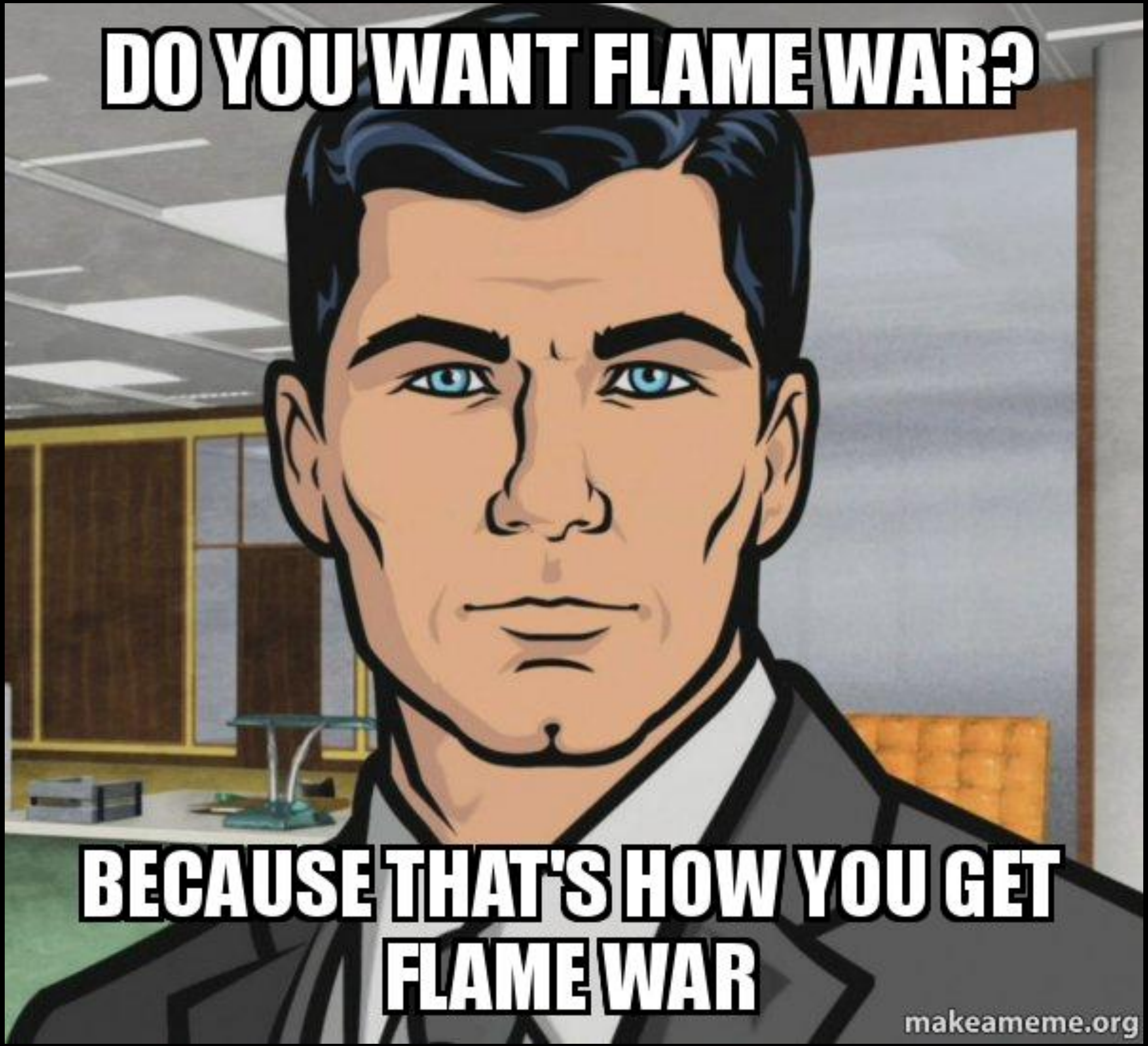
PROPERTY-BASED TESTING

THE GOAT ON UNIT TESTING

“the idea of "unit tests" appeals to me only rarely, when I’m feeling my way in a totally unknown environment and need feedback about what works and what doesn’t. Otherwise, lots of time is wasted on activities that I simply never need to perform or even think about.”

-- Donald Knuth


DO YOU WANT FLAME WAR?
BECAUSE THAT'S HOW YOU GET FLAME WAR
makeameme.org

ALL GOOD TALKS SHOULD START BY INCITING A FLAME WAR ...

so as currently practiced I think DEK is right.

The Horrors of Unit Testing

- Tests that run forever
- Tests that fail randomly
- Tests that verify wrong behavior
- Tests which are not obviously correct
- Tests that read GB source-controlled binary files
- Tests that give no context when they fail

We need an opinionated philosophy about what we're trying to accomplish with unit testing.

Unit testing is our <u>current best mechanism</u> of transmitting information about the probability of correct operation of the code between team members and through time.

Aumann disagreement theory

“Bayesian agents with common priors can never “agree to disagree”: if their opinions about any topic are common knowledge, then those opinions must be equal... for two agents with a common prior to agree to within ε about the expectation of a [0,1] variable with high probability over their prior, it suffices for them to exchange $O(1/\varepsilon^2)$ bits.”

-- Scott Aaronson

KEY QUESTION

How do we communicate the <u>most</u> information about correctness of our code with the least amount of extraneous code and computation?

IS THIS A GOOD UNIT TEST?

```
EXPECT_DOUBLE_EQ(std::sin(1.2), 0.9320390859672263);
```

IS THIS A GOOD UNIT TEST?

```
EXPECT_DOUBLE_EQ(std::sin(1.2), 0.9320390859672263);
```

If this test fails, is the test wrong, or is the code wrong? We have no clue.

IS THIS A GOOD UNIT TEST?

```
EXPECT_DOUBLE_EQ(std::sin(1.2), 0.9320390859672263);
```

If this test fails, is the test wrong, or is the code wrong? We have no clue.

<u>This test transmits zero bits of information about correctness of std::sin.</u>

DEFINITION

A <u>property</u> is something we expect to hold for all valid inputs to a program.

A PROPERTY-BASED TEST FOR `std::sin`

```
double x = dis(rd);
EXPECT_DOUBLE_EQ(std::sin(-x), -std::sin(x))
  << " sine is an odd function, but std::sin is not at x = "
  << x;
EXPECT_LE(std::abs(std::sin(x)), 1)
  << " |sin(x)|≤1, but std::sin(x) has magnitude above unity at x = "
  << x;
EXPECT_DOUBLE_EQ(1, std::sin(x)*std::sin(x) + std::sin(x+M_PI)*std::sin(x+M_PI))
  << "sin(x)² + cos(x)² = 1, but std::sin does not satisfy this identity at x = " << x;
```

Is this a good test?

IS THIS A GOOD TEST?

IS THIS A GOOD TEST?

» The test itself is obviously correct (ht: Richard Powell)

IS THIS A GOOD TEST?

» The test itself is obviously correct (ht: Richard Powell)

» The test prints the requisite metadata to fix the problem should it fail.

IS THIS A GOOD TEST?

- The test itself is obviously correct (ht: Richard Powell)
- The test prints the requisite metadata to fix the problem should it fail.
- The test executes in under a microsecond. (std::sin calls require ~5ns.)

But we had to know something about `std::sin`!

BUT WE HAD TO KNOW SOMETHING ABOUT `std::sin`!

Exactly! Property-based testing forces us to develop a theory of our own code!

“. . . the developer is forced to think about what the code should do at a high level rather than grunt out a few unmotivated test cases.”

-- Joe "begriffs" Nelson

EXERCISE

`Hack up a property-based test for std::reverse.`

```cpp
#include <algorithm>
#include <vector>
#include <random>
#include <gtest/gtest.h>

TEST(std_reverse, square_to_identity) {
     std::random_device rd;
     auto seed = rd();
     std::mt19937_64 mt(seed);
     std::vector<long> v(mt() % 500 + 1);
     for (auto & x : v) {
          x = mt();
     }
     auto u = v;
     std::reverse(v.begin(), v.end());
     std::reverse(v.begin(), v.end());
     EXPECT_EQ(u.size(), v.size())
         << " std::reverse should not change the vector size, but does on seed " << seed << "\n";
     for (size_t i = 0; i < u.size(); ++i) {
         EXPECT_EQ(u[i], v[i])
       << "Two applications of std::reverse should return the original vector, but it differs at index "
       << i << " on a vector of length " << v.size() << " generated with seed " << seed;
     }
}
```

PRO-TIP FOR PBT ADVOCATES USING GOOGLETEST

```
$ ./reverse_test --gtest_repeat=-1 --gtest_break_on_failure
```

Wait . . . I hate tests that fail randomly . . .

JUST BECAUSE YOU'RE A HATER DOESN'T MEAN IT'S INEFFECTIVE!

Failing randomly is not a problem!

In fact, randomized testing transmits novel information every run.

The problem is lack of metadata required for reproduction.

Symmetric Properties

In the Erlang world, our std::reverse example is called a symmetric property; see Hebert for details.

I would probably call it a forward/inverse operation.

Boilerplate for forward/inverse operations

```
auto seed = rd();
auto input = generate_random_data(seed);
auto output = forward(input);
auto computed_input = inverse(output);
EXPECT_EQ(input, computed_input)
   << "Failure with seed " << seed;
```

EXERCISE

What are some common forward/inverse operations required in programming tasks?

EXAMPLES

- serialization/deserialization
- compress/decompress
- encrypt/decrypt
- std::random_shuffle/std::sort
- push/pop operations on lists or queues
- std::rotate (left by n, right by n)
- std::swap

DECOMPOSITION/SYNTHESIS

Whenever we break an object into simpler subobjects, we have a natural property-based test.

DECOMPOSITION/SYNTHESIS EXAMPLES

- Factor an integer n into pair (p, q); assert $n == pq$.
- Factor a matrix A into a pair (Q, R), assert $\|A - QR\|$ is small.
- `std::set_union(std::set_difference(A, B), std::set_intersection(A, B)) == A` for any A, B.

IDEMPOTENT OPERATIONS

In mathematics, we say an operator P is idempotent iff $P^2 = P$.

Example:

```
auto input = generate_random_data(seed);
auto sorted = std::sort(input.begin(), input.end());
auto double_sorted = std::sort(sorted.begin(), sorted.end());
EXPECT_EQ(double_sorted, sorted)
  << " A sorted list shouldn't change after under sorting, seed: " << seed;
```

Idempotent operations are highly amenable to property-based testing!

EXERCISE:

What idempotent operations are common in programming?

» `std::clamp`

» `std::abs`

» `rm -f foo.cpp`

» `touch foo.cpp`[1]

» `std::unique`

[1] Thanks to Richard Powell for pointing out that touch is only idempotent wrt file existence. It's not idempotent wrt the timestamp.

VOCABULARY OF PBT: MODELING

Testing code against a slower, obviously correct implementation is called "modeling".

EXAMPLES OF MODELING

» Testing parallel implementation against serial implementation.

» Testing a sparse linear solver against a dense solver

» Testing your hand-rolled ASM/ASIC against a readable Python implementation.

EXERCISE

What is a model-based test for std::nth_element?

```
auto v = generate_random(seed);
auto m = v.begin() + v.size()/2;
std::nth_element(v.begin(), m, v.end());
auto median_nth_element = v[v.size()/2];
std::sort(v.begin(), v.end());
EXPECT_EQ(v[v.size()/2], median_nth_element)
  << " Median of fully sorted list is not equal to median from std::nth_element "
  << " when using seed " << seed;
```

INVARIANTS

An invariant is something that should always be true thoughout a loop operation, in a class, or after a computation.

EXERCISE:

Find an invariant of `std::upper_bound`.

Invariant is documented, so this is easy:

"Returns an iterator pointing to the first element in the range [first, last) such that value < element .. . is true (i.e. strictly greater), or last if no such element is found."

```
#include <random>
#include <algorithm>

auto v = random_vector(length);
std::sort(v.begin(), v.end());
std::uniform_real_distribution<decltype(v[0])> unif(v.front(), v.back());
std::random_device rd;

auto seed = rd();
std::mt19937_64 mt(seed);
auto val = unif(mt);
auto it = std::upper_bound(v.begin(), v.end(), val);
if (it == v.end()) {
  EXPECT_EQ(val, v.back());
} else {
  EXPECT_LT(val, *it)
    << " Invariant violation with seed " << seed << "\n";
}
```

IF WE CAN

» Recognize idempotence

» Identify a forward/inverse operation

» Synthesize a decomposition

» Find a model

» Find an invariant

. . . we can do property-based testing.

VOCABULARY OF PBT: GENERATORS

Property-based testing requires random input data, but random data must be sampled from some distribution.

That our algorithms can interact with different distributions in very different ways has been recognized since von Neumann in 1947!

VOCABULARY OF PBT: GENERATORS

The nomenclature for generating useful randomized data in the property-based testing world is generators.

VOCABULARY OF PBT: SHRINKING

Once a failing test case is found, it is often a massive list of random data. It is often difficult to see what precisely went wrong.

Constructing simple failing cases from a complicated one found by randomization is called shrinking.

Shrinking Techniques

- Call std::round on all floating point values
- Split lists in half, or take every other element
- Shrink all integers into range [-9, 9].

PROPERTY-BASED TESTING IN C++

Shrinking is where we start to need library support for PBT.

Let's examine rapidcheck which is explicitly designed for property-based testing.

TEST THIS CODE:

```cpp
double dot_product(std::vector<double> const & x, std::vector<double> const & y) {
    double s; // bug! s is uninitialized!
    for(size_t i = 0; i < x.size(); ++i) {
        s += x[i]*y[i];
    }
    return s;
}
```

RAPIDCHECK CODE:

```
#include <vector>
#include <rapidcheck.h>

int main() {
  rc::check("dot_product is symmetric in its arguments.",
            [](const std::vector<double> &x, const std::vector<double>& y) {
              RC_ASSERT(dot_product(x, y) == dot_product(y,x));
            });

  return 0;
}
```

IMMEDIATE DETECTION WITH A MINIMAL TESTCASE:

```
Using configuration: seed=17304471087690044341

- dot_product is symmetric in its arguments.
Falsifiable after 1 tests

std::tuple<std::vector<double>, std::vector<double>>:
([], [])

~/rapidcheck_example/dot_product_test.cpp:17:
RC_ASSERT(dot_product(x, y) == dot_product(y,x))

Expands to:
2.15871e-314 == 3.01264e-314
Some of your RapidCheck properties had failures. To reproduce these, run with:
RC_PARAMS="reproduce=BoCZvR3XwJ3bkV3Y0BSazByc51WblRncpNGIp5GIpR3cgEmcnVXbl5Gdz5StTh8bFTdJwX7UI_WxUXC81OFyvVM1lAftTh8bFTdJwHAABAAAAAAA"
```

OK, GREAT. BUT DOES IT DETECT REAL BUGS?

I set myself the quixotic quest of writing a single, simple C++ class without bugs.

Bug #1 detected by property-based testing:

```
template <class RandomAccessContainer>
class polynomial {
public:
  polynomial(RandomAccessContainer &&coefficients)
      : c_{std::move(coefficients)} {
    while (c_.back() == 0 && c_.size() > 1) {
      c_.pop_back();
    }

    if (c_.size() == 0) {
      std::string msg = "At least one coefficient must be passed to the polynomial.";
      std::source_location location = std::source_location::current();
      std::string err = "Error at line " + std::to_string(location.line()) +
                        " of file " + location.file_name() + ": ";
      err += msg;
      throw std::domain_error(err);
    }
  }
};
```

What data causes this to segfault?

ANSWER

The call to c_.back() on an empty vector segfaults!

More problems with my polynomial class

```
template<typename Real=CoefficientType>
auto roots() const {
  if (c_.size() == 1) {
    return std::vector<std::complex<Real>>();
  }
  std::size_t n = c_.size() - 1;
  Eigen::Matrix<Real, Eigen::Dynamic, Eigen::Dynamic> C(n, n);
  for (std::size_t i = 0; i < n; ++i) {
    // c_.back() != 0 is a class invariant:
    C(i, n - 1) = -c_[i] / c_.back();
   }
   for (std::size_t i = 0; i < n - 1; ++i) {
     C(i + 1, i) = 1;
   }
   Eigen::EigenSolver<decltype(C)> es;
   es.compute(C, /*computeEigenvectors=*/ false);
   return es.eigenvalues();
}
```

Using the property $|p(r)| \ll 1$ for numerical roots, this code was immediately identified as buggy by PBT, but it succeeded most of the time!

Turns out, Eigen does not initialize matrices! The fix is this:

```
Eigen::Matrix<Real, Eigen::Dynamic, Eigen::Dynamic> C(n, n);
C << Eigen::Matrix<Real, Eigen::Dynamic, Eigen::Dynamic>::Zero(n,n);
```

YOUR PROGRAMS DON'T HAVE PROPERTIES?

Read Fred Hebert's book!

I didn't see a way to property test any of his examples before reading; after reading it was much more natural.

THERE ARE NO PANACEAS, PROGRAMMING IS TOO BIG A UNIVERSE

I still cannot figure out a way to property-test rendered images!

What are other domains where property-based testing is unhelpful?

ANOTHER ISSUE:

Property-based testing libraries tend to discover unsolved and even unsolvable problems.

Example: If $p(x) = x^{5000}$, then $p(10^{300})$ overflows. Not super helpful!

FUNDAMENTAL PROBLEM WITH PBT

We want our PBT library to probe inputs outside how we imagine the code will be used. But that often intersects with the set of input data we know cannot be used as it's unsolved in the literature.

PBT often just tells us that human beings aren't very smart.

I'M TOO DUMB AND LAZY TO USE WEAK IDEAS

Traditional "grunt-out unmotivated test cases" unit testing is a weak proxy for "my code probably works".

Fill your finite mental space with powerful abstractions!

ACKNOWLEDGEMENTS

Huge thanks to Chris Mc for helping me learn to use Conan, which was absolutely essential to incorporating boost, googletest, rapidcheck, and Eigen into a single usable codebase!

Also thanks to Pim Schellart and Richard Powell for reading a draft of these slides and providing invaluable feedback!

References

» Aaronson, Scott. "The complexity of agreement." Proceedings of the thirty-seventh annual ACM symposium on Theory of computing. 2005.

» Hebert, Fred. Property-Based Testing with PropEr, Erlang, and Elixir: Find Bugs Before Your Users Do. Pragmatic Bookshelf, 2019.

» von Neumann, J.; Goldstine, H.H. (1947). "Numerical inverting of matrices of high order". Bull. Amer. Math. Soc. 53 (11): 1021–1099.