

(threatening to fire someone because they do that stuff doesn't help)

The path out of fear is confidence.

- "I don't know if I'm right" -- test and find out
- "I don't trust those yokels down the hall" -- test and find out
- "I don't know if I'll be OK with this change" -- that's what tests are for
- "My job is safe if we can all do this"

Science

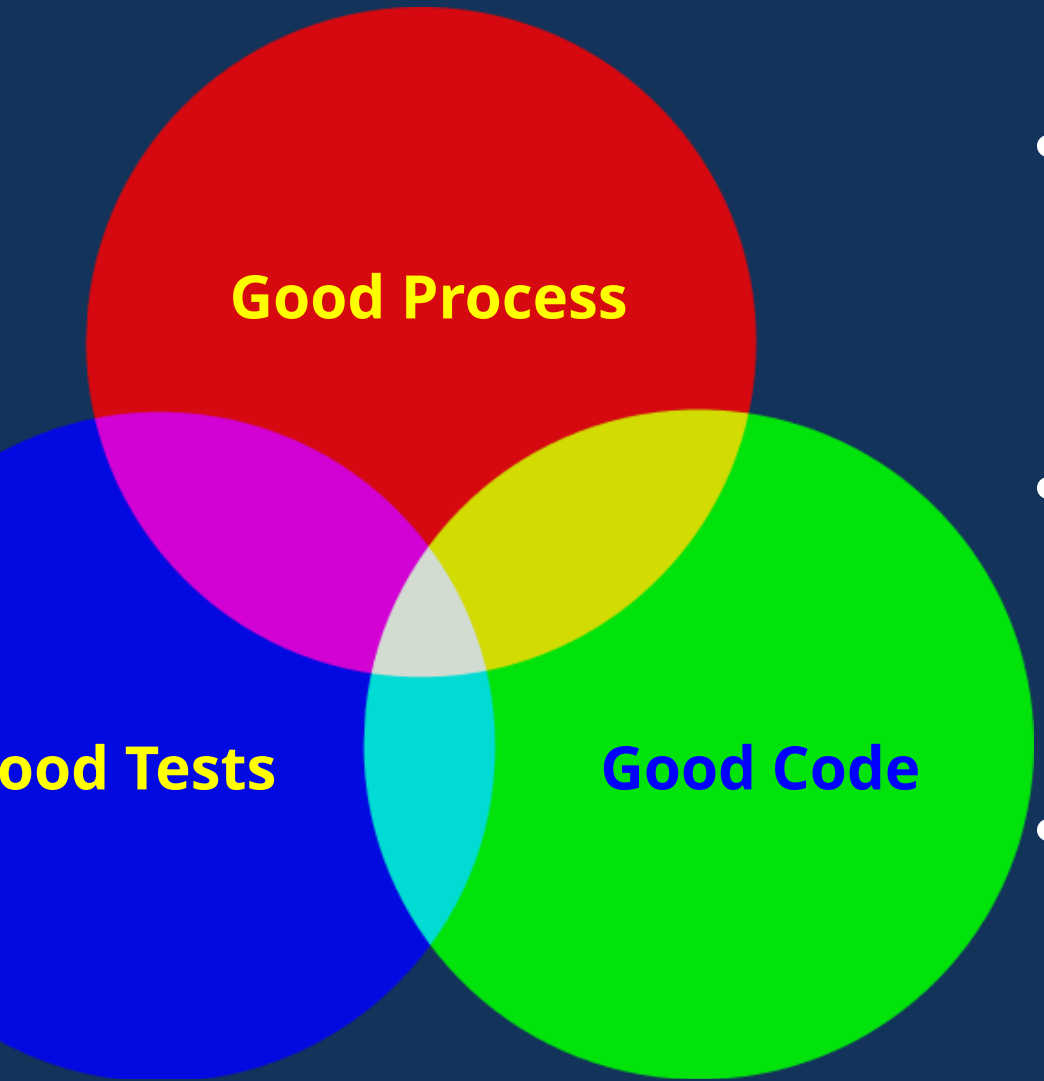
And we come back to science

We used to be afraid of solar eclipses: "What's eating the sun?"

Now we have a party.

We used to be afraid of comets

how to Learn Unit Testing



- Good Tests:
  - Replicable and Repeatable
  - Accurate and Precise
  - Falsifiable Hypotheses
- Good Code:
  - Readable
  - Maintainable
  - Documentary
- Good Process
  - Design
  - TDD

With a side helping of Black Box / White Box Philosophy

Science

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

-- Randall Munroe, XKCD

https://xkcd.com/701

You don't write unit tests to show your
code is correct

You write unit tests so that your code
becomes correct.

Science

Good tests kill flawed theories; we remain alive to guess again.

-- Karl Popper, "The Logic of Scientific Discovery", 1934

Good tests kill bugs! We remain alive to compile again!

Bad tests kill some bugs! We might remain alive to debug another day

No tests kills projects!

Rule 0

Write unit tests

Dave Steffen

Principal Software Engineer

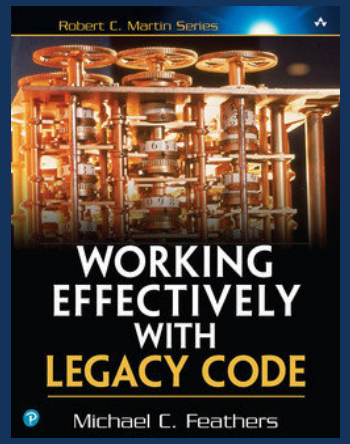
dsteffen@scitec.com

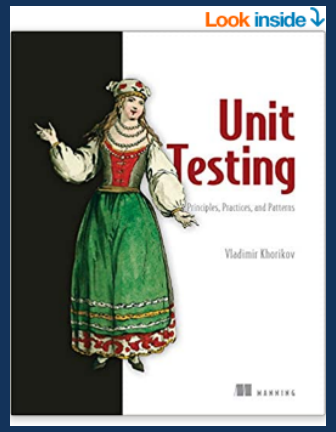
https://scitec.com/

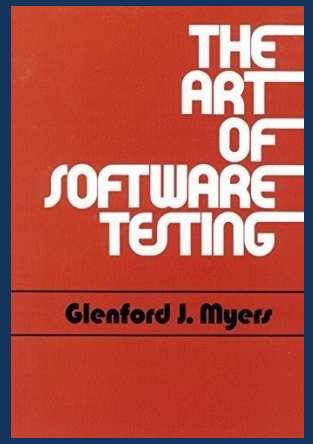
REFERENCES

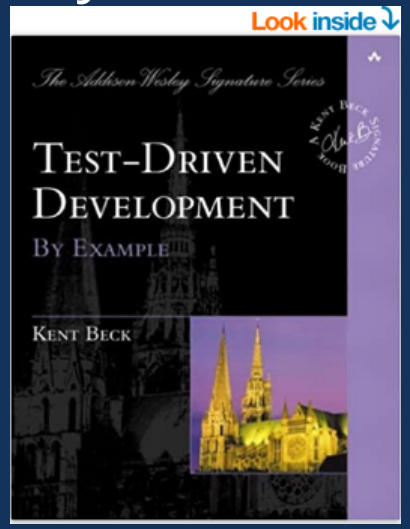
Unit Testing is a Big Topic

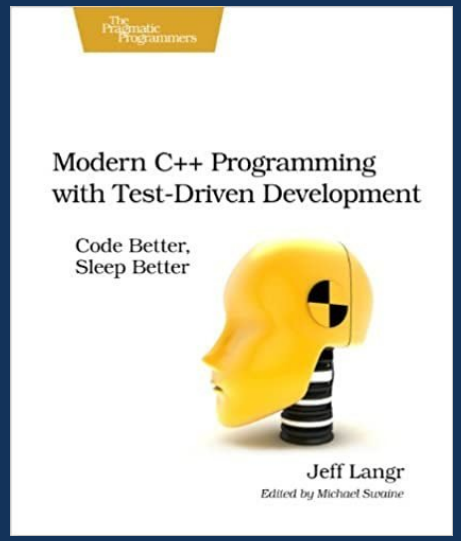
So many books…

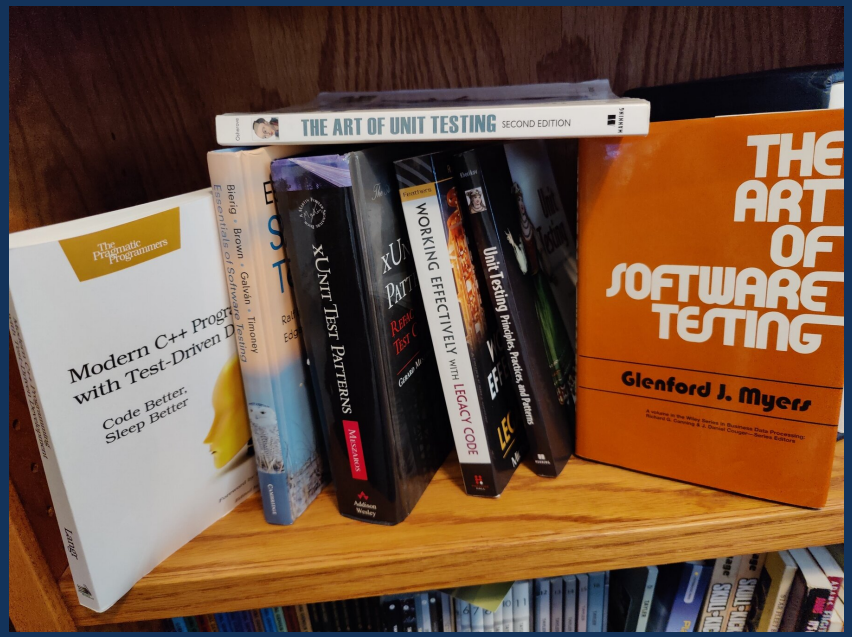












Another B2B on Unit Testing?

- 2019: Fedor Pikus (TDD) https://youtu.be/RoYljVOj2H8
- 2020: Ben Saks https://youtu.be/_OHE33s7EKw
- 2022: Amir Kirsh https://youtu.be/SAM4rWaIvUQ
- 2023: Phil Nash https://youtu.be/7_H4qzhWbnQ

Unit Testing is a Big Topic

So many talks...

- 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: *What we talk about when we talk about unit testing*

  Heisenbug 2020 https://youtu.be/-WWIeXmm4ec
- 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

Unit Testing is a Big Topic

So many talks...

Jon Jagger: *Testing as an equal 1st class citizen (to coding)*

https://www.youtube.com/embed/9lXFuf5IN7Q?enablejsapi=1

Another B2B on Unit Testing?

- 2020: Me, "The Science of Unit Tests" https://youtu.be/FjwayiHNI1w
- 2021: Me, "The Unit Tests Strike Back" https://youtu.be/N2YJ4D7O7Oc
- 2022: Me, "It's a Bug Hunt" https://youtu.be/P8qYIerTYA0
- 2015: Titus Winters & Hyrum Wright, "All Your Tests Are Terrible" https://youtu.be/u5senBJUkPc
- 2014: Matt Harget "Pragmatic Unit Testing in C++" https://youtu.be/Y8YVSohnlgY
- 2021: Igor Bogoslavskyi "Testing Compile-time Constructs in a Runtime Unit Testing Framework" https://youtu.be/hMn_dCae00g
- 2019: Manu Sánchez “Next generation unit testing using static reflection” https://youtu.be/8adO3fN1Igg

Test Driven Development

- 2020: Phil Nash, "Test Driven C++" https://youtu.be/N2gTxeIHMP0
- 2014: Peter Sommerlad "C++ Test-driven Development" https://youtu.be/YrGSQXZmAXs
- 2015: Phil Nash "Test Driven C++ with Catch https://youtu.be/gdzP3pAC6UI

Other Resources

- 2019: Michael T Starks, "Unit Testing: Prefer Children Over Friends (lightning talk) https://youtu.be/dDD_W4A93Pg

Frameworks

- 2020: Kris Jusiak ""unit"_test: Implementing a Macro-free Unit Testing Framework from Scratch in C++20" https://youtu.be/-qAXShy1xiE
- 2019: Wahid Tanner "650 line C++ unit test library" (Lightning Talk) https://youtu.be/nnlEQwQlHQg

Other Testing

- 2020: Barnabás Bágyi "Fuzzing Class Interfaces for Generating and Running Tests with libFuzzer" https://youtu.be/TtPXYPJ5_eE
- 2019: Clare Macrae "Quickly Testing Legacy C++ Code with Approval Tests" https://youtu.be/3GZHvcdq32s
- 2020 Clare Macrae "Quickly Testing Qt Desktop Applications with Approval

Other Conferences!

- Kevlin Henney
- ...

Prime Reacts "Thoughts about Unit Testing" https://youtu.be/KzV0mTqBcZA

(the "fiddling" bit is the only bit worth unit tesint

"Don't just describe what the test does either (we can read the code), tell us why it does this" -- Marit van Dijk

https://maritvandijk.com/blog/page/5/

"You are not writing your tests for a Ci/CD Pipeline, you are writing them for someone else" -- Gerard Meszaros