



+ 22

It's A Bug Hunt

Armor Plate Your Unit Tests

DAVE STEFFEN



20
22



Dave Steffen

Principal Software Engineer

dsteffen@scitec.com



www.scitec.com/join

Unit Testing is a Big Topic.

Aspects of Unit Testing

- **Why** to write unit tests
- **Process** for writing unit tests
- **Psychology** of writing unit tests
- Writing good unit test **code**
- ***Making unit tests good tests***

This Talk



Writing unit tests that are good tests is fundamental

Because "It's a Bug Hunt"

"It's A Bug Hunt"



"It's a bug hunt"

- Cpl. Dwayne Hicks (Michael Biehn)

"Aliens", 1986

Directed by James Cameron

20th Century Fox

A cautionary tale of a bug hunt gone bad

< There will be spoilers! >

Modern Science == Modern Testing

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

Take a page from experimental science

A good unit **test** is:

- Repeatable and Replicable
- Accurate
- Precise

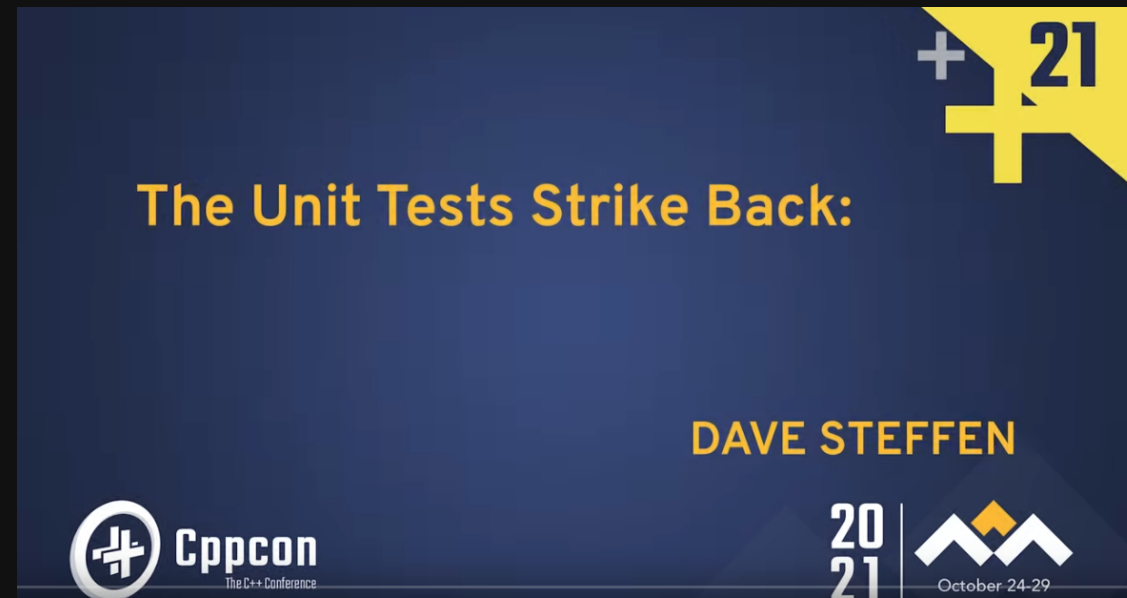
These are the armor plating for your unit tests.

Repeatable and Replicable

Repeatable: you get the same answer every time

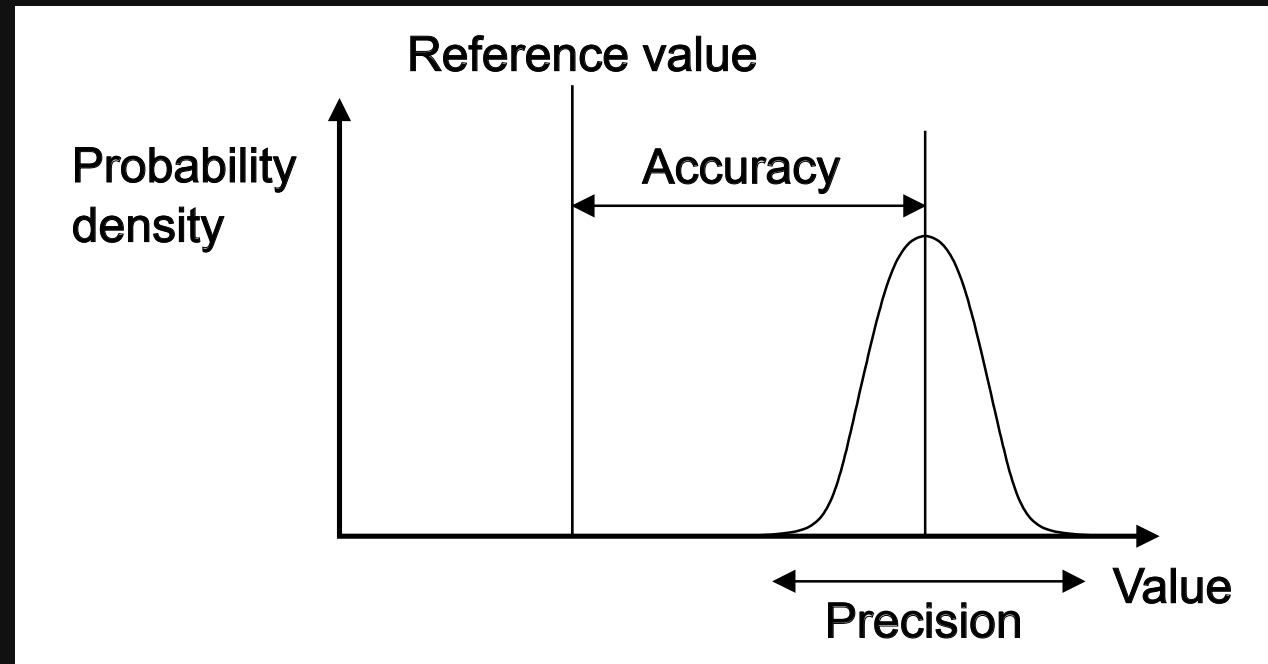
Replicable: your colleague gets the same answer you do

See my talk from last year:



Accuracy and Precision in Science : the Math

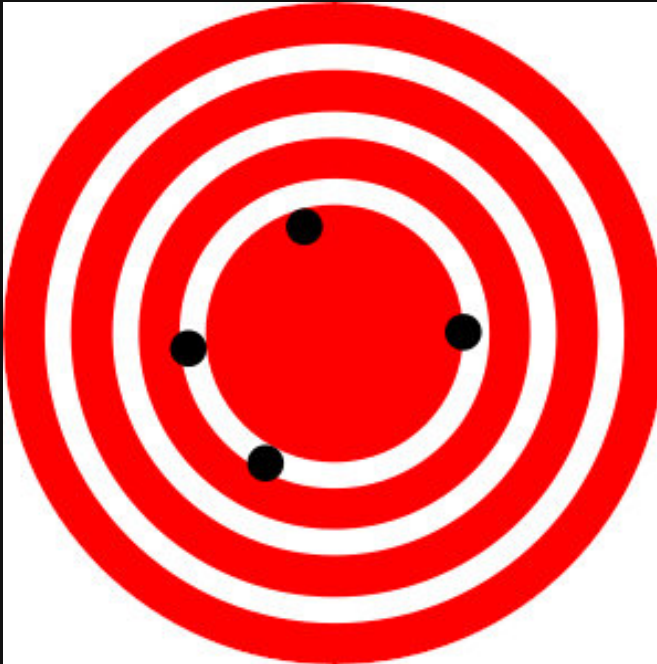
- **Accuracy**: measurements are close to a specific value
- **Precision**: measurements are close to each other
 - "Repeatability", "reproducibility"



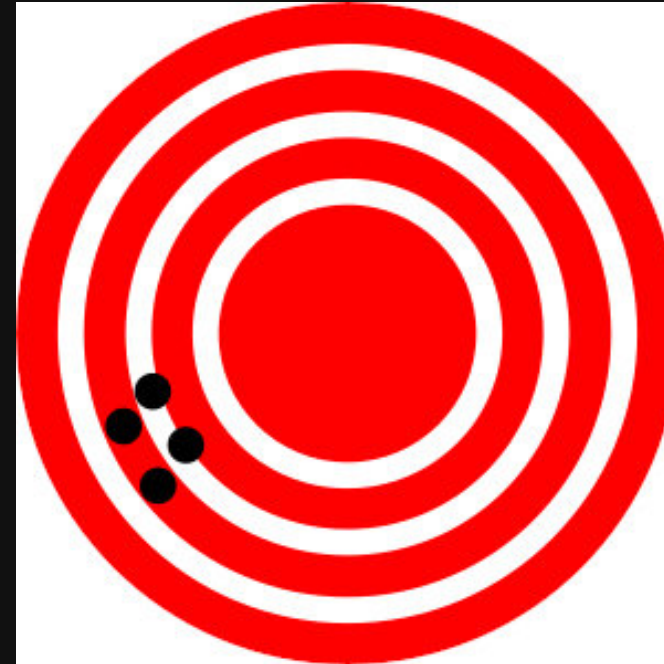
Pekaje at the English-language Wikipedia, GFDL
<<http://www.gnu.org/copyleft/fdl.html>>, via Wikimedia Commons

Accuracy and Precision in science: intuition

high accuracy
low precision



low accuracy
high precision



Accuracy == equipment is correct

Precision == equipment is reliable

Accuracy for Unit Tests: Math

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*

$$\text{"Rand Accuracy"} = \frac{P_T + N_T}{P_T + N_T + P_F + N_F}$$

Accuracy:

Fraction of test results that match reality of code

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

Precision for Unit Tests: Math

Precision, on the other hand, isn't mathematically defined for binary classifiers

For unit tests, we generalize precision to mean
"getting a maximal amount of useful
information about the problem"

Accuracy and Precision for Binary Classifiers: Intuition



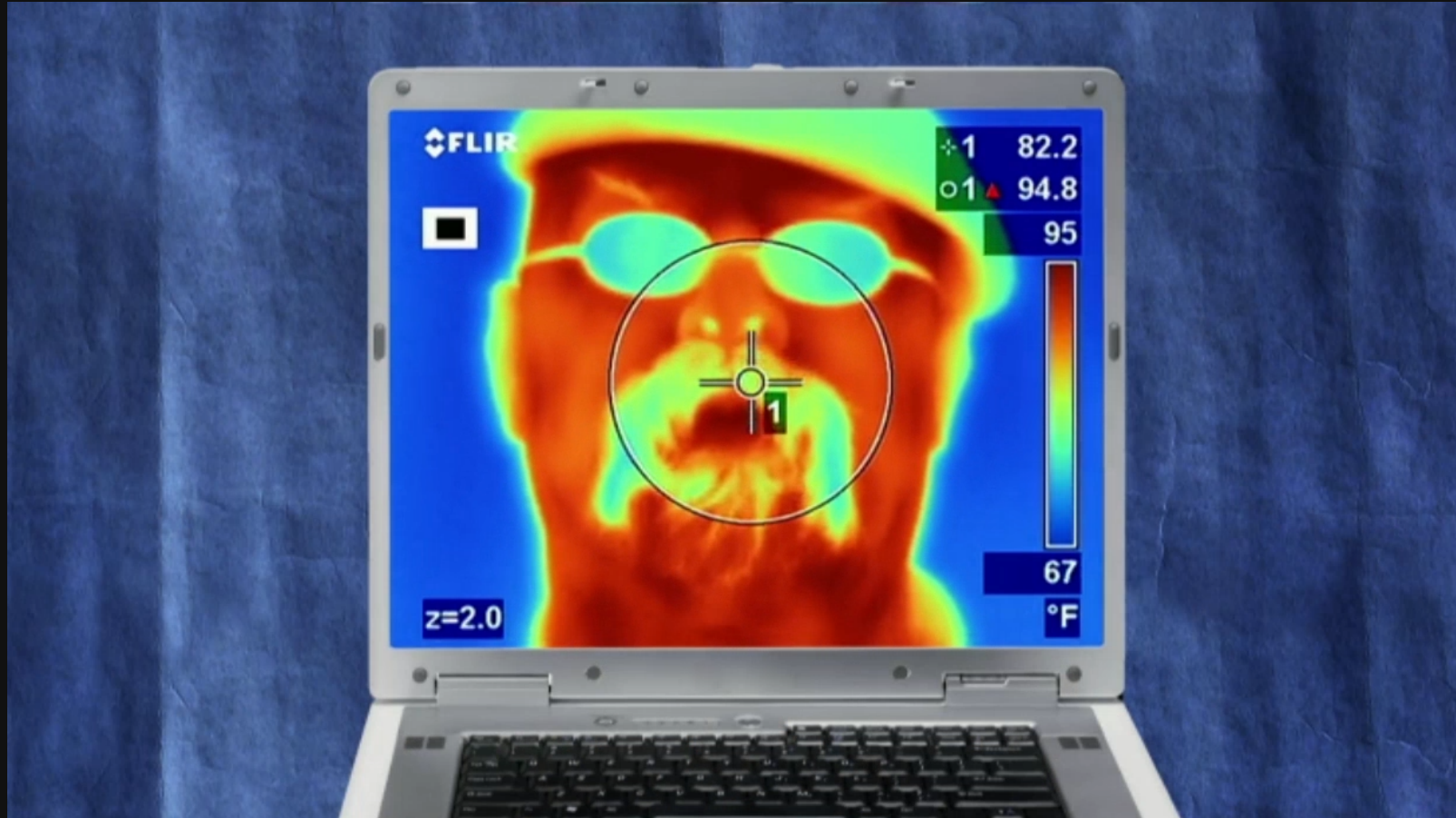
"There's something moving and it ain't us"

- Pvt. William Hudson
(Bill Paxton)

Motion Trackers: accurate but not precise

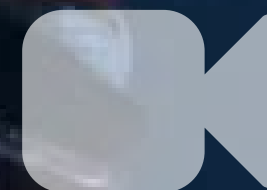
- **Accuracy High**: Never wrong about presence or absence of "something moving"
- **Precision Low**: problematic lack of useful information

Mythbusters Infrared



A real FLIR camera circa 2008

Colonial Marines FLIR



"Maybe they don't show up on infrared at all"

- Cpl. Dietrich
(Cynthia Dale Scott)

FLIR

Precise but not accurate

- **Precision high**: (presumably), would show exactly where targets are
- **Accuracy Low**: can't detect aliens

Accuracy and Precision in Unit Tests

Accuracy:

- when there's a bug, alarms sound
- when there's no bug, no alarms sound

Precision:

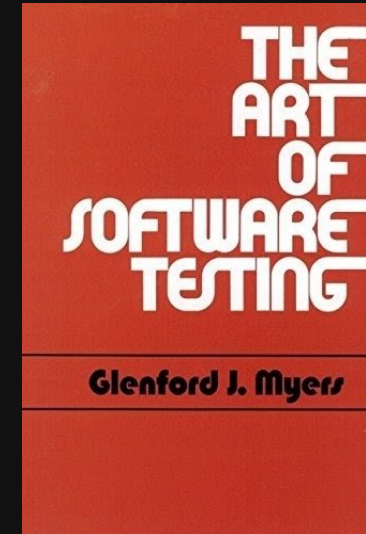
- When the alarm sounds you know where to look, what you're looking at, and what to be afraid of

Accuracy part 1: Completeness

Don't leave any place for bugs to hide

Goals for Completeness

- We have a finite number of tests
- We have a vast number of inputs
- How do we select our test cases?



Practically: we maximize accuracy by maximizing the chances of finding a bug per test case.

We can't test it all, but can we stack the deck in our favor?

Equivalence Partitioning

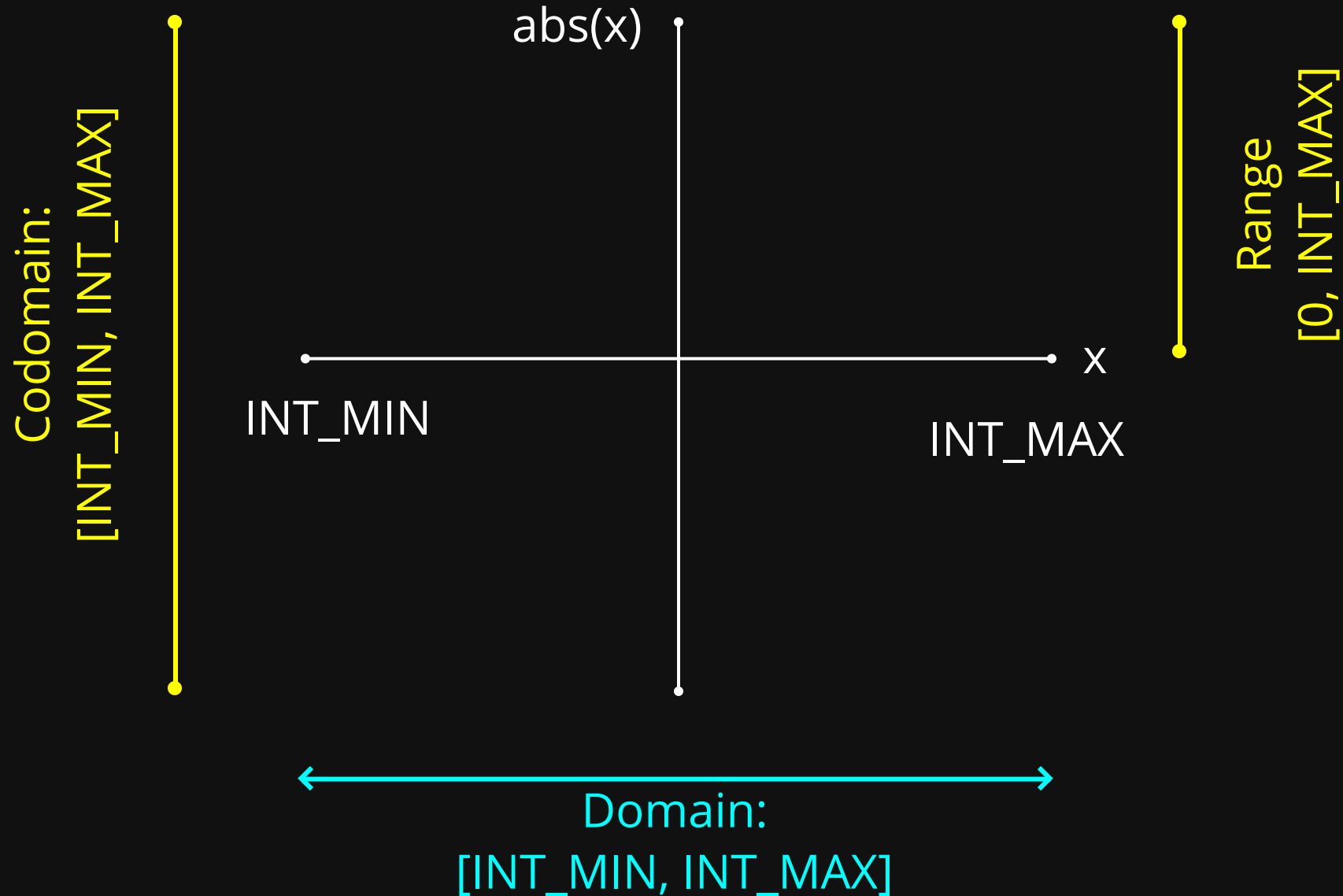
An Equivalence Class (EC) is a set of inputs that will all produce the same test result

In principle, testing one value from each EC should exercise all paths through the function.

Equivalence Partitioning is the act of identifying ECs.

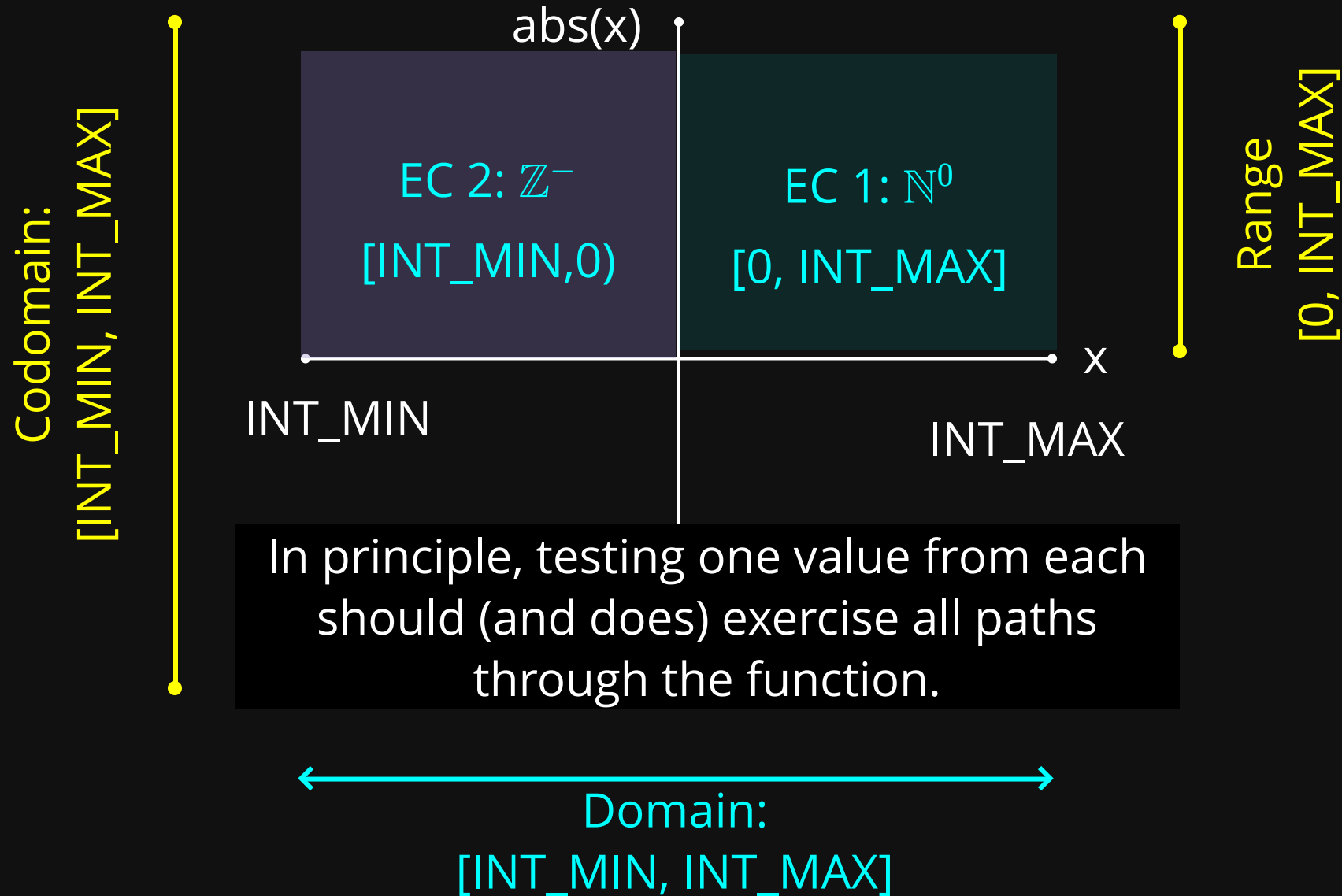
Step 1: Equivalence Partitioning

```
int abs(int x) { return (x < 0) ? -x : x; }
```



Step 1: Equivalence Partitioning

```
int abs(int x) { return (x < 0) ? -x : x; }
```



Step 2: Boundary Conditions

```
int abs(int x) { return (x < 0) ? -x : x; }
```

- For EC that are a range, check the boundaries
 - Places where behavior changes
 - Places where easy mistakes live (< vs <=)
 - Places next to these

Step 2: Boundary Conditions

```
int abs(int x) { return (x < 0) ? -x : x; }
```

EC1: [0, INT_MAX]

EC2: [INT_MIN, 0)

- Check INT_MAX: nothing new
- Check 0: nothing new
- Maybe, ± 1 : nothing new
- INT_MIN: whoops....

std::numeric_limits<int>::min = -2147483648

std::numeric_limits<int>::max = +2147483647

Signed integer range is not symmetric

abs(INT_MIN) is undefined behavior

Error Cases are separate ECs

```
int abs(int x) { return (x < 0) ? -x : x; }
```

Equivalence classes:

1. Non-negative integer: $[0, \text{INT_MAX}]$
2. Negative integer: $(\text{INT_MIN}, 0)$
3. Invalid integer: INT_MIN

As written, `abs(x)` has a narrow contract. No point in testing EC 3 because it's Undefined Behavior

Don't test out-of-contract input

Wide and Narrow Contracts

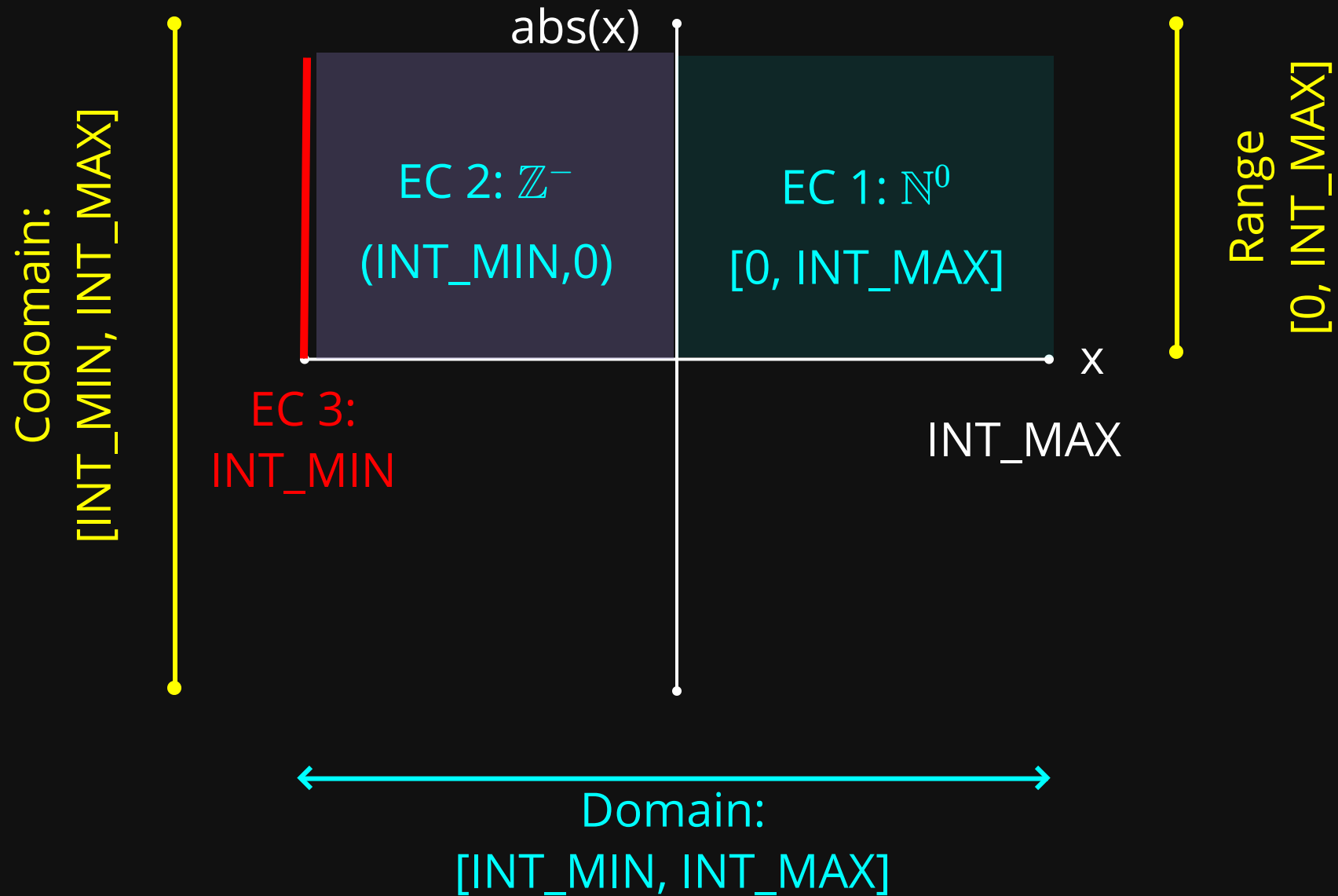
What if we rewrite as a wide contract?

```
int abs(int x) {  
    if (x == std::numeric_limits<int>::min())  
        throw std::domain_error("Can't take abs of INT_MIN")  
    return (x < 0) ? -x : x;  
}
```

This wide contract makes INT_MIN a runtime error.

Now, EC 3 is a valid equivalence class we should test.

Equivalence Partitions



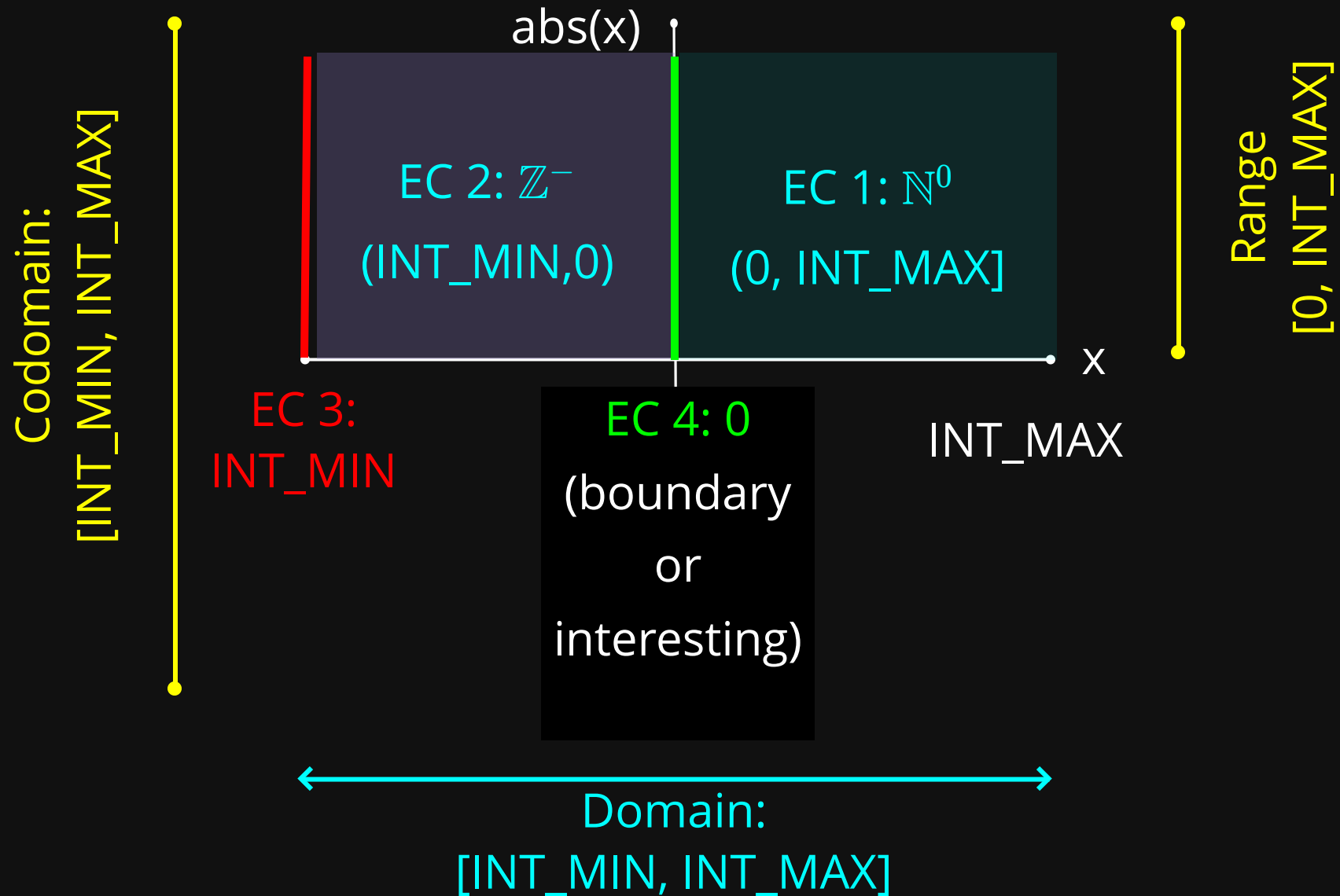
Step 3: Interesting Conditions

"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.

Equivalence Partitions



Is Equivalence Partitioning White Box?

Important point:

- "Black Box" testing tests only observable behavior
- "White Box" testing relies on implementation details

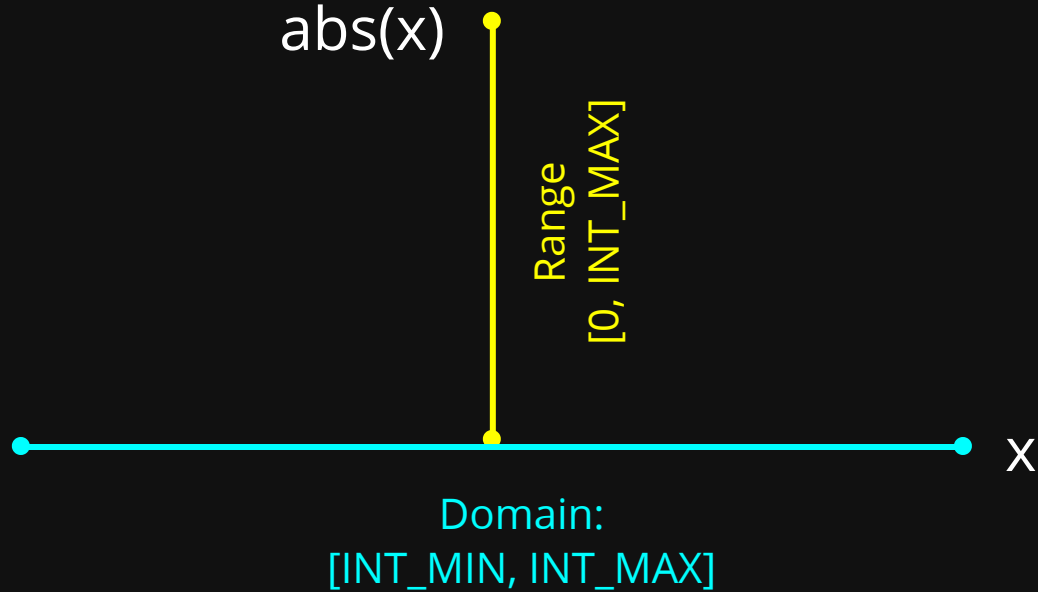
Should we derive ECs from knowledge of the implementation details?

If Yes: EC definitions (and therefore unit tests) may be coupled to implementation details.

If No: EC definitions decoupled from implementation *but may miss important corner cases*

"Black Box" ECs from output values

Check the *range* rather than the domain



Three obvious range cases:

- 0
- INT_MAX
- Everything in between

What input values produce these results?

- EC1 : 0
- EC 2: (0, INT_MAX)
- EC 3: (INT_MIN, 0)
- INT_MAX is in EC2
- INT_MIN is a problem

Example: operations on sequences

```
template <Iterator I> auto sort(I begin, I end) -> void;
```

Equivalence classes for range size:

- 0 / empty range : *always*
- 1 element : *always*
- Many elements: *always*
 - two elements? maybe

Equivalence classes for range contents:

- {2,2,2} // all same
- {1,2,3} // already sorted
- {3,2,1} // reverse order

Multiple Input Parameters

```
auto FeedCamel(Camel& c, Feed& f, Motorcycle& m) {...}
```

Let C = set of ECs for Camels

Let F = set of ECs for Feed

Let M = set of ECs for Motorcycles

(in principle) Test Suite = $C \otimes F \otimes M$

Do I have to do all of them?

- Success cases: one test including each EC is good enough (unless they interfere)
- Error cases: only one error EC per test

Equivalence Partitioning Roundup

EP as a technique draws a lot of ire online

- Lots of debate about language and definitions
- I *think* a lot of disappointment that it didn't live up to expectations.

My take:

EP is an excellent tool for identifying *possible* test cases.

It doesn't tell you what to test

It doesn't provide answers

It does help you ask the right questions

Accuracy part 2: Correctness

Correctness: does the test correctly identify:

- correct output as correct
- incorrect output as incorrect
- Correct Error handling

Correctness is part of Accuracy
also part of maintainability

Correctness 1: Brittle Tests

```
TEST(MyTest, Log)
{
    StartLogCapture();
    ...
    EXPECT_THAT(Logs() , HasSubstr("file.cc:421: Opened file"));
}
```

Maintenance problem: test "breaks" on a reformat

```
TEST(MyTest, Unordered)
{
    std::unordered_set<int> s {1,2,3,4,5};
    ...
    EXPECT_STREQ(to_string(s) , "1,2,3,4,5");
}
```

Maintenance problem: test "breaks" on a hash or
internal container change

Correctness 2: Too Much (computational) precision

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

3.141592741012573

```
TEST(Math, compute_pi) {  
    auto answer = 3.14159274;  
    EXPECT_NEAR( compute_pi() , answer , 1e-8);  
}
```

```
double compute_pi() { return std::acos(-1); } // 2
```

3.141592653589793

```
../test/Track/test_track.cpp:23: Failure  
The difference between compute_pi() and answer  
is 8.6410206989739891e-08, which exceeds 1e-8
```

Overspecifying "correct" produces *false positives* on *correct* code changes

Correctness 2: Too Little Precision

```
TEST(Math, compute_pi) {  
    auto answer = 3.14159;  
    EXPECT_NEAR( compute_pi() , answer , 2e-3);  
}
```

```
double compute_pi() { return 22.0 / 7; } //
```

3.142857142857143

Underspecifying "correct" lets the bugs through

"Correct" should contain only the information actually produced, or the quality actually needed.

Accuracy part 3: Validity

Beware of circular logic

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

TEST(Math, compute_pi) {
    float correct_value = std::acos(-1);
    EXPECT_TRUE( compute_pi() == correct_value );
}
```

You have just proven the algorithm you wrote is the algorithm you wrote

(this happens in science *all the time*)

Accuracy of this test isn't defined; it will always pass, so is not falsifiable

Accuracy pt 3: Validity

Contrast with two other similar situations

```
TEST(Math, compute_pi) {  
    EXPECT_NEAR( compute_pi(), 3.15149, 5e-6 );  
}
```

Value and error bound obtained from prototype or expert


Reference to known (or at least suspected) correct value obtained from reliable source

This is excellent, and frequently one of the first tests written

Beware floating point accuracy!

Accuracy pt 3: Validity

```
TEST(Math, compute_pi) {  
    EXPECT_EQ( compute_pi() , 3.141592741012573 )  
}
```



Value obtained by executing `compute_pi()` yesterday

- We haven't shown the code is *correct*
- We have shown it *hasn't changed behavior*

This is an **acceptance test**, not a unit test

Clare Macrae,
<https://claremacrae.co.uk/>

Accuracy

The result of your test matches reality of code

Test completely (Equivalence Partitioning)

Test Correctly (use no more information than you have, use no less than you need)

Test Validity: no circular logic; write falsifiable code and falsifiable tests

Precision

Precision:

Unit test results provide a maximum
amount of useful information:

How fast can we move from
"we know there's a problem"
to
"we know what and where the problem is"

Lack of Precision Is A Problem



"That can't be, that's
inside the room"

- Ellen Ripley (Sigourney Weaver)

High accuracy: Motion Trackers detect bugs

High range precision: they really are 4 meters away

Low angular precision: ... but in what direction?

Precision pt 1: Clarity

First rule of precision:

Use a good unit test framework.

Precision pt 1: Clarity

```
1 TEST(Alien_Hive, team_safety)
2 {
3     bool team_safety = false;
4     ASSERT_TRUE(team_safety);
5 }
```

- A passing test produces little or no output
- A failing test indicates
 - Test Suite
 - Test Case
 - Assertion that failed
 - Expected, and actual values
 - File and line number of failure

```
Running main() from ../googletest/src/gtest_main.cc
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from Alien_Hive
[ RUN      ] Alien_Hive.team_safety
../test/Track/test_track.cpp:12: Failure
Value of: team_safety
```

Actual: false

Expected: true

```
[  FAILED  ] Alien_Hive.team_safety (0 ms)
[-----] 1 test from Alien_Hive (0 ms total)
```

```
[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Alien_Hive.team_safety
```

Demand this of your unit test framework; accept nothing less

Precision part 2: Organization

```
1 TEST(Aliens_Threat_Model, Alien_Hive)
2 {
3     ...
4     std::vector<Alien> threat;
5     EXPECT_TRUE( threat.empty() );    // ???
6     ...
7 }
```

Consider that line.

- What are the chances it fails?
- What should you do if it does?
- What are the chances it fails but everything else still works?

Precision part 2: Organization

```
1 TEST(Alien_Threat_Model, Alien_Hive)
2 {
3     ...
4     our_custom_vector<Alien> threat;
5     EXPECT_TRUE( threat.empty() );    // ???
6     ...
7 }
```

Is this more reasonable?

- It may be reasonable for you to test other people's code *but not in your test suite*
- An isolated failure in `our_custom_vector` shows up in the Alien Hive test case.

Avoid red herrings

Precision part 2: Organization

If our_custom_vector fails, what do our tests report?

```
// extern_test.cpp

TEST(Other_Teams_Stuff,
     our_custom_vector_sanity)
{
    our_custom_vector<int> vec;
    EXPECT_TRUE( vec.empty() );
}
```

If only this fails,
everyone knows
where the bug is

```
TEST(Alien_Threat_Model, Alien_Hive)
{
    ...
    our_custom_vector<Alien> threat;
    ...
}
```

If both tests fail (more likely), at
least some of the output points at
the bug

Organize test cases and test suites to point clearly at (or at least strongly suggest) the problem

Mocks:

A(nother) Tale of Two Cities

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 Development*

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

Classicist

```
1 TEST(Aliens, Alien_Hive)
2 {
3     ...
4     our_custom_vector<Alien> threat;
5     EXPECT_TRUE( threat.empty() );    // not needed: tested elsewhere
6     ...
7 }
```

Mockist

```
1 TEST(Aliens, Alien_Hive)
2 {
3     ...
4     MOCK_custom_vector<Alien> threat;
5     EXPECT_TRUE( threat.empty() );    // not needed: it's your mock
6     ...                               // still tested elsewhere, right?
7 }
```

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.

Don't let them in through the ceiling!

Plus, have you tested your Mocks?

Beware circular reasoning!

Precision summary



Humans and Confusion

Humans are not good at handling
contradictory sensory input.

This problem is much worse when we are
tired, angry, or scared.

Don't confuse the tired, angry, frightened humans

Summary

Good tests are accurate and precise

Accuracy: test results match reality

- Bad Accuracy lets the bugs in: "Maybe they don't show up on infrared at all"

Precision: test results are useful

- Bad Precision is confusing: "That can't be, that's inside the room"

Armor Up



There are some bugs out there

Credits



SciTec is hiring

www.scitec.com/join

Editorial Search and Rescue:

Neil Sexton

Video Editing Assistance:

Nathan Paget

Steve Soule

Patience with Early Talk Prototypes:

Denver C++ Meetup

<https://www.meetup.com/north-denver-metro-c-meetup/>