Advanced Developers Conference C++
Development for Professionals!

C++ Test-Driven Development

with Cevelop and CUTE

Prof. Peter Sommerlad

ppedv

C++ TDD with Cevelop and CUTE

Prof. Peter Sommerlad
Director IFS Institute for Software
FHO HSR Rapperswil, Switzerland
ADCC++ May 2019

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

- **Cross-platform C++ IDE built by IFS**
  - any compiler, any (major) OS
- **C++ Refactoring support (started in 2006)**
  - built by IFS for Eclipse CDT (integrated from 2008 on)
- **Unit-Testing and TDD integrated (since 2007)**
- **Available for free as Cevelop (since 2015)**
  - might become fully open source 2020
- **Focus on code modernization and quality**
  - Checks and Quick-Fixes/Refactorings
  - Guidelines Checks (core, safety), C++11/14/17, pointer elimination

Cevelop

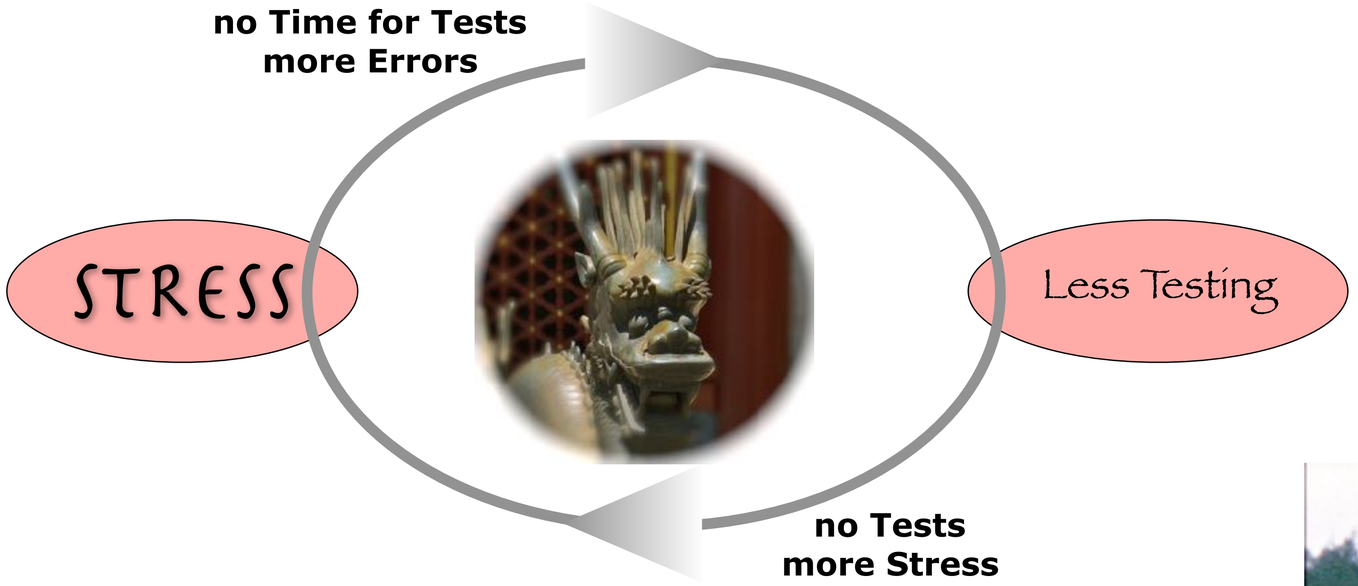
Your C++ deserves it

Download IDE at:
www.cevelop.com

- **More than 25% of C++ developers do not use a unit-testing framework***
- **Almost 90% of Embedded developers do not use a unit-testing framework***
- **Who does Unit Testing or other Test Automation?**

* Source: Anastasia Kazakova, JetBrains, ACCU 2019 talk: C++ ecosystem: For better, for worse

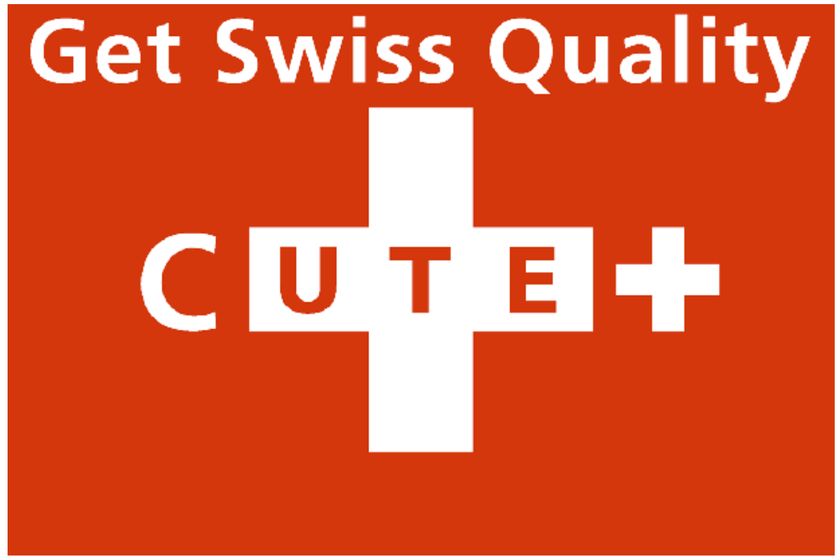
Without Test Automation: Vicious Circle



C++ Unit Testing with CUTE in Eclipse CDT

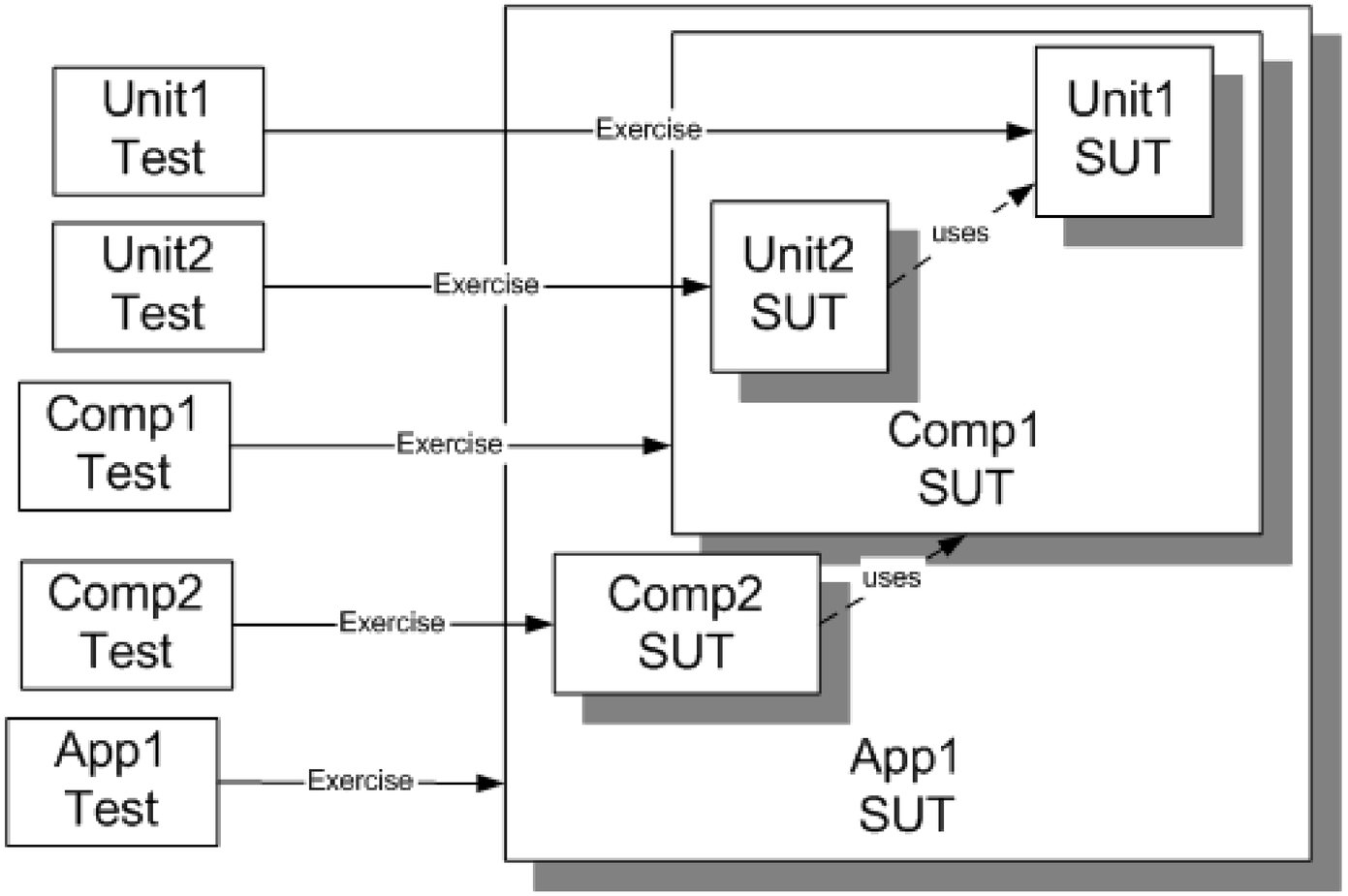
TDD Test-Driven Development and Refactoring

- **CUTE http://cute-test.com - free!!! Integrated into Cevelop!**
- **simple to use - a test is a function consisting of plain C++**
  - understandable also for C programmers, no special syntax
- **designed to be used with IDE support**
  - can be used without, but a slightly higher effort
- **deliberate minimization of #define macro usage**
  - macros make life harder for C/C++ IDEs and users



Some Terminology (source: xunitpatterns.com)

- **SUT System Under Test**

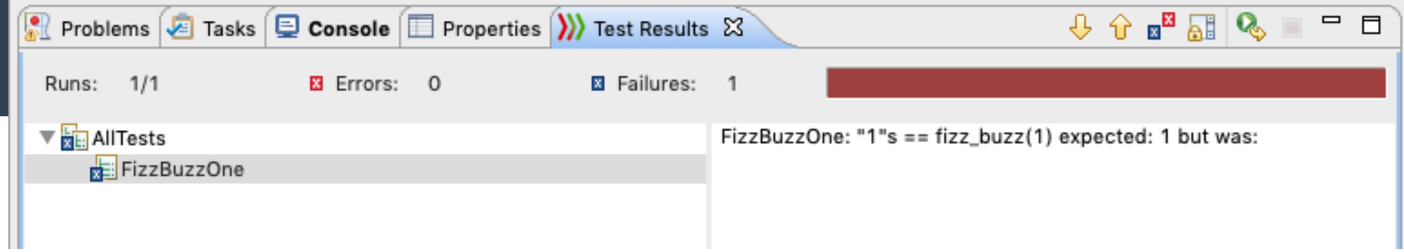


- **source: xunitpatterns.com - Gerard Meszaros**

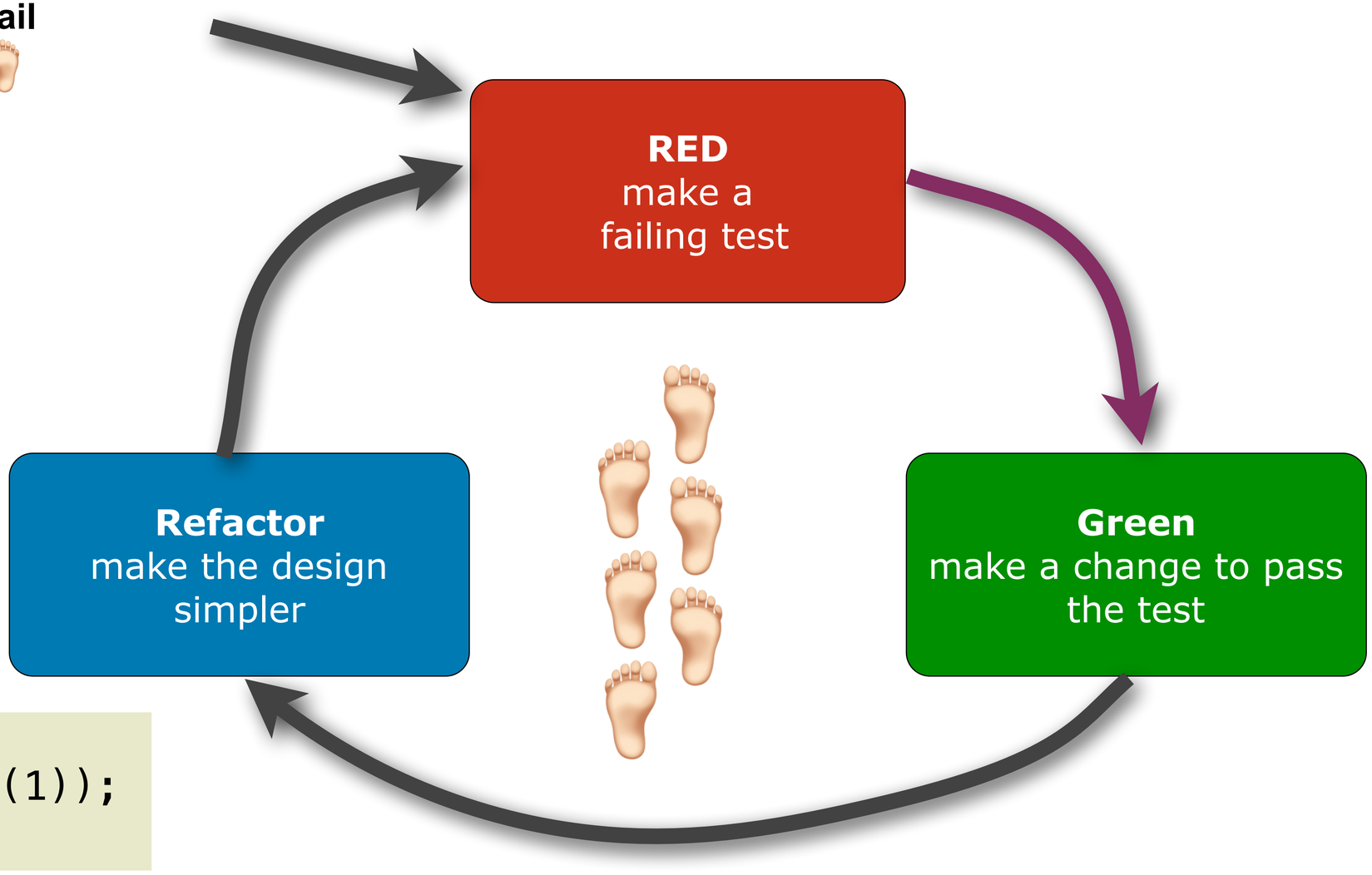
Fokus on the What not the How (as we use typically do as engineers)

What How

TDD Cycle



- Only write as much test code to fail
  - compile error is first failure
- In the beginning:
  - "Fake it 'til you make it"
- But
  - "Obvious Implementation"
  - just do it!
- Start
  - "ASSERT first"
  - ZOM: "Zero, One, Many"




```
void FizzBuzzOne() {
  ASSERT_EQUAL("1"s, fizz_buzz(1));
}
```

Kent Beck:

“Do the simplest thing that could possibly work”

When you don’t know what to do.

Ward Cunningham:

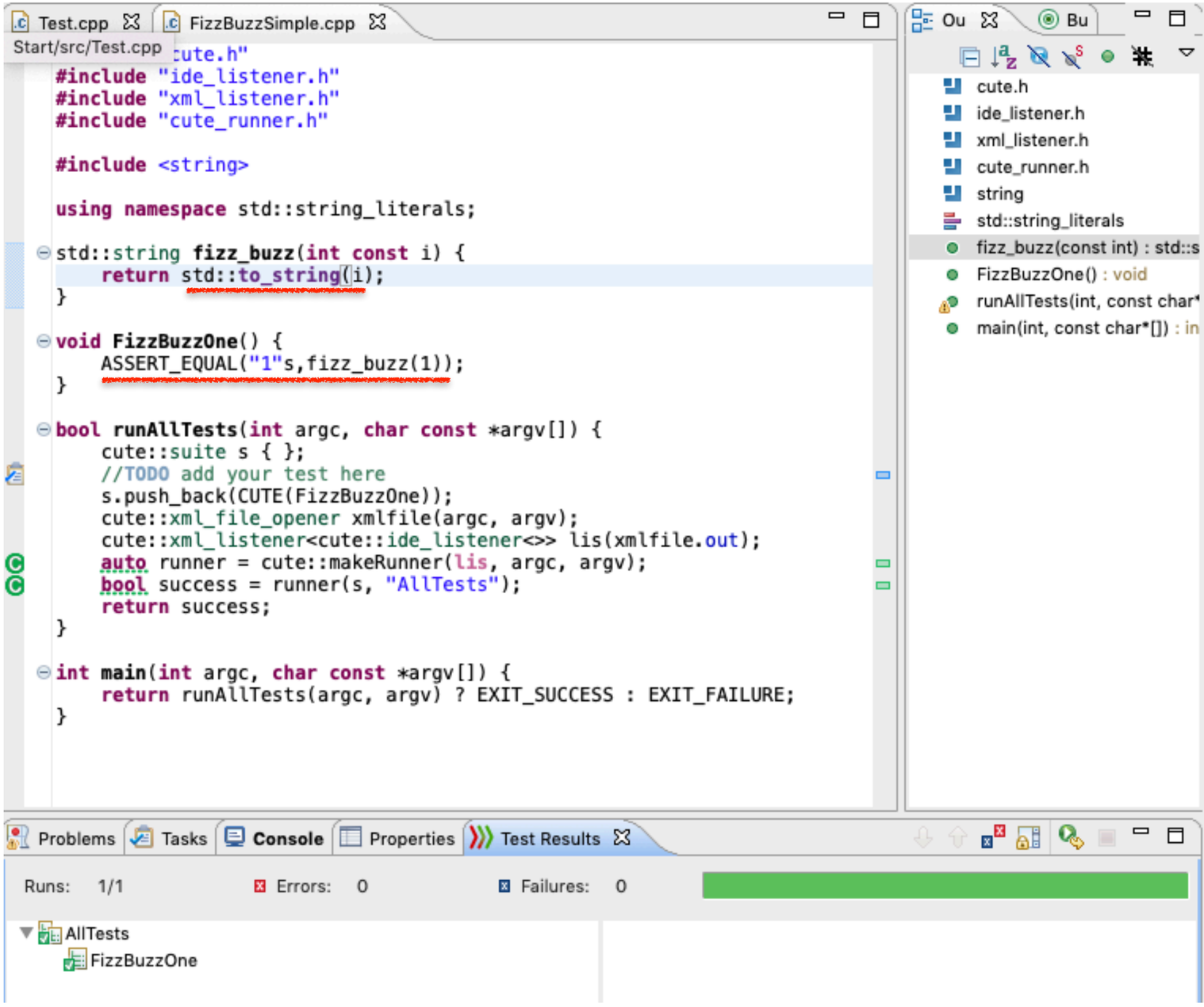
“Kent,
Do the simplest thing that could possibly work”

When you don’t know what to do.

Fake it 'til you make it - do not overuse: Obvious Implementation

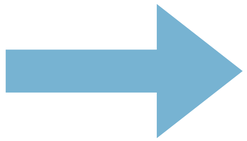
- **get forward quickly**
  - return "1";
- **obvious**
  - return to_string(i);
- **but incomplete**
  - that is OK!
- **Cevelop generates most of the code**
  - templates, hot-keys/menus
  - quick-fixes

mostly generated



Refactor Tests: similar test cases for function to data-driven-table tests


```
void FizzBuzzOne() {
   ASSERT_EQUAL("1"s, fizz_buzz(1));
}
void FizzBuzzTwo(){
   ASSERT_EQUAL("2"s, fizz_buzz(2));
}
void FizzBuzzThreeFizz(){
   ASSERT_EQUAL("Fizz"s, fizz_buzz(3));
}
//...
```



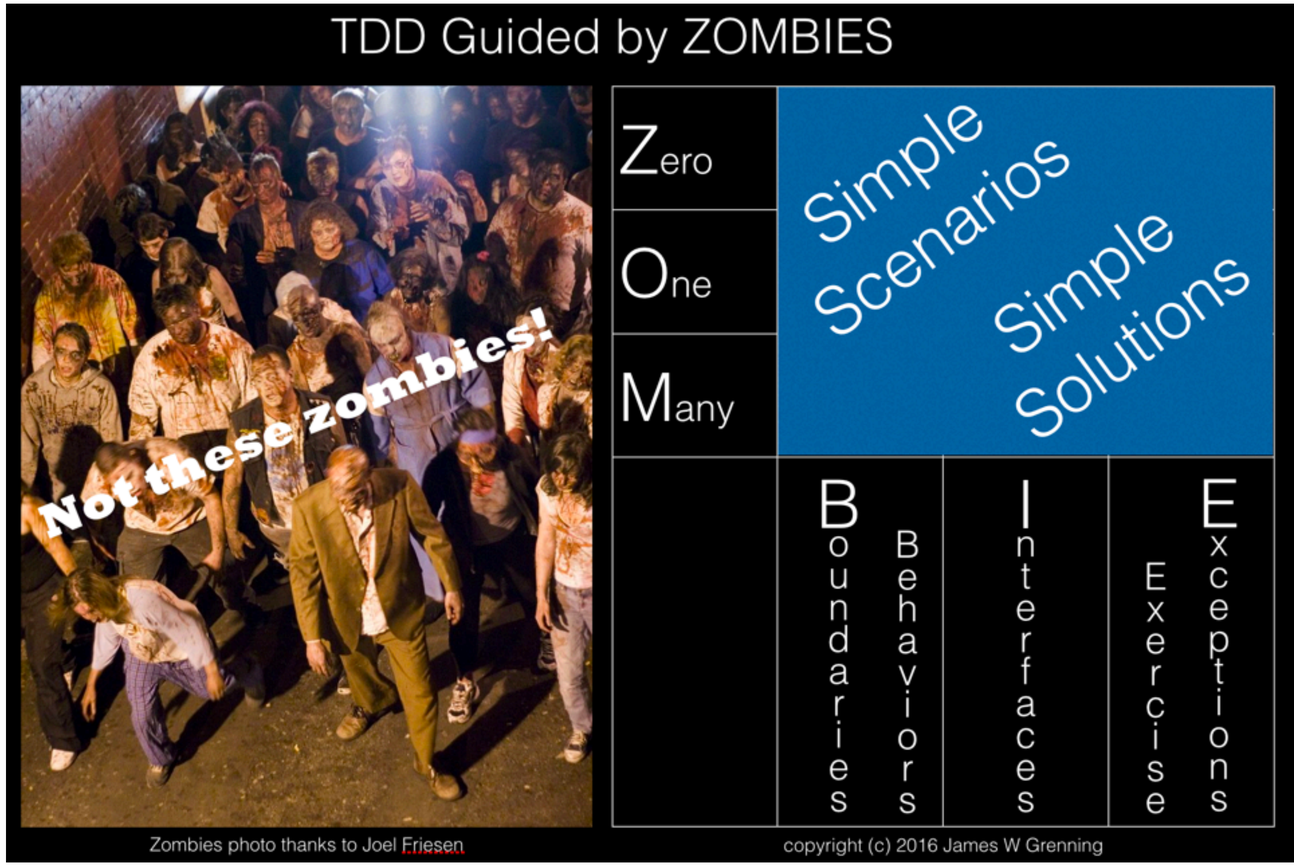
```
struct fizzbuzz_table {
   std::string const expected;
   int const input;
   cute::test_failure const fail; // to keep track of entry location
};

// DDT generates failure message linking back to table entry.
fizzbuzz_table const theFizzBuzzTests[]={
     {"1"s,    1,DDT()},
     {"2"s,    2,DDT()},
     {"Fizz"s, 3,DDT()},
     {"4"s,    4,DDT()},
     {"Buzz"s, 5,DDT()},
     {"Fizz"s, 6,DDT()},
     {"7"s,    7,DDT()},
     {"8"s,    8,DDT()},
     {"Fizz"s, 9,DDT()}
};
void FizzBuzzAllTable(){
   for(auto const &test:theFizzBuzzTests){
      ASSERT_EQUAL_DDT(test.expected, fizz_buzz(test.input), test.fail);
   }
}
```

- **Table easy to extend**
- **Demo?**

How do I write GUTs: Good Unit Tests?

- **If the code is correct, how would I know?**
- **How can I test this?**
- **What else could go wrong?**
- **Could a similar problem happen elsewhere?**

- **Beginners are often satisfied with “happy-path” tests**
  - error conditions and handling need tests (E)
- **Good tests provide better design!**
  - you are immediate victim of your design (not zombies)
- **How can tests be designed well?**
  - refactor them as well as the



source: J.W. Grenning

Some Guideline for GUTs

- **Three phase Test:**
  - Arrange - provide the SUT/infrastructure to work with, e.g., variables & objects
  - Act - exercise the SUT/function to test
  - Assert - prove that the result is what is required
- **ASSERT first**
- **Refactor common fixture**

Some books (based on Java) and Frameworks talk about 4-phase test: setup-exercise-verify-teardown
C++ has destructors for teardown!

```
struct ringbuffer {
   explicit ringbuffer(int  i = 0) {
   }

   int capacity() const {
      return int();
   }

   int size() const {
      return int();
   }

   bool empty() const {
      return true;
   }
};
```

identical fixture extract !

```
void testNewRingbufferIsEmpty() {
   ringbuffer const buf { };
   ASSERTM("initial buffer is empty", buf.empty());
}
void testNewRingbufferHasSizeZero(){
   ringbuffer const buf { };
   ASSERT_EQUAL(0,buf.size());
}
void testProvideCapacityToRingbuffer(){
   ringbuffer const buf { 2 };
   ASSERT_EQUAL(2,buf.capacity());
}
```

What about "fixtures" - setup/teardown?

- **C++ has constructors and destructors for that!**
  - or member initializers
- **If you put the same scaffolding in several test functions**
  - refactor to test class!
- **Registration creates fresh object for each test**
  - EmptyBufferTests{}.testNewRingbufferIsEmtpy
  - reuse of fixture objects possible

```
struct EmptyBufferTests {
   ringbuffer  buf { };
   void testNewRingbufferIsEmpty() const {
      ASSERTM("initial buffer is empty", buf.empty());
   }
   void testNewRingbufferHasSizeZero() const {
      ASSERT_EQUAL(0,buf.size());
   }
   void testPushIntoZeroCapacityRingbufferFails()  {
      ASSERT_THROWS(buf.push(42),std::logic_error);
   }
};
struct RingBufferSizeOneTests {
   ringbuffer buf{1};
   void testPushIncreasesSizeByOne(){
      buf.push(42);
      ASSERT_EQUAL(1,buf.size());
   }
   void testPopAfterPushReceivesValueInOneElementRingbuffer(){
      buf.push(42);
      ASSERT_EQUAL(42,buf.pop());
   }
   void testPopAfterPush1Pops1InOneElementRingbuffer(){
      buf.push(1);
      ASSERT_EQUAL(1,buf.pop());
   }
};
```

common fixture

common fixture

generated

```
s.push_back(CUTE_SMEMFUN(EmptyBufferTests, testNewRingbufferIsEmpty));
s.push_back(CUTE_SMEMFUN(EmptyBufferTests, testNewRingbufferHasSizeZero));
s.push_back(CUTE_SMEMFUN(EmptyBufferTests, testPushIntoZeroCapacityRingbufferFails));
s.push_back(CUTE_SMEMFUN(RingBufferSizeOneTests, testPushIncreasesSizeByOne));
s.push_back(CUTE_SMEMFUN(RingBufferSizeOneTests, testPopAfterPushReceivesValueInOneElementRingbuffer));
s.push_back(CUTE_SMEMFUN(RingBufferSizeOneTests, testPopAfterPush1Pops1InOneElementRingbuffer));
```

Should I really restrict myself to a single ASSERT? - YES, otherwise BUTs

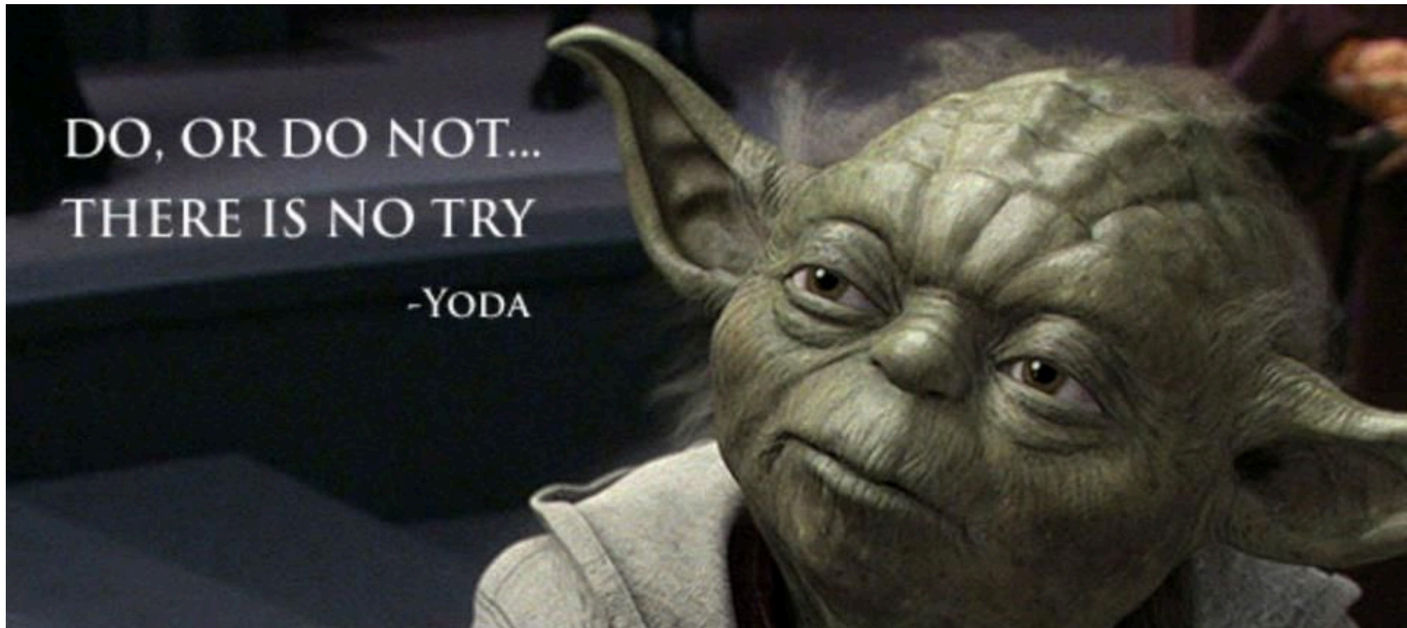
- **A GUT should have only one reason to fail**
- **Some Frameworks allow "checks" that continue when they fail and even encourage that**
  - gtest: EXPECT_* macros
  - catch2: CHECK() macro(s)
- **This encourages bad practice**
- **Consider data driven table tests**
  - or refactoring your tests
- **For testing complex conditions, write a custom check function**
  - but be pragmatic if simple enough

Do Not

Bad UT EXAMPLE

```
void testPushPopAndSize(){
    ringbuffer buf{2};
    ASSERT_EQUAL(0,buf.size());
    buf.push(42);
    ASSERT_EQUAL(1,buf.size());
    ASSERTM("buf not empty after push", not buf.empty());
    ASSERT_EQUAL(42,buf.pop());
    ASSERT_EQUAL(0,buf.size());
    ASSERTM("buf empty after pop",buf.empty());
}
```

side effect



- **Zero, One, Many - to get started**
- **Boundaries & Behavior**
  - C.O.R.R.E.C.T. boundary conditions
- **Interfaces & Independent Tests**
  - Allow for changing the implementation
  - Do not rely on test order
- **Exceptional Behavior Tests**
  - test error handling

- **Right-BICEP**
  - Right results
  - Boundary conditions
  - Inverse relationships
  - Cross check
  - Error conditions
  - Performance characteristics
- **P.I.R.A.T**
  - Professional
  - Independent
  - Repeatable
  - Automatic
  - Thorough

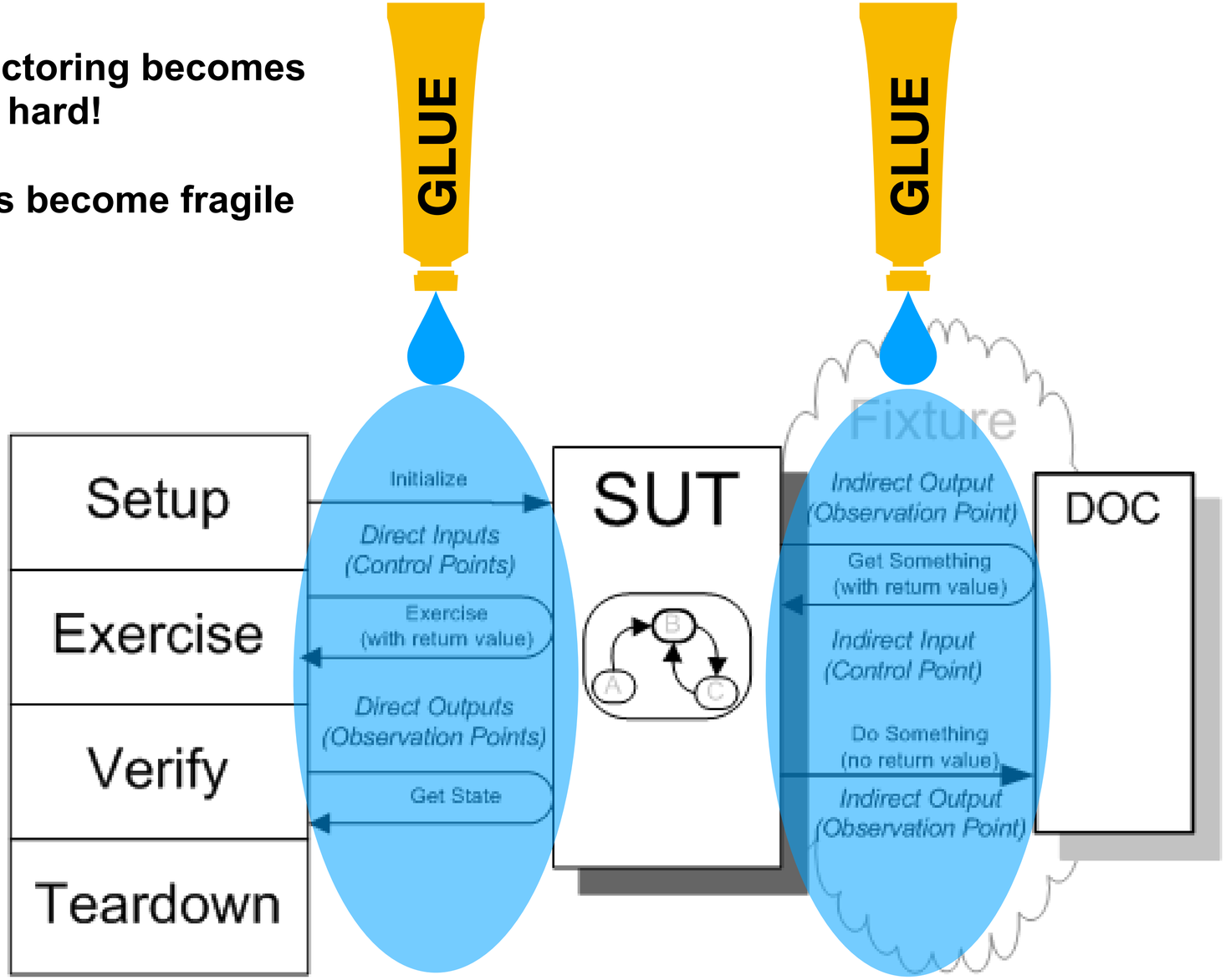
- **C.O.R.R.E.C.T. conditions for cross checking results and triggering errors**
  - Conformance
  - Ordering/Unordered
  - Range
  - Reference
  - Existence
  - Cardinality
  - Time

Literature:
- "Pragmatic Unit Testing" (Thomas&Hunt)
- "Test-Driven Development for Embedded" (Grenning)
- "xUnit Test Patterns" (Meszaros)
- "Test-Driven Development" (Beck)
- and more...

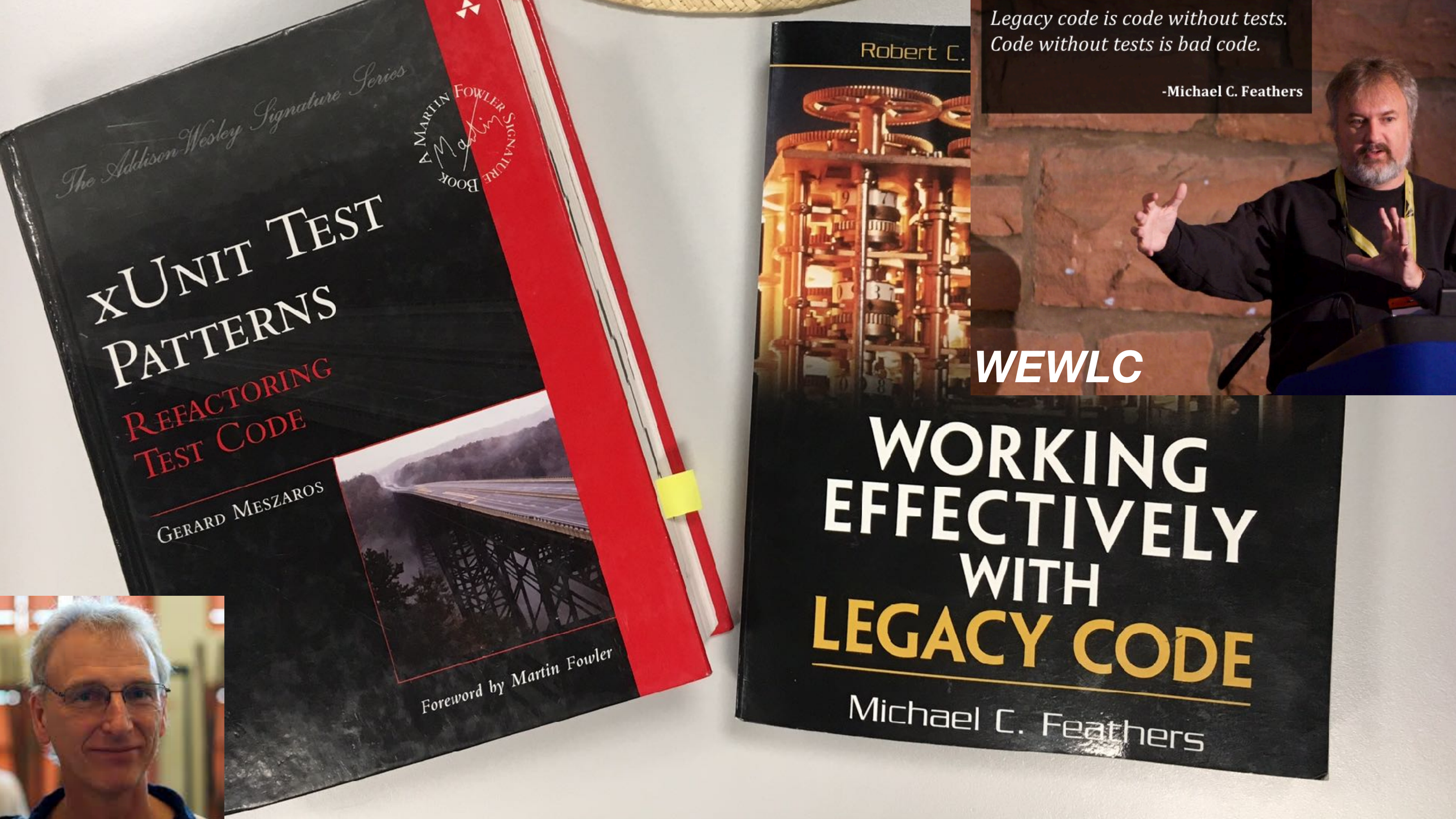
A word on Mocking - use only when really really needed!

- If you need to test legacy code - introduce seams and stub DOCs
- Take all the power C++ and IDEs provide to simplify that as needed
- Only if you have to test the use of a non-changeable stateful API - use mocks
- Be aware of the dangers of mocking (frameworks) for new code's design
- Do not mock unwritten code with a mocking framework (premature design)
- Remember: KISS applies to test automation as well

- **Refactoring becomes very hard!**
- **Tests become fragile**



See my CPPCon 2017 talk
https://www.youtube.com/watch?v=uhuHZXTRfH4


The Addison Wesley Signature Series
A Martin Fowler Signature Book
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler
Robert C.
Working Effectively with Legacy Code
Michael C. Feathers
Legacy code is code without tests.
Code without tests is bad code.
-Michael C. Feathers
WEWLC

Take Aways 🍔 🍟

- **Focus on the "What" not the "How"**
- **Automated good unit tests reduce developer stress**
- **TDD practices foster incremental development in small steps** 👣
- **An IDE can help providing scaffolding and refactoring**
- **Get started with ZOMBIES** 
- **Test for errors and boundary conditions**
- **Fokus on the interface of the SUT**
- **A test should have a single purpose/ASSERT**
- **Mock only what you do not control**


Advanced
Developers
Conference
++
Development for Professionals!

Fragen?

Vielen Dank!

Ich freue mich auf Feedback!

Peter Sommerlad