# Back To Basics
## Testing

PHIL NASH


# What is a test?

# What is a test?

operation

# What is a test?

operation

+

input values

# What is a test?

operation

+

input values

expected output values

# What is a test?

operation

input values

expected output values

```
ASSERT_EQ( add(1, 2), 3 );
```

# What is a test?

operation

input values

expected output values

I run my program

I type in some input

I expect it to show specific output

# What is a test?

operation

input values

expected output values

Invoke the compiler

give it my code

It compiles successfully

# What is a test?

# What is a test?

Unit

Integration

System

# What is a test?

System Tests

Integration Tests

Unit Tests

# What is a test?

The testing pyramid

System Tests

Integration Tests

Unit Tests

# What is a Unit Test?

# What is a **Unit Test?**

Michael Feathers

"A test is **not** a unit test if:

- It talks to the **database**
- It communicates across the **network**
- It touches the **file system**
- It **can't run at the same time** as any of your other unit tests
- You have to do special things to your **environment** (such as editing config files) to run it "

*Michael Feathers - A Set of Unit Testing Rules*

# What is a **Unit Test?**

Michael Feathers

> "Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness."

*Michael Feathers - A Set of Unit Testing Rules*

# What is a **Unit Test?**

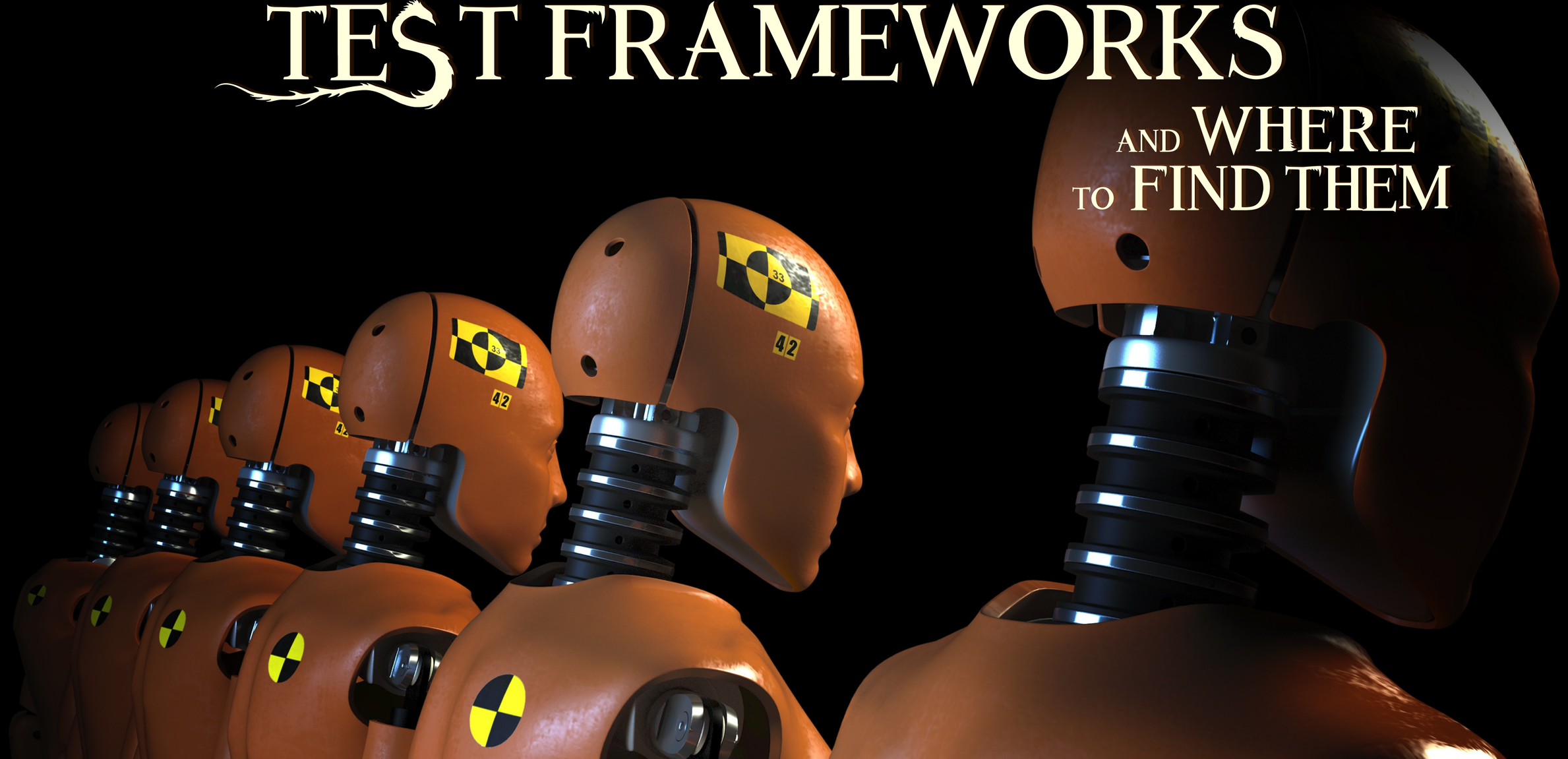
Michael Feathers

"Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness.

However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can **run fast whenever we make our changes**"

*Michael Feathers - A Set of Unit Testing Rules*

# FANTASTIC TEST FRAMEWORKS AND WHERE TO FIND THEM



# WHY USE A TEST FRAMEWORK

```
int add( int a, int b );
```

```c
#include <cassert>

int add( int a, int b );

int main() {
    assert( add( 1, 2 ) == 3 );
}
```

```c
#include <cassert>

int add( int a, int b );

int main() {
    assert( add( 1, 2 ) == 3 );
}
```

```
#include <cassert>


int add( int a, int b ){
    return a*b;
}


int main() {
   assert( add( 1, 2 ) == 3 );
}
```

```cpp
#include <cassert>

int add( int a, int b ){
    return a*b;
}

int main() {
   assert( add( 1, 2 ) == 3 );
}
```

```
Assertion failed: (add( 1, 2 ) == 3), function main,
  file /Users/phil/Dev/Scratch/AddTest/main.cpp, line 8.

Process finished with exit code 6
```

```cpp
#include <cassert>

int add( int a, int b ){
    return a*b;
}

int main() {
   assert( add( 1, 2 ) == 3 );
}
```

```
Assertion failed: (add( 1, 2 ) == 3), function main,
  file /Users/phil/Dev/Scratch/AddTest/main.cpp, line 8.

Process finished with exit code 6
```

```cpp
#include <cassert>

int add( int a, int b ){
    return a*b;
}

int main() {
   assert( add( 1, 2 ) == 3 );
}
```

```
Assertion failed: (add( 1, 2 ) == 3), function main,
  file /Users/phil/Dev/Scratch/AddTest/main.cpp, line 8.

Process finished with exit code 6
```

```cpp
#include <cassert>

int add( int a, int b ){
    return a*b;
}

int main() {
   assert( add( 1, 2 ) == 3 );
}
```

```
Assertion failed: (add( 1, 2 ) == 3), function main,
  file /Users/phil/Dev/Scratch/AddTest/main.cpp, line 8.

Process finished with exit code 6
```

## <cassert> pros:

- no dependencies
- some useful output (file, line, expr...)
- natural, familiar, syntax and usage

## <cassert> pros:

- no dependencies
- some useful output (file, line, expr...)
- natural, familiar, syntax and usage

## <cassert> cons:

## <cassert> pros:

- no dependencies
- some useful output (file, line, expr...)
- natural, familiar, syntax and usage

## <cassert> cons:

- not enough feedback (breakdown of expr?)

## <cassert> pros:

- no dependencies
- some useful output (file, line, expr...)
- natural, familiar, syntax and usage

## <cassert> cons:

- not enough feedback (breakdown of expr?)
- only console output

## <cassert> pros:

- no dependencies
- some useful output (file, line, expr...)
- natural, familiar, syntax and usage

## <cassert> cons:

- not enough feedback (breakdown of expr?)
- only console output
- no test names, groups/ suites

## <cassert> pros:

- no dependencies
- some useful output (file, line, expr...)
- natural, familiar, syntax and usage

## <cassert> cons:

- not enough feedback (breakdown of expr?)
- only console output
- no test names, groups/ suites
- aborts on first failure

en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

# List of unit testing frameworks

From Wikipedia, the free encyclopedia

This article contains embedded lists that **may be poorly defined, unverified or indiscriminate**. Please help to clean it up to meet Wikipedia's quality standards. Where appropriate, incorporate items into the main body of the article. *(April 2018)*

This page is a list of tables of code-driven unit testing frameworks for various programming languages. Some but not all of these are based on xUnit.

**Contents** [hide]

# List of unit testing frameworks

2.11 C++

84 C++ Test frameworks!

# CppUnit

CppUnit

**CppUnitLite**

CppUnit

CppUnitLite

**Boost.Test**

CppUnit

CppUnitLite

Boost.Test

**Google Test**

CppUnit

CppUnitLite

Boost.Test

Google Test

**Catch/ Catch2**

CppUnit

CppUnitLite

Boost.Test

Google Test

Catch/ Catch2

**doctest**

CppUnit

CppUnitLite

Boost.Test

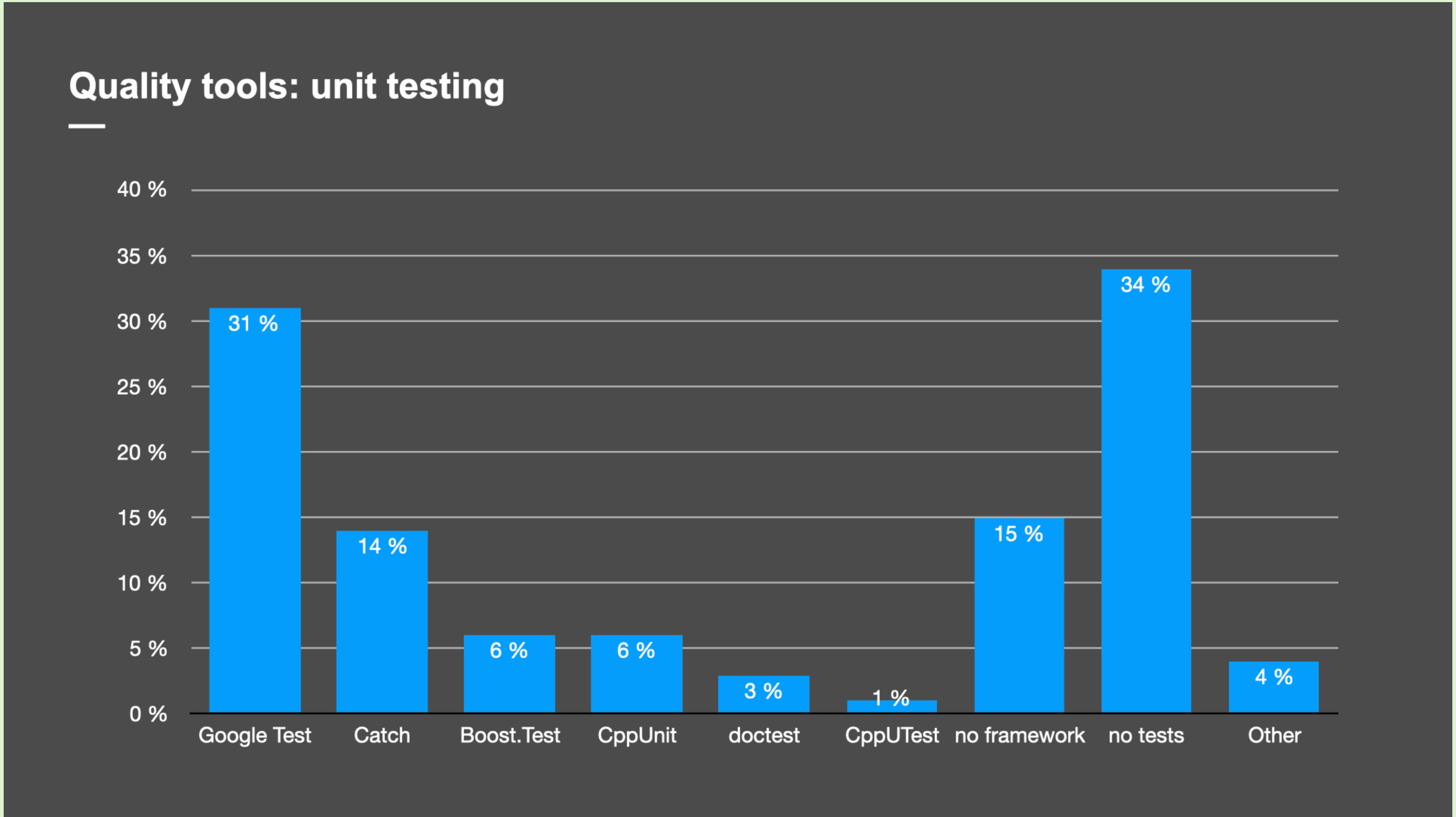
Google Test

Catch/ Catch2

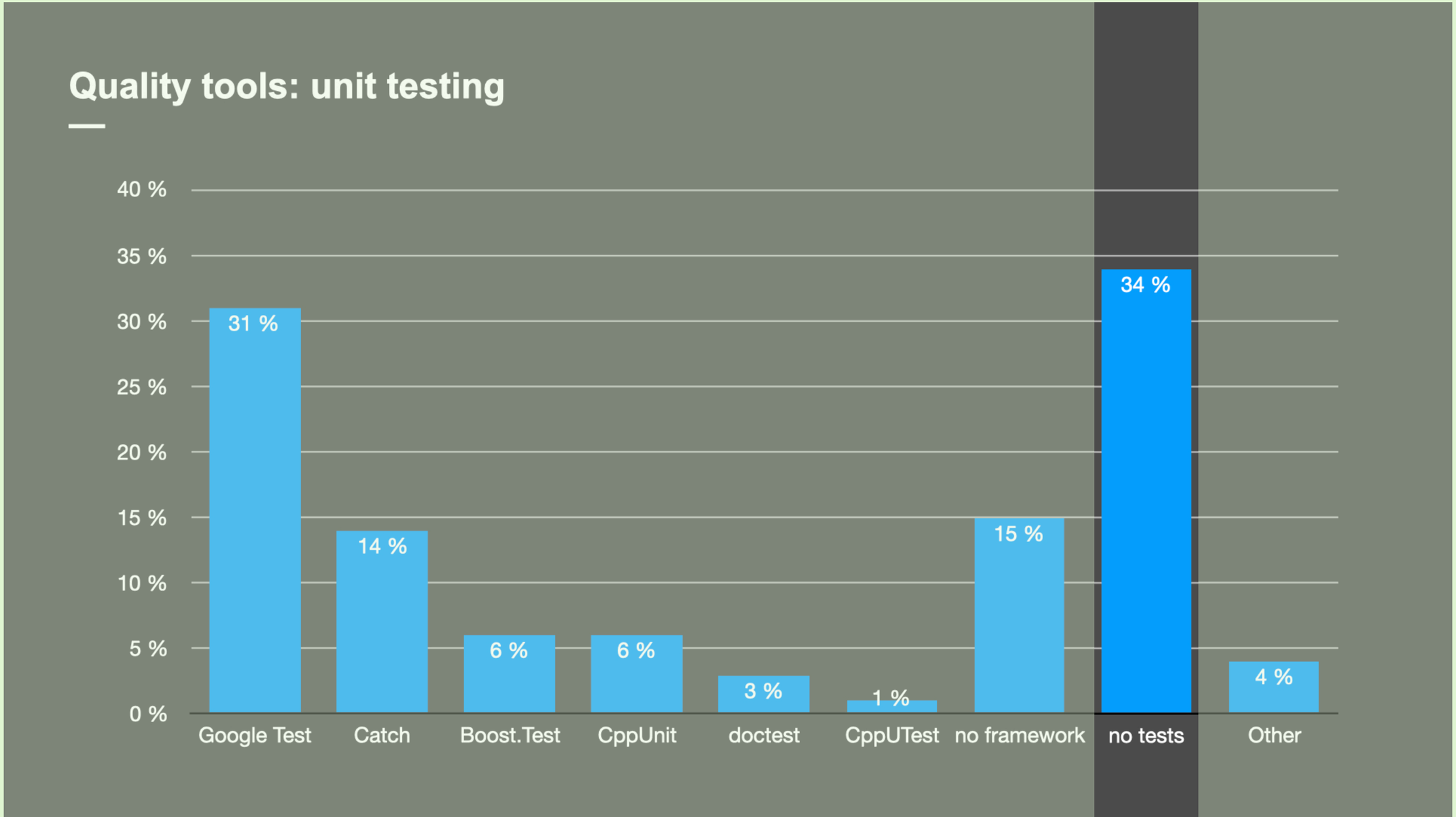
doctest

**UT**

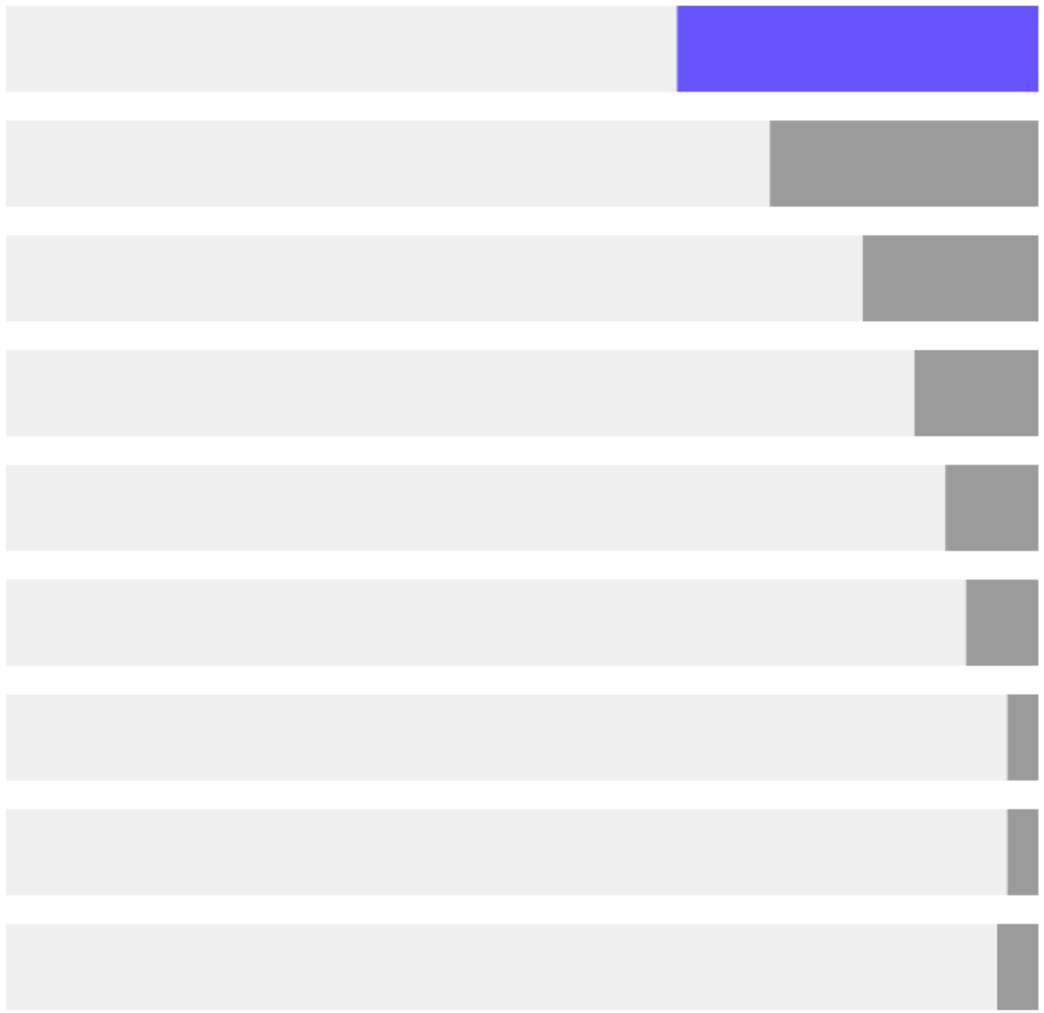
# JetBrains Developer Ecosystem report 2020



# JetBrains Developer Ecosystem report 2020



# JetBrains Developer Ecosystem report 2022



| | |
|---|---|
| **35%** | **Google Test** |
| 26% | I don’t write unit tests for C++ |
| 17% | I write unit tests but don’t use any frameworks |
| 12% | Catch |
| 9% | CppUnit |
| 7% | Boost.Test |
| 3% | CppUTest |
| 3% | doctest |
| 4% | Other |


Left-pad

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
#include "catch.hpp"

TEST_CASE("left pad") {
    /* .. */
}
```

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
#include "catch.hpp"

TEST_CASE("left pad") {
    /* .. */
}
```

```
#include "gtest/gtest.h"

TEST(left_pad, empty) {
    /* .. */
}
```

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
#include "catch.hpp"
TEST_CASE("left_pad pads strings to a minimum length") {

    /* .. */

}
```

```
#include "gtest/gtest.h"
TEST(left_pad_pads_strings_to_a_minimum_length, empty) {

    /* .. */

}
```

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
#include "catch.hpp"
TEST_CASE("left_pad pads strings to a minimum length") {
    REQUIRE(left_pad("", 0) == "");
}
```

```
#include "gtest/gtest.h"
TEST(left_pad_pads_strings_to_a_minimum_length,
     an_empty_string_with_zero_padding) {
    ASSERT_EQ(left_pad("", 0), "");
}
```

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
#include "catch.hpp"

TEST_CASE("left_pad pads strings to a minimum length") {

    REQUIRE(left_pad("", 0) == "");
    REQUIRE(left_pad("abc", 0) == "abc");

}
```

```
#include "gtest/gtest.h"

TEST(left_pad_pads_strings_to_a_minimum_length,
     not_padded_when_min_len_is_less_than_input_string_len) {

    ASSERT_EQ(left_pad("", 0), "");
    ASSERT_EQ(left_pad("abc", 0), "abc");

}
```

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
#include "catch.hpp"

TEST_CASE("left_pad pads strings to a minimum length") {

    SECTION("When padding is < string length, the existing string is returned") {

        REQUIRE(left_pad("", 0) == "");
        REQUIRE(left_pad("abc", 0) == "abc");
    }
}
```

```
#include "gtest/gtest.h"

TEST(left_pad_pads_strings_to_a_minimum_length,
     not_padded_when_min_len_is_less_than_input_string_len) {

    ASSERT_EQ(left_pad("", 0), "");
    ASSERT_EQ(left_pad("abc", 0), "abc");

}
```

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
#include "gtest/gtest.h"

TEST(left_pad_pads_strings_to_a_minimum_length,
     not_padded_when_min_len_is_less_than_input_string_len) {
   ASSERT_EQ(left_pad("", 0), "");
   ASSERT_EQ(left_pad("abc", 0), "abc");
}

TEST(left_pad_pads_strings_to_a_minimum_length,
     padded_when_min_len_equal_than_input_string_len) {
  ASSERT_EQ(left_pad("abc", 3), "abc");
}

TEST(left_pad_pads_strings_to_a_minimum_length,
     padded_when_min_len_greater_than_input_string_len) {
  ASSERT_EQ(left_pad("abc", 4), " abc");
}
TEST(left_pad_pads_strings_to_a_minimum_length,
    custom_pad_char_can_be_used) {
  ASSERT_EQ(left_pad("abc", 5, '*'), "**abc");
}
```

```
std::string left_pad(std::string const& str, size_t min_len, char pad_char=' ');
```

```
#include "catch.hpp"

TEST_CASE("left_pad pads strings to a minimum length") {
    SECTION("When padding is < string length, the existing string is returned") {
        REQUIRE(left_pad("", 0) == "");
        REQUIRE(left_pad("abc", 0) == "abc");
    }

    SECTION("When padding is == string length, the existing string is returned") {
        REQUIRE(left_pad("abc", 3) == "abc");
    }

    SECTION("When padding is > string length, the returned string is padded") {
        REQUIRE(left_pad("abc", 4) == " abc");
    }

    SECTION("String can be padded with a custom char") {
        REQUIRE(left_pad("abc", 5, '*') == "**abc");
    }
}
```

# Testing

## Best Practices

# Testing

## Best Practices

### Unit Tests should:

- Run fast
- Be deterministic & reproducible
- Be self-contained/ isolated
- Be consistent & reliable
- Be obvious & self-documenting
- Be focused on one thing (“single assert”)
- Use public interfaces

# Testing

## Best Practices

**All tests should:**

- Be well named
- Provide clear and useful feedback
- Have a clear reason
- Have low cyclomatic complexity

```
TEST_CASE( "Most recently used list" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

STATE EXPECTATIONS

# Unit Tests should be reproducible and deterministic

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

# Unit Tests should be reproducible and deterministic

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

"Arrange"

# Unit Tests should be reproducible and deterministic

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

"Arrange"

"Act"

# Unit Tests should be reproducible and deterministic

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

"Arrange"

"Act"

"ASSERT"

Tests should:

Have a single "Logical" Assert

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

Tests should:

Have a single "Logical" Assert

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

Single assert

Tests should:

Have a single "Logical" Assert

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;


    REQUIRE( list.empty() );
    REQUIRE( list.size() == 0 );

    list.add("item1");

    REQUIRE( list.empty() == false );
    REQUIRE( list.size() == 1 );

}
```

Dependency

Multiple asserts

# The Challenges & Pitfalls of Testing

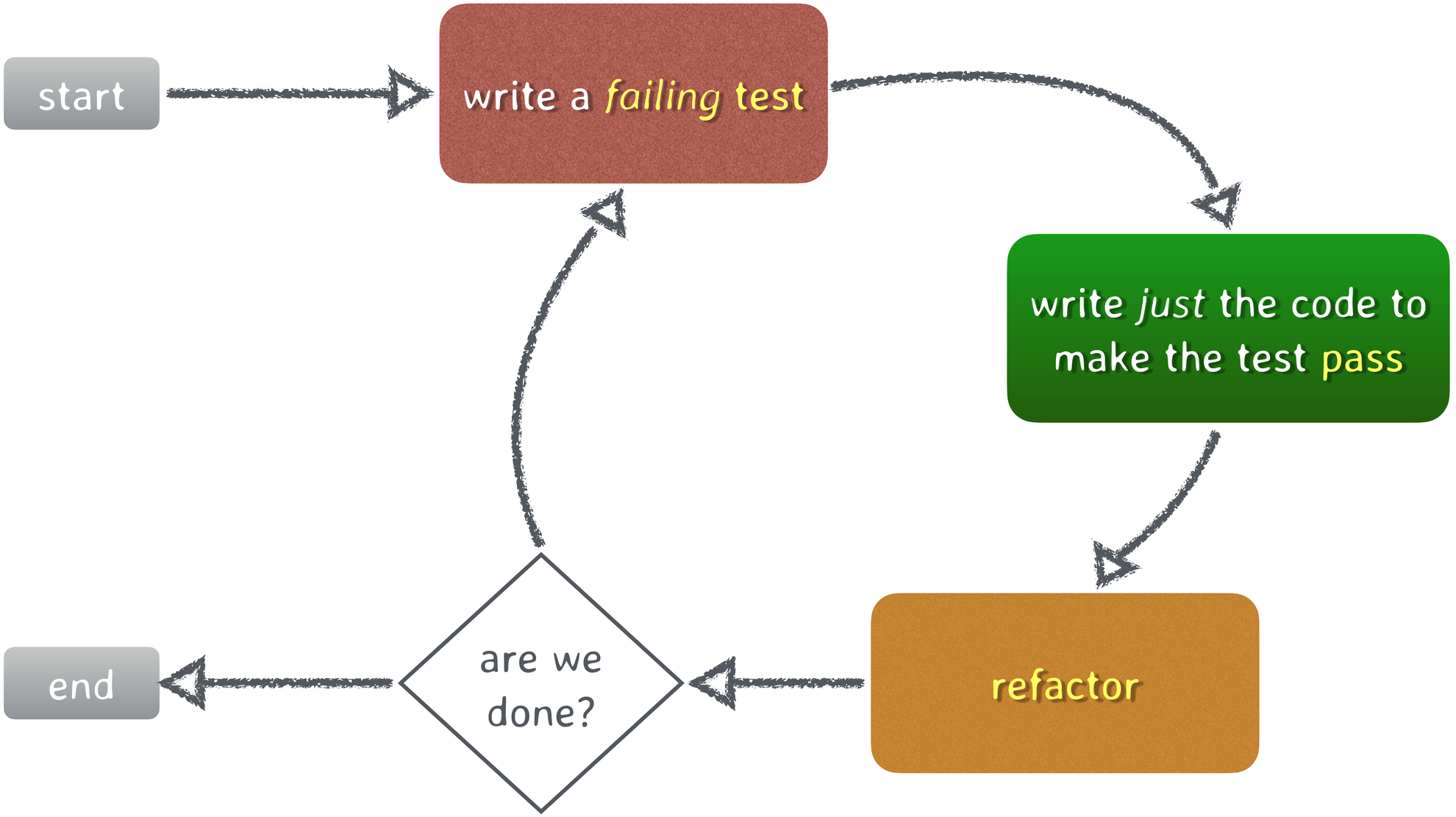
# The Challenges & Pitfalls of Testing

- Flakey tests
- Slow tests
- Brittle tests
- Hard to write

# What next?

# What next?

- Get comfortable with the best practices
- Become familiar with the design principles
- Pay attention to when testing is harder
- **Then** consider trying TDD


start
write a failing test
write just the code to make the test pass
refactor
are we done?
end


Design

# The SOLID principles

**SRP** Single Responsibility Principle

**OCP** Open-Closed Principle

**LSP** Liskov Substitution Principle

**ISP** Interface Segregation Principle

**DIP** Dependency Inversion Principle

Cppcon

The C++ Conference

# Breaking Dependencies: The SOLID Principles

Klaus Iglberger

2020

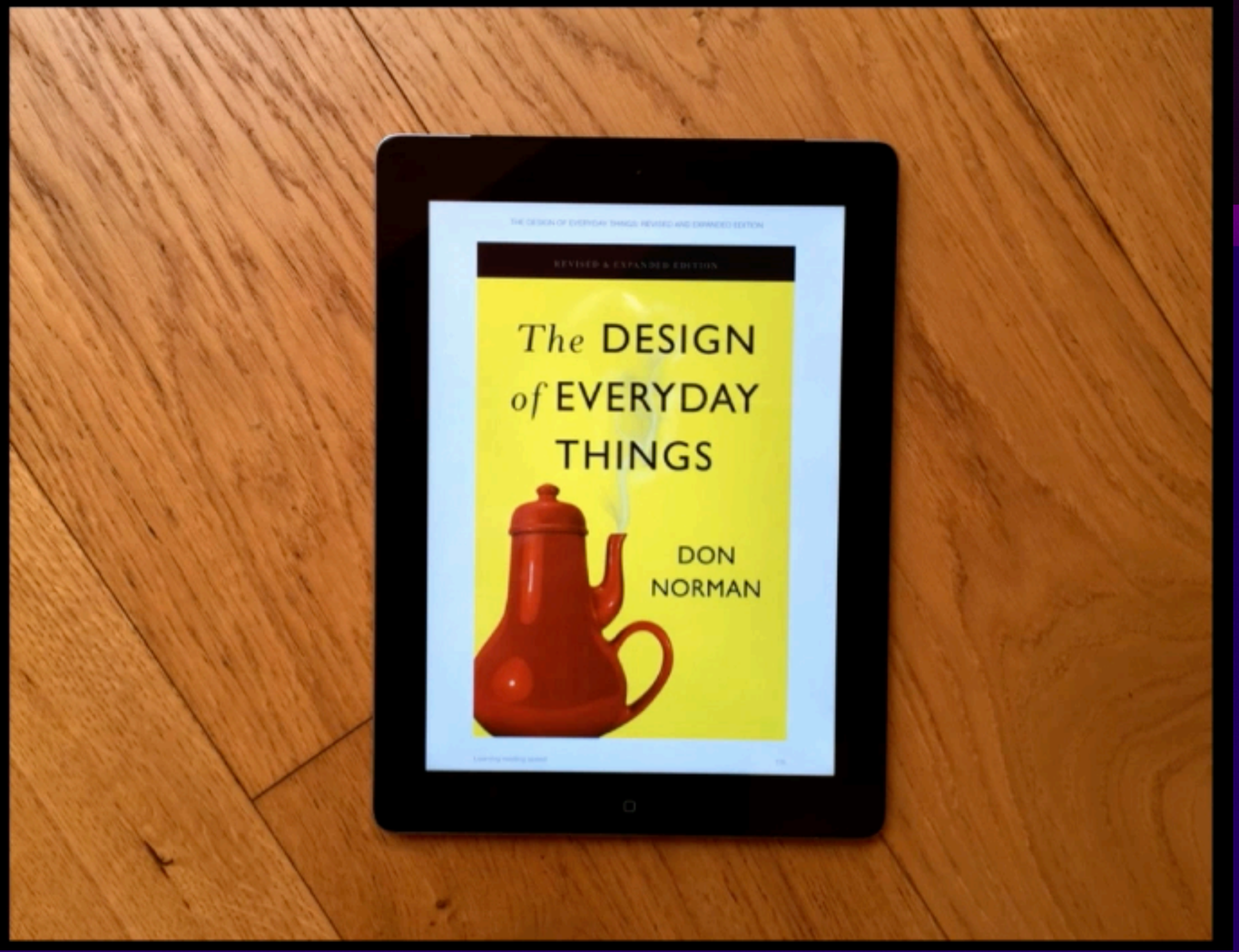
September 13-18

# Seeking Simplicity

**NDC** { London }

11-15 January 2016

Inspiring Developers
since 2008

# *Back To Basics*
# Testing

**levelofindirection.com**/refs/testing.html

**PHIL NASH**


@phil_nash

@mastodon@philnash.me