Untested C++ Considered Harmful

Markus Lohmayer

Abstract

After a motivation of software testing in section 1, both from the perspective of science and engineering, a short overview of different types of testing methods is given in section 2. In section 3 the use of unit testing frameworks is motivated, followed by an outline of their basic structure in section 4. The C++ software library Catch2 is introduced as an example in section 5. The ideas behind Test Driven Development and Behavior Driven Development are briefly presented in section 6 and section 7, respectively. Section 8 deals with the question of how to write testable code. An emphasis is put on synergies with other goals of software engineering.

1. Motivation: Why do we need Software Testing?

1.1. Software Testing in the name of Science

The scientific method demands that rigorous tests are carried out to ensure that scientists actually measure what they seek to measure in a specific experiment. In computational science, the computer is the experimental apparatus and the algorithms running on it represent the methods used for conducting the experiment. Analogously, computational scientists need to design software tests which ensure that they are computing what they seek to compute in a specific computational experiment. Hence, one may say that software tests are an essential ingredient of the scientific method in the 21st century.

Automated tests not only lead to an increased level of confidence in the correctness of computational results. They also help to overcome today's reproducibility crisis (cf. fig. 1), given that scientists publish their source code (including the automated tests). Hopefully the spirit of open-source software development will soon carry over to the computational science

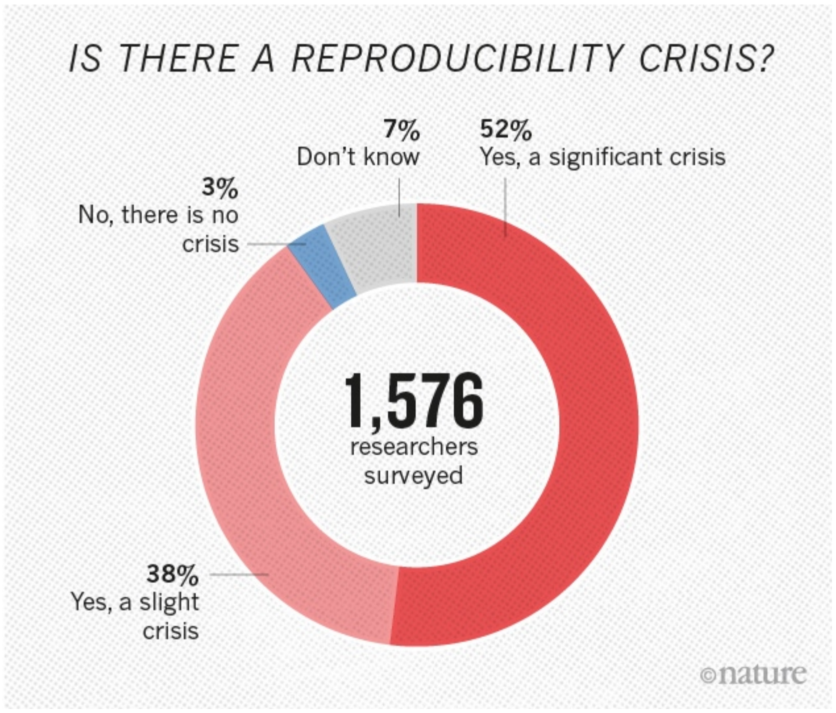

Figure 1: opinions of scientists on the current state of reproducibility [1]

community at large. If scientists invest sufficient time on improving code quality and increasing test coverage, it becomes much more likely that their work is reused and built upon (by other scientists) later on. Incentives to support such behavior should not be missing, since the long-term impact of scholarly work is paramount. The Journal of Open Source Software (JOSS) is a relatively young (* May 2016) project that shares exactly these goals.

1.2. Software Testing in the Name of Engineering

Automated tests are supposed to help programmers catch mistakes earlier in the development process. Hence, less time is probably spent on debugging and more time is spent on writing (test) code. Overall, an improved level of productivity should be the result, especially when the tests themselves are considered as a valuable by-product.

Beyond correctness, writing test code during the development process has other positive effects. In particular the following code qualities are expected to improve: Understandability, maintainability, flexibility and reusability. The overarching theme is a reduction of code dependencies.

2. Overview of Software Testing Methods

2.1. Static Methods

All testing methods that do not execute code are termed static. Code reviews and various kinds of static analysis tools belong to this category. Such tools can for example detect logical errors, syntax errors and formatting errors. As an example, the C++ linter tool clang-tidy may be mentioned.

2.2. Dynamic Methods

Dynamic testing methods on the other hand do execute code. These methods may be grouped into the categories on the following (non-exhaustive) list:

- Functional Tests
    - Sanity Tests
    - Unit Tests
    - Integration Tests
    - System Tests
    - others: Regression Tests, Smoke Tests, Installation Tests
- Non-functional Tests
    - Performance Tests, Usability Tests, Security Tests

In the sequel of this section the basic idea behind some of these terms shall be explained. The treatment here does not adhere to any existing standards.

Sanity Tests encode assumptions upon which the implementation of a certain function relies. These tests are interwoven with the code of the respective implementation. An example will be given at the beginning of the next section.

Unit Tests try to ensure the correct functioning of classes or free functions in isolation of other parts of the code base. These tests are small separate programs that each exercise and verify *one* specific aspect of the unit of code (class or free function) to which they pertain. Usually many short tests are necessary to ensure correctness of a specific unit of code. Therefore these tests are supposed to run fast. Input-output operations are hence usually avoided.

Integration Tests check the interoperability of different units of code. These tests are usually a bit more difficult to write.

System Tests verify if the entire system works together for a specific use case. Choosing an appropriate correctness measure is often not trivial.

Regression Tests are written whenever a bug is found. They pass once the bug is fixed and make sure that it doesn't creep back into the system later.

3. Why do we want to use a Testing Framework?

3.1. Sanity Tests

Sanity Tests are the most simple kind of tests, in the sense that they are *not* separate programs that exercise and verify certain aspects of the application. They are merely assertions interwoven with the application code. These assertions can be used to rule out obviously false results.

Example: function to compute a Rayleigh coefficient

```
#include <cassert>

float rayleigh_coefficient(Matrix A, Vector v)
{
  assert(
    A.shape[0] == A.shape[1] and
    A.shape[1] == v.shape[0]
  );
  return inner(v, inner(A, v)) / inner(v, v);
}
```

Sanity Tests are particularly useful for expressing the (otherwise implicit) assumptions on which a certain implementation rests upon. In that way they also serve as (additional) documentation. One advantage of assertions over exceptions is that they can be deactivated at compile time (by the `-DNDEBUG` compiler flag) in order to regain full performance.

3.2. Mere assertions are certainly not enough!

Consider the output of the above function, in case of failure:

```
Assertion failed: (A.shape[0] == A.shape[1] and
A.shape[1] == v.shape[0]), function
rayleigh_coefficient, file foo.cpp, line 6.
```

One major drawback of using the standard `assert` macro from `C` is that it does not report the values of `LHS` and `RHS` in e.g. `assert( LHS == RHS )`.

Beyond generating informative output for failures, a testing framework should support the following features:

- automated test discovery
- readable summary of all individual test results
- organization of tests (e.g. tags or test suites)
- sharing common setup code among different tests (e.g. fixtures)
- interoperability with continuous integration services

4. Architecture of xUnit Frameworks

The concept of a Testing Framework was introduced by Kent Beck. In 1989 he described SUnit for the Smalltalk programming language in an article [2]. Today many descendants exist: JUnit (for Java), CppUnit, Google Test (also for C++), unittest (for Python), pFUnit (for Fortran), …

These tools all share very similar architectural elements:

- Test Case: class that defines *one* particular test

- Test Runner: executable that finds and executes tests and reports results
- Test Fixture: class that defines setup (and teardown) code that can be used across multiple test cases
- Test Suite: class that groups together multiple test cases (which probably share one or more fixtures)

4.1. Structure of a Test Case

The structure of a test case is outlined in fig. 2. The System Under Test (SUT) is either a class or a free function which should be tested in some specific way. Various collaborators, i.e. instances of classes, may be necessary to run the test. These are termed the Dependent Other Components (DOCs). The work of instantiating one or multiple DOCs may be encapsulated by a fixture, which can then be used across multiple test cases, to avoid code duplication. In C++ teardown code should never be necessary because code should conform to the Resource Acquisition Is Initialization (RAII) principle.

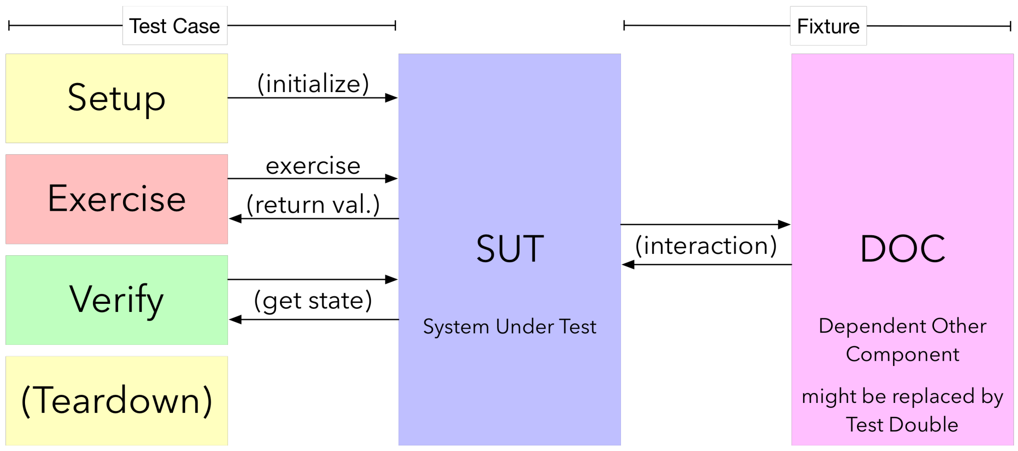

Figure 2: structure of a test case (cf. [7])

4.2. Test Doubles

If The DOCs themselves depend on other DOCs then it may be necessary to cut this possibly long chain of dependencies by introducing so-called

Test Doubles as DOCs. These are essentially objects specifically created for the purpose of testing, that are sufficiently similar to the actual DOCs but do themselves not depend on further DOCs.

There exist three different types of Test Doubles:

- Fake: a variant of the original DOC that has limited capabilities (e.g. a in-memory database instead of I/O and persistent storage)
- Stub: a replacement for the DOC that provides only few predefined answers to certain function calls required by the tests
- Mock: an object which holds and automatically verifies certain expectations.

Mocks are probably a bit harder to conceptualize. They essentially verify the behavior of the SUT w.r.t. the mocked DOC by encoding for instance a sequence of expected calls from the SUT to the mocked DOC. Software libraries (so-called mocking frameworks) exist to conveniently generate such objects. [4]

These definitions do not adhere to any (possibly existing) standard.

5. Introduction to a C++ Unit Testing Framework

This section gives a short introduction to the C++ testing framework Catch2. The software library features a declarative style for writing tests using macros. Per se, the use of macros is by no means beautiful but nevertheless a good solution for the problem at hand, given the absence of reflection capabilities in C++. It shall be mentioned here that Catch2 deviates a bit from the traditional xUnit style of writing tests.

Example: interface of the SUT (trapezoidal integrator)

```
#include <functional>

double trapz(
  std::function<double(double)> const & function,
  float lower_limit, float upper_limit, unsigned int bins
);
```

A test case implemented in Catch2 may take the following form:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "trapz.h"
#include <cmath>

TEST_CASE(
  "test trapz with derivative of atan function",
  "[integrators]"
)
{
  auto derivative = [](double x){ return 1 / (1 + x*x); };
  auto result = trapz(derivative, 0, 5, 500);
  REQUIRE(
    result == Approx( std::atan(5) ).epsilon(1e-4)
  );
}
```

The first line tells Catch2 to generate a main function, that serves as the test runner. As can be seen in the second line, a single header file suffices for using the library.

The first string after the `TEST_CASE` macro names the test case and the second string can be used to specify tags, which can then be used to selectively run tests that have certain tags. Tag names are always enclosed in square brackets. If one of the tags is `[!hide]` then the test is not executed by default. To run the test either its name or one of its tags must be specified as a command line argument of the test executable.

Two different macros are available to write assertions: `REQUIRE` stops execution of the test case after the first failure, whereas `CHECK` keeps the test case running. Specific assertion commands for different comparison operators are not required.

There exist various command line options when running the generated test executable: e.g. `-s` shows output also for successful tests. `-l` shows all available tests and `-t` shows all available tags. A complete list of options is displayed when the `-h` flag is used.

The above example comprises only of one single test case. In general a file can contain multiple test cases and each test case may have several nested sections.

Instead of relying on traditional class-based fixtures, nested sections are the preferred way to reuse setup code in Catch2. Sections carry a name just like test cases, but they do not carry tags. Code at the test-case level is executed once for each nested section. Section 7 shows a related example.

Remark: If a `TEST_CASE` block has multiple nested `SECTION` blocks then, in terms of the xUnit nomenclature, each `SECTION` block corresponds to a test case and the outer `TEST_CASE` block corresponds to a test suite that groups those test cases.

6. Test Driven Development

The main idea of Test Driven Development (TDD) is to always write unit tests before writing the corresponding application code. The resulting development cycle is outlined in fig. 3.

TDD leads to the following benefits:

- the process of writing tests drives the design of the application code in a positive way
- ideally all requirements / specifications are captured by tests
- rumors exist that the TDD cycle leads to improved productivity

Note however that an abundance of unit tests can lead to a false sense of security in particular if no or very few integration- and system-level tests are in place.

If tests are written first, then clearly testable code must be the result of the development process. The meaning behind the first item above will

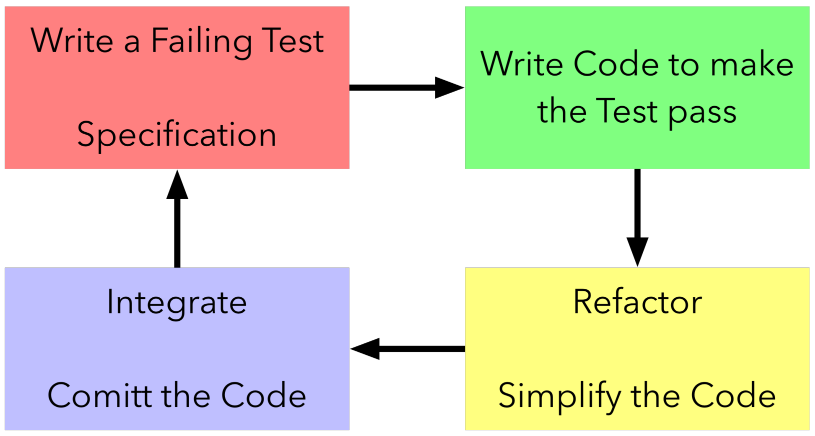

Figure 3: TDD development cycle (cf. [5])

become clear in section 8, which discusses synergies between testability and other desirable code qualities.

Take-away message: TDD must not be practiced religiously but always writing a good number of unit tests for every new unit of code before consuming its API elsewhere makes perfect sense! This way lots of problems that stem from avoidable code dependencies can be solved early in the development process when making changes is cheap.

7. Behavior Driven Development

The idea of Behavior Driven Development (BDD), as conceived by Dan North [9], is that tests might as well be thought of as executable specifications of program behavior.

Catch2 offers four macros to support this style of writing tests: `SCENARIO`, `GIVEN`, `WHEN` and `THEN` are used instead of `TEST_CASE` and `SECTION`.

A BDD-style test case then takes on the following form:

```
SCENARIO( "some name for the test", "[tag1][tag2]" )
{
  GIVEN( "some preconditions" )
  {
    /* setup something here ; note that this "fixture"
       is executed once before each WHEN block */

    REQUIRE( /* maybe check if preconditions are met */ );

    WHEN( "something is done" )
    {
      /* exercise the SUT in some way here */

      THEN( "some result is expected" )
      {
          REQUIRE( /* verify  */ );
      }
    }
    WHEN( "something else is done" )
    {
      /* exercise the SUT in some other way here */

      THEN( "another result is expected" )
      {
          REQUIRE( /* verify  */ );
      }
    }
  }
}
```

8. Writing Testable Code

8.1. Preliminary Overview of some Terminology

Low Coupling means that *different units* of code are largely independent of each other. In particular no forced dependencies exist, which makes it possible to replace one implementation of a component by another one.

High Cohesion means that elements within *one unit* of code belong together strongly. For example if some data members are accessed only by relatively few member functions then that class lacks cohesion.

The Single Responsibility Principle (SRP) dictates that one unit of code should focus only on a single concern [6]. It is the "S" in the SOLID design principles. Big units of code probably violate this principle.

The Single Level of Abstraction Principle (SLAP) commands that code operating on different levels of abstraction should not be mixed within one unit of code [6]. If this principle is violated then understanding the code requires mental (re-)construction of the abstractions that were not made explicit.

The Law of Demeter suggests that a class should communicate only with its direct dependencies. Formal statement:
Object `O` has member function `m`, then `m` only invokes:

- other member functions of `O`
- public member functions of data members of `O`
- member functions of objects that are passed as parameters of `m`
- functions on local objects created within `m`

Train wrecks like `function_argument.getX().getY().doZ()` violate this principle. [6]

Dependency Injection (DI) means that the responsibilities of *using* and *constructing* an object as a DOC are separated [6]. If a class `C` depends on another class `D` then `D` is not constructed by `C` but by another object dedicated to the task of object construction. This object then passes a reference to the instance of `D` as an argument when calling the constructor of `C`.

8.2. Seams

In his book "Working Effectively with Legacy Code" Michael Feathers coined the term "code seam": "A seam is a place where you can alter behavior in your program without editing in that place.” [3]

Modern and easily testable code has seams (i.e. test-enabling points) all over the place. Low coupling and high composability make it easy to replace components with test doubles. On the contrary, legacy code usually comes without any tests and it can be quite hard to find or introduce seams. A major topic of the aforementioned book is the question of how to create such test-enabling points in a legacy codebase. If certain dependencies are removed (i.e. seams are introduced) by making some small and hopefully safe modifications, tests can be written for the respective unit of code. Once enough tests are in place, larger modifications of the legacy system can be carried out with much higher confidence and even refactoring the codebase might become a reasonable goal.

Dependency Injection is the way how seams are naturally created. For legacy code on the other hand less straightforward techniques might unfortunately be necessary to introduce seams:

- Extract Interface Refactoring: based on dynamic polymorphism; causes virtual-function-lookup overhead
- Compile / Template Seam: extract a template parameter, with the original DOC as the default template parameter
- Link Seam: mainly used for replacing libraries with test doubles
- Preprocessor Seam

8.3. Lean Constructors

In the worst case a class instantiates its collaborators (DOCs) in its constructor. This leads to a deceptive API and forced dependencies, which is detrimental to understandability, testability, composability, flexibility and reusability.
Instantiating objects (either as SUT or DOC) is an ubiquitous task in testing. Therefore constructors should not do any work beyond assigning all *injected* dependencies to their respective fields (cf. SRP and DI). If any extra work is done, then chances are high that this work is unnecessary for the execution of most tests, leading to high overhead and thus slow

tests. If doing this work entails further dependencies then the situation gets even worse in terms of execution speed and also writing test code becomes unnecessarily complicated.

8.4. Lean Functions and Classes

Not only constructors but all units of code should adhere to the SRP and SLAP. If these principles are violated then

- important interactions are hidden inside a unit of code
- important seams for testing are missing
- the code becomes less easy to understand by just looking at the class definitions / function signatures, which reduces developer productivity
- code flexibility and reusability are severely limited because of lacking composability

Splitting code into many composable functions does not necessarily incur a performance penalty. In many cases the compiler will use function inlining and thus no function-call overhead will result. Having small units of code might also help the compiler to better optimize the code. Last but not least, it should be pointed out that usually only very few hotspots in the overall codebase are performance critical. The computer scientist Donald Knuth once said: "The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil in programming". Achieving low coupling usually is a much higher goal than minimizing function-call overhead.

When encapsulating functionality developers often have the choice between a free function and a member function. Free functions should always be preferred because they lead to less coupling and more flexible code. If a function does not need access to private data members then it should not have it. Member functions implicitly demand a valid `this` pointer. Requiring `this` as an unused argument might severely limit the achievable performance.
Good object-oriented design means to optimize for high cohesion, i.e.

to implement a minimal set of member functions which can efficiently perform all required operations on the data members. The purpose of a class is not to group together code but to achieve encapsulation of data only.

8.5. Object Graph Construction

It has been pointed out already, that all collaborators (DOCs) of a class must be passed as constructor arguments (DI). This makes code composable and dependencies become explicit through the API. It also makes code testable because certain DOCs can be replaced by test doubles. This is necessary if a DOC would cause too much overhead for a test or if a DOC depends on too many (transitive) DOCs and thus instantiation of a large object graph would be required to run a single test.

There exist three kinds of classes (and responsibilities):

- Service Objects (application logic)
- Value Objects (non-primitive data types)
- Factories, Builders, …(object graph construction)

Service Objects require references to all their collaborators as constructor arguments and do no instantiate other service objects themselves.

Value Objects encapsulate data and contain no application logic. They are instantiated in-line with application logic (just like primitive data types).

Factories, Builders, … are special objects for instantiating and wiring up a group of service objects. More information on so-called creational design patterns can be found in [8]. Objects responsible for object graph creation must not necessarily follow any particular design pattern like e.g. the factory pattern.
Higher modularity and composability means ending up with many smaller objects that use DI. This goes hand in hand with an increased need for wiring things up again. An interesting solution to save on boilerplate code is to use a DI framework such as [Boost].DI to inject dependencies.

8.6. Global State

Singletons and global variables lead to various adverse effects:

- singletons are forced dependencies and cannot be replaced by test doubles
- lower flexibility and reusability
- bad understandability and thus also maintainability
- no (safe) parallel execution of tests
- execution order of tests may matter

Note that singletons and all their internals (internals (internals (...))) are global state, i.e. global state is transitive in this sense.

In few cases global state may be acceptable, in particular if

- it is is an immutable object
- it is a primitive variable
- information flows only in one direction (e.g. in case of a logger)

8.7. Law of Demeter

The Law of Demeter, which should just be called the Guideline of Demeter, suggests that functions should ask exactly for what they need. Dependency Injection should not happen via any intermediary objects, as this would unnecessarily increase the coupling of components.

Example: Incomplete logging component

```
class Logger
{
  private:
    LogLevel level;

  public:
    Logger()
    : level( app_configuration_singleton.getLogLevel() ) {}
    Logger(AppConfigurationDatabase app_config_db)
    : level( app_config_db.getLogLevel() ) {}

    Logger(LogLevel level)
    : level(level) {}
};
```

The first constructor relies on global state and is therefore a very bad choice. The second try is better because it leads to weaker coupling. The third constructor is the best choice because it directly asks for what it needs and thus leads to minimal dependencies between the logging component and the rest of the codebase.
A common exception to the Guideline of Demeter are domain specific languages (DSLs). For examples consider the fluent builder pattern [8] or the DSL used by [Boost].DI.

8.8. Design for Testability and Synergies

Summing up, testable code

- is modular
- is highly composable
- has low coupling and high cohesion
- does not make unnecessary assumptions on how the code is used
- is more flexible
- has much higher chances to be reused
- does not have a deceptive API
- leads to increased productivity
- is easier to maintain

References

[1] M. Baker, *1,500 scientists lift the lid on Reproducibility*. `https://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970`, 2016.

[2] K. Beck, *Simple Smalltalk Testing: With patterns*. `http://swing.fit.cvut.cz/projects/stx/doc/online/english/tools/misc/testfram.htm`, 1989.

[3] M. Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2004.

[4] M. Fowler, *Mocks aren't Stubs*. `https://martinfowler.com/articles/mocksArentStubs.html`, 2007.

[5] J. Langr, *Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better*, O'Reilly UK Ltd, 2013.

[6] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 ed., 2008.

[7] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison Wesley, 2007.

[8] D. Nesteruk, *Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design*, Apress, 2018.

[9] D. North, *Introducing BDD*. `https://dannorth.net/introducing-bdd/`, 2006.