Clean testing

Good practices in general coding

Alessandro Sciarra

Z02 – Software Development Center

PUNCH Young Academy – PUNCH4NFDI

24 May 2024

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

AxelKrypton

PUNCH 4NFDI

CRC-TR 211 Strong-interaction matter under extreme conditions

Prelude

Leave me a feedback

Was the talk clear, useful, inspiring?

Any general comment?

Drop me an email! → ✉ Alessandro

Disclaimer

Slides are quite full of text for later reading.

Most example are oversimplified to fit on a slide.

Tests shown before the TDD section are probably bad.

Let's discuss!

Outline

1. A short recap about clean code
2. Why (automated) testing?
3. Different types of testing
4. Tests goals – properties – good principles
5. The curse of the **unit** word
6. The weaknesses of white- and black-box testing
7. Test the behaviour, not the implementation
8. Test driven development as discipline
9. Take-home message

A short recap about clean code

What we explored in the clean code talk

- Take advantage of your IDE
- Use meaningful names
- Limit comments and care about formatting
- Function and classes should have a single responsibility (SRP)
- Integration Operation Separation Principle (IOSP)
- Don't repeat yourself (DRY)
- Keep it simple, stupid (KISS)
- Beware of optimisation
- The boyscout rule
- Keep improving

Why testing?

A mind-blowing changing point!

Lots of software should rather work...

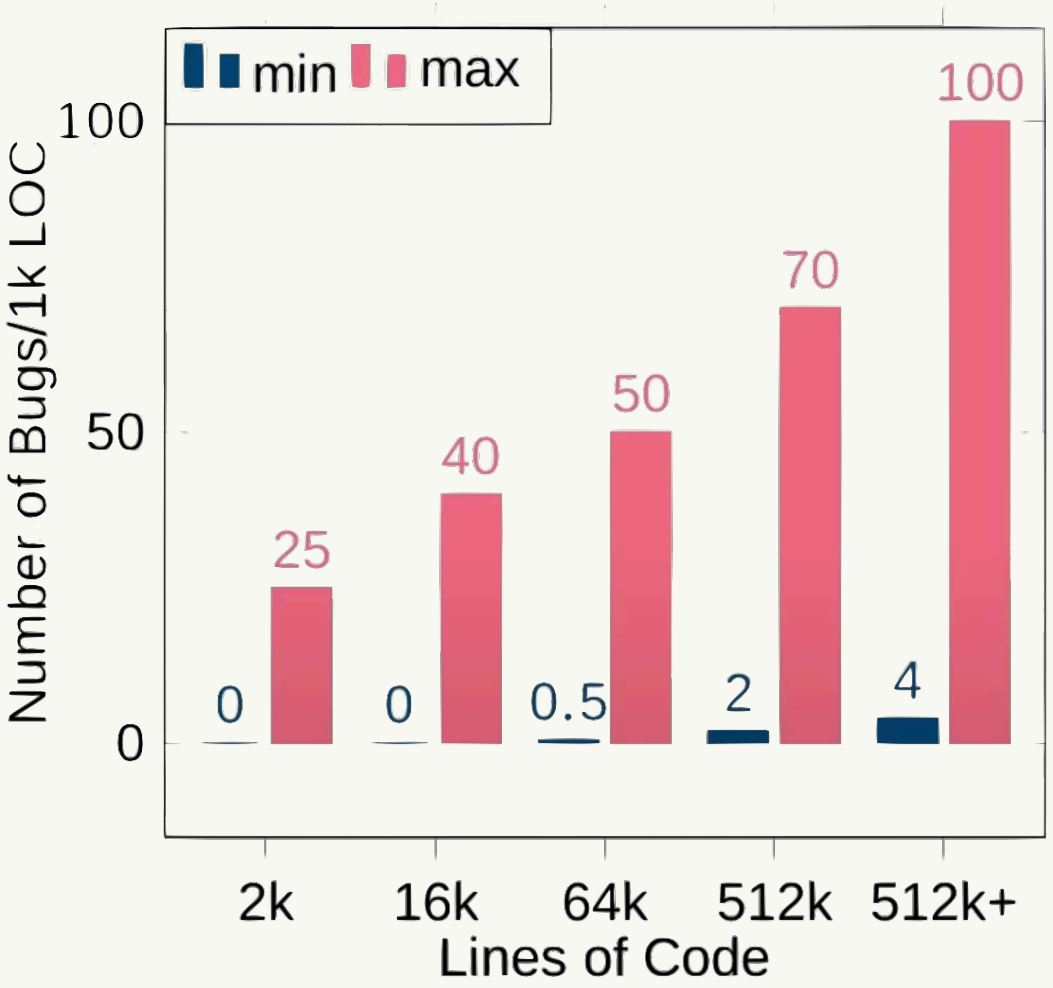
In our society you cannot spend 60 seconds without interacting with a software system

1. In modern car there is software that controls the steering wheel...

2. Think of medical devices, aeroplanes, lifts, etc.
3. Aren't you upset when an app on your phone stalls or crashes?!

An old, simple fact

Taken from Casper Jones, «Program quality and programmer productivity» – 1977



The unfortunate tendency in many cases

- It gives reasonable results $\Rightarrow$ it works! 😱
- Manual (and possibly not systematic) testing is unfortunately still common practice in our community!
- Even if done in a systematic way, you will never run your tests enough!

An remarkable story about manual testing

I have been using an own manually tested "agenda-script" to track my working time **for months** and then one day it suddenly crashed. Where was the problem?

The unfortunate tendency in many cases

- It gives reasonable results $\Rightarrow$ it works! 😱
- Manual (and possibly not systematic) testing is unfortunately still common practice in our community!
- Even if done in a systematic way, you will never run your tests enough!

An remarkable story about manual testing

I have been using an own manually tested "agenda-script" to track my working time **for months** and then one day it suddenly crashed. Where was the problem? I assumed **every** day has 24 hours... 😇

The unfortunate tendency in many cases

- It gives reasonable results $\Rightarrow$ it works! 😱
- Manual (and possibly not systematic) testing is unfortunately still common practice in our community!
- Even if done in a systematic way, you will never run your tests enough!

Theorem:

Prove that, with $n, x, y, z \in \mathbb{N} > 0$ and $n > 2$, the equation $x^n + y^n = z^n$ has no solutions.

Would you accept this as proof or even argument?

$2^3 + 3^3 \neq 4^3$ and $1^4 + 5^4 \neq 6^4$ and $4^5 + 2^5 \neq 5^5$

The unfortunate tendency in many cases

- It gives reasonable results $\Rightarrow$ it works! 😱
- Manual (and possibly not systematic) testing is unfortunately still common practice in our community!
- Even if done in a systematic way, you will never run your tests enough!

Now, software testing is not like mathematics...

...but would you accept a physics fact **without rigorous evidence**?

Consider one of your programs. Does it work?

How do you prove that a software works?

Consider one of your programs. Does it work?

How do you prove that a software works?

You don't. You try proving it doesn't work... and fail!

Consider one of your programs. Does it work?

How do you prove that a software works?

You don't. You try proving it doesn't work... and fail!

The scientific method

1. Make an hypothesis $\mathcal{C}$ $\longrightarrow$ «My code is correct»
2. Tests attempt to show $!\mathcal{C}$
3. Confidence in $\mathcal{C}$ tracks thoroughness of tests

Testing code is science (cf. Popper's falsification principle)

Uncle Bob's magic button

Uncle Bob's magic button

Fearless competence

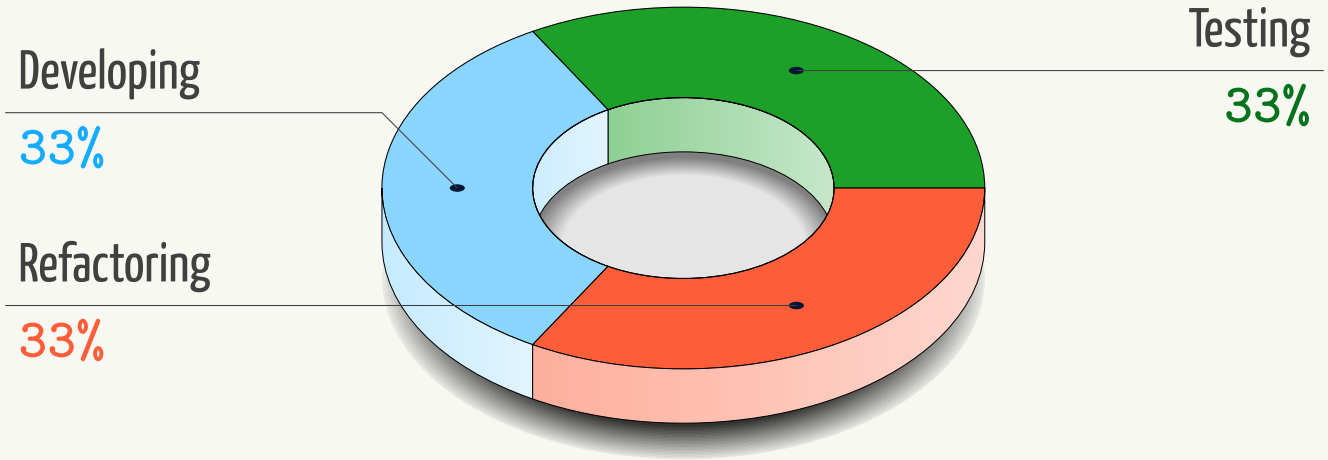
Are you afraid to touch the code? Are you afraid to improve the code? Do you have that little echo in your ear saying – Ah, if it ain't broken, don't fix it? I expect you to improve the code, all of the time, fearless competence. How do you get fearless competence?

Tests! What if I had a button, a little button I could push on my keyboard and some lights would blink and science-fiction sounds would come out of my laptop for a few minutes and then a little green light would light up and that green light told me that the system worked. And I believed it.

Robert C. Martin

The correct mental approach to testing

- Testing is part of the natural evolution of the code (not an overhead)
- Test code has to fulfil the same clean code rules as production code
- Tests are the key to be free to change (improve) the production code!
- Keep balance between adding new code, writing tests and refactoring

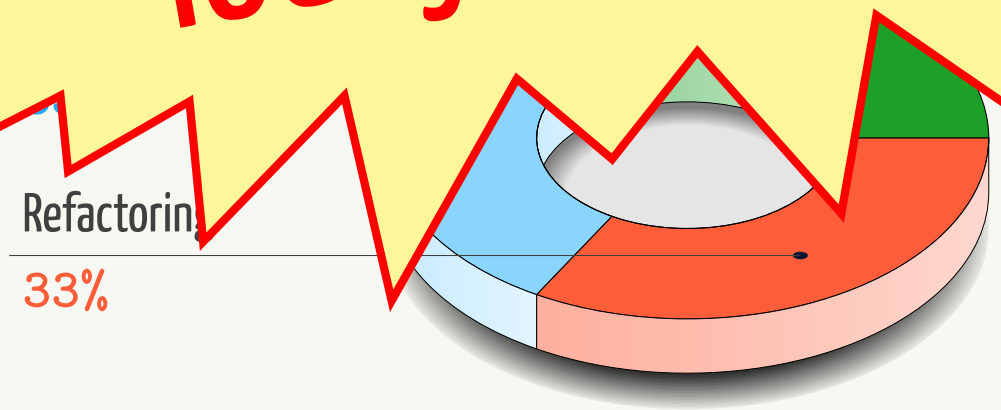


Slide taken from the Clean Code talk.

The correct mental approach to testing

- Testing is part of the natural evolution of the code (not an overhead)
- Test code has to fulfil the same clean c e rules as production ode
- Tests are th ey to be free t ng ro h d n de!
- Keep b lance n ng

Today much more!



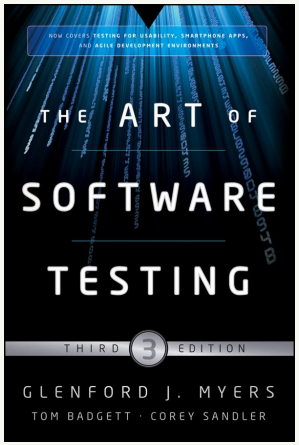
Slide taken from the Clean Code talk.

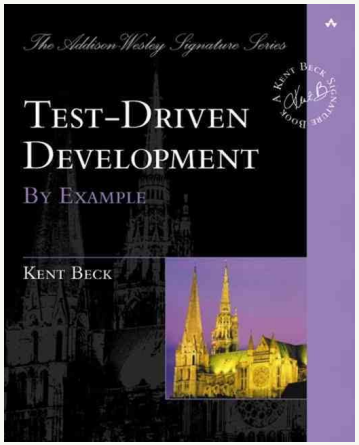
Different types of testing

Software testing

Test code is just as important as production code

- Testing levels
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing





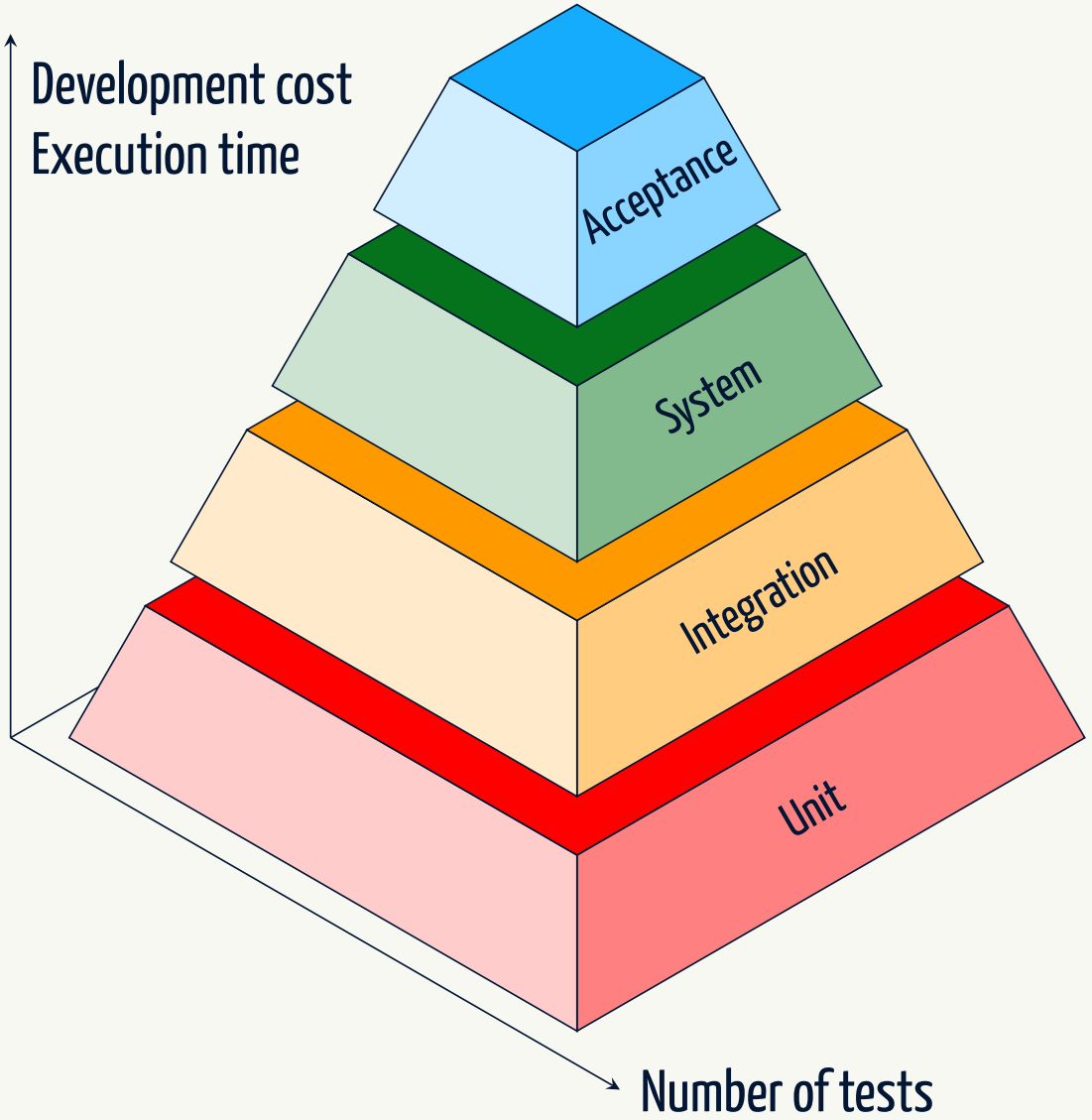
- Testing types and techniques
  - Manual testing
  - Automated testing
  - Continuous testing
  - Functional testing
  - Mutation testing
  - Fuzzy testing
  - ...

- Plenty of videos/resources on the web

🔗 CppCon2015 – All your tests are terrible...
🔗 CppCon2020 – The science of unit tests
🔗 CppCon2020 – Back to Basics: Unit Tests
🔗 TDD, Where Did It All Go Wrong
🔗 TDD for those who don't need it
...

The videos linked here above (as well as the references therein) were mainly used to prepare this talk.

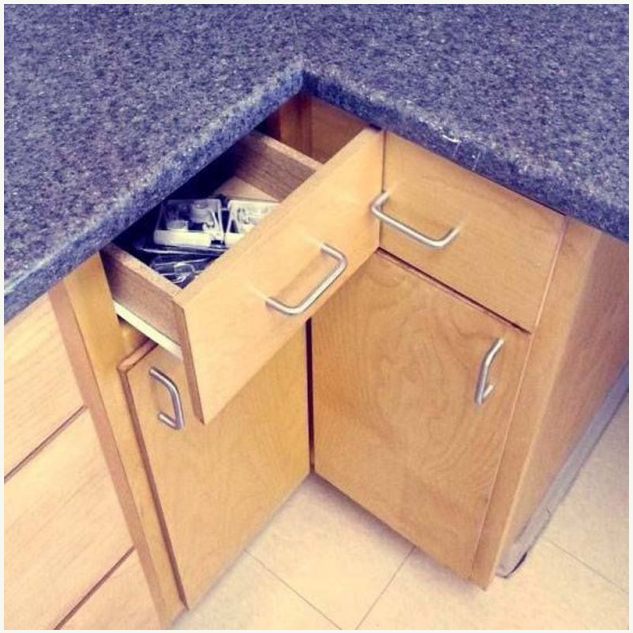
The tests pyramid



Unit tests are not enough!

Integration tests

Once you are sure that all components work correctly alone, be sure they work as expected together!

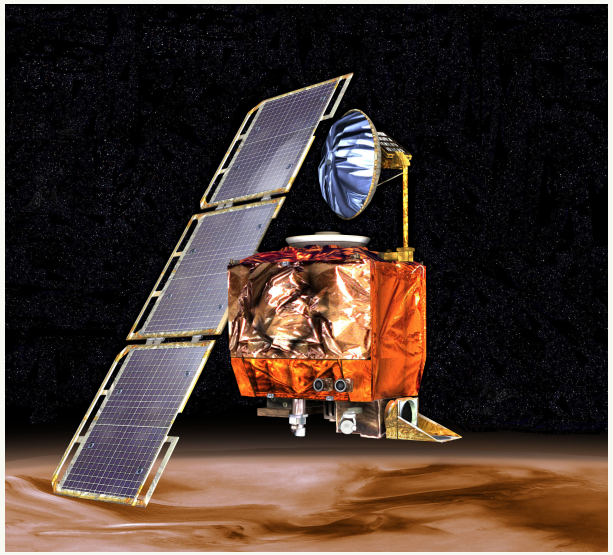


I already mentioned this in the Clean code talk.

Unit tests are not enough!

Integration tests

Once you are sure that all components work correctly alone, be sure they work as expected together!



Mars Climate Orbiter lost in 1999 – By NASA/JPL/Corby Waste – © Public Domain

I already mentioned this in the Clean code talk.

And once integration tests pass?

System testing

System testing tests a completely integrated system to verify that the system meets its requirements.

Wikipedia

Acceptance testing

Formal testing with respect to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether to accept the system.

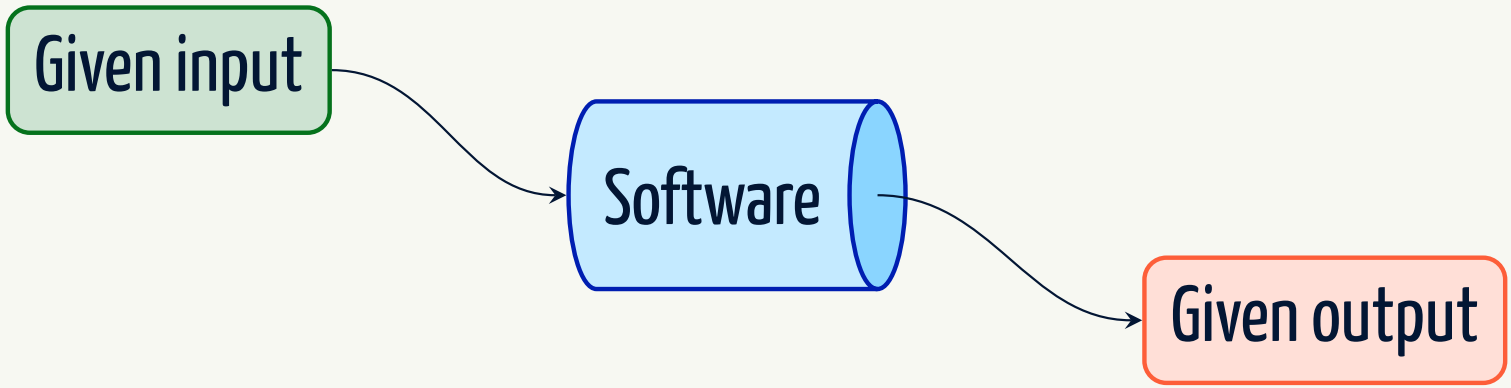
Standard Glossary of Terms used in Software Testing

In smaller projects

You might not need all categories, but, if you drop any, do it consciously!

A technique for system/acceptance tests

> **Functional tests**
>
> They tend to answer the question of "can the user do this" or "does this particular feature work".
>
> Wikipedia



- Consider to implement these tests in an external script!
- This is a good starting point to work with legacy code without tests!

Tests goals – properties – good principles

General idea and overview

There is slightly different advice around, pick your favourite:

Write F.I.R.S.T tests first

Fast:	To be run frequently
Independent:	To make failures meaningful
Repeatable:	To be run everywhere with same outcome
Self-validating:	To be automatised
Timely:	To make production code testable

General idea and overview

There is slightly different advice around, pick your favourite:

Google's perspective

Readable:	Tests are correct by inspection
Correct:	Do not rely on known bugs and test real scenarios
Complete:	Test most edge cases
Documenting:	Demonstration of how the API works
Resilient:	Repeatable, independent, hard to break, hermetic, etc.

General idea and overview

There is one not-negotiable **requirement of tests**, though:

General idea and overview

There is one not-negotiable **requirement of tests**, though:

A small interlude: Unit test frameworks for C++

Essential for self-validation and automation!

1 Boost Test Library

Documentation

```
#define BOOST_TEST_MODULE My Test
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_CASE(first_test)
{
    int i = 1;
    BOOST_TEST(i);
    BOOST_TEST(i == 2);
}
```

```
Running 1 test case...
test_file.cpp(8): error: in "first_test": check i == 2 has
    ↪ failed [1 != 2]

*** 1 failure is detected in the test module "My Test"
```

A small interlude: Unit test frameworks for C++

Essential for self-validation and automation!

1. Boost Test Library — Documentation
2. GoogleTest — User's guide

```
#include <gtest/gtest.h>
#include "src/factorial.h"

TEST(FactorialTest, HandlesZeroInput) {
    EXPECT_EQ(factorial(0), 1);
}

TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(factorial(1), 1);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

A small interlude: Unit test frameworks for C++

Essential for self-validation and automation!

1. Boost Test Library — Documentation
2. GoogleTest or Catch2 — User's guide or GitHub project

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

unsigned int factorial(unsigned int number)
{
    return number > 1 ? factorial(number - 1) * number : 1;
}

TEST_CASE("Factorials are computed", "[factorial]") {
    REQUIRE(factorial(0)  == 1);
    REQUIRE(factorial(1)  == 1);
    REQUIRE(factorial(3)  == 6);
    REQUIRE(factorial(10) == 3628800);
}
```

A small interlude: Unit test frameworks for C++

Essential for self-validation and automation!

1. Boost Test Library — Documentation
2. GoogleTest or Catch2 — User's guide or GitHub project
3. Vir's Unit Test Framework — GitHub repository

```
#include <vir/test.h>

TEST(test_name) {
    int test = 3;
    COMPARE(test, 2) << "more " << "details";
    //COMPARE(test, 2).on_failure("more ", "details");
}
```

```
FAIL: ┌ at tests/testfile.cpp:5 (0x40451f):
FAIL: │ test (3) == 2 (2) -> false more details
FAIL: └ test_name

Testing done. 0 tests passed. 1 tests failed.
```

A small interlude: Unit test frameworks for C++

Essential for self-validation and automation!

1. Boost Test Library — Documentation
2. GoogleTest or Catch2 — User's guide or GitHub project
3. Vir's Unit Test Framework — GitHub repository

...and so many mores! — List on Wikipedia

Not all have the same weight and functionality

Pick your favourite depending on your needs and project:

1. Natural choice if you already depend on Boost
2. More complex but also more powerful
3. Very simple and lightweight

If you use a different language, you can find at the Wikipedia link lists for most programming languages.

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs

Having tests passing is simple...

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs

Having tests passing is simple...

```
int cube(int x) {
    //TODO: To be implemented
    return 0;
}

TEST(cube_test) {
    VERIFY(0 == cube(2));
    VERIFY(0 == cube(3));
    VERIFY(0 == cube(42));
}
```

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs

...better to have them failing!

```
int cube(int x) {
    //TODO: To be implemented
    return 0;
}

TEST(cube_test) {
    VERIFY(8 == cube(2));
    VERIFY(27 == cube(3));
    VERIFY(74088 == cube(42));
}
```

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs
- Never mock the code that you are testing

This is probably useless

```
class MockCoffeeMachine : public CoffeeMachine {
    //For simplicity we assume always in temperature
    double warmUp() override {
        return constants::brewingT;
    }
}

BOOST_AUTO_TEST_CASE(warmUp_test) {
    MockCoffeeMachine machine{};
    BOOST_REQUIRE_CLOSE(machine.warmUp(),
                        constants::brewingT, 0.1);
}
```

Having tests is not enough, you must trust them!

Few words about tests correctness

- Never rely on known bugs
- Never mock the code that you are testing

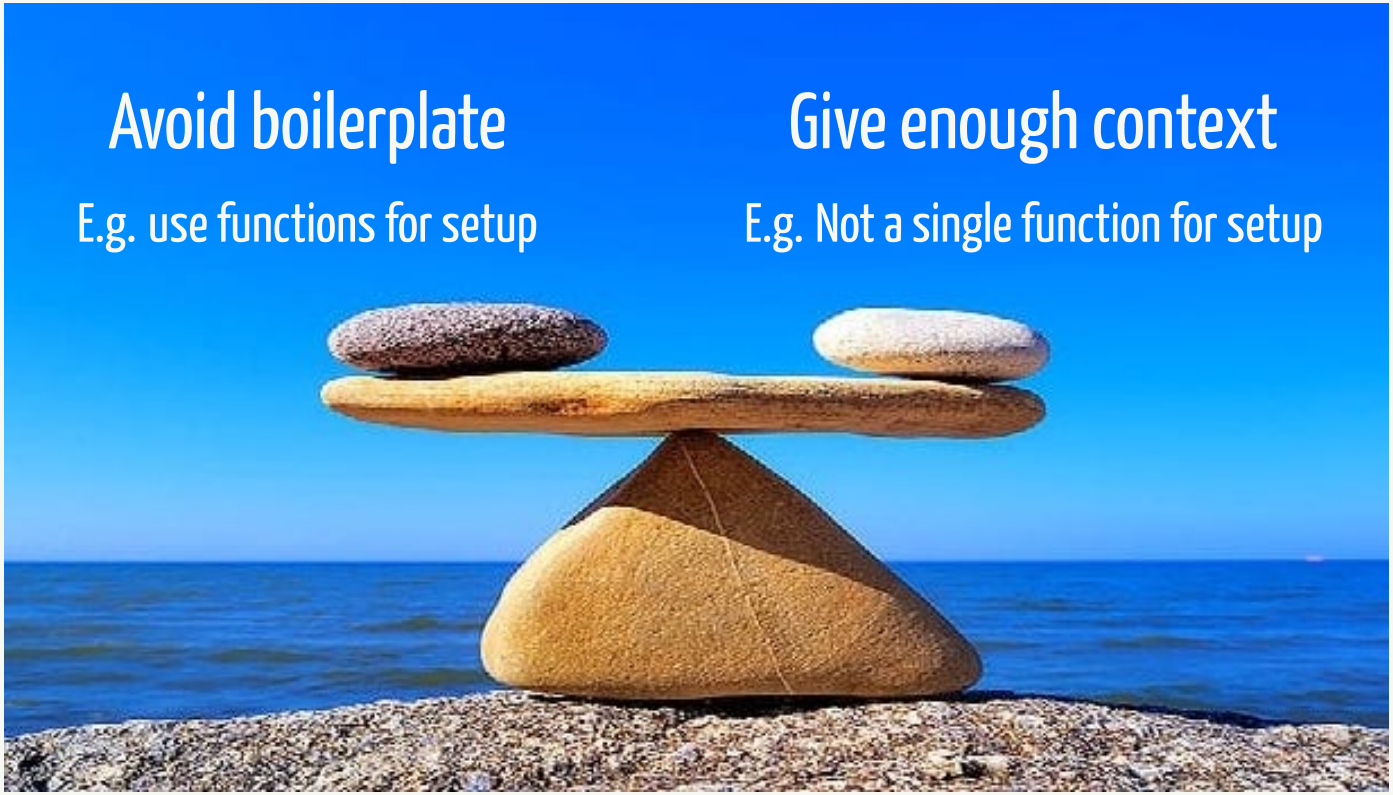
Think of a clear reaction procedure in your team

What should happen in your project when a test fails?
...and when a bug that was not caught by tests is found?

↓

Never miss any chance to improve your tests!

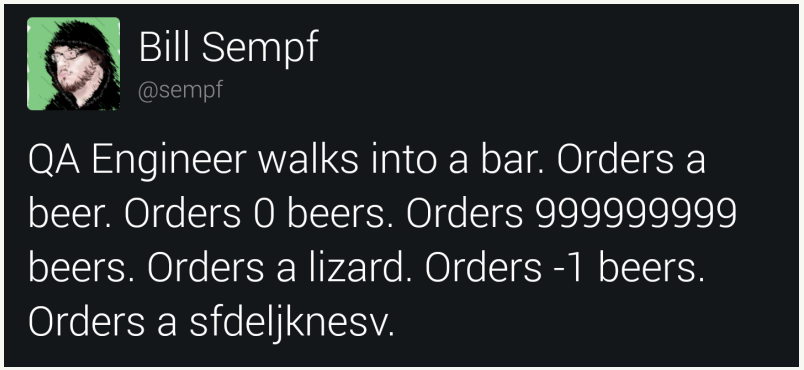
Tests readability



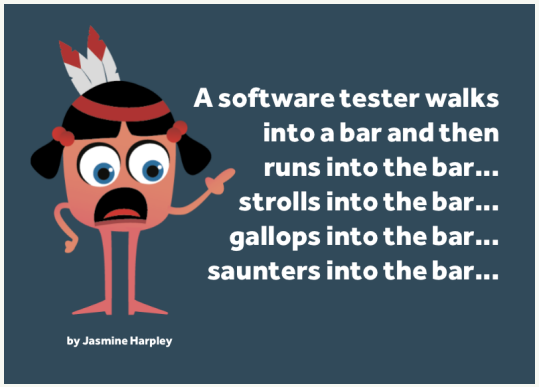
Remember: Tests are correct by inspection!

Tests completeness: What should I test and how?

- Values



- Actions



...and anything that makes sense!

I already mentioned this in the Clean code talk.

Tests completeness: What should I test and how?

A very nice example from Titus Winter

```
TEST(FactorialTest, BasicTests) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(120, Factorial(5));
}
```

Tests completeness: What should I test and how?

A very nice example from Titus Winter

```
int Factorial(int n) {
    if (n == 1) return 1;
    if (n == 5) return 120;
    return -1; // TODO(goofus): figure this out.
}

TEST(FactorialTest, BasicTests) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(120, Factorial(5));
}
```

Tests completeness: What should I test and how?

A very nice example from Titus Winter

```
int Factorial(int n) {
    if (n == 1) return 1;
    if (n == 5) return 120;
    return -1; // TODO(goofus): figure this out.
}

TEST(FactorialTest, BasicTests) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(120, Factorial(5));
    EXPECT_EQ(1, Factorial(0));
    EXPECT_EQ(479001600, Factorial(12));
    EXPECT_EQ(std::numeric_limits<int>::max() , Factorial(13));
    EXPECT_EQ(1, Factorial(0));
    EXPECT_EQ(std::numeric_limits<int>::max() , Factorial(-10));
}
```

This will pretty much naturally emerge when doing TDD

The hidden value of tests

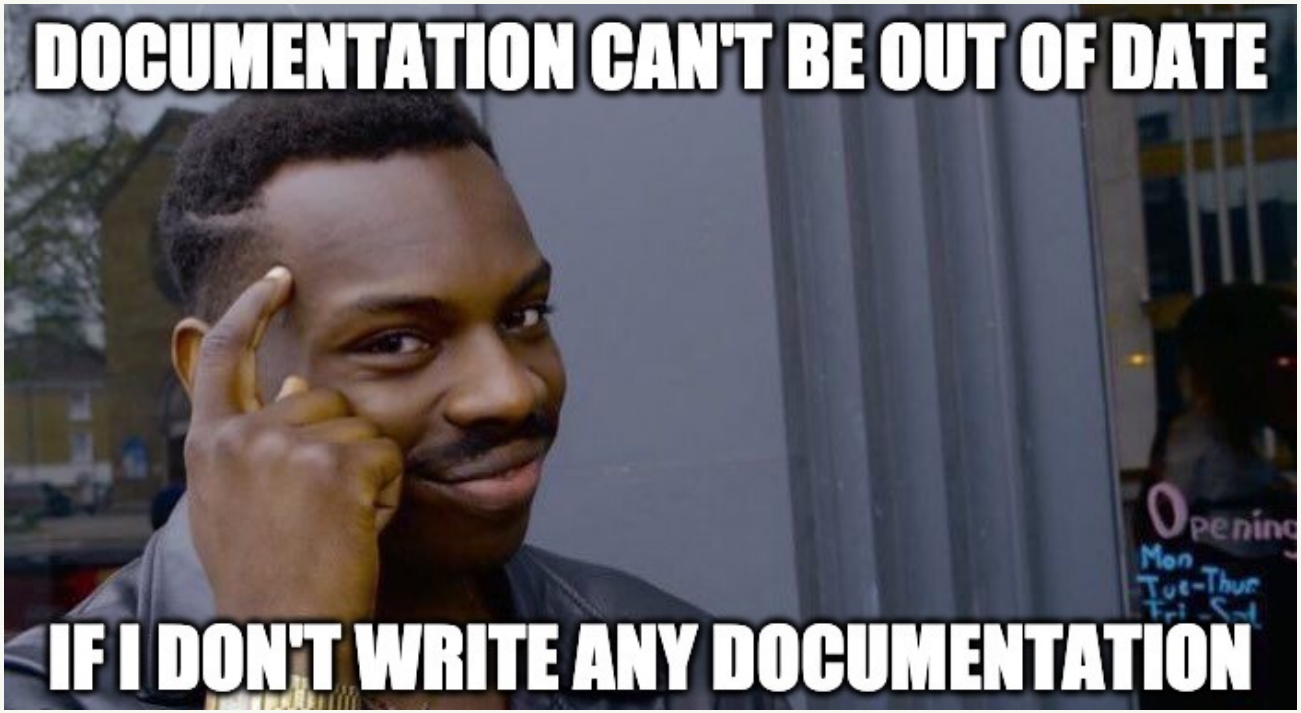
Few words about tests as API usage examples

Of course, a meme, do not take it (too) seriously, but...

The hidden value of tests

Few words about tests as API usage examples

Of course, a meme, do not take it (too) seriously, but...



...good tests should show how to use your code!

In this sense, it should be avoided to use implementation details in tests (cf. white box testing downsides).

Tests resilience: Google advice

- No flaky tests → Tests should always give the same outcome
- No brittle tests → One failure should not trigger many failures
- Tests reference values should not come from the system under test!
- Changing tests order should never change the outcome
- Tests should be as hermetic as possible → No I/O, no network, etc.
- No deep dependence → Avoid any implicit assumption

Let's see some examples!

This list and all GoogleTest examples are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

A sneaky flaky test: What happens if the test-system is overloaded?

```
TEST(UpdaterTest, RunsFast) {
    Updater updater;
    updater.UpdateAsync();
    SleepFor(Seconds(.5)); // Half a second should be *plenty*.
    EXPECT_TRUE(updater.Updated());
}
```


This list and all GoogleTest examples are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

A sneaky flaky test: What happens if the test-system is overloaded?

```
TEST(UpdaterTest, RunsFast) {
    Updater updater;
    updater.UpdateAsync();
    SleepFor(Seconds(.5)); // Half a second should be *plenty*.
    EXPECT_TRUE(updater.Updated());
}
```

An infamous brittle pattern

```
TEST(Logger, LogWasCalled) {
    StartLogCapture();
    EXPECT_TRUE(Frobber::Start());
    EXPECT_THAT(
      Logs(), Contains("file.cc:421: Opened file frobber.config" )
    );
}
```


This list and all GoogleTest examples are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

Where are the reference values coming from?

```
BOOST_AUTO_TEST_CASE(Dirac_M)
{
    const hmc_float refs[4] =
        { 2610.3804893063798, 4356.332327032359,
          2614.2685771909237, 4364.1408252701831 };
    test_fermionmatrix<physics::fermionmatrix::Dslash>(refs);
}
```


This list and all GoogleTest examples are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

Where are the reference values coming from?

```
BOOST_AUTO_TEST_CASE(Dirac_M)
{
    const hmc_float refs[4] =
        { 2610.3804893063798, 4356.332327032359,
          2614.2685771909237, 4364.1408252701831 };
    test_fermionmatrix<physics::fermionmatrix::Dslash>(refs);
}
```

A non-hermetic test: What if run twice in parallel?

```
TEST(Server, StorageTest) {
    StorageServer* server = GetStorageServerHandle();
    auto my_val = rand();
    server->Store("testkey", my_val);
    EXPECT_EQ(my_val, server->Load("testkey"));
}
```


This list and all GoogleTest examples are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

Tests resilience: Google advice

Deep dependence example

```
class File {
  public:
    // ...
    virtual bool StatWithOptions(Stat_t* stat, StatOptions opts) {
        return this->Stat(stat); // In base class ignore options
    }
};
TEST(Filesystem, FSUsage) {
    // ... and call to StatWithOptions
    EXPECT_CALL(file, Stat(_)).Times(1);
}  // This relies on a call to base StatWithOptions!!
```

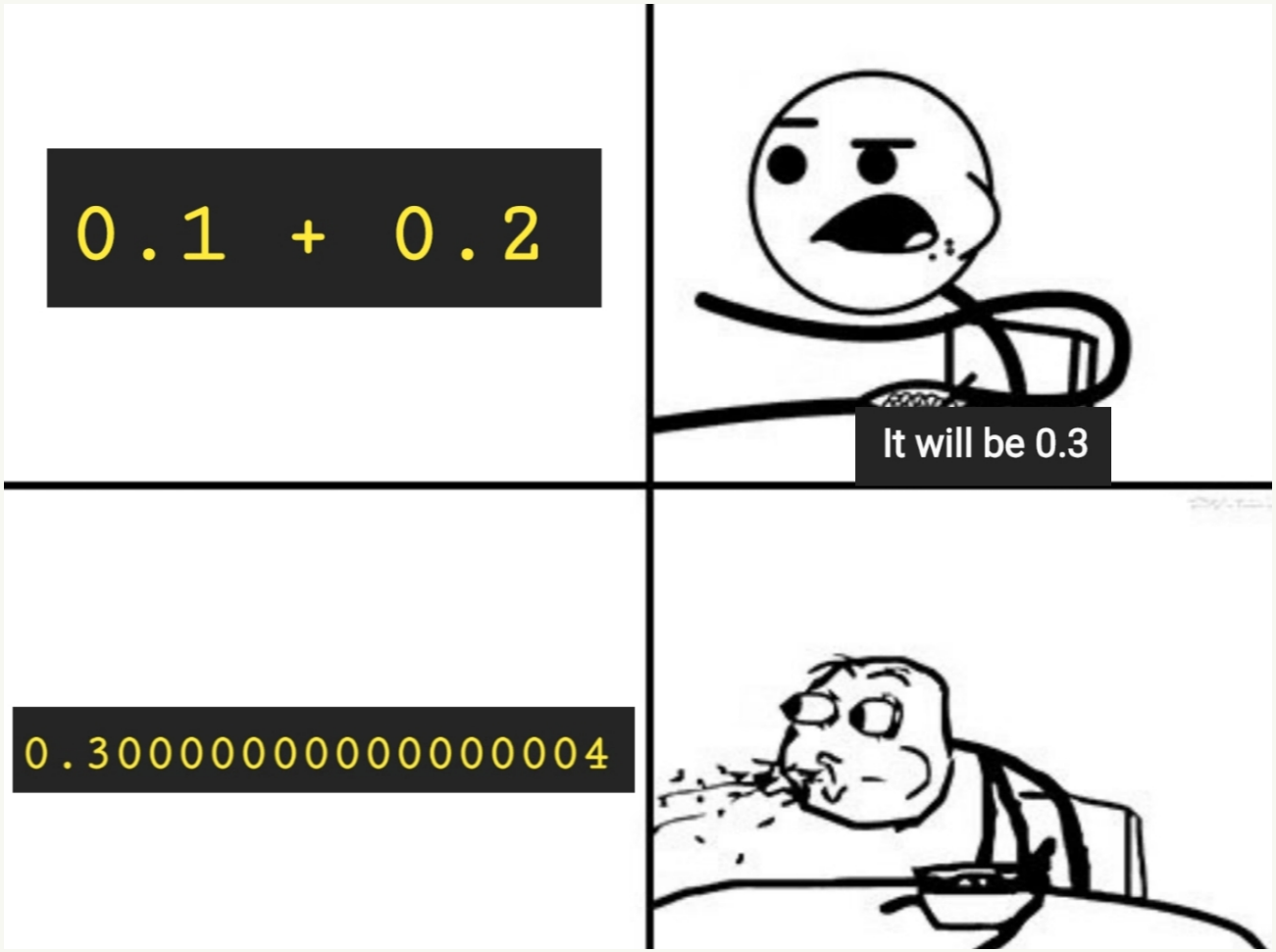
The law of implicit interfaces (AKA Hyrum's law)

Given enough users of your public interface, soon or later someone will start implicitly relying on your implementation details (speed, memory consumption, etc.).

Hyrum Wright

This list and all GoogleTest examples are taken from Titus Winters' and Hyrum Wright's talk at CppCon2015.

A slide about tests precision



A slide about tests precision

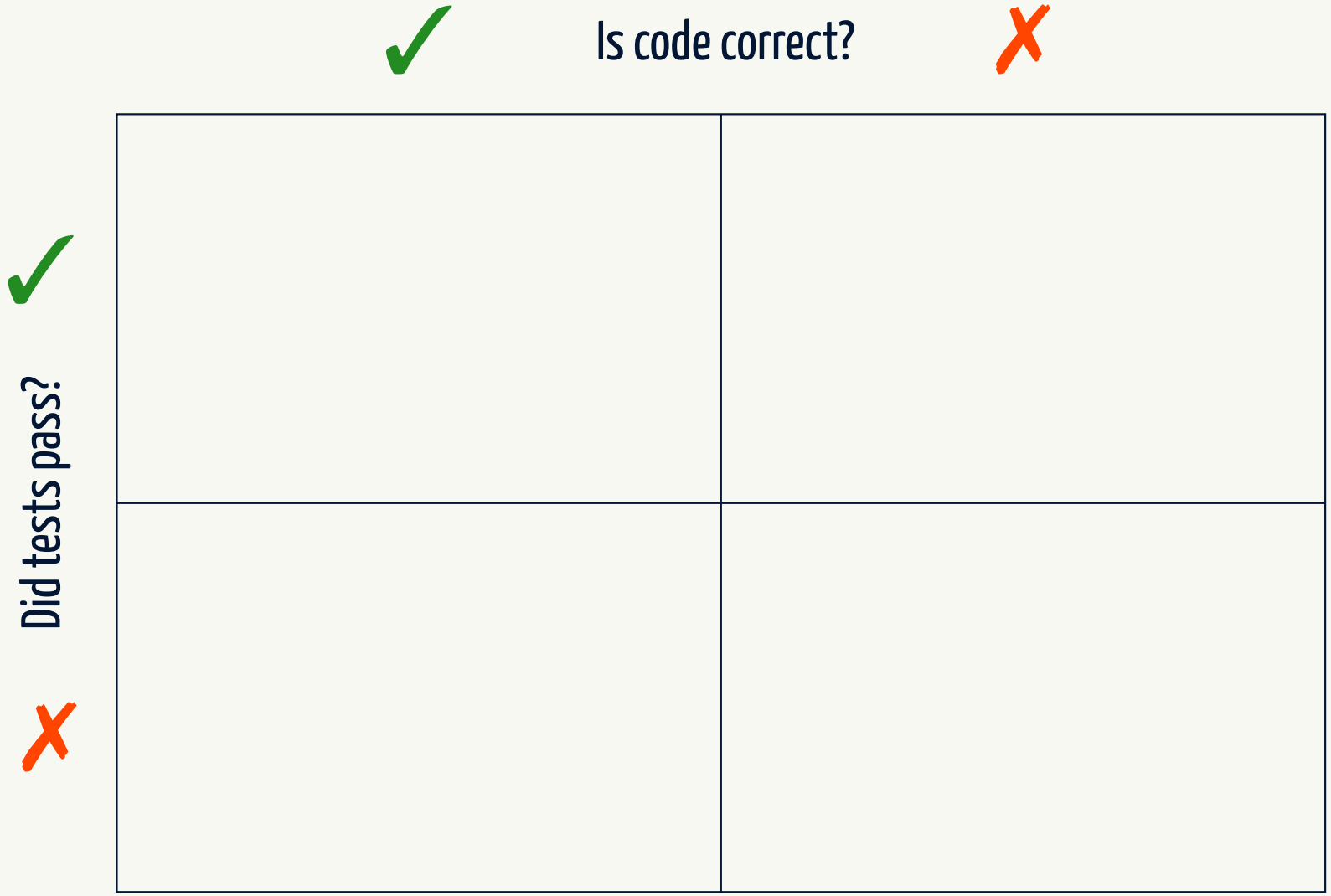
- We all know that machine precision is finite!
- Comparing floating point numbers has to be done carefully
- If you know a reference value exactly, put in the code more than the needed digits $\longrightarrow \pi = 3.14159265358979323846264 3$
- Be aware of possible compiler techniques, e.g. FMA { fused multiply-add }
  ↳ This could make your test fail if you require too high precision
- Requiring too high precision makes tests brittle
- Requiring too low precision makes tests useless
- After all it is a judgement call!

Few more words about reproducibility

How can I make my tests reproducible?

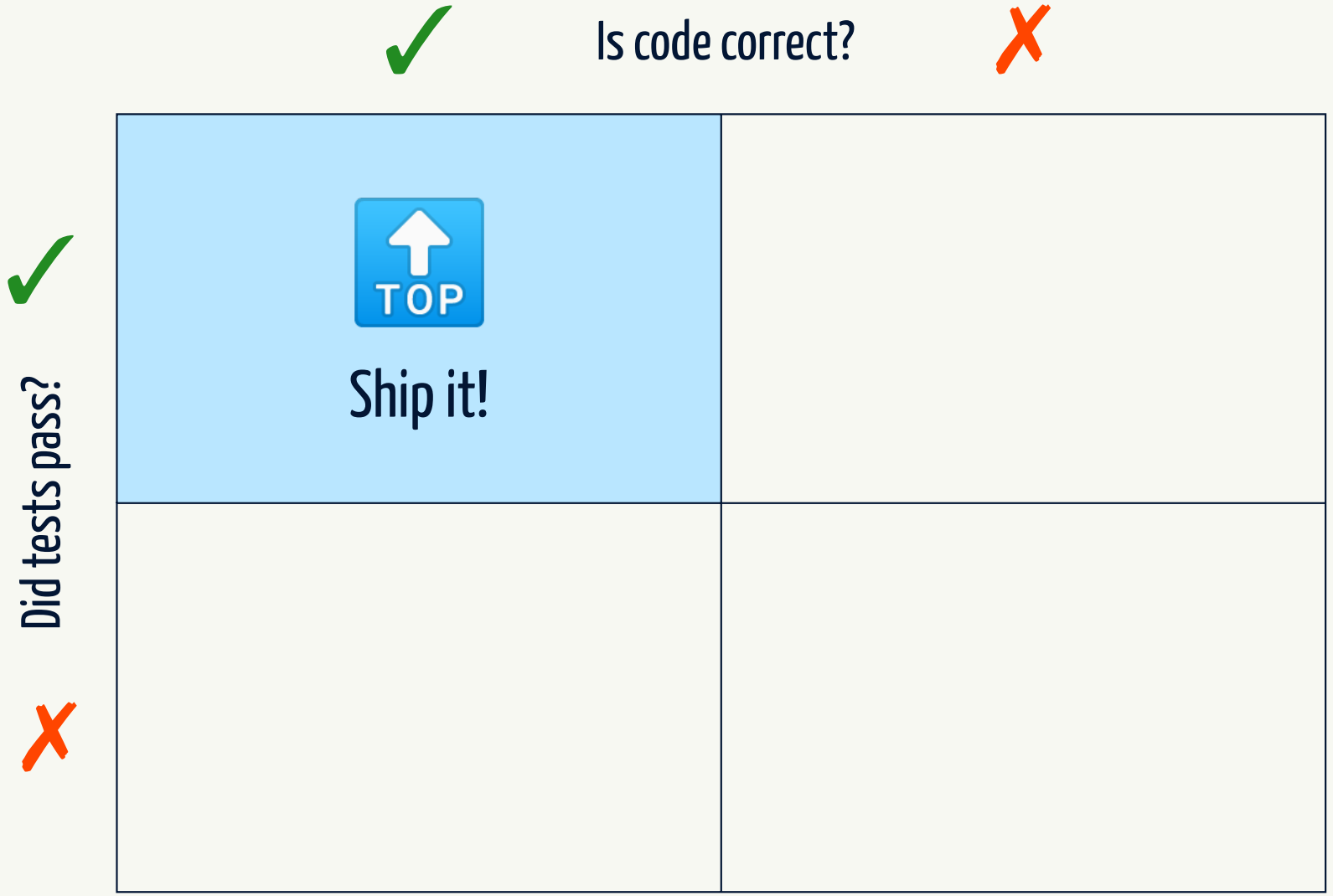
1. Isolate them from non-deterministic aspects { e.g. network, database, random numbers }
   ↳ This is not always possible!
2. Measure unwanted environment effect and subtract it in tests
3. Detect if test can be run in the present environment
   ↳ If not, either fail or handle situation
4. Handle intrinsic noise by statistics
   ↳ E.g. if random numbers are needed, measure the failure frequency and re-run the test a number of times large enough to consider a failure as such!

Tests accuracy and reacting to failures



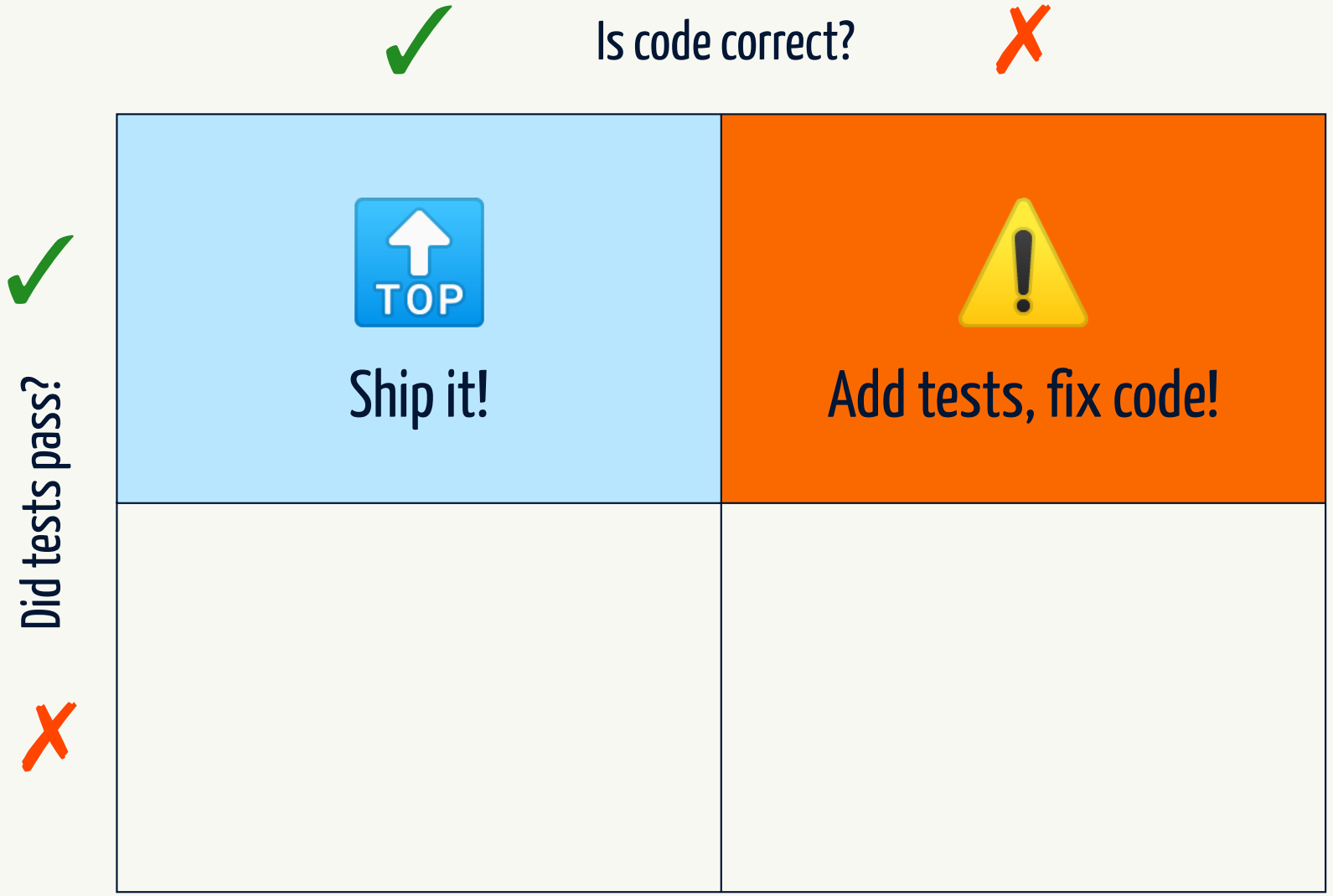
Tests accuracy means how much tests outcome matches code correctness.

Tests accuracy and reacting to failures



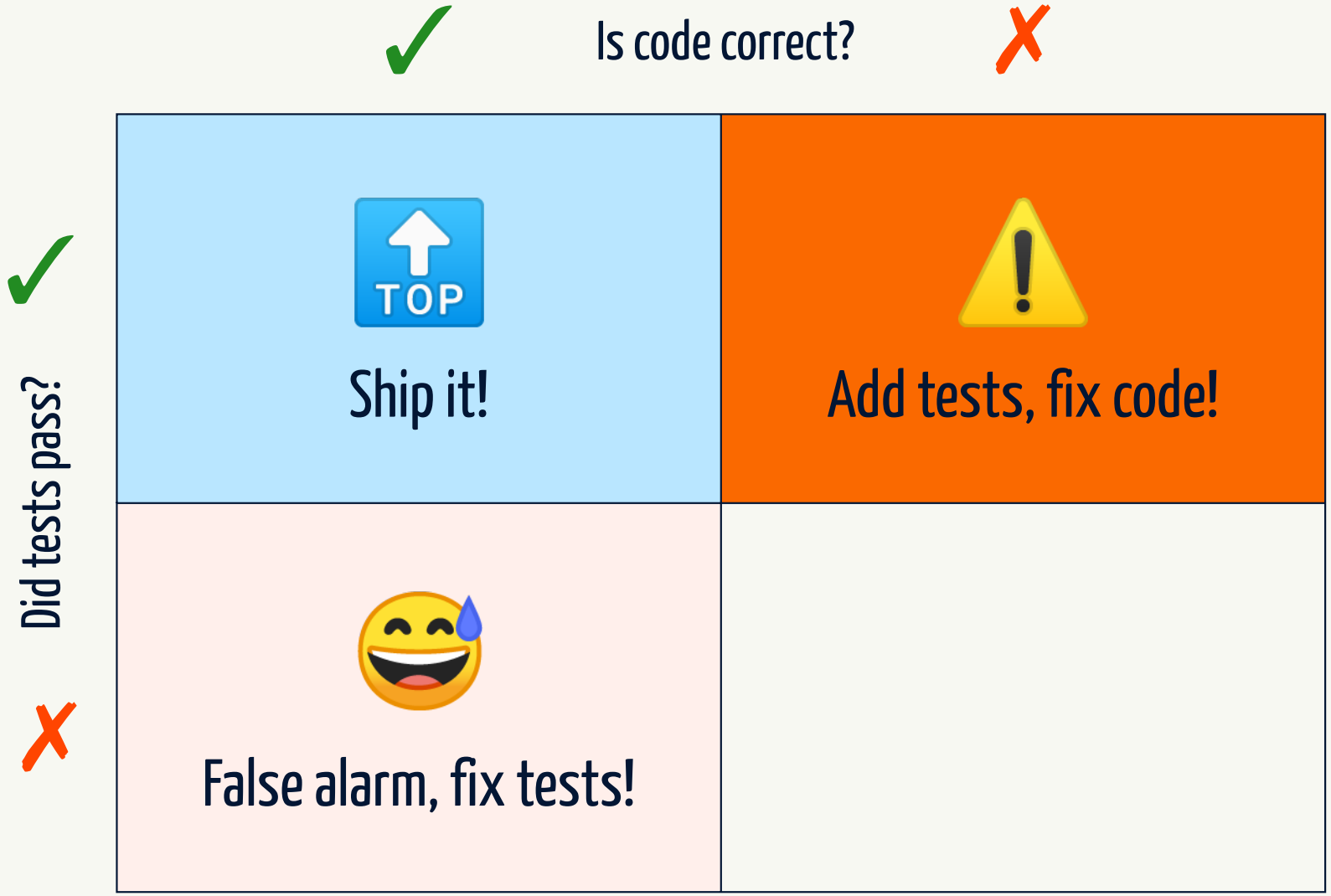
Tests accuracy means how much tests outcome matches code correctness.

Tests accuracy and reacting to failures



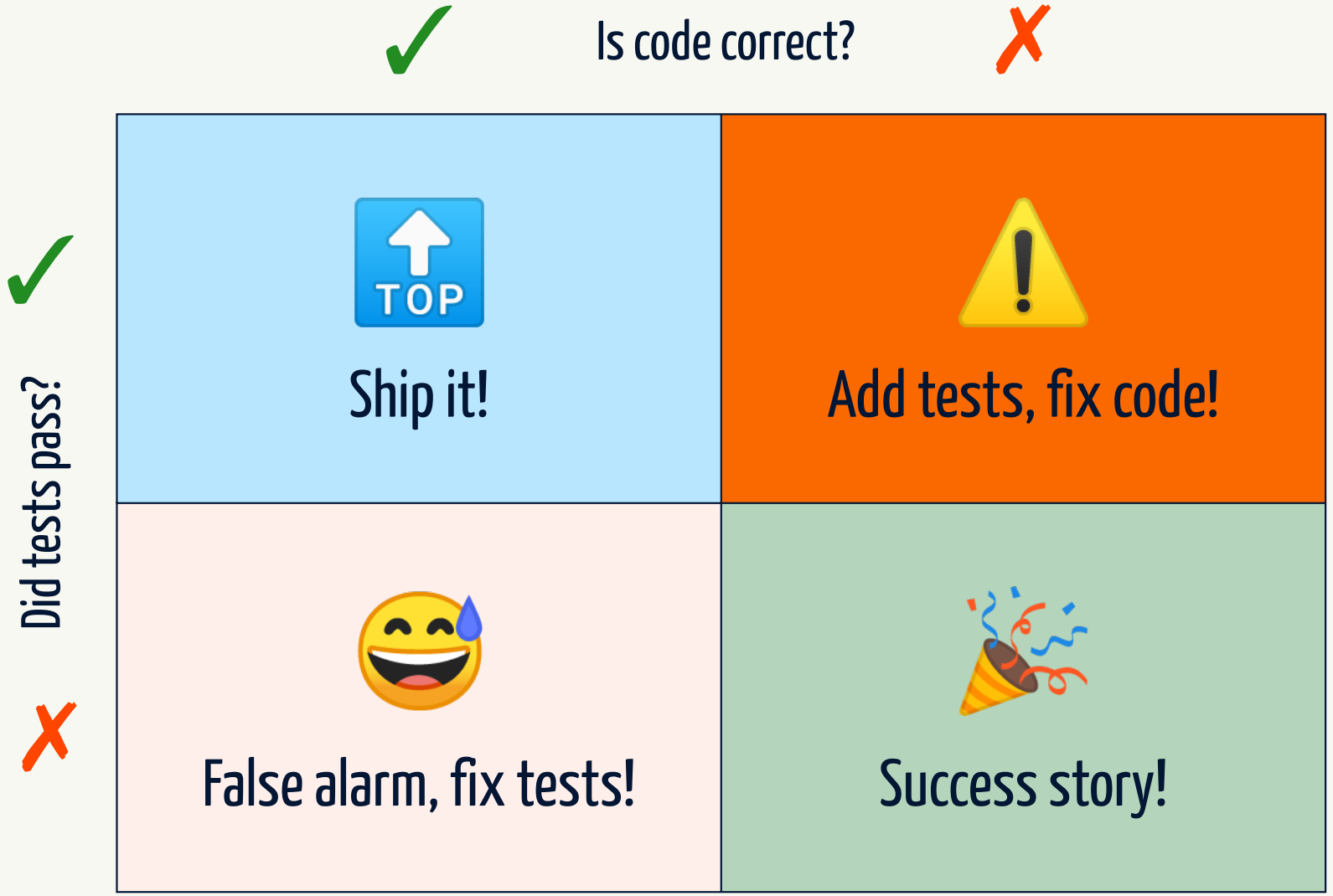
Tests accuracy means how much tests outcome matches code correctness.

Tests accuracy and reacting to failures



Tests accuracy means how much tests outcome matches code correctness.

Tests accuracy and reacting to failures



Tests accuracy means how much tests outcome matches code correctness.

The curse of the **unit** word

What is a **unit** in unit testing?

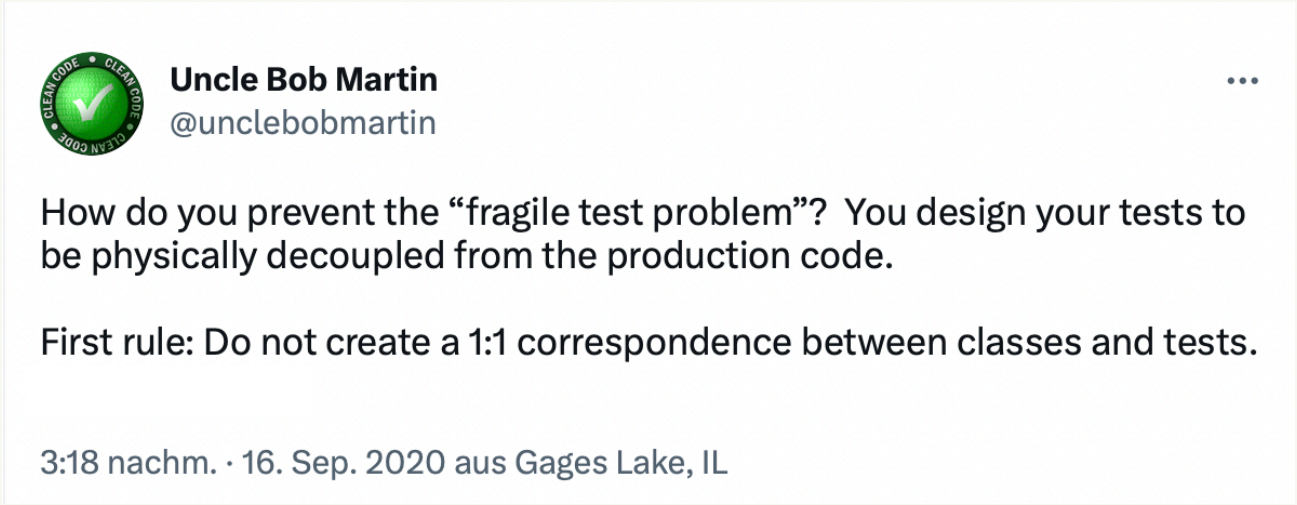
> A group of related functions often called a module.
>
> — Wikipedia

> A unit test should test a requirement, only test against the stable contract of the API.
>
> — Ian Cooper

A unit is not necessarily a class or a function

This is possibly one of the most misunderstood concepts in software engineering in the last decade. Sometimes a unit is a class or a function, but there should be a good reason for such a correspondence. **More often a unit should be (the public API of) a module.**

What is a **unit** in unit testing?



Uncle Bob Martin
@unclebobmartin

How do you prevent the “fragile test problem”? You design your tests to be physically decoupled from the production code.

First rule: Do not create a 1:1 correspondence between classes and tests.

3:18 nachm. · 16. Sep. 2020 aus Gages Lake, IL

A unit is not necessarily a class or a function

This is possibly one of the most misunderstood concepts in software engineering in the last decade. Sometimes a unit is a class or a function, but there should be a good reason for such a correspondence. **More often a unit should be (the public API of) a module.**

Wait, what's wrong
in testing each single class?

It depends

> **TDD, Where Did It All Go Wrong**
>
> A couple of times we were very proud of the fact that we produced some code that went to production with almost zero defects. [...] I looked at this test suite. One things I noticed was that we had about three times as much test code as production code.
>
> Ian Cooper

- Some classes are simply implementation details
  - ↳ You would like to freely change them
- Imagine a module with few classes working together { in your implementation, today }
  - ↳ Does it make sense to mock all but one several times?
- If a class happens to be the API of a module, probably that's a unit

White- and black-box testing!?

I'll test a single class next in examples,
but you now know what a **unit** is!

White box testing

- You allow yourself in using all code in your tests
  - ↳ even private methods and members
  - ↳ implementation details are used to test interface

White box testing

- You allow yourself in using all code in your tests
- In some languages (e.g. Python) easier than in others (e.g. C++)

Consider this class

```
// File Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Drink();
    bool Fill();
  private:
    bool isEmpty_ = true;
};
```

Test file

```
#include "vir/test.h"
#include "Cup.h"


TEST(Cup::Cup) {
    Cup cup{};
    VERIFY(cup.isEmpty_);
}
TEST(Cup::IsEmpty) { /*...*/ }
TEST(Cup::Fill)    { /*...*/ }
TEST(Cup::Drink)   { /*...*/ }
```

This does not compile!

I took this example from Dave Steffen's talk at CppCon2020.

White box testing

- You allow yourself in using all code in your tests
- In some languages (e.g. Python) easier than in others (e.g. C++)

Consider this class

```
// File Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Drink();
    bool Fill();
  private:
    bool isEmpty_ = true;
};
```

Test file

```
#include "vir/test.h"
#include "Cup.h"
#define private public // UB!!!

TEST(Cup::Cup) {
    Cup cup{};
    VERIFY(cup.isEmpty_);
}
TEST(Cup::IsEmpty) { /*...*/ }
TEST(Cup::Fill)    { /*...*/ }
TEST(Cup::Drink)   { /*...*/ }
```

This relies on undefined behaviour!!!

I took this example from Dave Steffen's talk at CppCon2020.

White box testing

- You allow yourself in using all code in your tests
- In some languages (e.g. Python) easier than in others (e.g. C++)

Consider this class

```
// File Cup.h

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

Test file

```
#include "vir/test.h"
#include "Cup.h"
struct CupTester { /*...*/ };

TEST(Cup::Cup) {
    Cup cup{};
    CupTester tester{cup};
    VERIFY(tester.isEmpty_);
}
TEST(Cup::IsEmpty) { /*...*/ }
TEST(Cup::Fill)    { /*...*/ }
TEST(Cup::Drink)   { /*...*/ }
```


I took this example from Dave Steffen's talk at CppCon2020.

White box testing

- You allow yourself in using all code in your tests
- In some languages (e.g. Python) easier than in others (e.g. C++)
- Something to consider to start working on legacy code!

White box testing

- You allow yourself in using all code in your tests
- In some languages (e.g. Python) easier than in others (e.g. C++)
- Something to consider to start working on legacy code!
- What happens when you change your implementation?
  - ↳ Compilation might break
  - ↳ If not, some tests will probably fail
  - ↳ You likely need to change your tests code
  - ↳ For large refactoring, it might be boring/frustrating

Black-box testing

- What about just testing the public interface?

Consider this class

```
// File Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Drink();
    bool Fill();
  private:
    bool isEmpty_ = true;
};
```

Source code same as before

Test file

```
#include "vir/test.h"
#include "Cup.h"

TEST(Cup::Cup)      { /*...*/ }
TEST(Cup::IsEmpty) { /*...*/ }
TEST(Cup::Fill)     { /*...*/ }
TEST(Cup::Drink)    { /*...*/ }
```

Black-box testing

- What about just testing the public interface?

Consider this class

```
// File Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Drink();
    bool Fill();
  private:
    bool isEmpty_ = true;
};
```

Source code same as before

Test file

```
#include "vir/test.h"
#include "Cup.h"

TEST(Cup::Cup) {
    Cup cup{};
    VERIFY(cup.IsEmpty());
}
TEST(Cup::IsEmpty) { /*...*/ }
TEST(Cup::Fill)    { /*...*/ }
TEST(Cup::Drink)   { /*...*/ }
```

Black-box testing

- What about just testing the public interface?

Consider this class

```
// File Cup.h

class Cup {
  public:
    Cup();
    bool IsEmpty();
    bool Drink();
    bool Fill();
  private:
    bool isEmpty_ = true;
};
```

Source code same as before

Test file

```
#include "vir/test.h"
#include "Cup.h"

TEST(Cup::Cup) {
    Cup cup{};
    VERIFY(cup.IsEmpty());
}
TEST(Cup::IsEmpty){
    Cup cup{};
    VERIFY(cup.IsEmpty());
}
TEST(Cup::Fill)    { /*...*/ }
TEST(Cup::Drink)   { /*...*/ }
```

Wait, we wrote the same code twice!

Black-box testing

- What about just testing the public interface?

> **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.
>
> Dave Steffen

Black-box testing

- What about just testing the public interface?

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.

Dave Steffen

So, what to do then?

1. Ignore the problem { Be aware: Tests are not independent, one bug triggers many failures! }
2. Declare one member function "correct by inspection" { and start from there }
3. Abandon black-box testing as impractical

Test the behaviour, not the implementation

Do not test methods! Test behaviour!

Back to our cup example:

```
#include "vir/test.h"
#include "Cup.h"

TEST(A new cup is empty)              { /*...*/ }
TEST(An empty cup can be filled)      { /*...*/ }
TEST(A filled cup is full)            { /*...*/ }
TEST(Drinking empties a filled cup)   { /*...*/ }
```

- Test names describe the unit specifications
- Suddenly, we are completely independent from implementation details
- OK, so how does test code look like?

Do not test methods! Test behaviour!

Back to our cup example:

```
#include "vir/test.h"
#include "Cup.h"

TEST(A new cup is empty){
    Cup cup{};
    VERIFY(cup.IsEmpty());
}
TEST(An empty cup can be filled)     { /*...*/ }
TEST(A filled cup is full)           { /*...*/ }
TEST(Drinking empties a filled cup) { /*...*/ }
```

Do not test methods! Test behaviour!

Back to our cup example:

```
#include "vir/test.h"
#include "Cup.h"

TEST(A new cup is empty){
    Cup cup{};
    VERIFY(cup.IsEmpty());
}
TEST(An empty cup can be filled)    { /*...*/ }
TEST(A filled cup is full)          { /*...*/ }
TEST(Drinking empties a filled cup) { /*...*/ }
```

- Wait!! This is exactly the same code that wasn't OK few minutes ago!?
  - ↳ How is it that changing the name makes it fine?
  - ↳ How should this solve the black-box conundrum?

Do not test methods! Test behaviour!

The bottom line

We are now testing consistency of interface, not correctness of implementation!

Do not test methods! Test behaviour!

The bottom line

We are now testing consistency of interface, not correctness of implementation!

Let's say it once more:

We are now testing consistency of interface, not correctness of implementation!

Do not test methods! Test behaviour!

The bottom line

We are now testing consistency of interface, not correctness of implementation!

Let's say it once more:

We are now testing consistency of interface, not correctness of implementation!

It is not a philosophical statement!

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

Is this code correct or wrong?

```
// File Cup.h

class Cup {
  public:
    Cup()           { isEmpty_ = false; }
    bool IsEmpty()  { return !isEmpty_; }
    bool Drink()    { /*...*/ }
    bool Fill()     { /*...*/ }
  private:
    bool isEmpty_ = false;
};
```

As testing behaviour passes, the code is correct!

Is this code correct or wrong?

```
// File Cup.h

class Cup {
  public:
    Cup()           { isEmpty_ = false; }
    bool IsEmpty()  { return !isEmpty_; }
    bool Drink()    { /*...*/ }
    bool Fill()     { /*...*/ }
  private:
    bool isEmpty_ = false;
};
```

```
#include "vir/test.h"
#include "Cup.h"
TEST(A new cup is empty)              { /*...*/ }
TEST(An empty cup can be filled)      { /*...*/ }
TEST(A filled cup is full)            { /*...*/ }
TEST(Drinking empties a filled cup)   { /*...*/ }
```

Test driven development as discipline

It is a discipline

It is a discipline

- Do not bring it immediately into your daily work
- Learn it in small exercise projects:
  - Implement a `Stack` object
  - Implement a file parser
  - Write a 1D integration tool with few different algorithms
  - Simulate the 2D Ising model with a Monte Carlo algorithm
  - Implement a simple game (e.g. hangman, minesweeper)
  - [...]
- Convince yourself about it
- Get a feeling and choose "your way"

Finally

Get rid of the "there-is-no-time-to-write-tests" attitude!

The purist original approach

Test driven development main rules

Robert C. Martin

The purist original approach

Test driven development main rules

1. You're not allowed to write any production code until you have first written a test that fails because the production code doesn't exist.

Robert C. Martin

The purist original approach

Test driven development main rules

1. You're not allowed to write any production code until you have first written a test that fails because the production code doesn't exist.
2. You're not allowed to write more of a test than is sufficient to fail. And not compiling is failing.

Robert C. Martin

The purist original approach

Test driven development main rules

1. You're not allowed to write any production code until you have first written a test that fails because the production code doesn't exist.
2. You're not allowed to write more of a test than is sufficient to fail. And not compiling is failing.
3. You're not allowed to write any more production code than is sufficient to pass the currently failing test.

Robert C. Martin

The purist original approach

Test driven development main rules

1. You're not allowed to write any production code until you have first written a test that fails because the production code doesn't exist.
2. You're not allowed to write more of a test than is sufficient to fail. And not compiling is failing.
3. You're not allowed to write any more production code than is sufficient to pass the currently failing test.

Robert C. Martin

How would it be a world in which everything always worked one minute ago?

Writing the factorial in TDD

Source file

```
// Empty file
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

 TEST(Nothing){}
```

It passes

Writing the factorial in TDD

Source file

```
// Empty file
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
```

It doesn't compile

Writing the factorial in TDD

Source file

```
void factorial(int n){}
```

Notice the function signature!

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
```

It fails

Writing the factorial in TDD

Source file

```
void factorial(int n)
{
  throw std::logical_error();
}
```

Still no need to return anything!

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
```

It passes

Writing the factorial in TDD

Source file

```
void factorial(int n)
{
  throw std::logical_error();
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
}
```

It fails

Writing the factorial in TDD

Source file

```
int factorial(int n)
{
  if(n<0)
    throw std::logical_error();
  else
    return 1;
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
}
```

It passes

Writing the factorial in TDD

Source file

```
int factorial(int n)
{
  if(n<0)
    throw std::logical_error();
  else
    return 1;
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
  COMPARE(120, factorial(5));
}
```

It fails

Writing the factorial in TDD

Source file

```
int factorial(int n)
{
  if(n<0)
    throw std::logical_error();
  else if(n==1)
    return 1;
  else
    return n * factorial(n-1);
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
  COMPARE(120, factorial(5));
}
```

It passes

Writing the factorial in TDD

Source file

```
int factorial(int n)
{
  if(n<0)
    throw std::logical_error();
  else if(n==1)
    return 1;
  else
    return n * factorial(n-1);
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
  COMPARE(120, factorial(5));
  COMPARE(1, factorial(0));
}
```

It fails

Writing the factorial in TDD

Source file

```
int factorial(int n)
{
  if(n<0)
    throw std::logical_error();
  else if(n<2)
    return 1;
  else
    return n * factorial(n-1);
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
  COMPARE(120, factorial(5));
  COMPARE(1, factorial(0));
}
```

It passes

Writing the factorial in TDD

Source file

```
int factorial(int n)
{
  if(n<0)
    throw std::logical_error();
  else if(n<2)
    return 1;
  else
    return n * factorial(n-1);
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
  factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
  COMPARE(120, factorial(5));
  COMPARE(1, factorial(0));
  COMPARE(479001600,
          factorial(12));
}
```

It still passes

Writing the factorial in TDD

Source file

```
int factorial(int n)
{
  if(n<0)
    throw std::logical_error();
  else if(n<2)
    return 1;
  else
    return n * factorial(n-1);
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"
#include <limits>

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
    factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
  COMPARE(120, factorial(5));
  COMPARE(1, factorial(0));
  COMPARE(479001600,
          factorial(12));
  COMPARE(INT_MAX,
          factorial(13));
}
```

It fails

In the code `INT_MAX` has been used for space reasons. The C++11 limit `std::numeric_limits<int>::max()` should be preferred, though.

Writing the factorial in TDD

Source file

```
#include <limits>

int factorial(int n)
{
  if(n<0)
    throw std::logical_error();
  else if(n<2)
    return 1;
  else if(n>=13)
    return INT_MAX;
  else
    return n * factorial(n-1);
}
```

Test file

```
#include <vir/test.h>
#include "factorial.h"

TEST_CATCH(Negative_factorial,
           std::logical_error)
{
    factorial(-17);
}
TEST(FactorialTest) {
  COMPARE(1, factorial(1));
  COMPARE(120, factorial(5));
  COMPARE(1, factorial(0));
  COMPARE(479001600,
          factorial(12));
  COMPARE(INT_MAX,
          factorial(13));
}
```

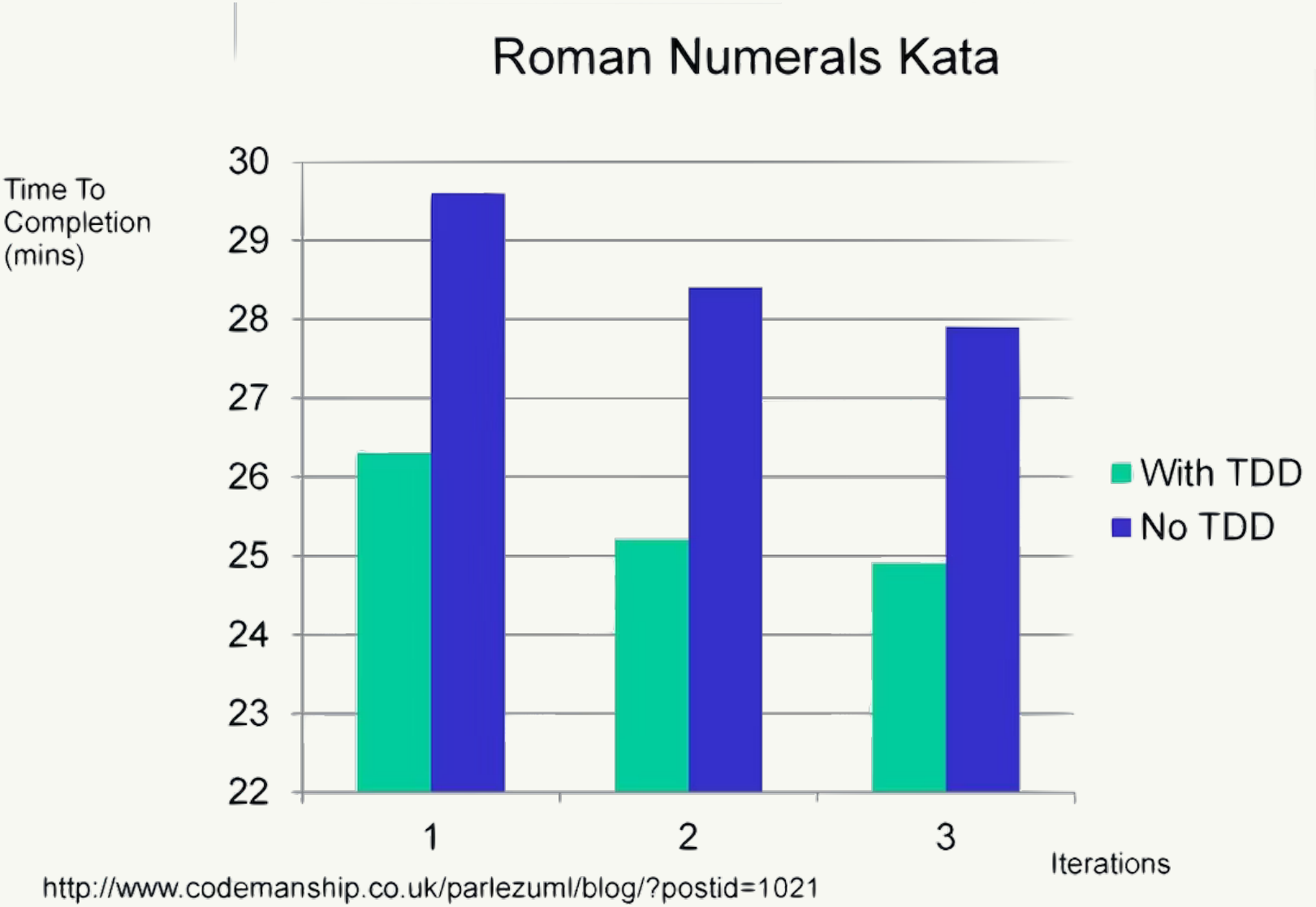
It passes

In the code `INT_MAX` has been used for space reasons. The C++11 limit `std::numeric_limits<int>::max()` should be preferred, though.

How strict should you be with yourself?

The more edits you do at once, the more dangerous it is!

Disciplines feel slow, they are not slow



http://www.codemanship.co.uk/parlezuml/blog/?postid=1021

The Red – Green – Refactor approach

What Kent Beck proposed

Red: Write a little test that doesn't work, and perhaps doesn't even compile at first.

Green: Make the test work quickly, committing whatever sins necessary in the process.

Refactor: Eliminate all of the duplication created in merely getting the test to work.

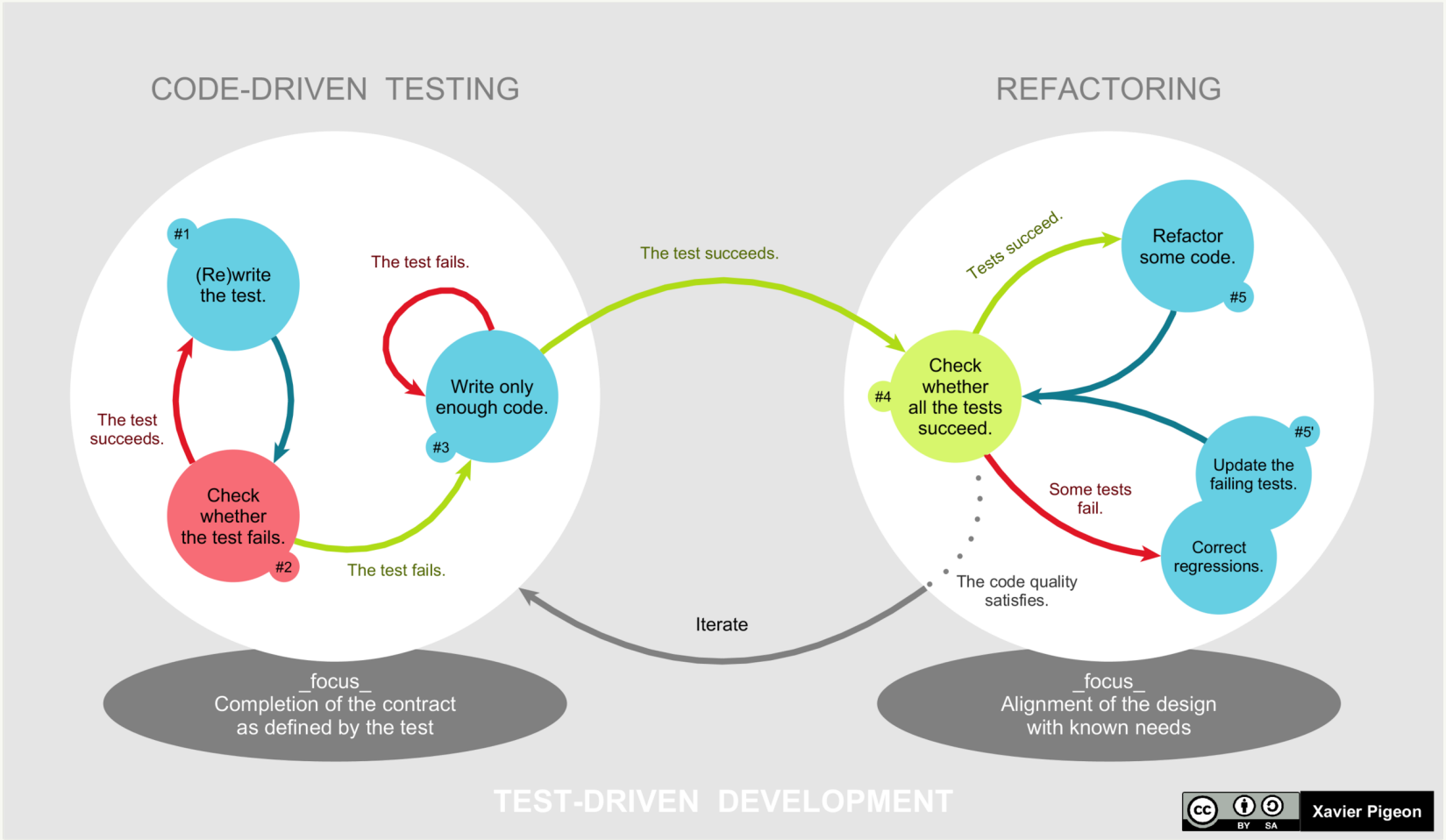
Ian Cooper

About testing the implementation

Sometimes it is useful to probe the implementation details because they are difficult to understand. You can do that, but **delete the tests afterwards!**

Ian Cooper

The Red – Green – Refactor approach



Behaviour driven development in its original idea

- Dan North's original idea stemmed from TDD
  - Test method names should be sentences
  - An expressive test name is helpful when a test fails
  - Determine the next most important behaviour
  - Requirements are behaviour, too
  - BDD provides a "ubiquitous language" for analysis

Behaviour driven development in its original idea

- Dan North's original idea stemmed from TDD
- Use cases become stories and each test implements an acceptance criterion for the story

Example of use case as a story

Title: Customer withdraws cash

As a customer,
I want to withdraw cash from an ATM,
so that I don't have to wait in line at the bank.

Behaviour driven development in its original idea

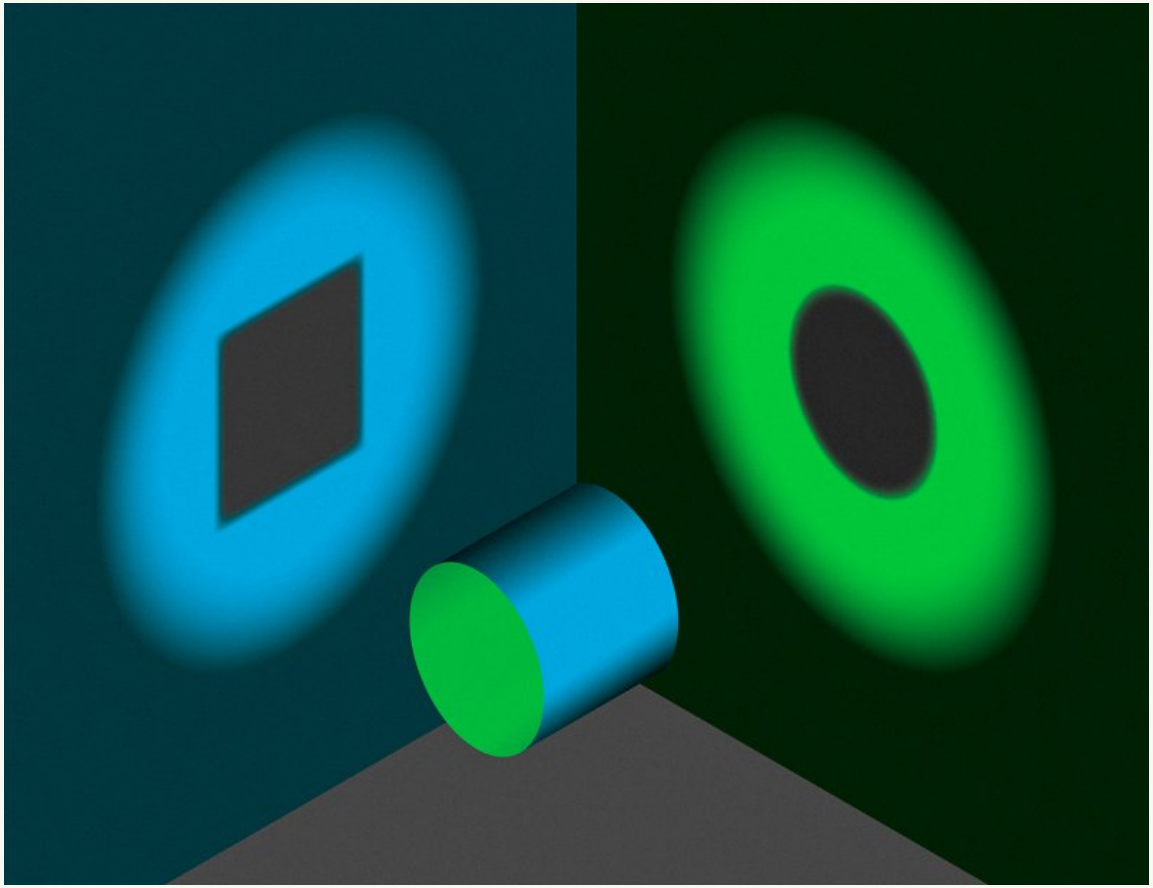
- Dan North's original idea stemmed from TDD
- Use cases become stories and each test implements an acceptance criterion for the story

Example of story acceptance criterion

Scenario: Account is overdrawn past the overdraft limit

Given the account is overdrawn
And the card is valid
When the customer requests cash
Then ensure a rejection message is displayed
And ensure cash is not dispensed
And ensure the card is returned

TDD and BDD



After all they are disciplines to deal with the same idea!

Take-home message

Wrapping up the advice

1. «I do not have time to write tests» **does not make any sense!**

Wrapping up the advice

1. «I do not have time to write tests» **does not make any sense!**
2. Writing tests is an inseparable part of the code development

Wrapping up the advice

1. «I do not have time to write tests» **does not make any sense!**
2. Writing tests is an inseparable part of the code development
3. Tests are the only way to change code still ensuring its correctness

Wrapping up the advice

1. «I do not have time to write tests» **does not make any sense!**
2. Writing tests is an inseparable part of the code development
3. Tests are the only way to change code still ensuring its correctness
4. Test the behaviour, not the implementation

Wrapping up the advice

1. «I do not have time to write tests» **does not make any sense!**
2. Writing tests is an inseparable part of the code development
3. Tests are the only way to change code still ensuring its correctness
4. Test the behaviour, not the implementation
5. Depending on the size of the project, you can go one or another way

Wrapping up the advice

1. «I do not have time to write tests» **does not make any sense!**
2. Writing tests is an inseparable part of the code development
3. Tests are the only way to change code still ensuring its correctness
4. Test the behaviour, not the implementation
5. Depending on the size of the project, you can go one or another way
6. If you do not want to strictly do TDD, still envision your tests and API usage before implementing any code

Wrapping up the advice

1. «I do not have time to write tests» **does not make any sense!**
2. Writing tests is an inseparable part of the code development
3. Tests are the only way to change code still ensuring its correctness
4. Test the behaviour, not the implementation
5. Depending on the size of the project, you can go one or another way
6. If you do not want to strictly do TDD, still envision your tests and API usage before implementing any code
7. Now you are aware of how writing code should be done!

Wrapping up the advice

1. «I do not have time to write tests» **does not make any sense!**
2. Writing tests is an inseparable part of the code development
3. Tests are the only way to change code still ensuring its correctness
4. Test the behaviour, not the implementation
5. Depending on the size of the project, you can go one or another way
6. If you do not want to strictly do TDD, still envision your tests and API usage before implementing any code
7. Now you are aware of how writing code should be done!
8. Testing (and hence writing) code is doing science

And now?

If this is not your reaction right now, it will be soon!

And now?

Thanks for listening!

Questions? Feedback?