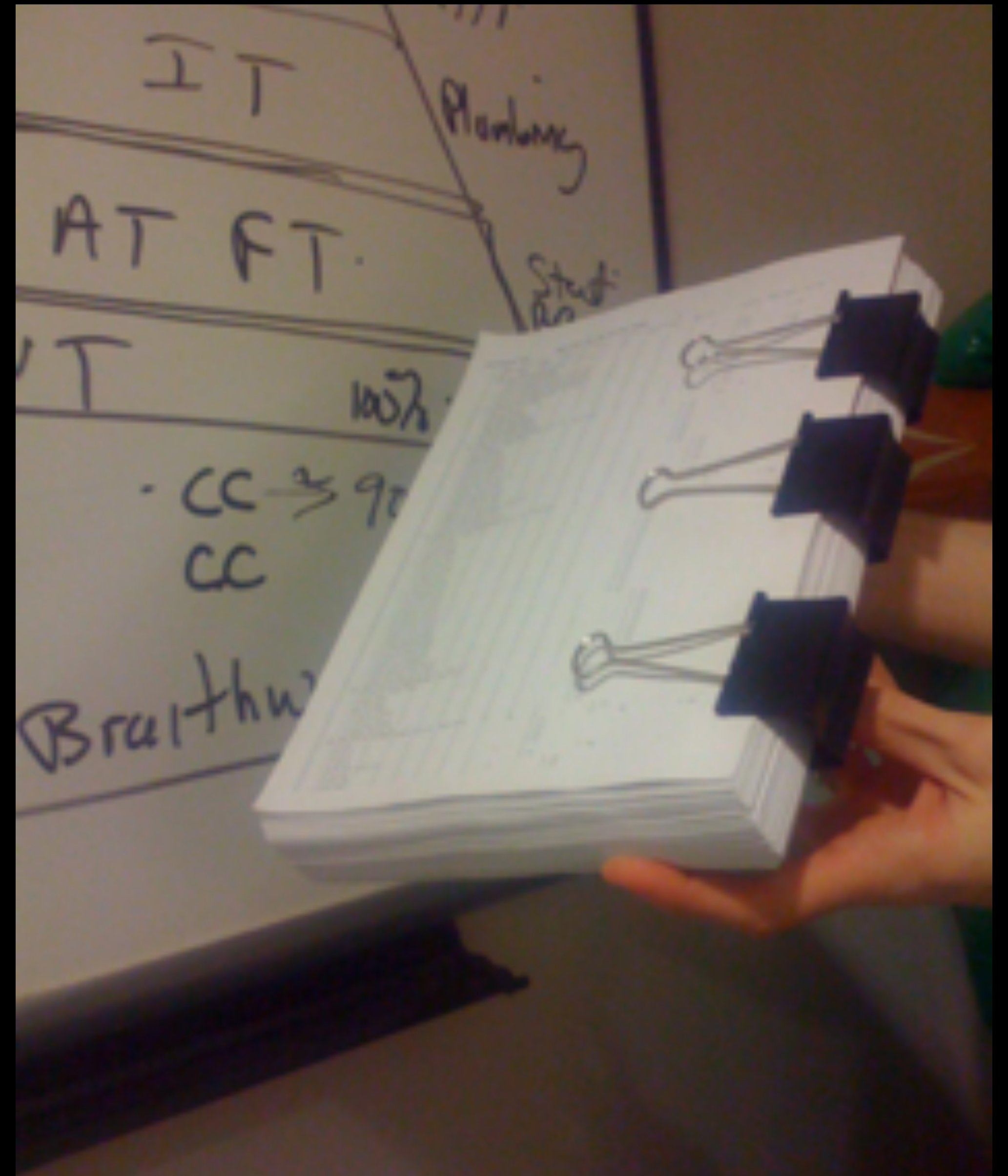


# To Catch(2) a mocking Bug!

Quick overview of Catch Test Framework  
and even quicker overview of FakeIt mock Framework

# Who knows what this is?

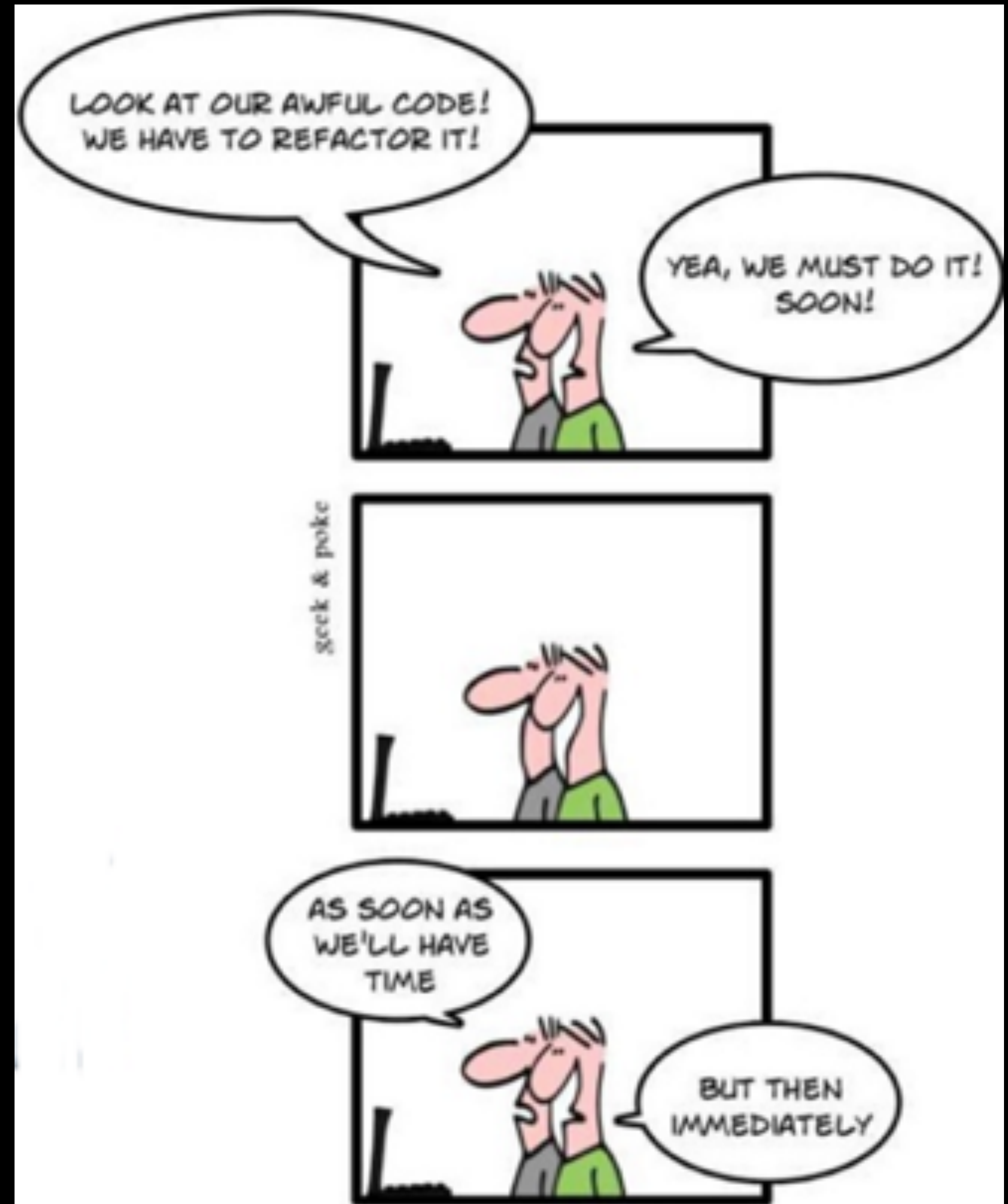
Quality Assurance : List of manual tests to be done every time a new Release is being prepared for shipping.



# A Story of Testing

- Pressure Regulator Testing saga
- More than a Year of Testing and Bug Corrections
- Continually had to retest things that kept breaking
- Ended up testing about 70% of the capabilities of the product.
- Some Bug found in the field were “Known bugs”.
- Became scared of making changes, to avoid breaking anything.

# Code ... without unit tests



# Code ... without unit tests

- Fear of Refactoring
- Fear of adding new features
- Cannot make sure your feature working now will not be corrupted by someone else working on the code later. (Beyoncé rule)

# Is it necessary?

- We already have manuel tests
- We have a great Q.A. department
- We don't have time to write tests
- It's hard to setup
- ...

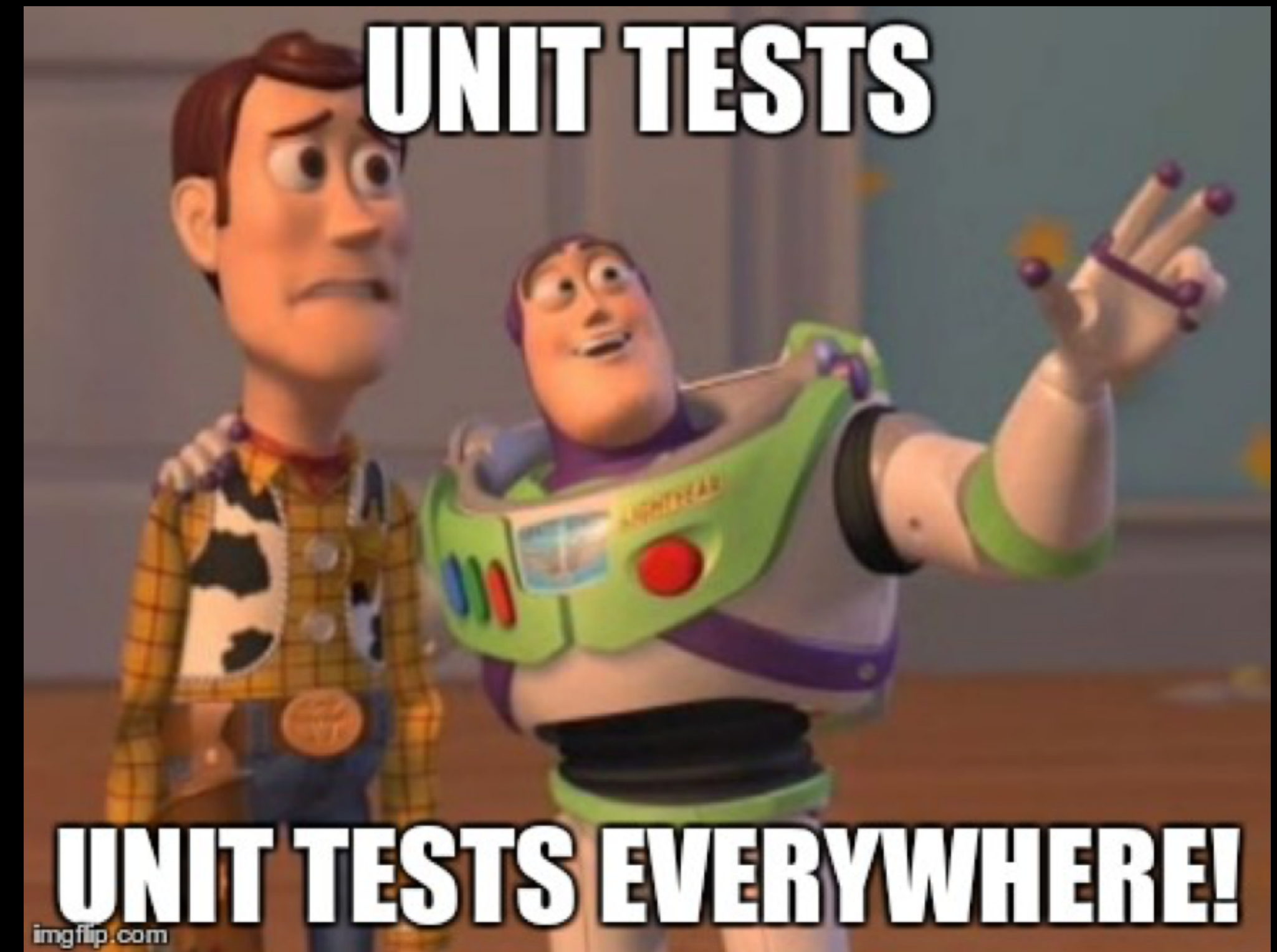


# Code with unit Tests

Eliminates fear

Can refactor with confidence

But how to ...  
reach Good coverage?



# TDD

Test-Driven Development  
Test First



# TESTING

I FIND YOUR LACK OF TESTS DISTURBING.



# TDD

- RED : Write a Failing Test
- GREEN : Make the test pass (Make the code work)
- REFACTOR : Make the code better
- Repeat!



# Tests Frameworks

- Google Tests

- Boost.Test

- Catch

- CppTest

- CppUTest

- CxxTest

- CppUnit

- More on [Wikipedia](#)

- How to choose a Framework???





# Catch Features

## Key Features

---

- Quick and Really easy to get started. Just download catch.hpp, `#include` it and you're away.
- No external dependencies. As long as you can compile C++11 and have a C++ standard library available.
- Write test cases as, self-registering, functions (or methods, if you prefer).
- Divide test cases into sections, each of which is run in isolation (eliminates the need for fixtures).
- Use BDD-style Given-When-Then sections as well as traditional unit test cases.
- Only one core assertion macro for comparisons. Standard C/C++ operators are used for the comparison - yet the full expression is decomposed and lhs and rhs values are logged.
- Tests are named using free-form strings - no more couching names in legal identifiers.

Demo



# Catch Features

## Key Features

---

- Quick and Really easy to get started. Just download catch.hpp, `#include` it and you're away.
- No external dependencies. As long as you can compile C++11 and have a C++ standard library available.
- Write test cases as, self-registering, functions (or methods, if you prefer).
- Divide test cases into sections, each of which is run in isolation (eliminates the need for fixtures).
- Use BDD-style Given-When-Then sections as well as traditional unit test cases.
- Only one core assertion macro for comparisons. Standard C/C++ operators are used for the comparison - yet the full expression is decomposed and lhs and rhs values are logged.
- Tests are named using free-form strings - no more couching names in legal identifiers.



# Assertion Macros

ASSERT( condition );	VS REQUIRE( condition );
ASSERT_TRUE( condition );	VS REQUIRE( condition );
ASSERT_FALSE( condition );	VS REQUIRE_FALSE( condition );
ASSERT_EQ( val1, val2 );	VS REQUIRE( val1 == val2 );
ASSERT_NE( val1, val2 );	VS REQUIRE( val1 != val2 );
ASSERT_LT( val1, val2 );	VS REQUIRE( val1 < val2 );
ASSERT_LE( val1, val2 );	VS REQUIRE( val1 <= val2 );
ASSERT_GT( val1, val2 );	VS REQUIRE( val1 > val2 );
ASSERT_GE( val1, val2 );	VS REQUIRE( val1 >= val2 );
ASSERT_STREQ( str1, str2 );	VS REQUIRE( str1 == str2 );
ASSERT_FLOAT_EQ( val1, val2 );	VS REQUIRE( val1 == Approx(val2) );

- Only one core macro for assertion, but full expression is decomposed and reported
- No need to remember which value is the “expected” and the “actual”.  
(Usually the first value is the “expected” value, but I often mix them).
- Approx Class to easily compare Floating point numbers

# Assertion Macros

```
require( expression );  
check( expression );
```

- Another Macro for Non-Fatal Assertions : CHECK
- Fatal Assertion vs Non-Fatal Assertion

# Catch Features

## Key Features

---

- Quick and Really easy to get started. Just download catch.hpp, `#include` it and you're away.
- No external dependencies. As long as you can compile C++11 and have a C++ standard library available.
- Write test cases as, self-registering, functions (or methods, if you prefer).
- Divide test cases into sections, each of which is run in isolation (eliminates the need for fixtures).
- Use BDD-style Given-When-Then sections as well as traditional unit test cases.
- Only one core assertion macro for comparisons. Standard C/C++ operators are used for the comparison - yet the full expression is decomposed and lhs and rhs values are logged.
- Tests are named using free-form strings - no more couching names in legal identifiers.

# Test Fixtures?

- xUnit (Most Test frameworks follow xUnit patterns) introduces the idea of a Test Fixtures, to have common Setup and Cleanup (Teardown) facilities for a group of tests.
- Catch, while supporting this way of working, introduces a new (better?, more suited for C++?) way of working.
- Utilises the stack, with scope, to setup and cleanup tests in a much more natural way.



# Sections Please!

```
TEST_CASE( "vectors can be sized and resized", "[vector]" ) {
    std::vector<int> v( 5 );

    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );

    SECTION( "resizing bigger changes size and capacity" ) {
        v.resize( 10 );

        REQUIRE( v.size() == 10 );
        REQUIRE( v.capacity() >= 10 );
    }

    SECTION( "reserving bigger changes capacity but not size" ) {
        v.reserve( 10 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );

        SECTION( "reserving smaller again does not change capacity" ) {
            v.reserve( 7 );

            REQUIRE( v.capacity() >= 10 );
        }
    }
}
```



# Sections Please!

```
TEST_CASE( "vectors can be sized and resized", "[vector]" ) {
    std::vector<int> v( 5 );

    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );

    SECTION( "resizing bigger changes size and capacity" ) {
        v.resize( 10 );


        REQUIRE( v.size() == 10 );
        REQUIRE( v.capacity() >= 10 );
    }

    SECTION( "reserving bigger changes capacity but not size" ) {
        v.reserve( 10 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );

        SECTION( "reserving smaller again does not change capacity" ) {
            v.reserve( 7 );

            REQUIRE( v.capacity() >= 10 );
        }
    }
}
```



# Sections Please!

```
TEST_CASE( "vectors can be sized and resized", "[vector]" ) {
    std::vector<int> v( 5 );

    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );

    SECTION( "resizing bigger changes size and capacity" ) {
        v.resize( 10 );

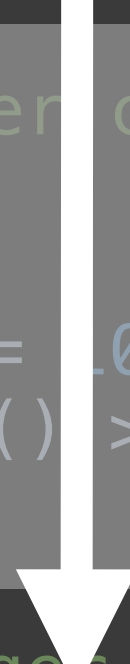
        REQUIRE( v.size() == 10 );
        REQUIRE( v.capacity() >= 10 );
    }

    SECTION( "reserving bigger changes capacity but not size" ) {
        v.reserve( 10 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );

        SECTION( "reserving smaller again does not change capacity" ) {
            v.reserve( 7 );

            REQUIRE( v.capacity() >= 10 );
        }
    }
}
```



# Catch Features

## Key Features

---

- Quick and Really easy to get started. Just download catch.hpp, `#include` it and you're away.
- No external dependencies. As long as you can compile C++11 and have a C++ standard library available.
- Write test cases as, self-registering, functions (or methods, if you prefer).
- Divide test cases into sections, each of which is run in isolation (eliminates the need for fixtures).
- Use BDD-style Given-When-Then sections as well as traditional unit test cases.
- Only one core assertion macro for comparisons. Standard C/C++ operators are used for the comparison - yet the full expression is decomposed and lhs and rhs values are logged.
- Tests are named using free-form strings - no more couching names in legal identifiers.



# BDD

- Behaviour Driven Development is a style of TDD focusing on the behaviour of modules, using the AAA (Arrange, Act, Assert) explicitly with the “Given - When - Then” syntax.
- SCENARIO Macro is equivalent to the TEST\_CASE MACRO, except that it add the prefix Scenario to the output.
- GIVEN, WHEN and THEN are MACROS equivalent to SECTIONS, except that they add a prefix to the output to the reporters.

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {  
  
    GIVEN( "A vector with some items" ) {  
        std::vector<int> v( 5 );  
  
        REQUIRE( v.size() == 5 );  
        REQUIRE( v.capacity() >= 5 );  
  
        WHEN( "the size is increased" ) {  
            v.resize( 10 );  
  
            THEN( "the size and capacity change" ) {  
                REQUIRE( v.size() == 10 );  
                REQUIRE( v.capacity() >= 10 );  
            }  
        }  
        WHEN( "the size is reduced" ) {  
            v.resize( 0 );  
  
            THEN( "the size changes but not capacity" ) {  
                REQUIRE( v.size() == 0 );  
                REQUIRE( v.capacity() >= 5 );  
            }  
        }  
        WHEN( "more capacity is reserved" ) {  
            v.reserve( 10 );  
  
            THEN( "the capacity changes but not the size" ) {  
                REQUIRE( v.size() == 5 );  
                REQUIRE( v.capacity() >= 10 );  
            }  
        }  
        WHEN( "less capacity is reserved" ) {  
            v.reserve( 0 );  
  
            THEN( "neither size nor capacity are changed" ) {  
                REQUIRE( v.size() == 5 );  
                REQUIRE( v.capacity() >= 5 );  
            }  
        }  
    }  
}
```

Demo

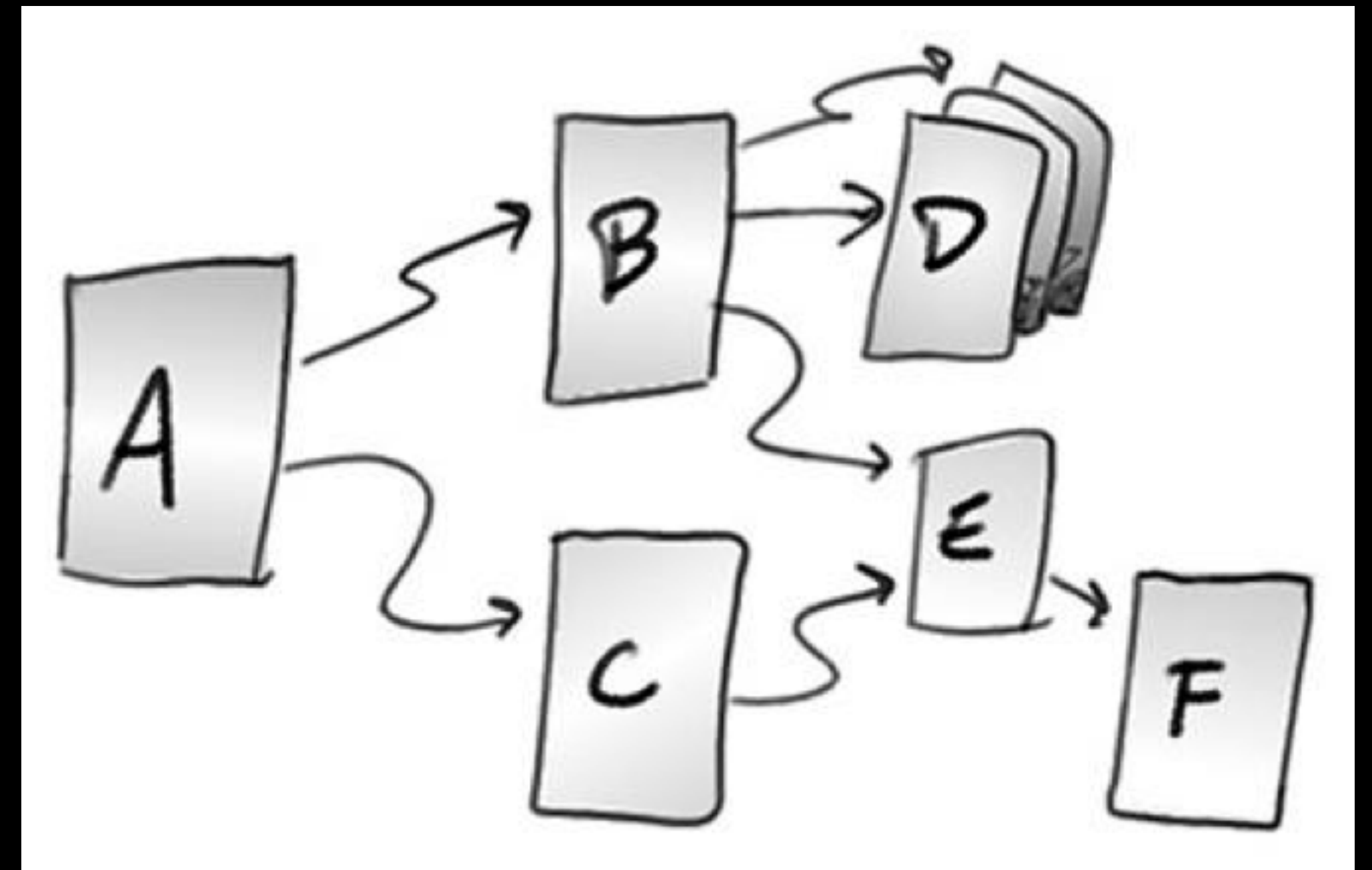


# Other Catch Features

- Tags (for easy groupings of related TEST\_CASEs)
- Assertion Macros (Verify Throwing Exceptions), Logging Macros
- Matchers (Hamcrest style matchers for Strings, Vectors, and extensible to your own types)
- Reporters and Listeners
- Supplying your own main, with command-line parser extensible.

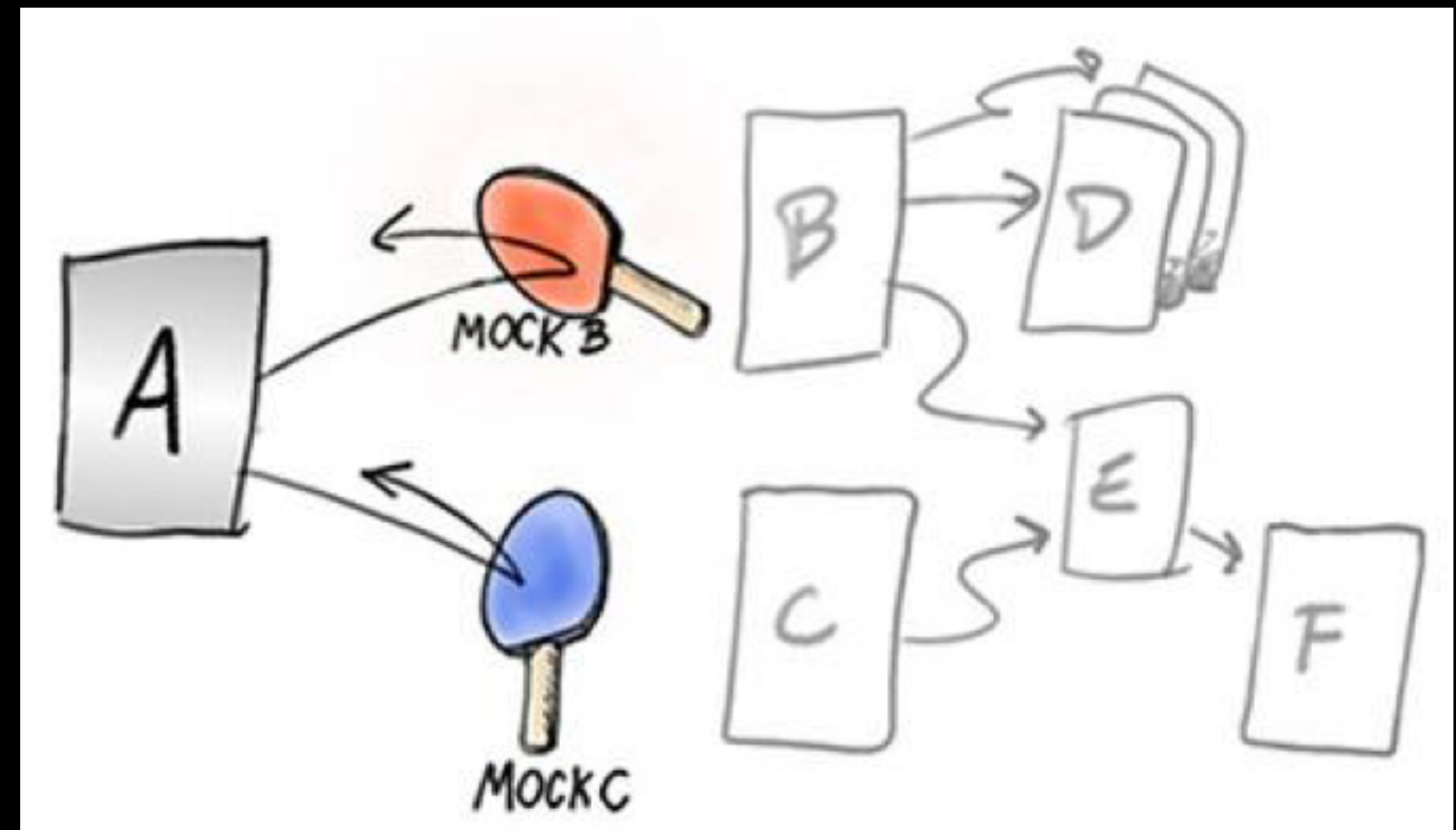
# Mocking? Why?

- Modules have dependencies to other module.
- We want to test in isolation.
- We would like to replace B and C by mocks (Stubs, Fakes, Spies, and Mocks)



# Mocking? Why?

- Modules have dependencies to other module.
- We want to test in isolation.
- We would like to replace B and C by mocks (Stubs, Fakes, Spies, and Mocks)



# Mock Frameworks

- Just like Test frameworks, there are lots of Mocking frameworks.
- Googlemock (part of googletest now)
- Boost.Turtle (to go with Boost.Test)
- Trompeloeil (c++14 header-only, works well with catch)
- FakeIt (header-only, works well with googletest, boost.test, catch, ...)

Demo



# FakeIt

- No need to implement all the virtual functions (even pure virtual ones). Only need to define those that will be used.
- Easy syntax for setup and verification
- Can even Spy existing object.

# Fakelt - Limitations

- Currently only GCC, Clang and MSC++ are supported.
- Can't mock classes with multiple inheritance.
- Can't mock classes with virtual inheritance.
- Currently mocks are not thread safe.
- Official Fakelt works with Catch 2.0.1 (myrgy/Fakelt is a fork working with catch 2.2.3 — Pull Request will probably occur soon in Official Fakelt)

# References

- <https://github.com/catchorg/Catch2>
- <https://github.com/google/googletest>
- <https://github.com/eranpeer/Fakelt>
- <https://github.com/myrgy/Fakelt> (Fork working with Catch 2.2.3)