Software Testing

What is Software Testing?

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. - Wikipedia

- Does the software meet the specified requirement?
- Does it respond timely?
- Can it be installed and run on all intended environments?
- Does it produce the expected output on all kinds of input?

Why test software?

A study in 2002 conducted by NIST conluded:

- software bugs cost the U.S. economy ~$60 billion every year
- ⅓ of the cost could be avoided with better testing

Time introduced	**Time detected**				
	Requirements	Design	Programming	Testing	Release
Requirements	1x	3x	5-10x	10x	10-100x
Design		1x	10x	15x	25-100x
Programming			1x	10x	10-25x

Testing Levels

Unit tests

Unit tests seek to verify intended functionality of a specific section of code. It is typically applied at the function- or class-level.

These tests are usually written by the developer of the function. There can be multiple test cases within a single unit test to check for corner cases or branches (if/else) in the code.

Don’t guarantee software correctness alone, but verify that the basic building blocks work.

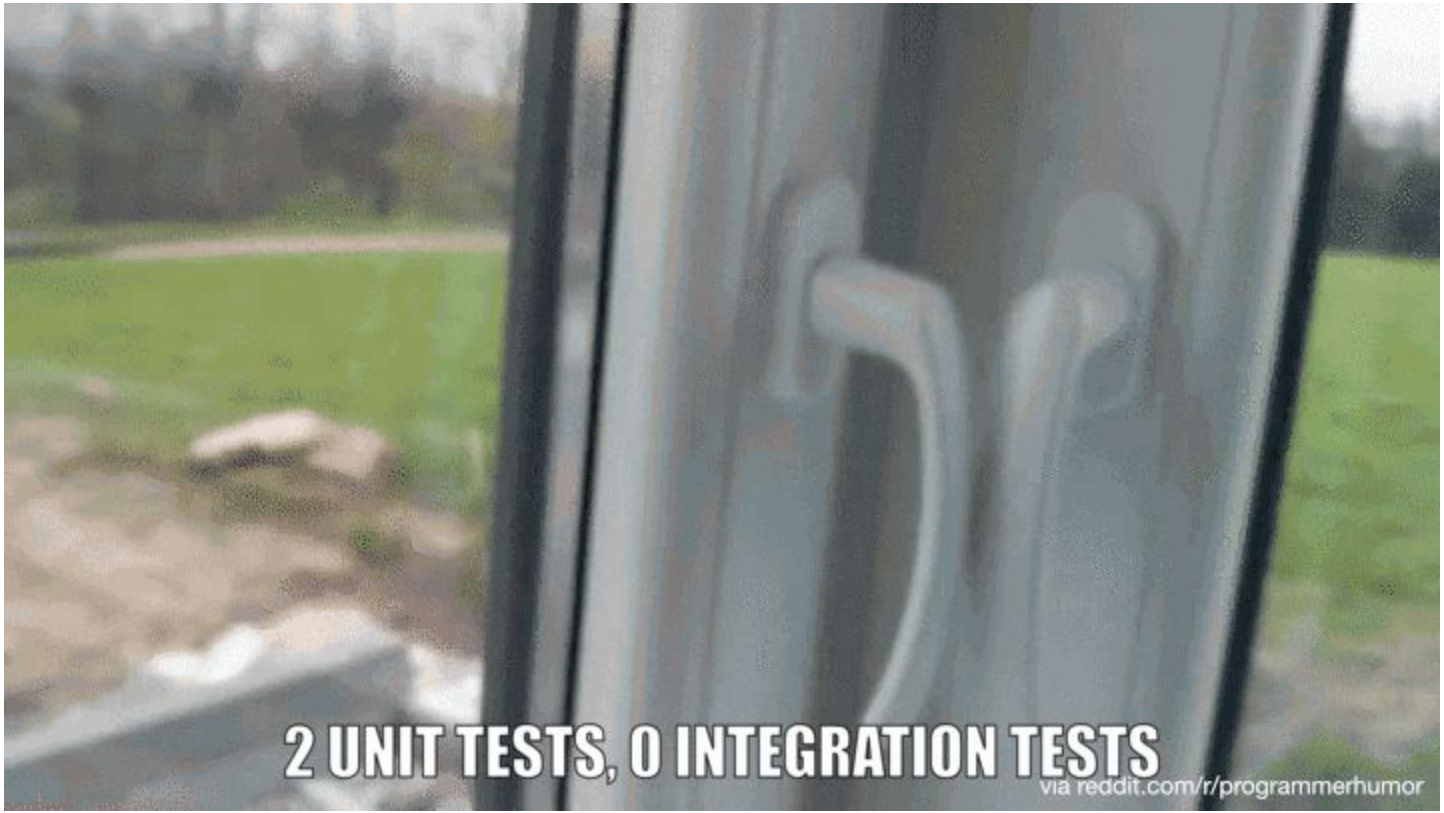
Integration tests

Integration tests are performed to validate the interface(s) between software components.

It is good practice to test components against each other in pairs. It is also possible to stick everything together at once (*big bang*), this however makes it harder to locate eventual issues.

Usually applied in a hierarchical fashion, plugging together ever larger groups of components until the whole system has been integrated.

Optionally finished with a system test, which checks the design requirements from the start against the finished software.


2 UNIT TESTS, 0 INTEGRATION TESTS
via reddit.com/r/programmerhumor

Testing Tools

Unit testing with CMake

- enable_testing()
  Enables the add_test() command for the current and all subdirectories.
- add_test(NAME <name> COMMAND <command> [<arguments>])
  Adds a test with a given name to the test script of the current directory.
  Tests added this way can be run by issuing …
- ctest [-C <config>] [--verbose]
  Calls all tests added using add_test. Enables monitoring and archiving of test suites, and much more. We will stick to the simple call and its direct results.

On Makefile-based CMake Generators, enable_testing() adds a target *test* that can be built like any other, i.e. `make test`.
Calling ctest is preferred, though. See next slide.

Continuous Integration (CI) using GitLab/Github

Run all (affected) tests on every commit. This helps to detect bugs earlier and keep the code base clean. Reviewers of merge requests can easily reject commits that did not pass.

CI is controlled via a simple YAML script.

- Creates stages of your build and test setup.
- Creates concrete jobs and assigns them to stages.
- Jobs are run in parallel (if possible).

You will receive an email whenever a commit fails.

Example file: .gitlab-ci.yml

```
image: rikorose/gcc-cmake
stages:
  - build
  - test
cache:
  paths:
    - "build/*"
# job build
build:
  stage: build
  script:
    - mkdir -p build
    - cd build
    - cmake ..
    - cmake --build .
```

Example file: .gitlab-ci.yml

```
# job test
test:
  stage: test
  script:
    - cd build
    - ctest --verbose
```

Catch - Test-Driven Development in C++

Catch^2 by Phil Nash

- C++
- Automated
- Test
- Cases
- Header-only (single file)

Also supports Objective-C, for those of you developing for Mac OS.

Catch^2 - Key Features

- single header file → easy to get started: download the file, and you're ready to go
- no external dependencies (besides a C++11 conformant compiler)
- only a single assertion macro for comparisons - expressions decomposition at compile time with meaningful, rich error messaging and logging
- Floating point approximate comparisons

Example

```
#define CATCH_CONFIG_MAIN
#include <catch.hpp>
#include <stdexcept>

int factorial(int n) {
	if (n < 0) throw std::invalid_argument("n must be non-negative");
	return (0 == n) ? 1 : n * factorial(n - 1);
}

TEST_CASE("Computing factorial of integers", "[factorial]") {
	REQUIRE(factorial(3) == 6);
}
```

Basics

- #define CATCH_CONFIG_MAIN must be set before including the header
  This way catch takes care of defining a main function that calls all our tests.
- test cases are created using the TEST_CASE() macro.
  It takes two strings: the first being a free form descriptive name, the second a set of tags, e.g. “[a][b][c]”
  It is possible to only run tags with a certain flag later on.
- the REQUIRE macro takes boolean expressions that have to be true in order for the TEST_CASE to succeed
- TEST_CASES can be grouped in sections

Catch-Sections

```
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( "resizing smaller changes size but not capacity" ) {
        v.resize( 0 );

        REQUIRE( v.size() == 0 );
        REQUIRE( v.capacity() >= 5 );
    }
}
```

BDD-style test cases

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
   GIVEN( "A vector with some items" ) {
      std::vector<int> v( 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 );
         }
      }
}
```

BDD-output

Scenario: vectors can be sized and resized

Given: A vector with some items

When: more capacity is reserved

Then: the capacity changes but not the size

Many more features

For all features consult the reference, if needed.