Test Driven Development:

Pair programming to the max

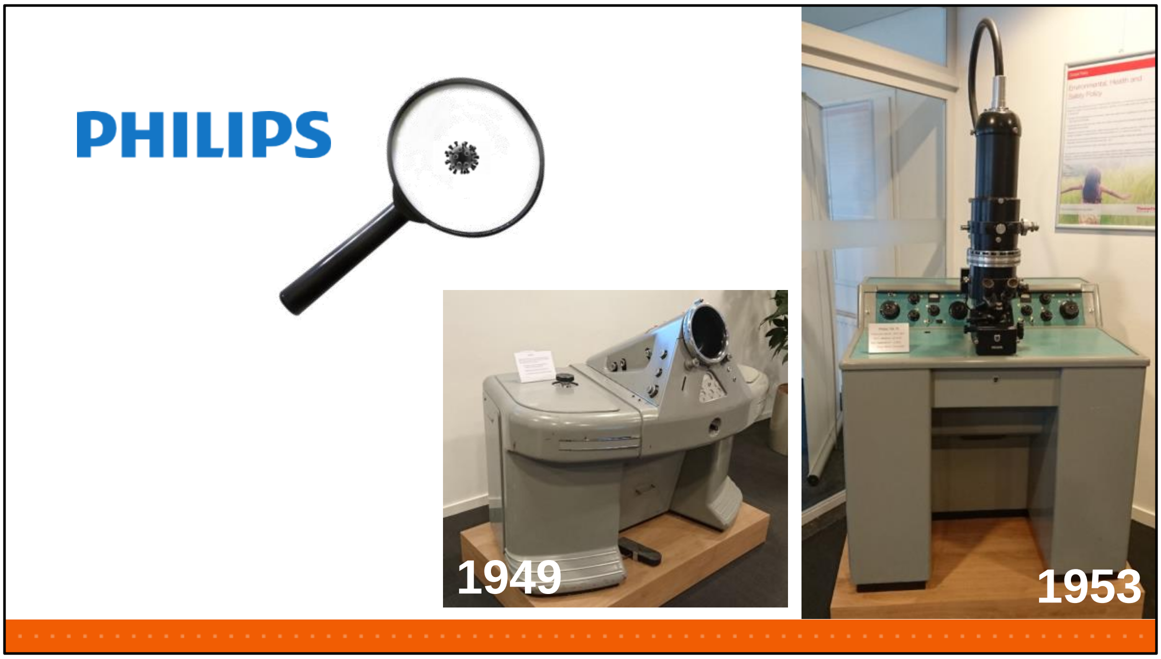
SIOUX

SOURCE OF YOUR TECHNOLOGY

Klaas van Gend, 040coders.nl, March 15, 2018

Klaas van Gend

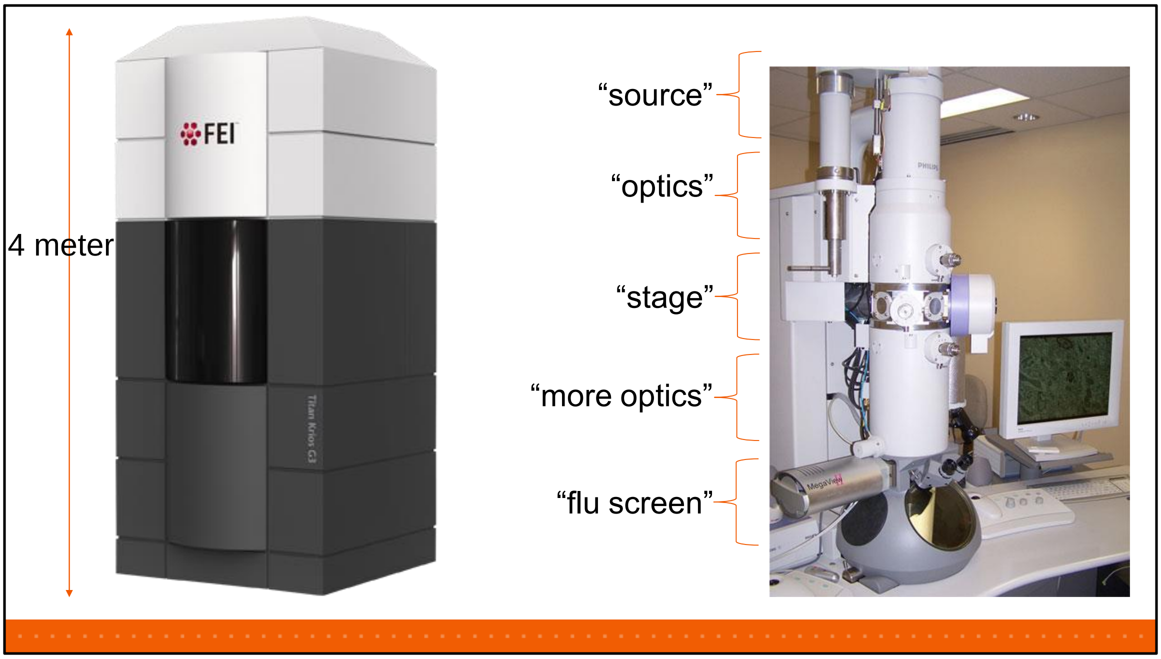
- Hobby:
- Hobby: **{040coders.nl}**
- Hobby:



PHILIPS

1949

1953



4 meter

“source”

“optics”

“stage”

“more optics”

“flu screen”

FEI acquired Philips Electron Optics and kept building bigger and better electron microscopes.

Now part of

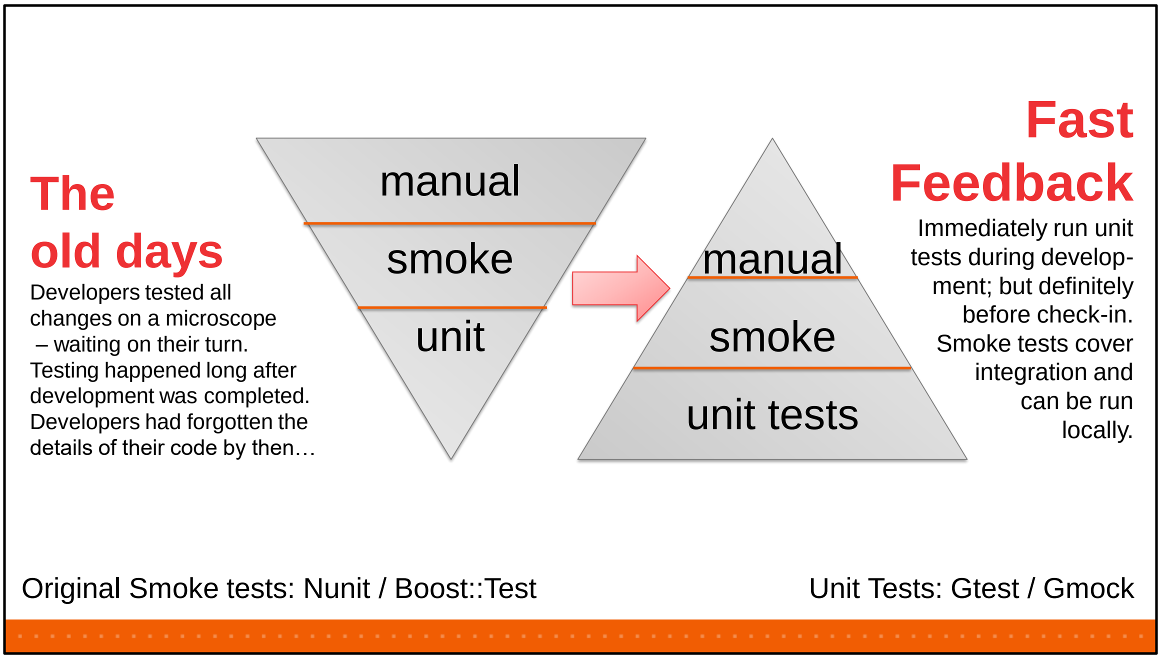
ThermoFisher SCIENTIFIC

Nobel Prize Chemistry 2017 awarded to 3 developments surrounding the Titan Krios

Thermo Fisher acquired FEI in 2016.
Last year, they were indirectly awarded for their efforts by a Nobel prize for Chemistry, awarded to three researchers that stood at the base of the Titan Krios and Cryo-Electron Microscopy – which is a huge breakthrough in “life science”, able to make detailed pictures of e.g. large complex proteins.

On the PC runs a lot of applications, including ‘Acquisition Server’, approx. 400kloc, C++11, written using Visual Studio 2013. In total in Eindhoven around 90 SW developers, on AcqSvr 16. Klaas is Scrum Master and Senior developer of the “Scanning Team”. The other team is the “Camera Team”. Other groups build the software for optics, sample management, vacuum, GUI, etc.

Then, some components were end-of-life and an important part of the rack needed a redesign. Unfortunately, Acquisition Server was very closely tied to the old hardware. We’re now almost at the end of a 3-man 2 year project to unmarry the software, abstract the hardware and make Acquisition Server hardware independent – it must be able to control both the old and the new hardware.
All normal software deliveries had to continue. Essentially, we had to remodel the shop while it was open – without interfering with the customers.
To start with that large redesign, we needed to be sure that we wouldn’t break anything. So we started by looking at our tests.



The old days

Developers tested all changes on a microscope – waiting on their turn. Testing happened long after development was completed. Developers had forgotten the details of their code by then…

Fast Feedback

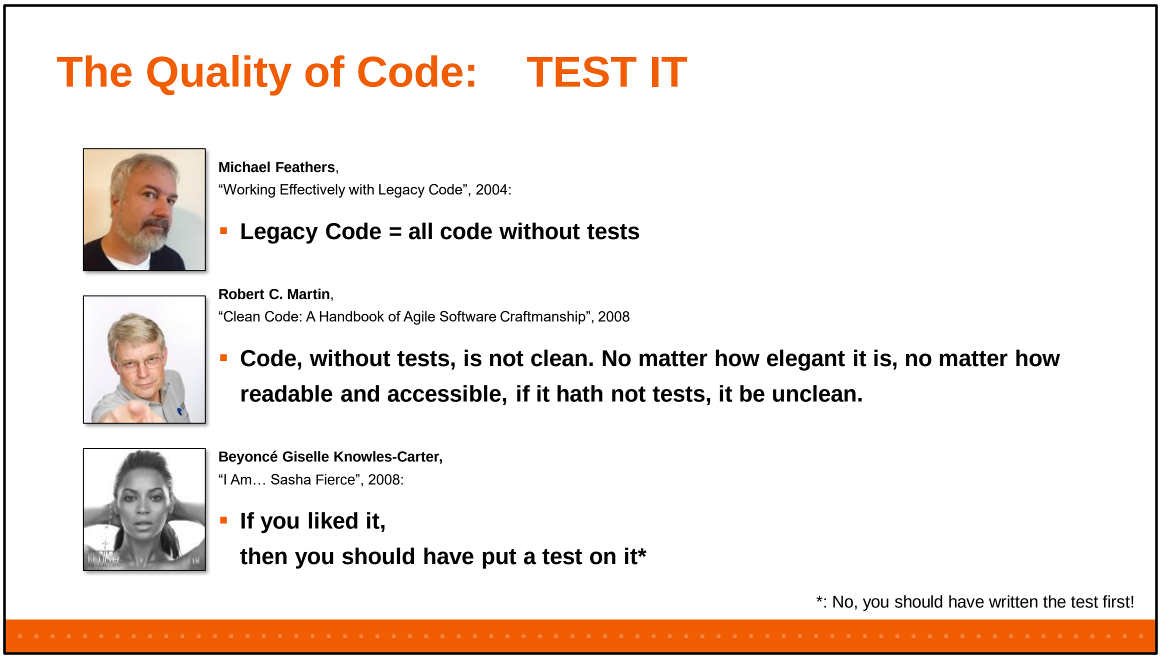
Immediately run unit tests during development; but definitely before check-in. Smoke tests cover integration and can be run locally.

Original Smoke tests: Nunit / Boost::Test

Unit Tests: Gtest / Gmock

Why not stick with Boost::Test? Mocking sounded very alluring…

The problem of only having integration tests?
There’s no decomposition of the test code, whereas there is decomposition of the code.
This means that all tests touch roughly the same code – breaking a single piece breaks many tests or nothing at all.
One breaking unit test shows very clearly where your issue is.



The Quality of Code: TEST IT

Michael Feathers,
“Working Effectively with Legacy Code”, 2004:

- **Legacy Code = all code without tests**

Robert C. Martin,
“Clean Code: A Handbook of Agile Software Craftmanship”, 2008

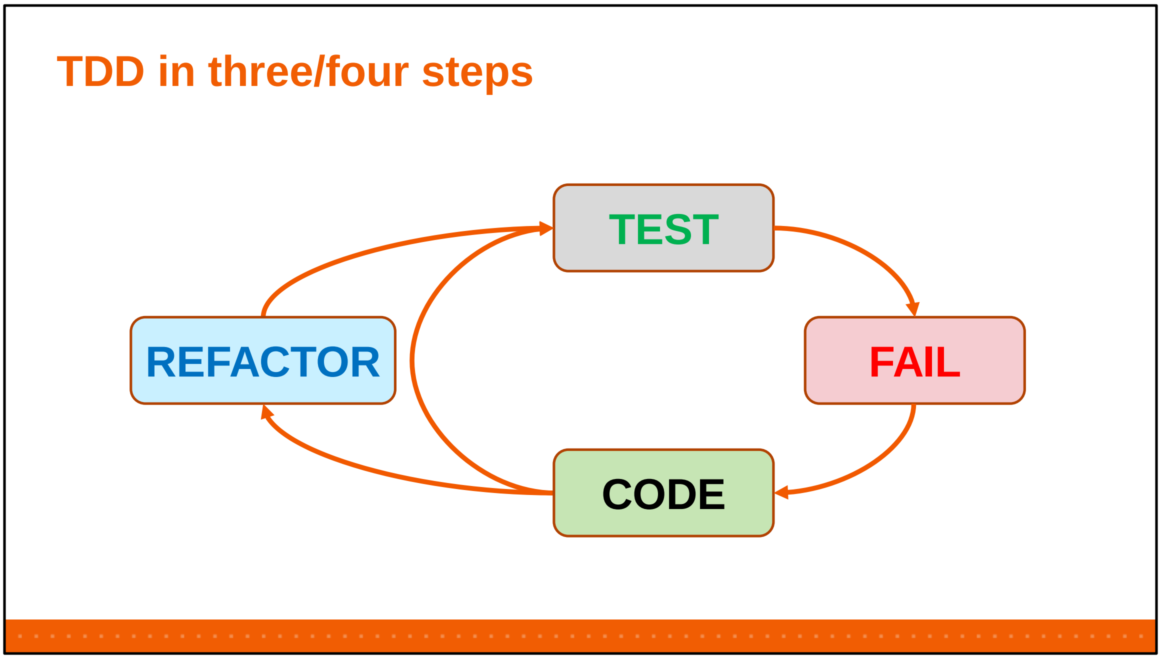
- **Code, without tests, is not clean. No matter how elegant it is, no matter how readable and accessible, if it hath not tests, it be unclean.**

Beyoncé Giselle Knowles-Carter,
“I Am… Sasha Fierce”, 2008:

- **If you liked it,**
  then you should have put a test on it*

*: No, you should have written the test first!

We’ll talk more about TDD and legacy code later.



TDD in three/four steps

TEST

FAIL

CODE

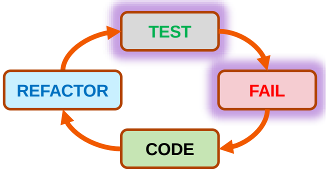
REFACTOR

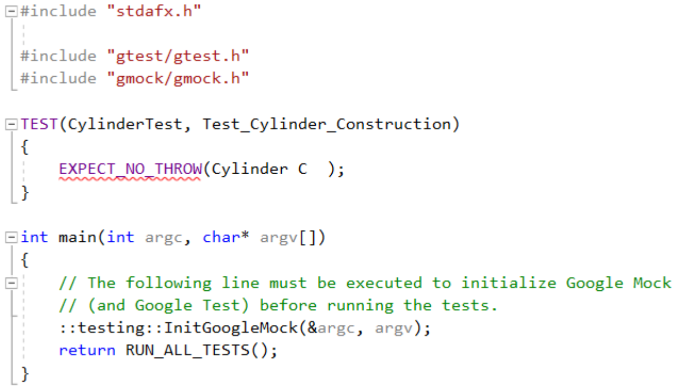
Write a TEST -> make it FAIL -> add just enough CODE -> write a new TEST that FAILS -> add just enough more CODE -> REFACTOR and run again.

Your first test…

- Remember:
  - ***First*** write the test
    - The test must **FAIL**

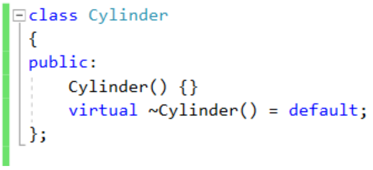
      (in this case: compile fail)
  - ***Then*** the implementation



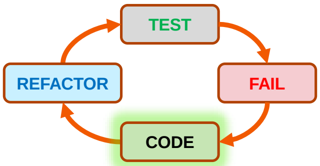


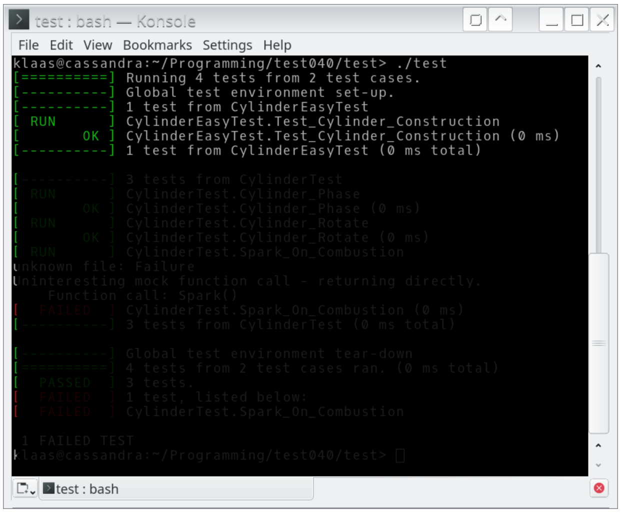
```
#include "stdafx.h"

#include "gtest/gtest.h"
#include "gmock/gmock.h"

TEST(CylinderTest, Test_Cylinder_Construction)
{
    EXPECT_NO_THROW(Cylinder C  );
}

int main(int argc, char* argv[])
{
    // The following line must be executed to initialize Google Mock
    // (and Google Test) before running the tests.
    ::testing::InitGoogleMock(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Add code, *just until the test passes…*

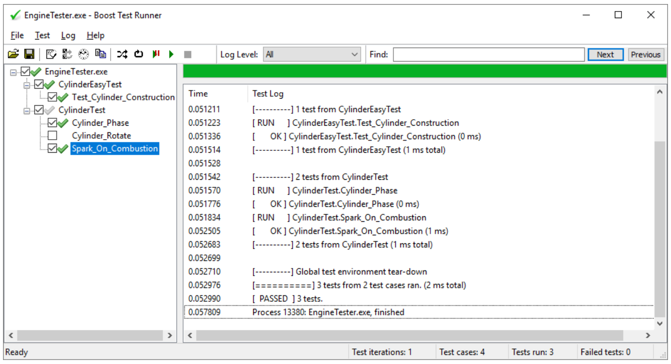


```
class Cylinder
{
public:
    Cylinder() {}
    virtual ~Cylinder() = default;
};
```





```
klaas@cassandra:~/Programming/test040/test> ./test
[==========] Running 4 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 1 test from CylinderEasyTest
[ RUN      ] CylinderEasyTest.Test_Cylinder_Construction
[       OK ] CylinderEasyTest.Test_Cylinder_Construction (0 ms)
[----------] 1 test from CylinderEasyTest (0 ms total)

[----------] 3 tests from CylinderTest
[ RUN      ] CylinderTest.Cylinder_Phase
[       OK ] CylinderTest.Cylinder_Phase (0 ms)
[ RUN      ] CylinderTest.Cylinder_Rotate
[       OK ] CylinderTest.Cylinder_Rotate (0 ms)
[ RUN      ] CylinderTest.Spark_On_Combustion
unknown file: Failure
Uninteresting mock function call - returning directly.
    Function call: Spark()
[  FAILED  ] CylinderTest.Spark_On_Combustion (0 ms)
[----------] 3 tests from CylinderTest (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 2 test cases ran. (0 ms total)
[  PASSED  ] 3 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] CylinderTest.Spark_On_Combustion

 1 FAILED TEST
klaas@cassandra:~/Programming/test040/test>
```

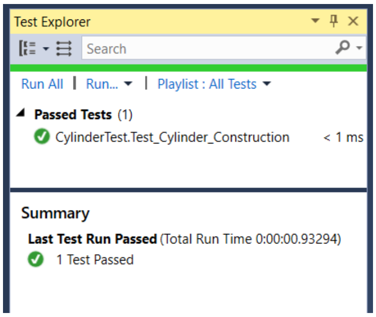


- ☑ ✓ EngineTester.exe
  - ☑ ✓ CylinderEasyTest
    - ☑ ✓ Test_Cylinder_Construction
  - ☑ CylinderTest
    - ☑ ✓ Cylinder_Phase
    - □ Cylinder_Rotate
    - ☑ ✓ Spark_On_Combustion

```
Time        Test Log
0.051211    [----------] 1 test from CylinderEasyTest
0.051223    [ RUN      ] CylinderEasyTest.Test_Cylinder_Construction
0.051336    [       OK ] CylinderEasyTest.Test_Cylinder_Construction (0 ms)
0.051514    [----------] 1 test from CylinderEasyTest (1 ms total)
0.051528
0.051542    [----------] 2 tests from CylinderTest
0.051570    [ RUN      ] CylinderTest.Cylinder_Phase
0.051776    [       OK ] CylinderTest.Cylinder_Phase (0 ms)
0.051834    [ RUN      ] CylinderTest.Spark_On_Combustion
0.052505    [       OK ] CylinderTest.Spark_On_Combustion (1 ms)
0.052683    [----------] 2 tests from CylinderTest (1 ms total)
0.052699
0.052710    [----------] Global test environment tear-down
0.052976    [==========] 3 tests from 2 test cases ran. (2 ms total)
0.052990    [  PASSED  ] 3 tests.
0.057809    Process 13380: EngineTester.exe, finished
```

Ready | Test iterations: 1 | Test cases: 4 | Tests run: 3 | Failed tests: 0

https://github.com/djeedjay/BoostTestUi



Passed Tests (1)

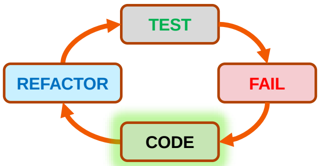
✓ CylinderTest.Test_Cylinder_Construction < 1 ms

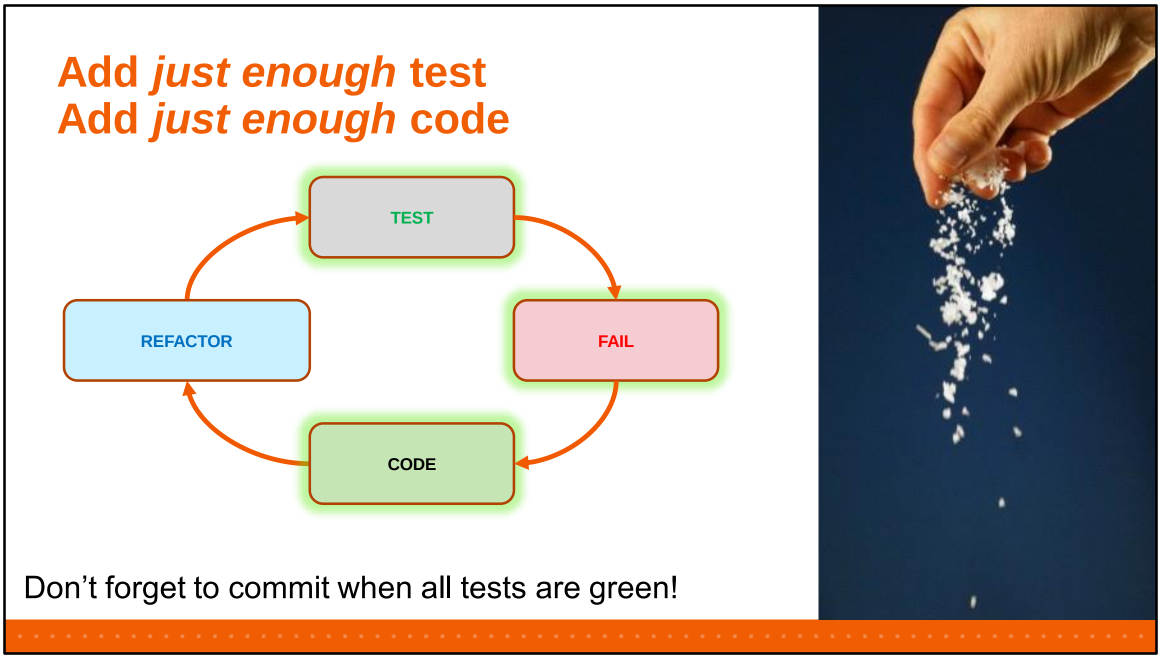
Summary

Last Test Run Passed (Total Run Time 0:00:00.93294)

✓ 1 Test Passed

Visual Studio 2013+





Add *just enough* test
Add *just enough* code

TEST

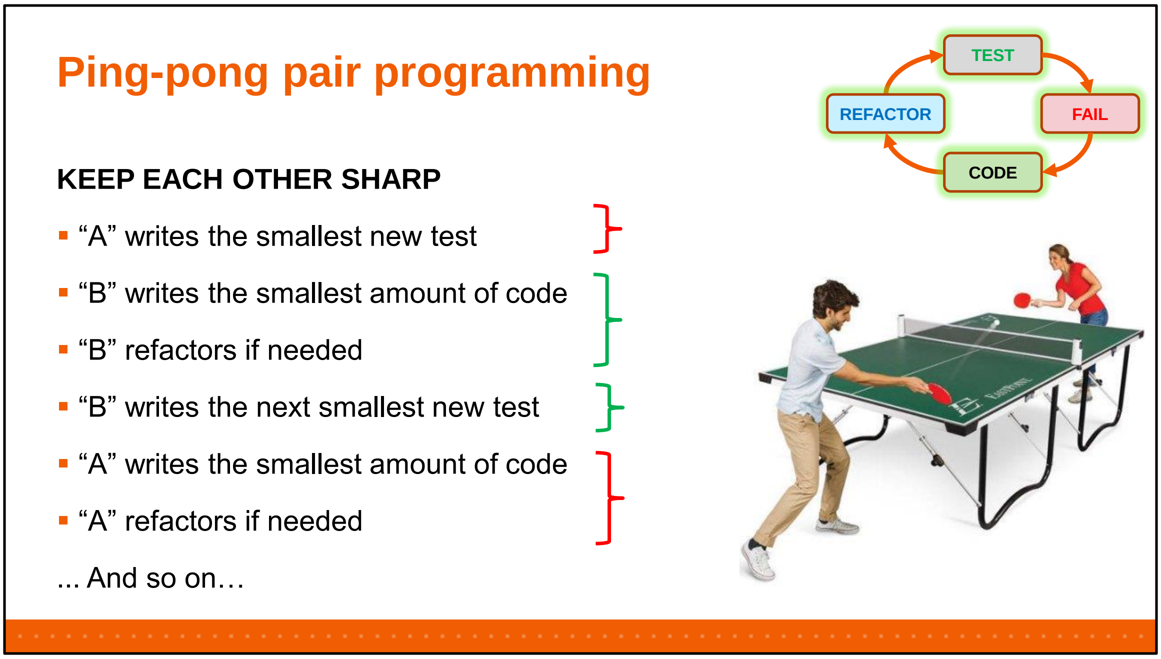
FAIL

CODE

REFACTOR

Don’t forget to commit when all tests are green!

TDD changed our work flow: we now push much more but smaller changesets to our repositories.



Ping-pong pair programming

TEST
FAIL
CODE
REFACTOR

KEEP EACH OTHER SHARP

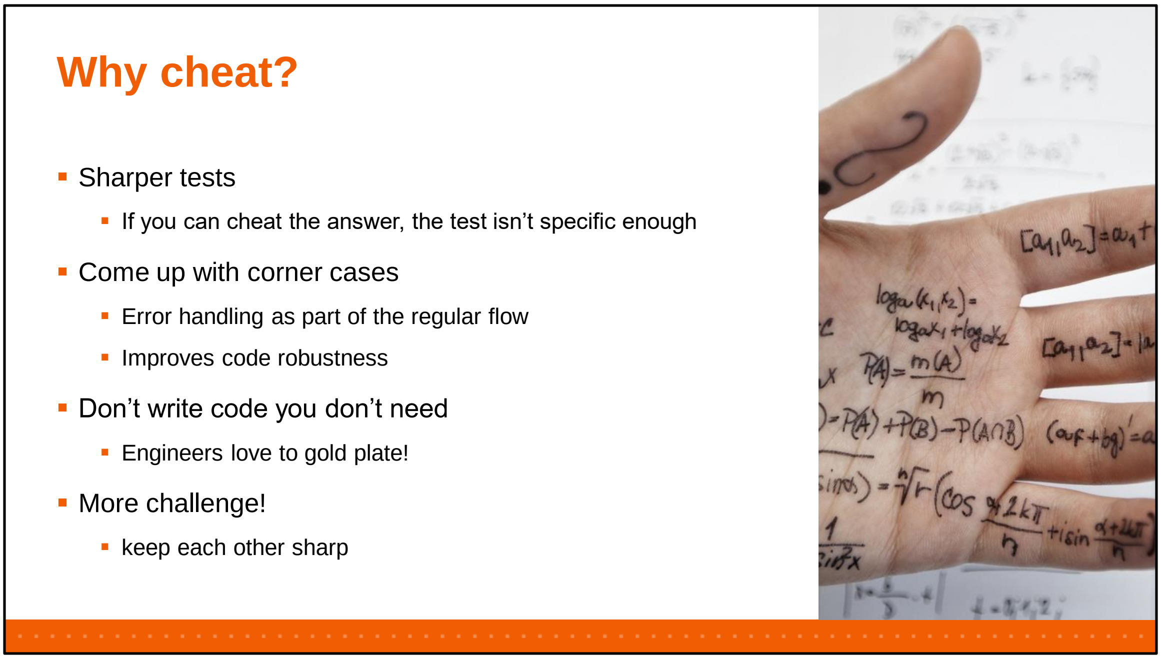
- “A” writes the smallest new test
- “B” writes the smallest amount of code
- “B” refactors if needed
- “B” writes the next smallest new test
- “A” writes the smallest amount of code
- “A” refactors if needed

... And so on…

One of the benefits of ping-pong pair programming is that you have two engineers at the peak of their abilities – up to 6 hours per day.
Are you really productive for 6 hours? Research has shown that most engineers only do “real work” for about 2 hours a day.

Preframe: but… what makes ping-pong really shine?
I’m going to tell you why our best developers switched on, next!

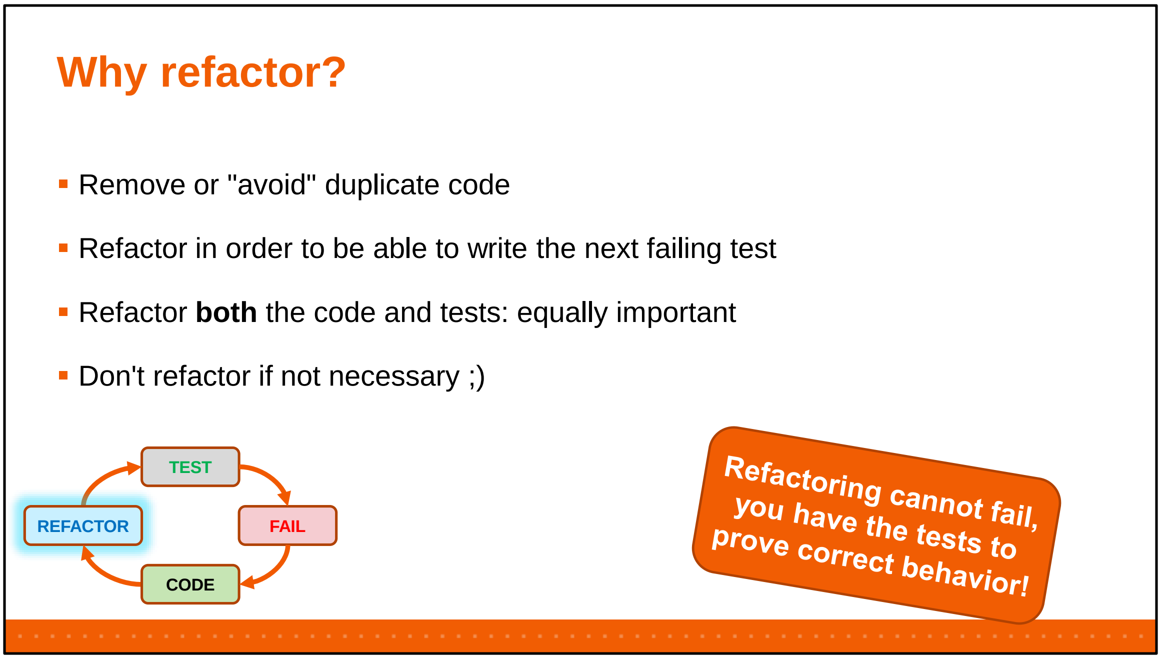
CHEAT – while writing tests and while writing code !!!



Why cheat?

- Sharper tests
  - If you can cheat the answer, the test isn’t specific enough
- Come up with corner cases
  - Error handling as part of the regular flow
  - Improves code robustness
- Don’t write code you don’t need
  - Engineers love to gold plate!
- More challenge!
  - keep each other sharp

One of my dear co-workers also likes to cheat the reverse way – he sometimes just removes code while all tests keep functioning. That’s of course very bad.



Why refactor?

- Remove or "avoid" duplicate code
- Refactor in order to be able to write the next failing test
- Refactor **both** the code and tests: equally important
- Don't refactor if not necessary ;)

TEST
FAIL
CODE
REFACTOR

Refactoring cannot fail, you have the tests to prove correct behavior!

Refactoring is VERY important and cannot go wrong.
After all, you’re doing small steps, right?
And you have a set of good tests – so if you mess up, it will show.

Why keep the cycle short?

- Forces to write only a few lines
- No Need for Debugging
  - You only added a few lines, right?
- Committing + Delivering/Pushing often
  - Helps keeps merges simple

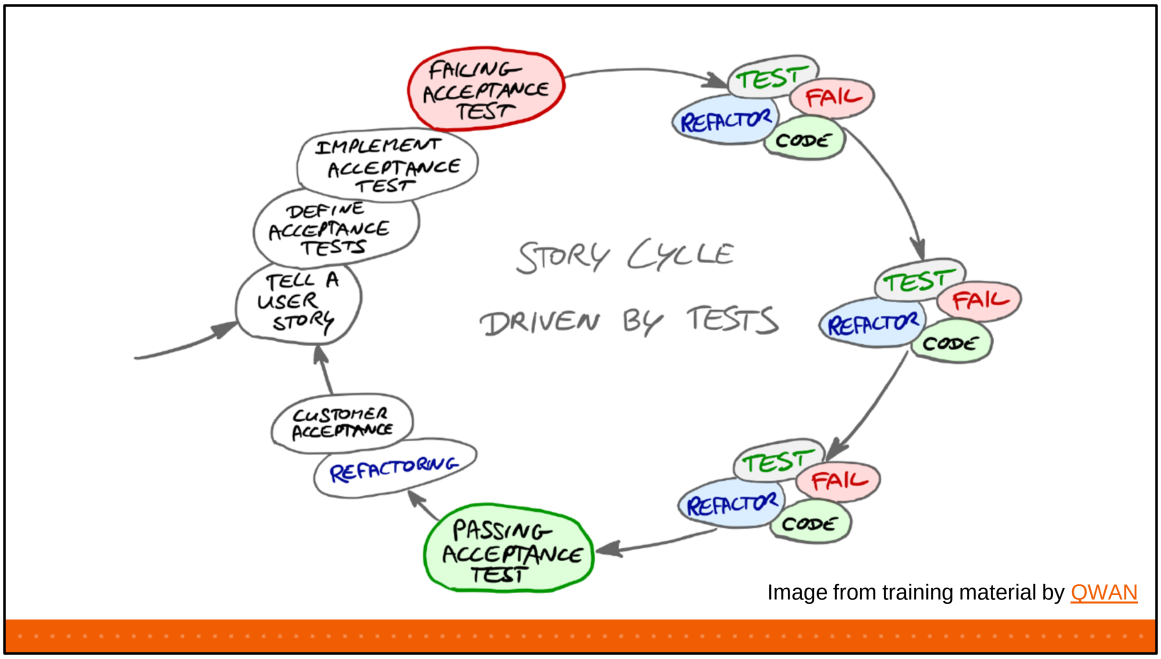
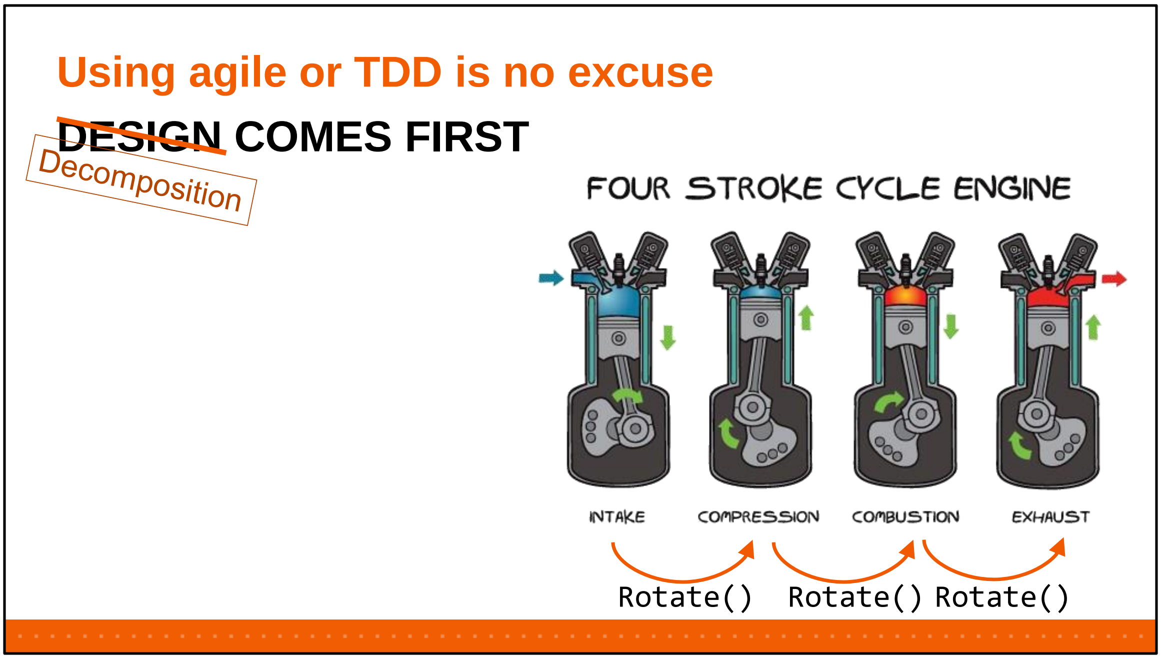


Image from training material by QWAN

TDD also works very well in the larger picture.

It grows…



Using agile or TDD is no excuse

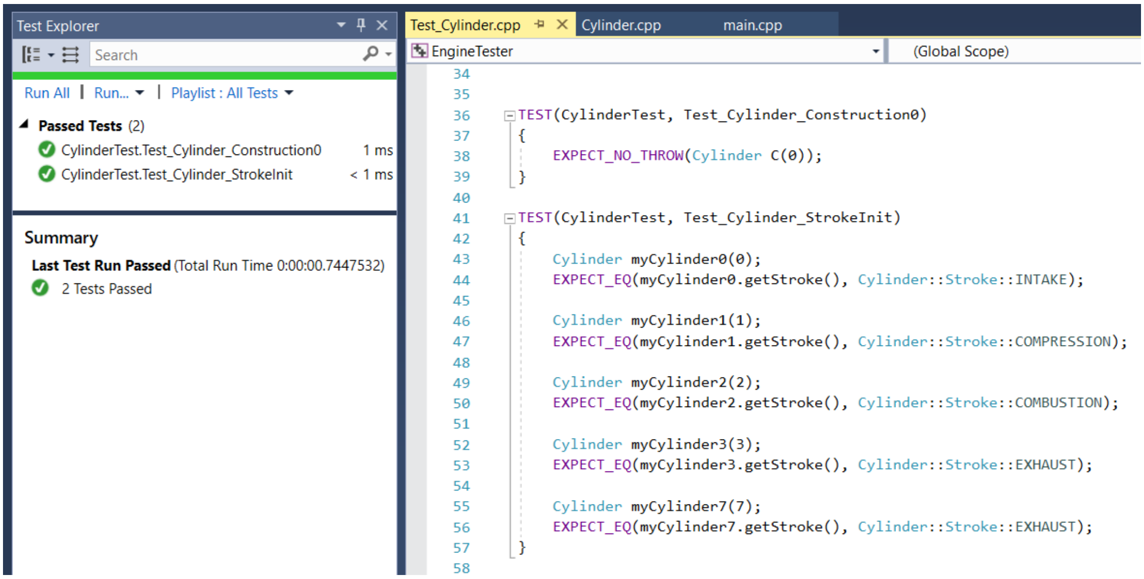
~~DESIGN~~ COMES FIRST

Decomposition

Rotate() Rotate() Rotate()

Obviously, start with a design (or at least a decomposition) in mind (or on paper).

In our experience, we often wind up somewhere different – better.
Usually more (but smaller) classes with better defined responsibilities.



Passed Tests (2)

- ✓ CylinderTest.Test_Cylinder_Construction0 1 ms
- ✓ CylinderTest.Test_Cylinder_StrokeInit < 1 ms

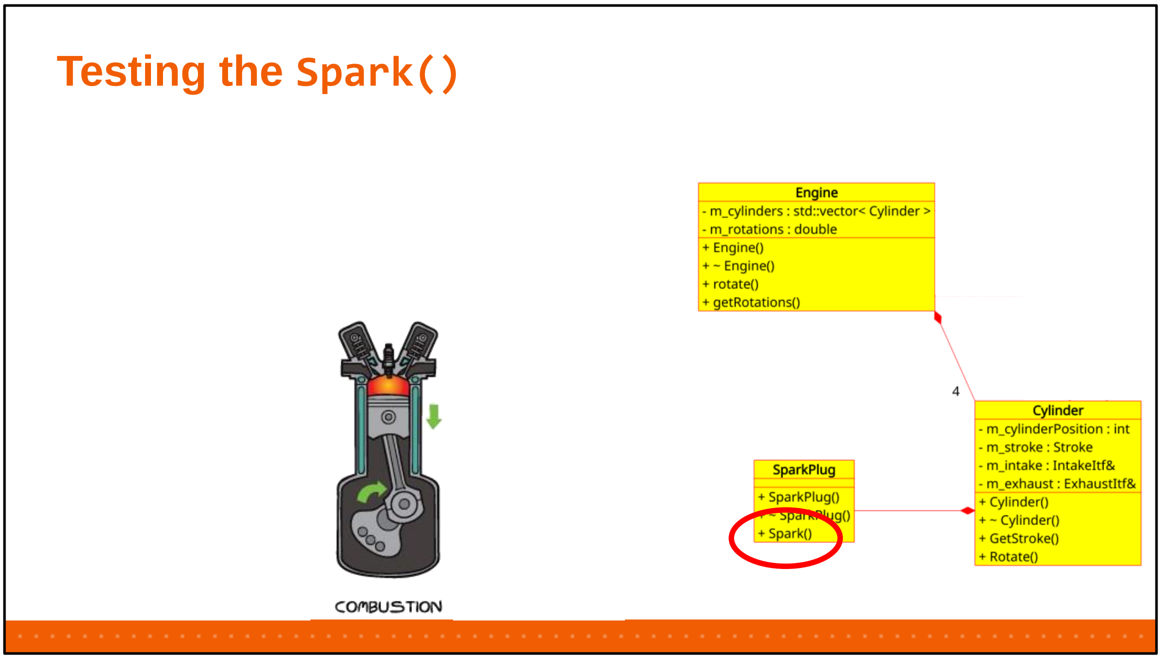
Summary

Last Test Run Passed (Total Run Time 0:00:00.7447532)

✓ 2 Tests Passed

```cpp


TEST(CylinderTest, Test_Cylinder_Construction0)
{
    EXPECT_NO_THROW(Cylinder C(0));
}

TEST(CylinderTest, Test_Cylinder_StrokeInit)
{
    Cylinder myCylinder0(0);
    EXPECT_EQ(myCylinder0.getStroke(), Cylinder::Stroke::INTAKE);

    Cylinder myCylinder1(1);
    EXPECT_EQ(myCylinder1.getStroke(), Cylinder::Stroke::COMPRESSION);

    Cylinder myCylinder2(2);
    EXPECT_EQ(myCylinder2.getStroke(), Cylinder::Stroke::COMBUSTION);

    Cylinder myCylinder3(3);
    EXPECT_EQ(myCylinder3.getStroke(), Cylinder::Stroke::EXHAUST);

    Cylinder myCylinder7(7);
    EXPECT_EQ(myCylinder7.getStroke(), Cylinder::Stroke::EXHAUST);
}

```

Yup, there’s code duplication in this test.



Testing the `Spark()`

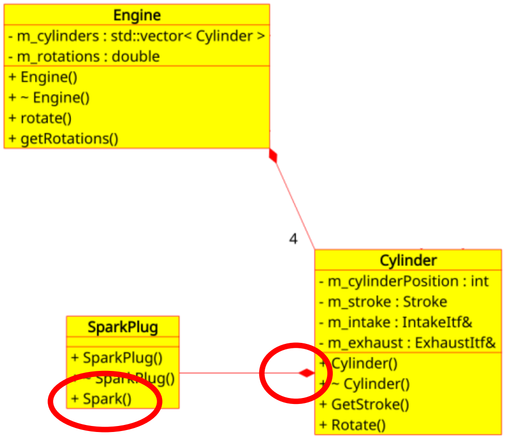
Engine
- m_cylinders : std::vector< Cylinder >
- m_rotations : double
+ Engine()
+ ~ Engine()
+ rotate()
+ getRotations()

4

Cylinder
- m_cylinderPosition : int
- m_stroke : Stroke
- m_intake : IntakeItf&
- m_exhaust : ExhaustItf&
+ Cylinder()
+ ~ Cylinder()
+ GetStroke()
+ Rotate()

SparkPlug
+ SparkPlug()
+ ~ SparkPlug()
+ Spark()

Testing the `Spark()`



Engine

- m_cylinders : std::vector< Cylinder >
- m_rotations : double

+ Engine()
+ ~ Engine()
+ rotate()
+ getRotations()

4

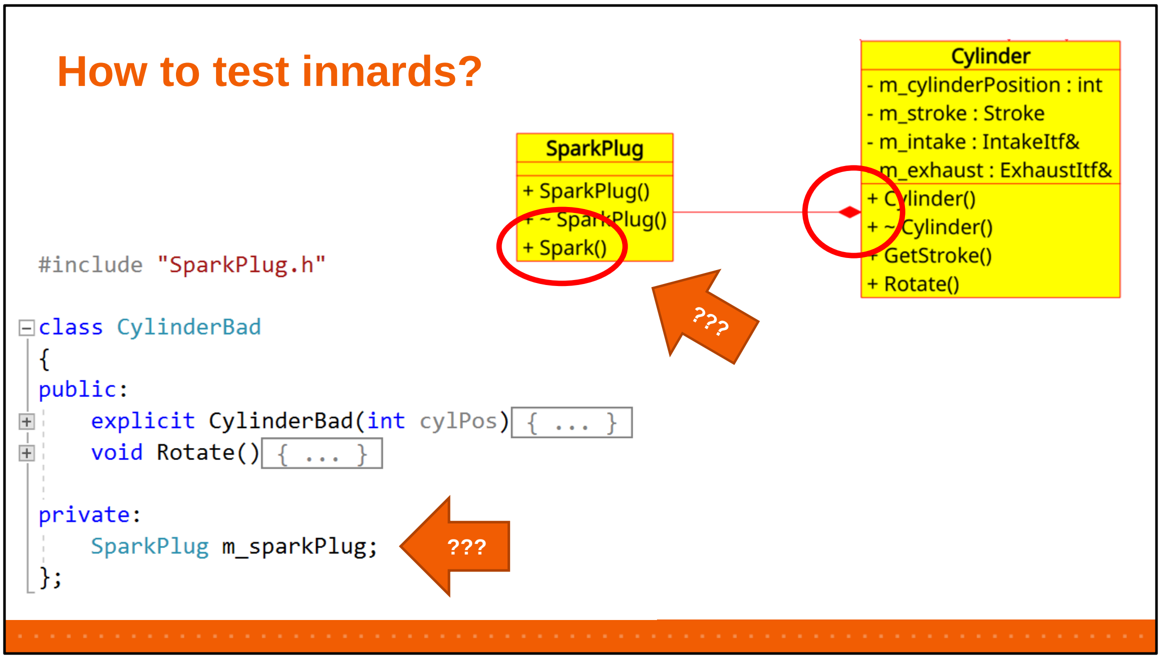
Cylinder

- m_cylinderPosition : int
- m_stroke : Stroke
- m_intake : IntakeItf&
- m_exhaust : ExhaustItf&

+ Cylinder()
+ ~ Cylinder()
+ GetStroke()
+ Rotate()

SparkPlug

+ SparkPlug()
+ ~ SparkPlug()
+ Spark()



How to test innards?

SparkPlug

+ SparkPlug()
+ ~SparkPlug()
+ Spark()

Cylinder

- m_cylinderPosition : int
- m_stroke : Stroke
- m_intake : IntakeItf&
- m_exhaust : ExhaustItf&

+ Cylinder()
+ ~Cylinder()
+ GetStroke()
+ Rotate()

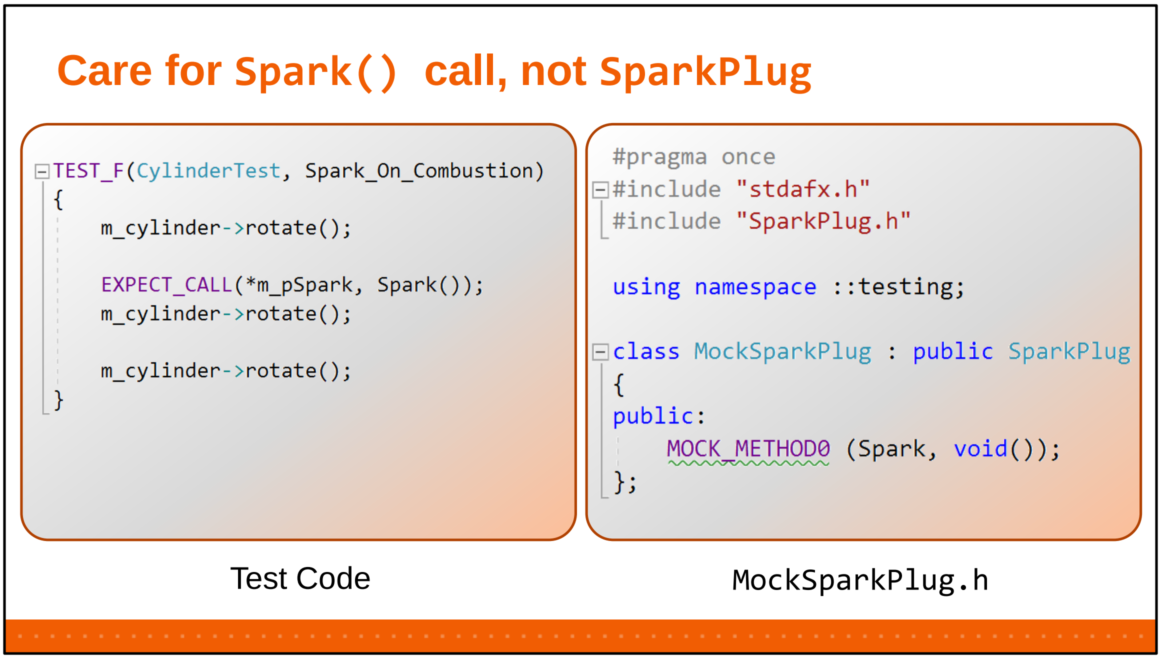
???

```
#include "SparkPlug.h"

class CylinderBad
{
public:
    explicit CylinderBad(int cylPos) { ... }
    void Rotate() { ... }

private:
    SparkPlug m_sparkPlug;
};
```

???

So, how do we get to test that the sparkplug was sparked?

The next part is key to TDD: a change in how your construct your objects and test them.



Care for Spark() call, not SparkPlug

```
TEST_F(CylinderTest, Spark_On_Combustion)
{
    m_cylinder->rotate();

    EXPECT_CALL(*m_pSpark, Spark());
    m_cylinder->rotate();

    m_cylinder->rotate();
}
```

Test Code

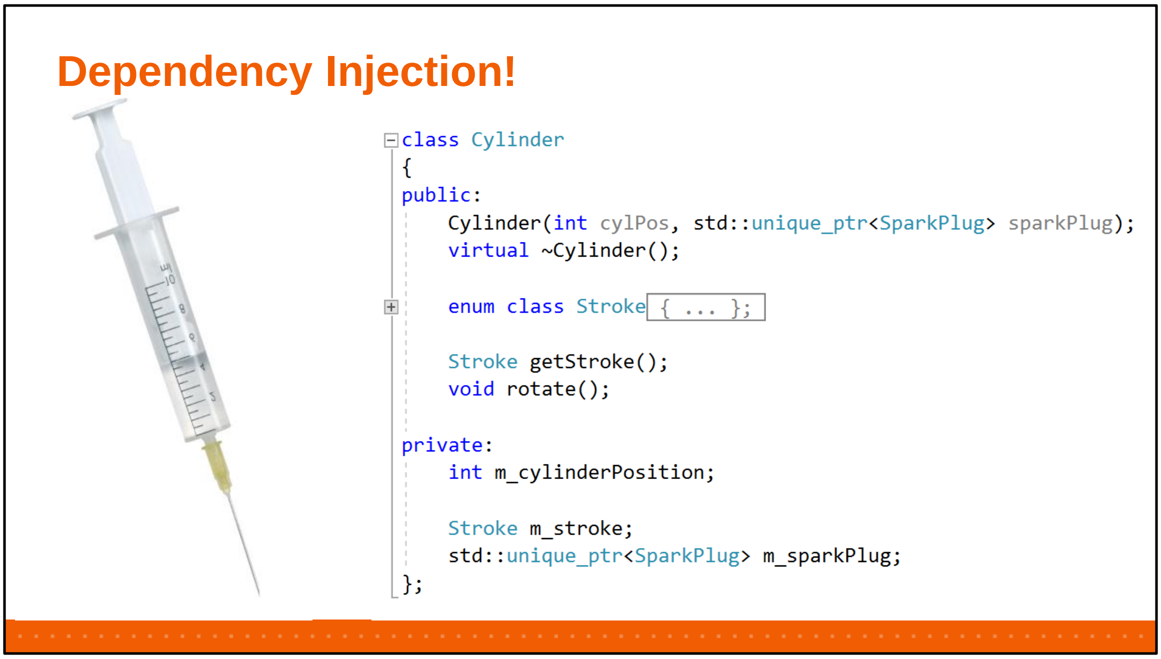
```
#pragma once
#include "stdafx.h"
#include "SparkPlug.h"

using namespace ::testing;

class MockSparkPlug : public SparkPlug
{
public:
    MOCK_METHOD0 (Spark, void());
};
```

MockSparkPlug.h

For now, let’s keep the test understandable – it’s “not the best test ever”.

First rotate turns from INTAKE to COMPRESSION
Second rotate turns from COMPRESSION to COMBUSTION
We only expect a call to Spark() at COMBUSTION.

Google Mock & Google Test do the work.
Note that we have to write the “mock” ourselves – that’s automated by “HippoMocks”.

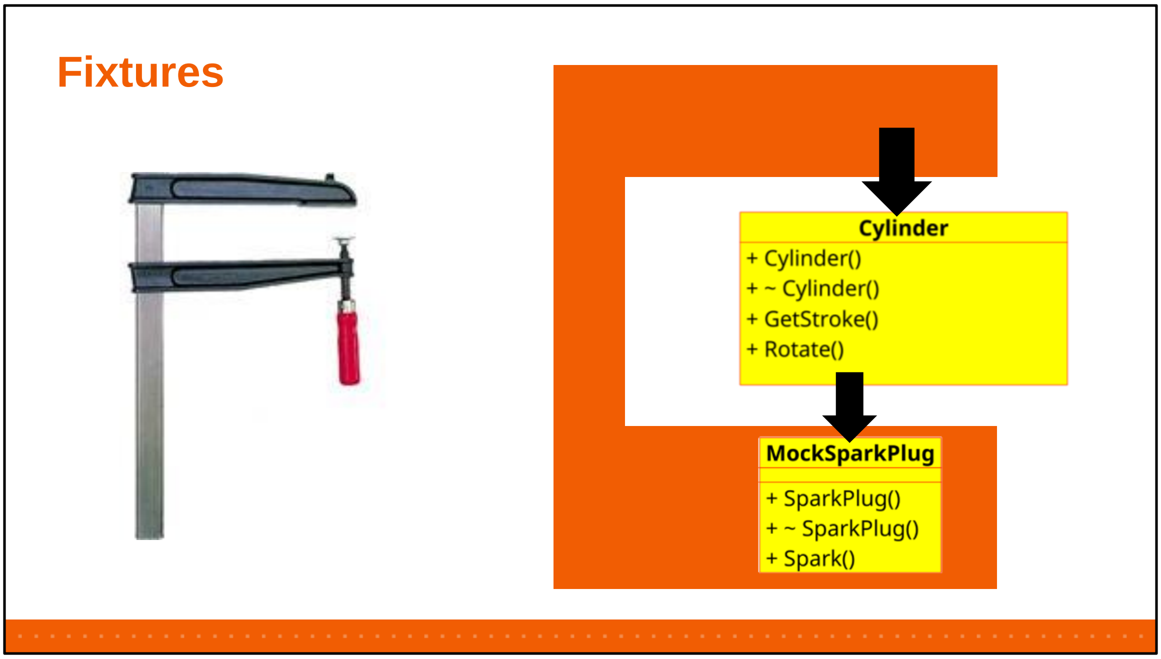


Dependency Injection!

```
class Cylinder
{
public:
    Cylinder(int cylPos, std::unique_ptr<SparkPlug> sparkPlug);
    virtual ~Cylinder();

    enum class Stroke { ... };

    Stroke getStroke();
    void rotate();

private:
    int m_cylinderPosition;

    Stroke m_stroke;
    std::unique_ptr<SparkPlug> m_sparkPlug;
};
```

Dependency injection: instead of having the class create its innards, we provide them in a “Factory”.
The advantage: really improves testability.
The disadvantage: the Factories get more complex.

In our code, the Factories roughly are the only ones not following the ‘4-5 lines per function’ and ‘max 100 lines per file’ rules.



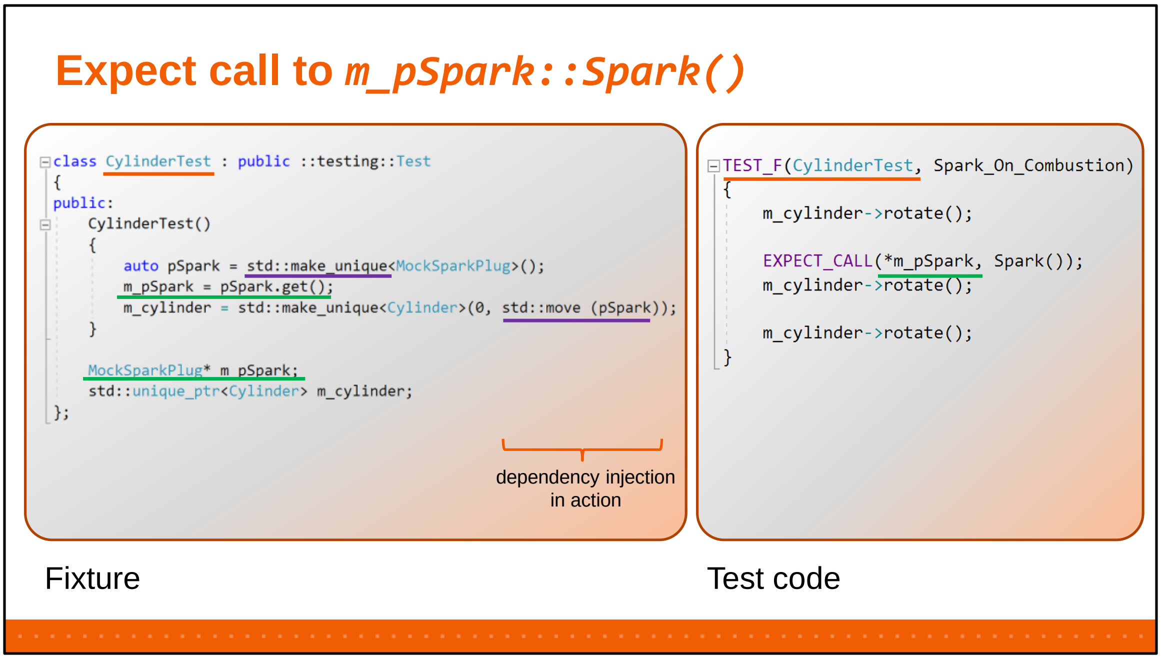
Fixtures

Cylinder

+ Cylinder()
+ ~ Cylinder()
+ GetStroke()
+ Rotate()

MockSparkPlug

+ SparkPlug()
+ ~ SparkPlug()
+ Spark()



Expect call to `m_pSpark::Spark()`

```
class CylinderTest : public ::testing::Test
{
public:
    CylinderTest()
    {
        auto pSpark = std::make_unique<MockSparkPlug>();
        m_pSpark = pSpark.get();
        m_cylinder = std::make_unique<Cylinder>(0, std::move (pSpark));
    }

    MockSparkPlug* m_pSpark;
    std::unique_ptr<Cylinder> m_cylinder;
};
```

dependency injection
in action

Fixture

```
TEST_F(CylinderTest, Spark_On_Combustion)
{
    m_cylinder->rotate();

    EXPECT_CALL(*m_pSpark, Spark());
    m_cylinder->rotate();

    m_cylinder->rotate();
}
```

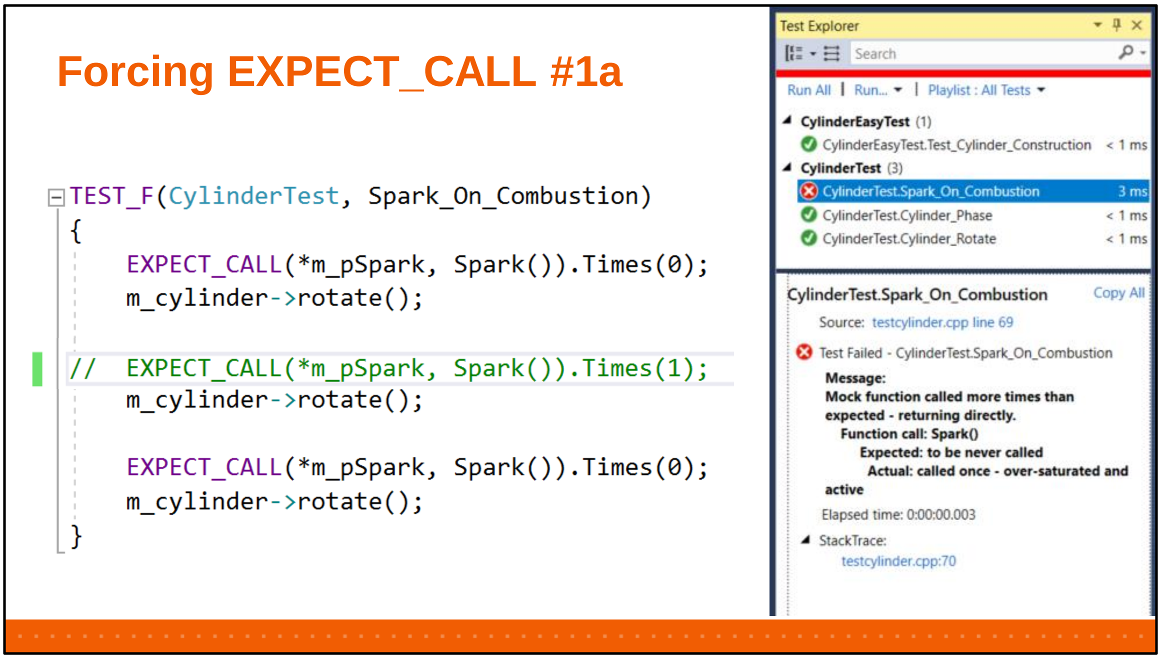
Test code

Test fails if this call didn’t happen.

The TEST_F macro creates a new subclass of the Fixture called CylinderTest_Spark_on_Combustion – so you can access any public member of the Fixture as your own.
For every test, the fixture is destructed and constructed again – because every test is in a different class.

This code also shows a common issue with passing unique_ptr: you loose the contents after construction. So we have to keep an old-fashioned pointer around for future use.

But… Pay attention to the real requirement: the call must happen AFTER the first rotate(), but we didn’t specify whether during/after 2nd or 3rd or during destruction…
Ordering is important here!!!
So, putting the EXPECT before for the first rotate() has the same effect.
Note that putting the EXPECT between the 2nd and the 3rd will fail the test.



Forcing EXPECT_CALL #1a

```
TEST_F(CylinderTest, Spark_On_Combustion)
{
    EXPECT_CALL(*m_pSpark, Spark()).Times(0);
    m_cylinder->rotate();

//  EXPECT_CALL(*m_pSpark, Spark()).Times(1);
    m_cylinder->rotate();

    EXPECT_CALL(*m_pSpark, Spark()).Times(0);
    m_cylinder->rotate();
}
```

CylinderEasyTest (1)
- CylinderEasyTest.Test_Cylinder_Construction < 1 ms

CylinderTest (3)
- CylinderTest.Spark_On_Combustion 3 ms
- CylinderTest.Cylinder_Phase < 1 ms
- CylinderTest.Cylinder_Rotate < 1 ms

CylinderTest.Spark_On_Combustion

Source: testcylinder.cpp line 69

Test Failed - CylinderTest.Spark_On_Combustion

```
Message:
Mock function called more times than
expected - returning directly.
    Function call: Spark()
         Expected: to be never called
           Actual: called once - over-saturated and
active
```

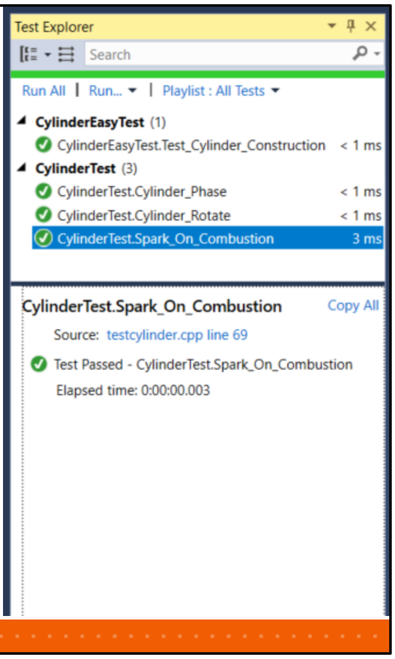
Elapsed time: 0:00:00.003

StackTrace:
testcylinder.cpp:70

One way to enforce EXPECT_CALL to work, is by providing a “cardinality” – Times(0) means it has to never be called.

Forcing EXPECT_CALL #1b

```
TEST_F(CylinderTest, Spark_On_Combustion)
{
    EXPECT_CALL(*m_pSpark, Spark()).Times(0);
    m_cylinder->rotate();

    EXPECT_CALL(*m_pSpark, Spark()).Times(1);
    m_cylinder->rotate();

    EXPECT_CALL(*m_pSpark, Spark()).Times(0);
    m_cylinder->rotate();
}
```



```
◢ CylinderEasyTest (1)
    ✓ CylinderEasyTest.Test_Cylinder_Construction   < 1 ms
◢ CylinderTest (3)
    ✓ CylinderTest.Cylinder_Phase                   < 1 ms
    ✓ CylinderTest.Cylinder_Rotate                  < 1 ms
    ✓ CylinderTest.Spark_On_Combustion                3 ms
```

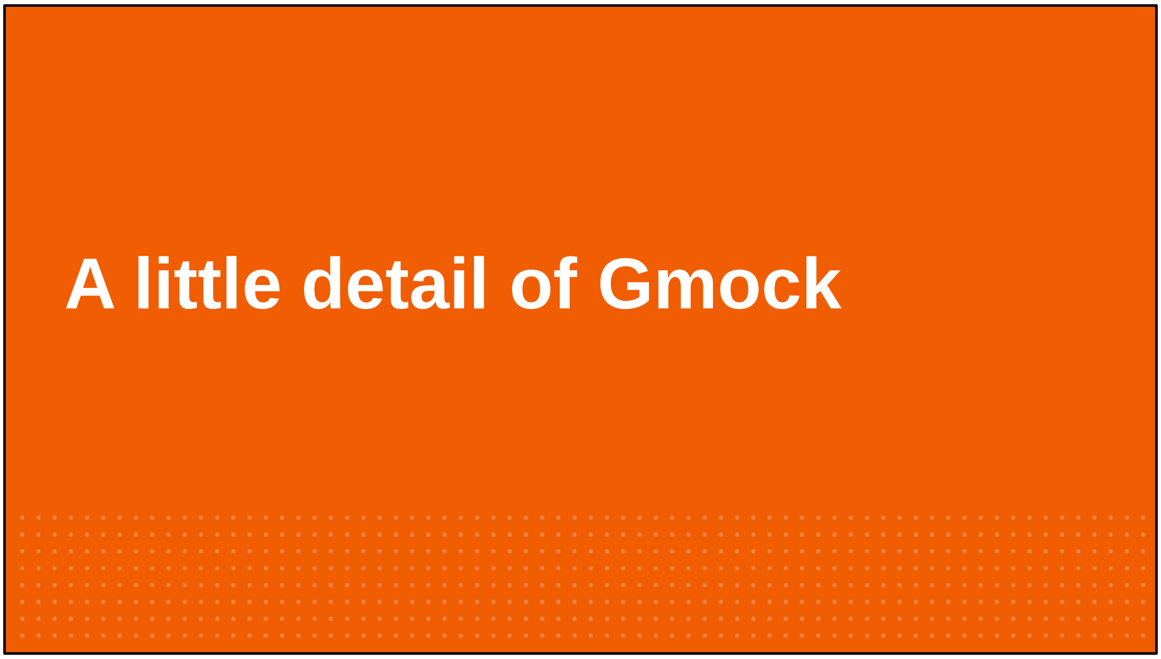
CylinderTest.Spark_On_Combustion

Source: testcylinder.cpp line 69

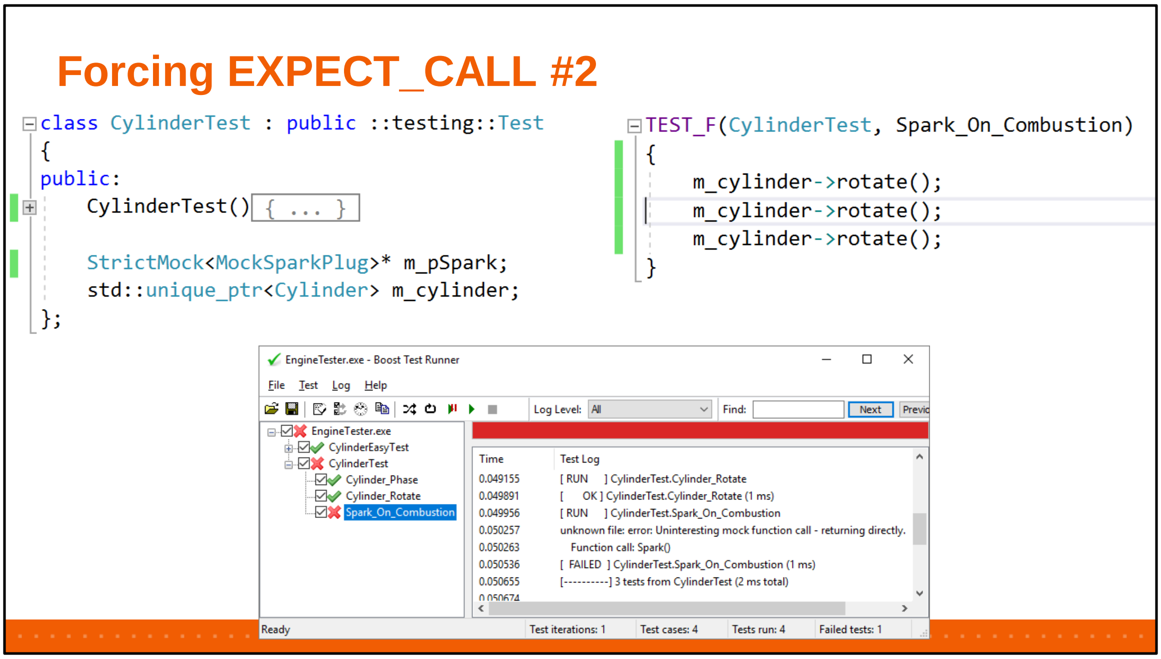
✓ Test Passed - CylinderTest.Spark_On_Combustion

Elapsed time: 0:00:00.003

One way to enforce EXPECT_CALL to work, is by providing a “cardinality” – Times(0) means it has to never be called.



A little detail of Gmock



Forcing EXPECT_CALL #2

```
class CylinderTest : public ::testing::Test
{
public:
    CylinderTest() { ... }

    StrictMock<MockSparkPlug>* m_pSpark;
    std::unique_ptr<Cylinder> m_cylinder;
};
```

```
TEST_F(CylinderTest, Spark_On_Combustion)
{
    m_cylinder->rotate();
    m_cylinder->rotate();
    m_cylinder->rotate();
}
```

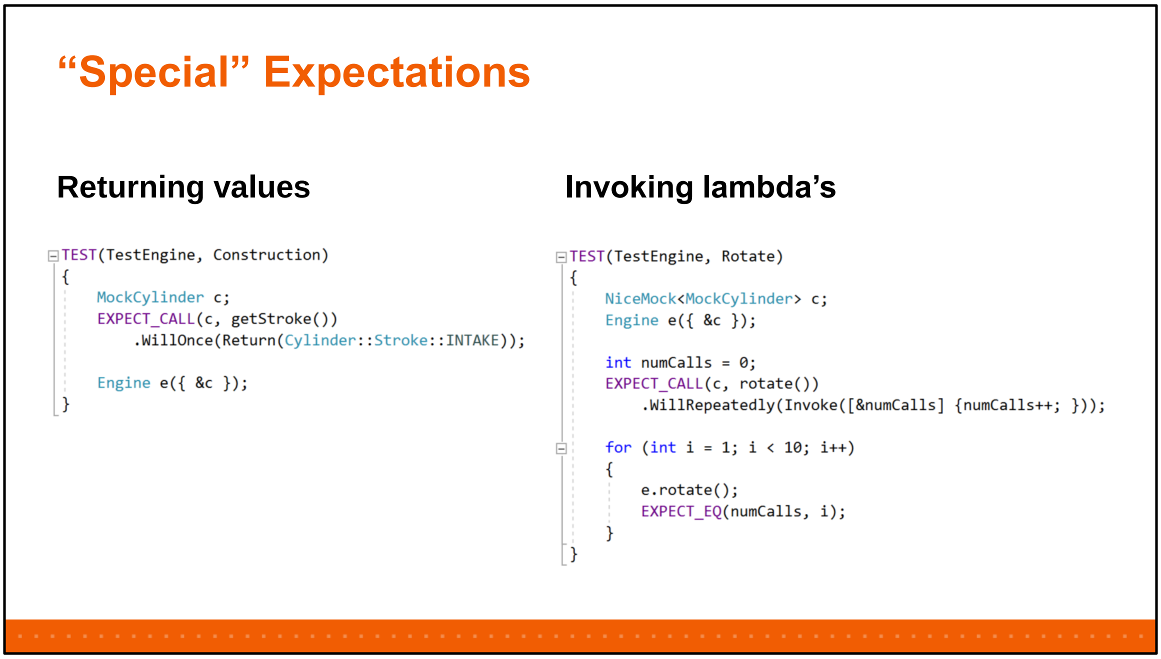
EngineTester.exe - Boost Test Runner

- ☑ EngineTester.exe
  - ☑ CylinderEasyTest
  - ☑ CylinderTest
    - ☑ Cylinder_Phase
    - ☑ Cylinder_Rotate
    - ☑ Spark_On_Combustion

```
Time        Test Log
0.049155    [ RUN      ] CylinderTest.Cylinder_Rotate
0.049891    [       OK ] CylinderTest.Cylinder_Rotate (1 ms)
0.049956    [ RUN      ] CylinderTest.Spark_On_Combustion
0.050257    unknown file: error: Uninteresting mock function call - returning directly.
0.050263        Function call: Spark()
0.050536    [  FAILED  ] CylinderTest.Spark_On_Combustion (1 ms)
0.050655    [----------] 3 tests from CylinderTest (2 ms total)
0.050674
```

Ready | Test iterations: 1 | Test cases: 4 | Tests run: 4 | Failed tests: 1

Another way is to use a StrictMock (the reverse is a NiceMock).



“Special” Expectations

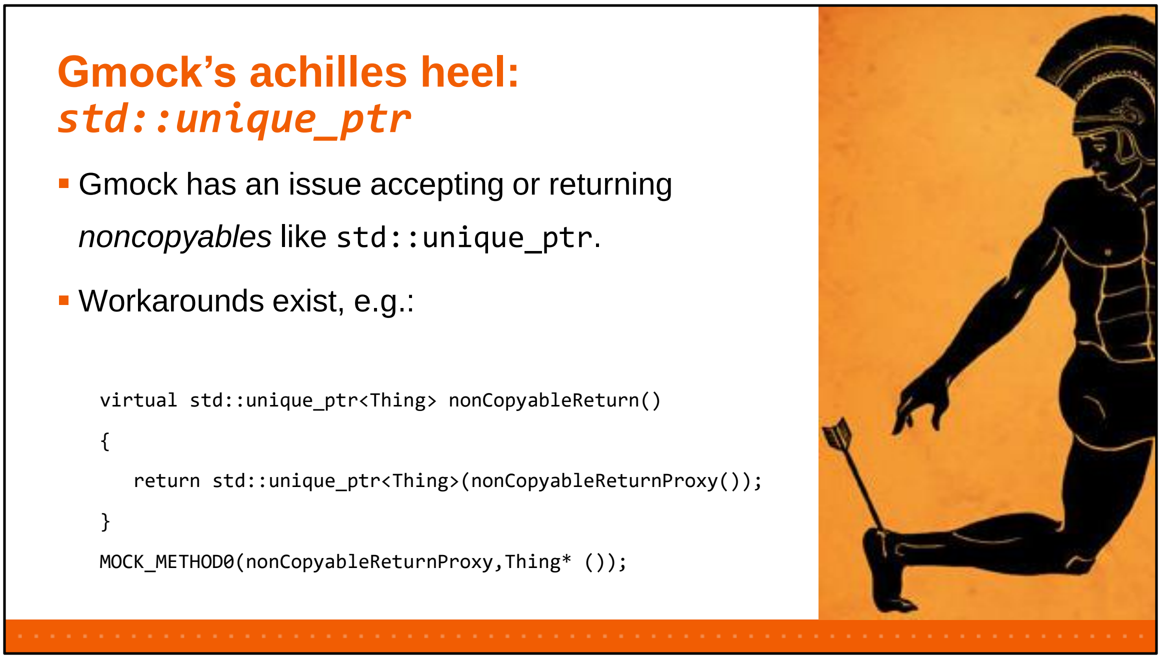
Returning values

```
TEST(TestEngine, Construction)
{
    MockCylinder c;
    EXPECT_CALL(c, getStroke())
        .WillOnce(Return(Cylinder::Stroke::INTAKE));

    Engine e({ &c });
}
```

Invoking lambda’s

```
TEST(TestEngine, Rotate)
{
    NiceMock<MockCylinder> c;
    Engine e({ &c });

    int numCalls = 0;
    EXPECT_CALL(c, rotate())
        .WillRepeatedly(Invoke([&numCalls] {numCalls++; }));

    for (int i = 1; i < 10; i++)
    {
        e.rotate();
        EXPECT_EQ(numCalls, i);
    }
}
```

WARNING: play close attention:
WillOnce(): exactly 1 time;
WillRepeatedly(): **0 or more** !!!

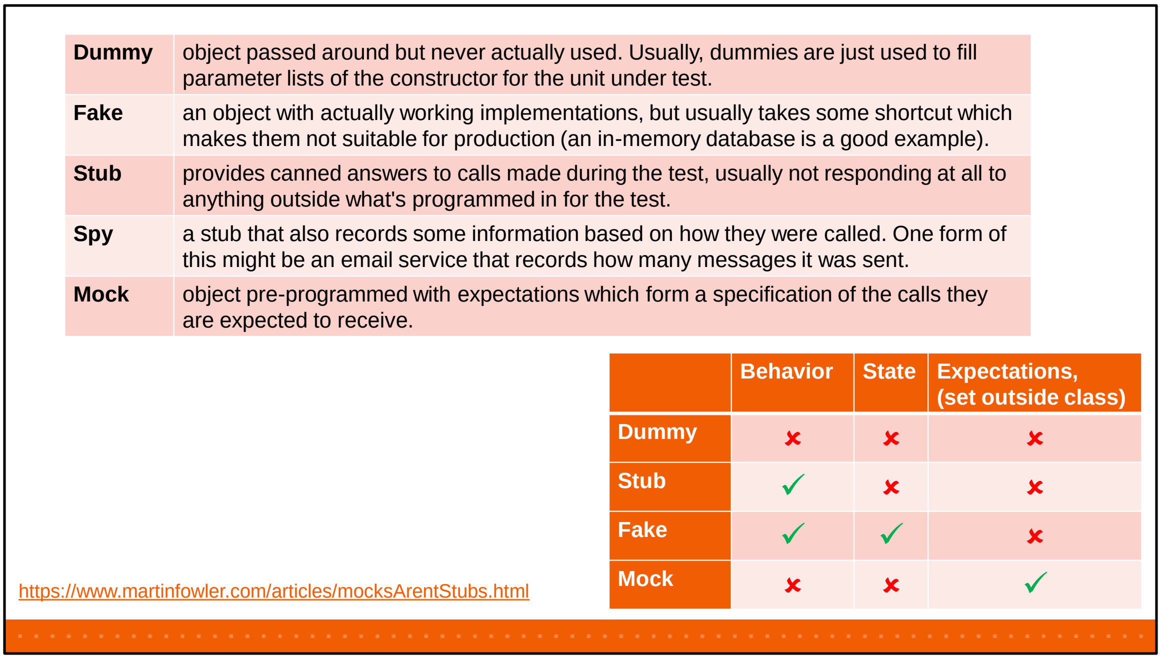


Gmock’s achilles heel: *`std::unique_ptr`*

- Gmock has an issue accepting or returning *noncopyables* like `std::unique_ptr`.
- Workarounds exist, e.g.:

```
virtual std::unique_ptr<Thing> nonCopyableReturn()
{
   return std::unique_ptr<Thing>(nonCopyableReturnProxy());
}
MOCK_METHOD0(nonCopyableReturnProxy,Thing* ());
```

There’s a lot more to google mock and test that I cannot show. The documentation is very good.

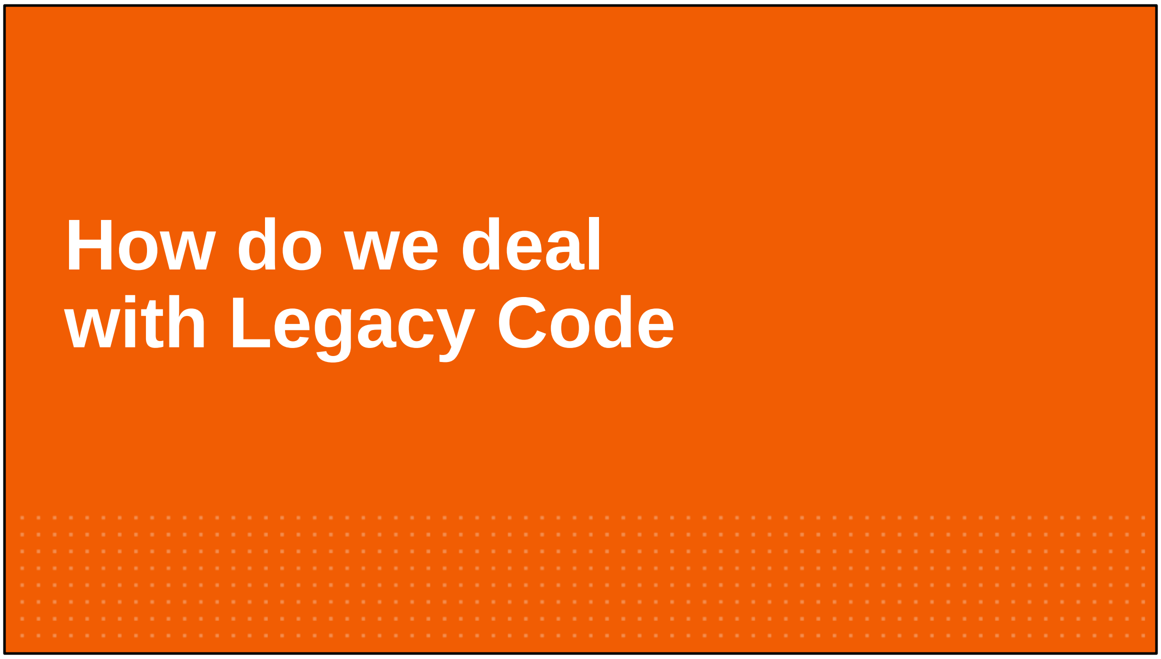


Dummy	object passed around but never actually used. Usually, dummies are just used to fill parameter lists of the constructor for the unit under test.
Fake	an object with actually working implementations, but usually takes some shortcut which makes them not suitable for production (an in-memory database is a good example).
Stub	provides canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
Spy	a stub that also records some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
Mock	object pre-programmed with expectations which form a specification of the calls they are expected to receive.

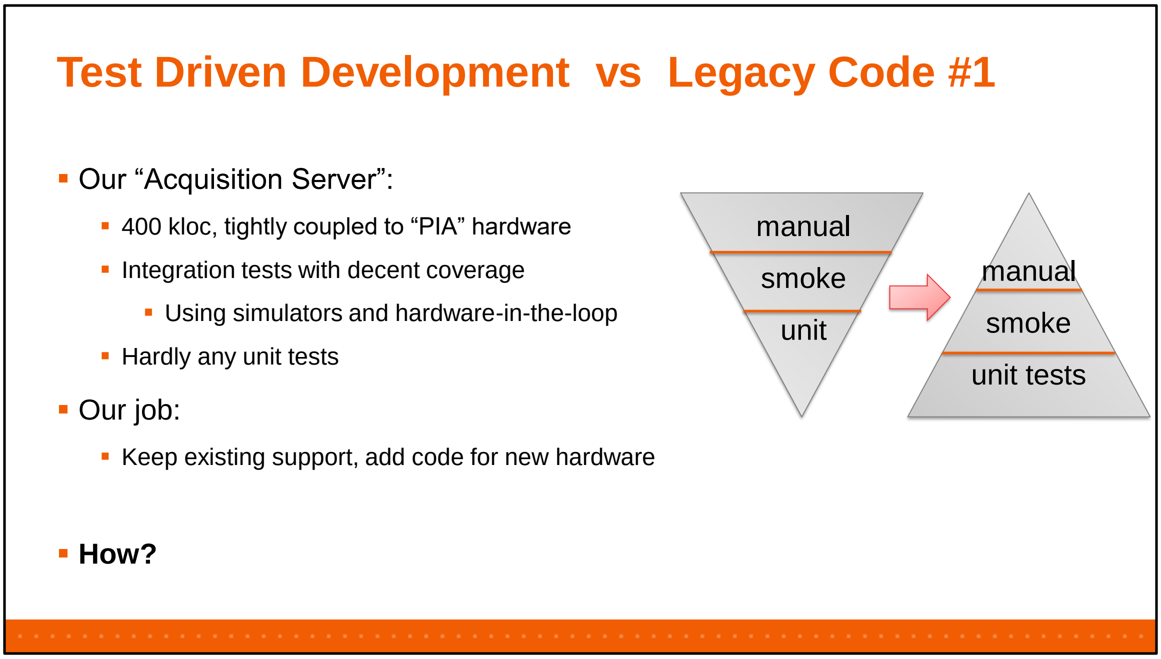
	Behavior	State	Expectations, (set outside class)
Dummy	✗	✗	✗
Stub	✓	✗	✗
Fake	✓	✓	✗
Mock	✗	✗	✓

https://www.martinfowler.com/articles/mocksArentStubs.html

These two tables are part of our internal "cheat sheet" – detailing what features of gmock and gtest we use where, how and why. It's a subset of all that's possible.



How do we deal with Legacy Code



Test Driven Development vs Legacy Code #1

- Our “Acquisition Server”:
  - 400 kloc, tightly coupled to “PIA” hardware
  - Integration tests with decent coverage
    - Using simulators and hardware-in-the-loop
  - Hardly any unit tests
- Our job:
  - Keep existing support, add code for new hardware
- **How?**

manual

smoke

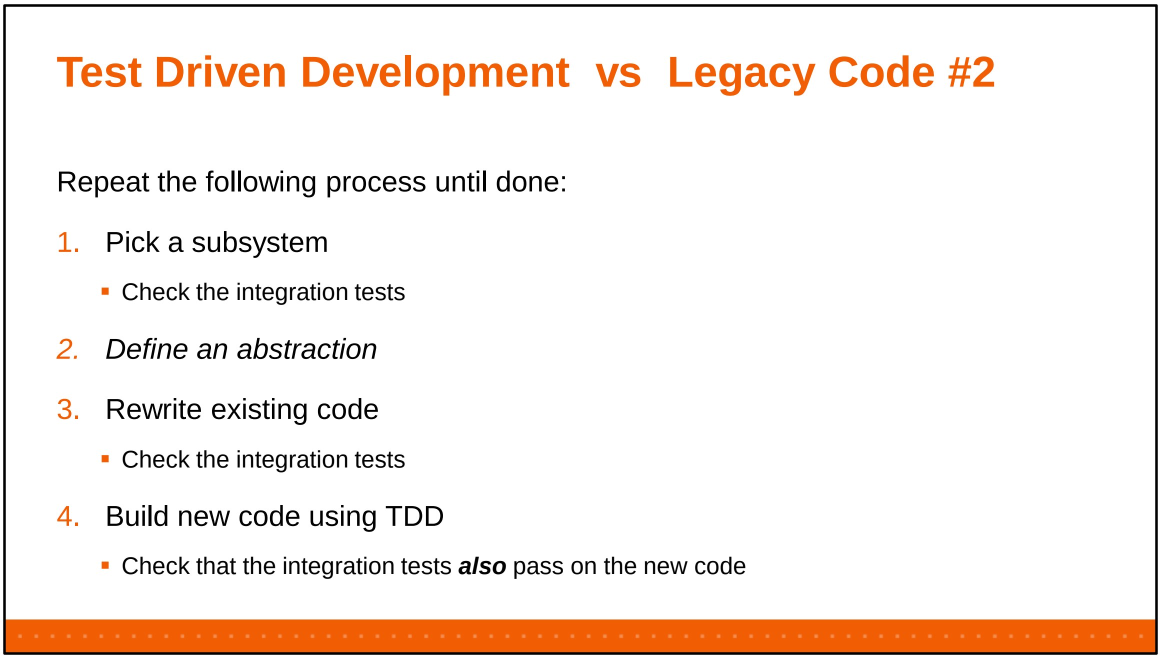
unit

manual

smoke

unit tests

Remember Michael Feathers statement?

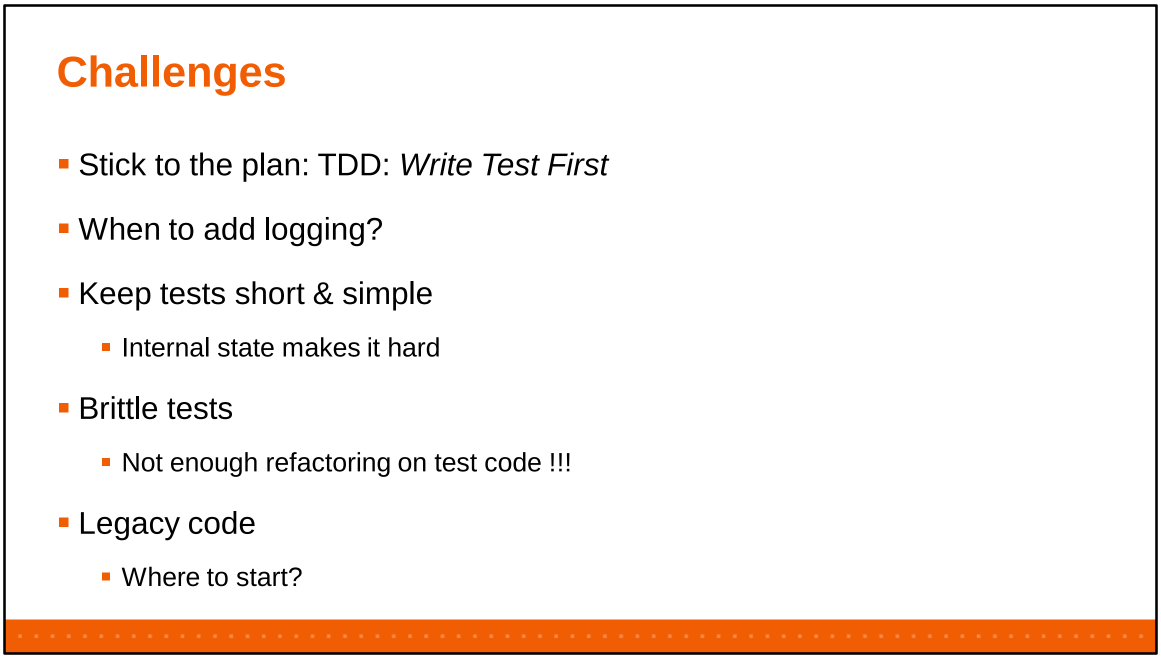


Test Driven Development vs Legacy Code #2

Repeat the following process until done:

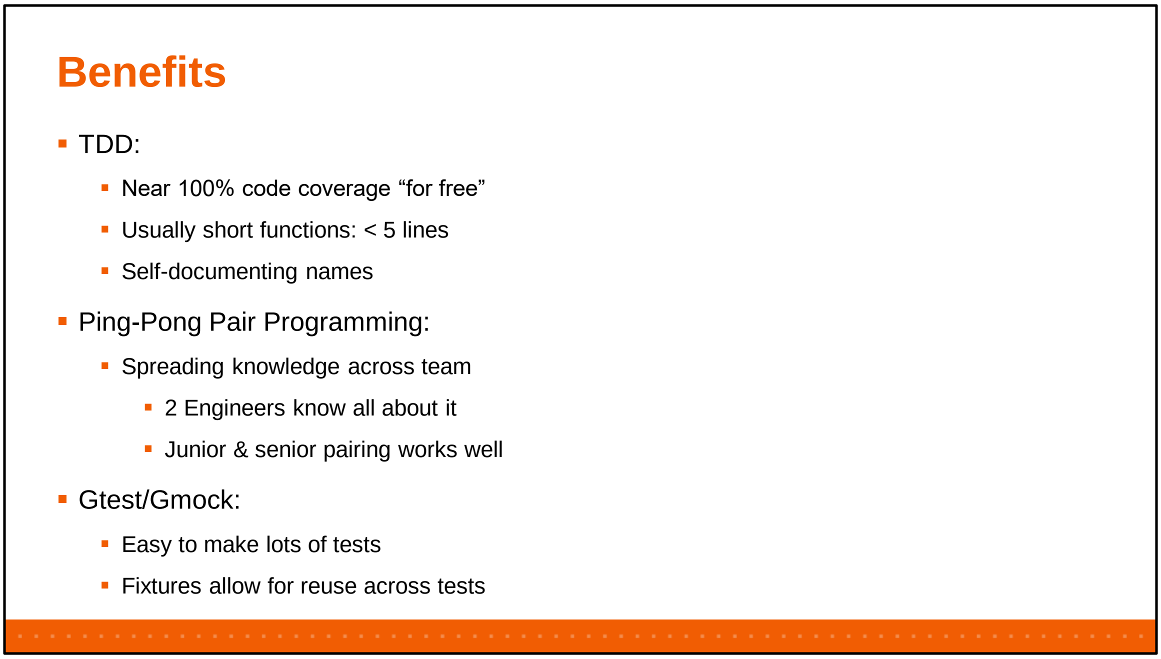
1. Pick a subsystem
   - Check the integration tests
2. *Define an abstraction*
3. Rewrite existing code
   - Check the integration tests
4. Build new code using TDD
   - Check that the integration tests ***also*** pass on the new code

Again, refactoring was key to our success: we had working tests – albeit at a higher level – and could start from there.



Challenges

- Stick to the plan: TDD: *Write Test First*
- When to add logging?
- Keep tests short & simple
  - Internal state makes it hard
- Brittle tests
  - Not enough refactoring on test code !!!
- Legacy code
  - Where to start?



Benefits

- TDD:
  - Near 100% code coverage “for free”
  - Usually short functions: < 5 lines
  - Self-documenting names
- Ping-Pong Pair Programming:
  - Spreading knowledge across team
    - 2 Engineers know all about it
    - Junior & senior pairing works well
- Gtest/Gmock:
  - Easy to make lots of tests
  - Fixtures allow for reuse across tests

Klaasvan.gend@Thermofisher.com

Klaas.van.gend@Sioux.eu

Klaas@040coders.nl

ThermoFisher
SCIENTIFIC

SIOUX
SOURCE OF YOUR TECHNOLOGY

www.sioux.eu