Testing Tests

Improving test methodology, step by step

About myself 🙋‍♂️

- Peter Hrenka, studied Computer Science and Mathematics in Tübingen
- Long time Linux user
- Currently Staff Engineer at Bosch Cross-Domain Computing Solutions
- Core Circle Member in the Bosch Developer Advocate Network
- Developing an embedded C++ library for Bosch Automotive (XC-AS)
  - Main features containers and math
- Started career as 3D Software Developer for a Finite Element Tool
- Hobbies
  - HAM Radio 📡 📻 Operator/Trainer
  - 🎹 Player

BOSCH

Scope of this talk

- Topics and examples are related my day-job
  - C++ Template library for Containers and Mathematics (VFC)
- The focus topic will be on unit tests
  - Some concepts may be applicable for other testing tasks
- I will be using Googletest for the examples
- This is not an overview of research but rather a practitioners view and opinion
  - I do not claim that anything is a new idea
  - Not everything I will present has been proven in use (yet)

BOSCH

Requirements for Testing

```
template<typename Iter>
void sort_sequence(Iter start, Iter stop)
{
    auto first = *start;
    for (auto it = start; it != stop; ++it) {
        *it = first++;
    }
}

TEST(sort, test)
{
    MyVector<int> vec { 5, 1, 8 };
    sort_sequence(vec.begin(), vec.end());
    EXPECT_TRUE(std::is_sorted(vec.begin(), vec.end()));
}
```

BOSCH

Requirements

- Naming matters!
- Requirements matter!!
- Every program is correct according to some specification:
  - "The function `sort_sequence` shall generate an ascending sequence in the given range where the generated sequence starts with the first element of the given sequence and the following elements are incremented by the value 1 each"
- No test without specification
  - Become aware of implicit assumptions
  - If a function contains "sort" it is expected to only perform a permutation of the original elements

First Break

- Yes, we can test template functions and classes templates
  - We have to clarify the requirements
  - We have to think about the runtime test data
  - We have to define the compile-time variations we want to test
- Basic testing patterns seem inadequate to the expressive power of templates
  - Copy & paste is not the answer
- More structure and clarity would be nice
- Guiding Question: How do I know when my testing is done?

Our requirements towards tests

- Easy to generalize
  - Reduce usage of "magic numbers"
- Easy to extend
  - Both in the runtime and in the compile-time dimension
- Easy to identify what is tested

"Templated Tests"

- Using `googletest` Link

```
#include <vector>
#include "gtest/gtest.h"

template <class T>
class BasicTypesTest : public testing::Test {};

using BasicTypes = testing::Types<char, int, float>;
TYPED_TEST_SUITE(BasicTypesTest, BasicTypes);

TYPED_TEST(BasicTypesTest, TestContainerConstruction)
{
    MyVector<TypeParam> vec{ TypeParam{1}, TypeParam{2}, TypeParam{3}};
    EXPECT_EQ(vec.size(), 3);
}
TYPED_TEST(BasicTypesTest, TestContainerAccess) {
    MyVector<TypeParam> vec{ TypeParam{1}, TypeParam{2}, TypeParam{3}};
    EXPECT_EQ(vec[0], TypeParam{1});
}
```

Issues with "Templated Tests"

- Heavy usage of structural macros
  - Hard to customize per test, need a new test suite for every combination
  - Sometimes bad interaction with C++ `namespace`s
- "Magic" name `TypeParam`
  - There can be only one template parameter
- Problems with modularity and extendability
  - Which macros can we put in the header and which in a `.cpp` file?
  - We want to be able to add more types in independent projects
- Where to handle runtime variations?

Our approach

- Avoid using the top-level macros
- Use the fixture base class `testing::test` directly
- Implement the missing infrastructure without the use of macros
- More control over instantiations

```cpp
/// container_test.hpp
#include "gtest/gtest.h"

namespace vector_tests {

template <class T>
class BasicTypesTest : public ::testing::Test {}; // no change

template<typename Type>
class TestContainerConstruction : public BasicTypesTest<Type>
{
  void TestBody() override { // interface of googletest
    MyVector<Type> vec{ Type{1}, Type{2}, Type{3}};
    EXPECT_EQ(vec.size(), 3);
  }
};

template<typename Type, int Size>
class TestContainerAccess : public BasicTypesTest<Type>
{
  void TestBody() override {
    MyVector<Type> vec{ Type{1}, Size};
    EXPECT_EQ(vec[Size-1], Type{1});
  }
};
}
```

```cpp
/// cpp
#include "container_test.hpp"

template<typename... Types>
struct Instantiate { /* call internal gtest APIs  to register tests*/ };

namespace vector_tests {

Instantiate<
  TestContainerConstruction<char>,
  TestContainerConstruction<int>,
  TestContainerConstruction<float>
> g_basicConstruction;

Instantiate<
  TestContainerAccess<char, 10>,
  TestContainerAccess<int, 10>,
  TestContainerAccess<float, 100>
> g_basicAccess;
}
```

Advantages of custom instantiations

- Instances can be grouped by topic of function
- more freedom to distribute test on separate compilation units
  - e.g. customer-specific files
- Complete freedom over number and kind of template parameters for test classes
- no issues with namespaces

Test Fixtures

- Typically, all the setup for the test can be done in the constructor of the fixture class
- Specific tests derive from the fixture and re-use the members

```
template <class Type>
struct TemplateFixture : public ::testing::Test {
  MyVector<Type> classes[4]{
    MyVector<Type>{},            // empty
    MyVector<Type>{0},           // one element
    MyVector<Type>(1, 100),      // many identical
    MyVector<Type>{2, 3, 5, 7}  // four primes
  };
};
```

BOSCH

The `EquivalenceClasses` pattern

```
/// container_test.hpp
template <class T>
class TemplateFixture : public ::testing::Test {
  using Type = T;

  struct EquivalenceClasses
  {
    EquivalenceClasses() { /* initialize members */ }
    T classes[];
  };
};

template<typename Type>
class TestContainerAccess : public TemplateFixture<Type>
{
  void TestBody() {
    using EquivalenceClasses = typename TestContainerAccess::EquivalenceClasses;
    EquivalenceClasses eq{};
    for (auto eqClass : eq.classes) { ... }
};
```

Advantages of equivalence class pattern

- Can be decoupled from test class infrastructure
- Can be instantiated multiple times in the same test function
- Can be instantiated as `const`

Example: Testing a copy assignment operator - Test

```
// Test class using equivalence classes
template<typename Type>
class TestCopyAssignment : public TemplateFixture<Type>
{
  void TestBody() {
    using EquivalenceClasses = typename TestCopyAssignment::EquivalenceClasses;
    const EquivalenceClasses eqSrc{};
    for (const auto& src : eqSrc.classes) {
      EquivalenceClasses eqDest{};  // re-construct for every src
      for (auto& dest : eqDest.classes) {
        dest = src;
        EXPECT_EQ(src.size(), dest.size());
        EXPECT_EQ(src, dest);
      }
    }
  }
};
```

Example: Testing a copy assignment operator - Instances

```
namespace container_tests {

// cpp file for basic data types
TestInstance<TestCopyAssignment<char>,
             TestCopyAssignment<int>,
             TestCopyAssignment<float>
            > g_basicCopyAssignment;
}
```

Testing Example

```
int add(int a, int b) { return a+b; }

TEST(addition, all)
{
    for (int i=0; i<std::numeric_limits<int>::max(); ++i)
    {
        int res{i};
        for (int j=0; j<std::numeric_limits<int>::max(); ++j)
        {
            EXPECT_EQ(add(i,j), res);
            ++res;  // parallel construction
        }
    }
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 143
  Killed - processing time exceeded
  Program terminated with signal: SIGKILL
  Running main() from gtest_main.cc
  [==========] Running 1 test from 1 test suite.
  [----------] Global test environment set-up.
  [----------] 1 test from addition
  [ RUN      ] addition.all
```

- Too slow for Compiler Explorer
  - Golden Rule of Compiler Explorer: If it does not run on Compiler Explorer, it is too long for a unit test.

BOSCH

Fundamental Limitation of Testing

- It is impossible to test everything
  - Some notable exceptions for functions with a limited parameter space (e.g. `square`)
  - Everything with more than 32 bit worth of data is too much
- Most of the time, we cannot test everything
- Incomplete tests can not prove the correctness
- Even if we could, it might not be a good idea
  - financial and environmental impact may not be justified
- Keep your unit tests in the single digit number of seconds

Choosing equivalence classes

- Equivalence classes can be
  - A representative object state, e.g. empty, one element, many elements
  - A combination of parameters to call a (member) function
  - A combination of both (object state and parameters)
- Prefer runtime variation over compile-time variation where possible
- For algorithms consider the special floating point values
  - ±0, ±∞, ± `NaN`

Choosing types

- For containers consider
  - Builtin primitive types
  - Types with different alignments
  - `const` qualified types
  - Types with deleted special member functions
    - Move-only types
    - Non-movable and non-copyable types
- For mathematical functions consider
  - Signed types
  - Floating points types
  - Non-primitive math objects, e.g. vectors, matrices, unit types

Break number two

- We have some nicer way of writing tests
  - Run-time vs. compile-time variations
- Are we finally done now?
- Seems like we're moving farther away
  - Not everything can be tested
  - Be careful not to test too much
- Conflicting goals
  - Do not take too much time
  - But test "as well as possible"

Coverage to the rescue?

- Coverage: Measure the code paths taken during testing
- Often measured as percentage of covered code lines relative to the total number of code lines
- Definitely useful!
- But: Coverage is often used as a measure to ensure that a test is "complete"
- A common interpretation:
  - 100% coverage $\rightarrow$ test complete / good

Problems with coverage in presence of templates

- Template specializations
- `enable_if` and other SFINAE constructs
- Compiler explorer as a ad-hoc coverage tool (only works for templates!)

```
void visited() {}

template<typename Type, int Capacity>
class Container
{
    Type mData[Capacity];
public:
    Container()
    {
        visited();
    }
};

template<typename Type>
class Container<Type,0>
{
public:
    Container()
    {
        visited();
    }
};


TEST(Container, constructor)
{
    Container<int,10> c_i10;
    Container<int,0> c_i0;
}
```

Interpretations of template coverage

There are two main interpretation of template-aware coverage:

1. The line must be covered for **all** template instantiations of a class
2. The line must be covered for **some** template instantiations of a class

(line can also be a statement or decision, depending on the type of coverage)

Interpretation 1 is very strong and quickly becomes unfeasible when metaprogramming techniques are used.

```cpp
template<typename Type>
Type square(Type val)
{
    return val*val;
}

TEST(square, coverage)
{
    square(0);
}
```

Is coverage the answer?

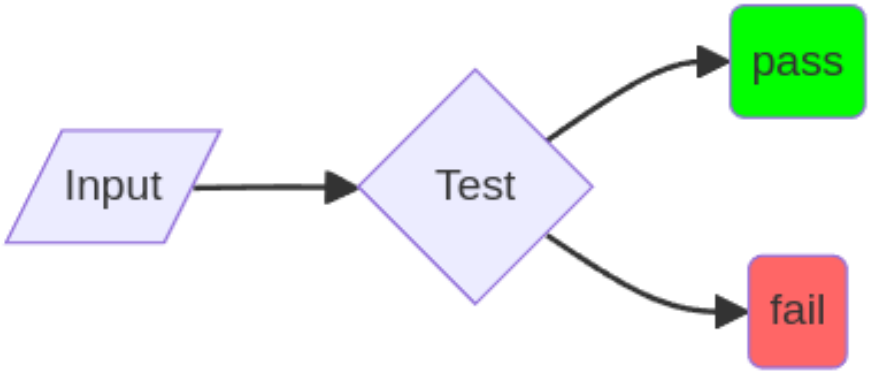
No.

Here the test does not even have a test macro that could verify an expectation.

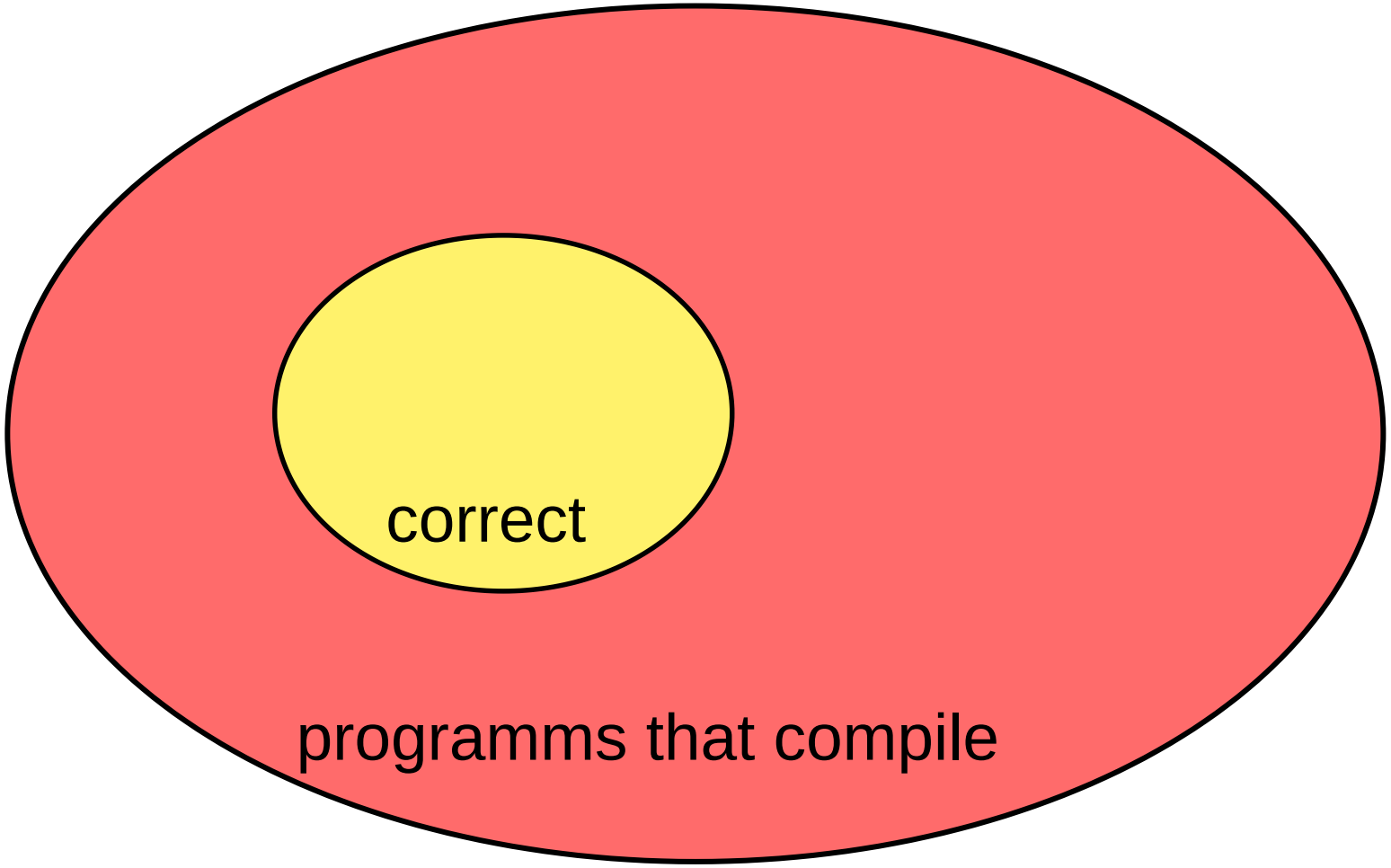
Coverage does not prove that the test is appropriate. Only that you have not accidentally missed anything.

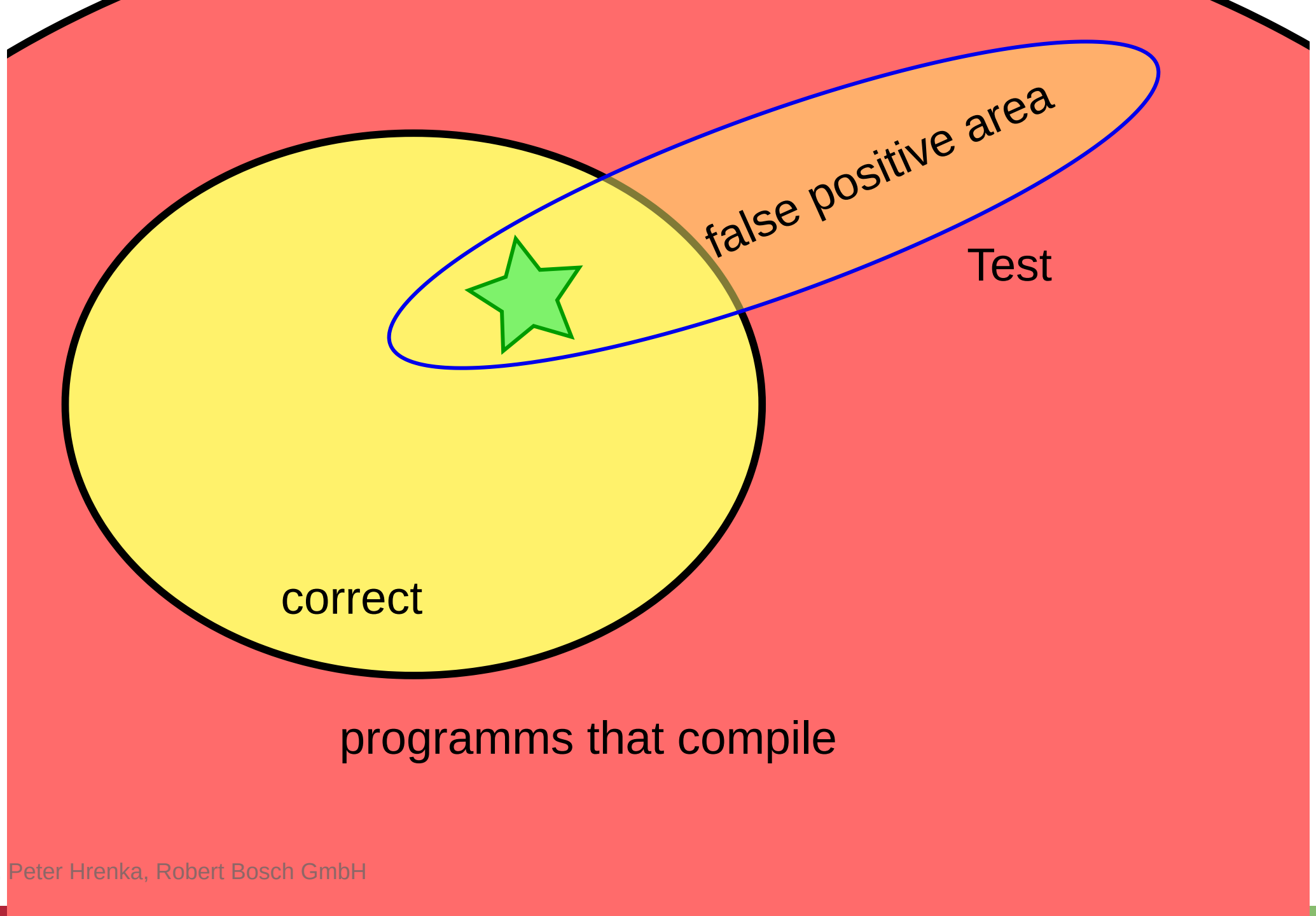
What is Testing, anyway?

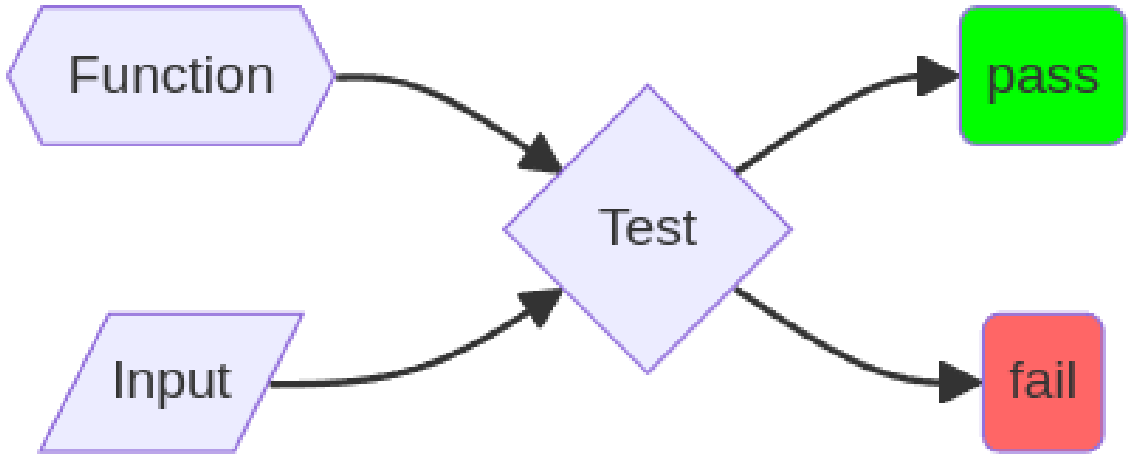
- High Level View: A test is a filter for correct programms
  - Good programs pass -> 🟢
  - Bad programs clog the filter -> 🔴
- Usual API
  - Success: Progress output, green colors, return code 0
  - Failure: Error output, expected and actual values, return code != 0



Space of all programs




false positive area
Test
correct
programms that compile



A generalized approach

- Consider the function under test to be a parameter to the test!
- Passing the correct function we expect "pass"
- Passing a defective function we expect a "fail"

"Short-selling" tests

- Provide obviously inadequate functions to the test expecting failure
- Only a test that can (provably) fail is a good test!

Good negative testing functions

- Empty functions
- Functions returning or writing constants
- Functions with a similar interface but different semantics
  - Drop or add parameters
  - Write small wrapper or mockups
- Old, buggy versions

BOSCH

Adversarial Testing

```
// Guard object to signal failure to detect
template<bool good>
struct AdversarialGuard;

template<>
struct AdversarialGuard<true> { // normal test mode
  void add(bool) {}
};

template<>
struct AdversarialGuard<false> {  // adversarial mode
  void add(bool pass) { if (!pass) ++numFail; }
  ~AdversarialGuard()
  {
      EXPECT_NE(numFail, 0);      // we must have at least one failure
  }
  int numFail{0};
};
```

Adversarial Testing Support Macros

Unfortunately, there is no infrastructure customizing googletest the way I need it, so let's just define some new macros:

```
#define ADV_EXPECT_TRUE(x)  if constexpr (!good) { guard.add(x); } else EXPECT_TRUE(x)
#define ADV_EXPECT_FALSE(x) if constexpr (!good) { guard.add(!x); } else EXPECT_FALSE(x)
// ... and everything else you need
```

- Note the lack a trailing semicolon so we can still use error output using `operator<<` as usual.

Example: Sorting

Test Data

```
template <class Type>
struct TemplateFixture : public ::testing::Test {
  struct EquivalenceClasses
  {
    MyVector<Type> classes[4]{
      MyVector<Type>{},            // empty
      MyVector<Type>{0},           // one element
      MyVector<Type>(1, 100),      // many identical
      MyVector<Type>{2, 3, 5, 7}  // four primes
    };
  };
};
```

Example: https://godbolt.org/z/eqocbEovn

Adversarial test for sorting

```
template<typename SortFunction, bool good, typename Type>
struct TestSort : public TemplateFixture<Type>
{
  void TestBody() {
    AdversarialGuard<good> guard;
    typename TestSort::EquivalenceClasses inputs;
    auto sorter = SortFunction{};
    for (auto& data : inputs.classes) {
        sorter(data.begin(), data.end());
        ADV_EXPECT_TRUE(std::is_sorted(data.begin(), data.end()));
    }
  }
};
```

Test function wrappers

```
struct Sort {
    template<typename Iter>
    void operator()(Iter begin, Iter end) {
        std::sort(begin, end);
    }
};

struct NonSort {
    template<typename Iter>
    void operator()(Iter begin, Iter end) { /* I am feeling lucky */ }
};

struct Reverse {
    template<typename Iter>
    void operator()(Iter begin, Iter end) {
        std::reverse(begin, end);
    }
};
```

Test instances

```
TestInstance<
  // normal tests
  TestSort<Sort, true, char>,
  TestSort<Sort, true, int>,
  // adversarial tests
  TestSort<NonSort, false, char>,
  TestSort<Reverse, false, char>
> g_sortInstances;
```

Oh no! A failure!

```
[----------] 1 test from TestSort<NonSort, false, char>
[ RUN      ] TestSort<NonSort, false, char>.
/app/example.cpp:35: Failure
Expected: (numFail) != (0), actual: 0 vs 0
[  FAILED  ] TestSort<NonSort, false, char>. (1 ms)
[----------] 1 test from TestSort<NonSort, false, char> (1 ms total)
```

Analysis of adversarial test failure

- No failures were recorded for the adversarial test case `NonSort`
- We would expect the test to be able to detect if a sort function does nothing
- Why was is not detected that nothing has changed?
- All inputs were already sorted 🤦
    - Easy to fix by adding unsorted equivalence class

Re-using old examples for profit

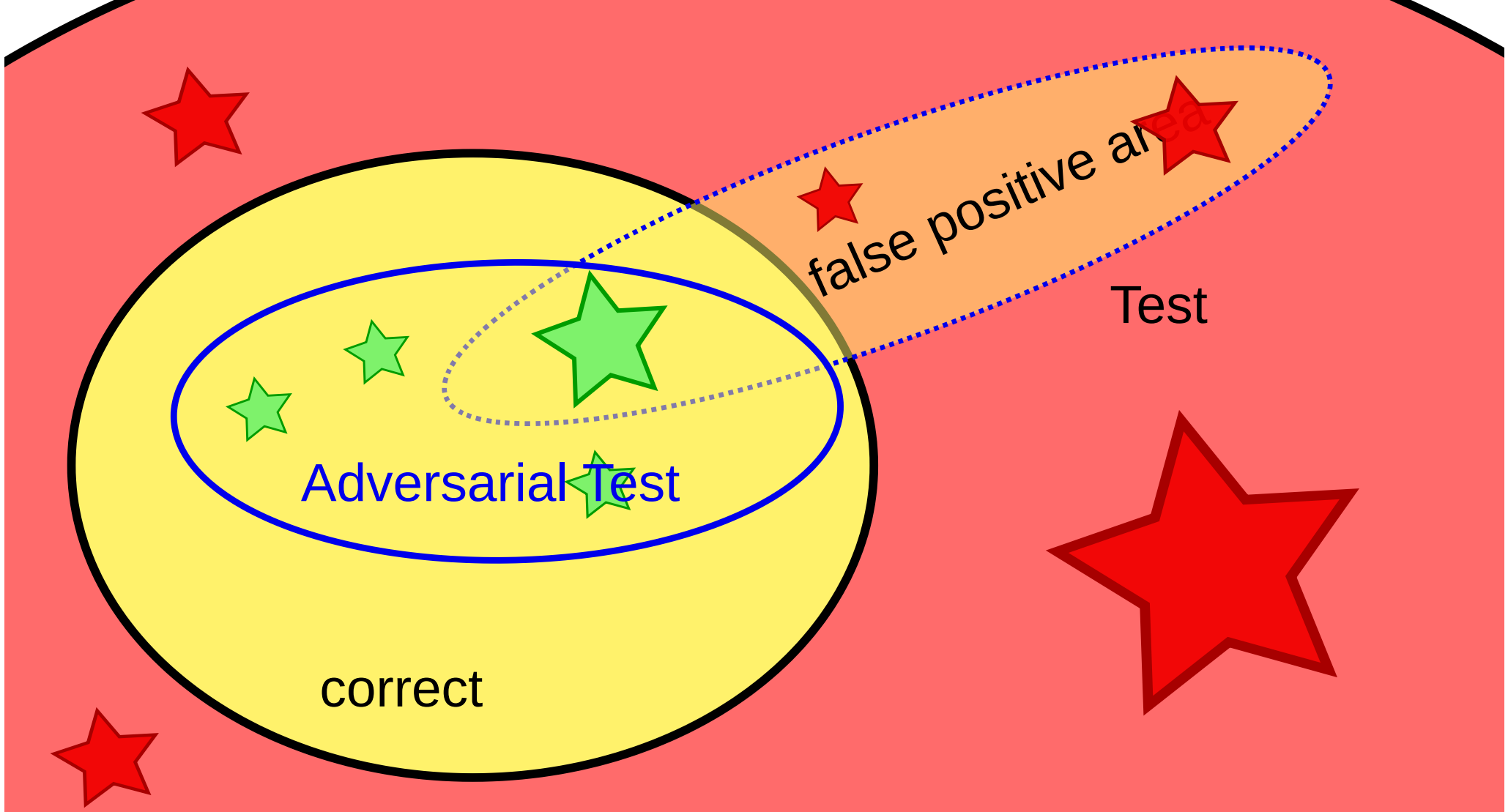
```
struct GenerateSequenceFromFirst
{
    template<typename Iter>
    void operator()(Iter start, Iter stop)
    {
        if (start == stop) return; // we have empty containers, so we need this
        auto first = *start;
        for (auto it = start+1; it != stop; ++it)
        {
            *it = first++;
        }
    }
};
```

Analyzing another adversarial test failure

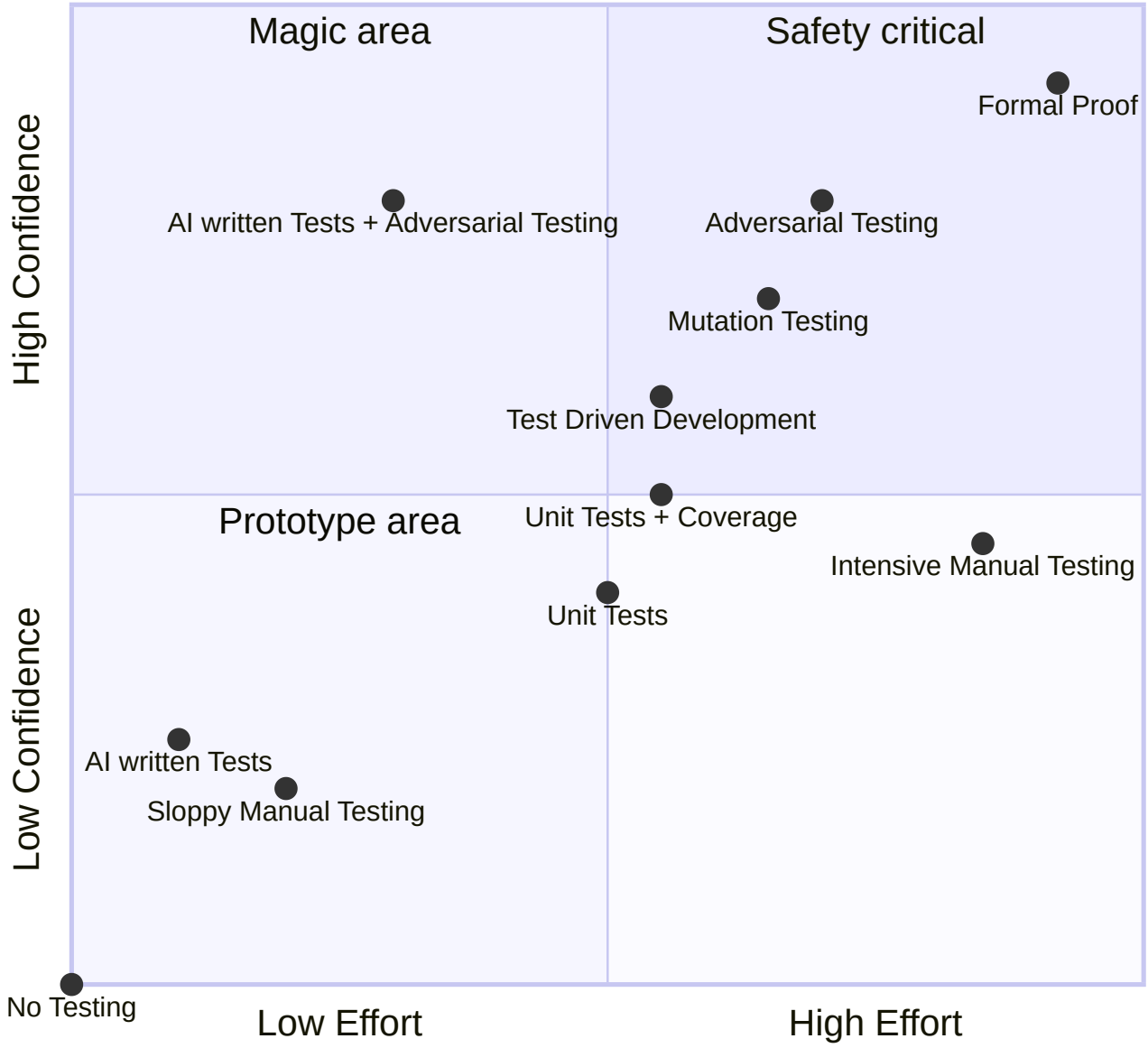
- The test does not check if the original items are present in the result

```
template<typename SortFunction, bool good, typename Type>
struct TestSort : public TemplateFixture<Type>
{
  void TestBody() {
    AdversarialGuard<good> guard;
    typename TestSort::EquivalenceClasses inputs;
    auto sorter = SortFunction{};
    for (auto& data : inputs.classes) {
        const auto copy = data;
        sorter(data.begin(), data.end());
        ADV_EXPECT_TRUE(std::is_sorted(data.begin(), data.end()));
        for (auto& oldItem : copy) {
            ADV_EXPECT_TRUE(std::find(data.begin(), data.end(), oldItem)!=data.end());
        }
    }
  }
};
```

Solution: https://godbolt.org/z/v9xz465Tq


false positive area
Test
Adversarial Test
correct
programms that compile

Testing Methodologies



How to test the test of the test?

- Hold-out set: Keep some specimens for later
  - specimens can be data sets or test functions (both good and bad)
- Mutation testing: Tweak the implementations
  - Cool if it can be automated, but changing an algorithm does not mean that it fails to fulfill the specification
  - Expected high manual effort for verification
- Time will tell: Higher level product metrics
- Double blind tests

Conclusion

- It is possible to separate equivalence class definition and testing code using a common test framework
  - only small additions are required
- The quality of the tests can be improved by providing adversarial implementations that must be detected by the test
  - Simple examples lead to significant improvement of test quality (and by extension of the code)
- High potential as a verification step of AI generated Testing Code

Thanks to

- Matt Godbolt for compiler explorer
- The MARP framework for a wonderful way to write presentations
- Tübix for providing this great conference
- The people attending ACCU who gave me inspiration and ideas for this talk
- The marvelous MOM team at Bosch
- Reza Ahmadi for teaching me a lot about testing and responsibility diffusion
- The vibrant DAN developer community at Bosch

BOSCH

Questions?

Backup slides

So you really want to know how to hack googletest

```
template< typename... TestClasses >
struct TestInstance {};

template< typename TestClass, typename... TestClasses >
struct TestInstance< TestClass, TestClasses... >
    :TestInstance<TestClasses...>
{
    TestInstance()
    {
        MakeAndRegisterTestInfo(
            testing::internal::GetTypeName<TestClass>().c_str(),
            "",
            nullptr,
            nullptr,
            testing::internal::CodeLocation("", 0),
            testing::internal::GetTypeId<TestClass>(),
            TestClass::SetUpTestCase,
            TestClass::TearDownTestCase,
            new testing::internal::TestFactoryImpl< TestClass > );
    }
};
```