OOP (C++): Testing

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania
dlucanu@info.uaic.ro

Object Oriented Programming 2019/2020

Plan

A Giants's Dialog

From SOFTWARE ENGINEERING TECHNIQUES Report on a conference sponsored by the NATO SCIENCE COMMITTEE Rome, Italy, 27th to 31st October 1969:

Hoare: One can construct convincing proofs quite readily of the ultimate futility of exhaustive testing of a program and even of testing by sampling. So how can one proceed? The role of testing, in theory, is to establish the base propositions of an inductive proof. You should convince yourself, or other people, as firmly as possible that if the program works a certain number of times on specified data, then it will always work on any data.

. . .

Dijkstra: Testing shows the presence, not the absence of bugs.

http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF

On TDD

“One of the primary reasons I switched to TDD is for improved test coverage, which leads to 40%-80% fewer bugs in production. This is my favorite benefit of TDD. It’s like a giant weight lifting off your shoulders.”

(Eric Elliott, TDD Changed My Life)

Test-driven development (TDD): Definition

“Test-driven development” refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

It can be succinctly described by the following set of rules:

- write a “single” unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write “just enough” code, the simplest possible, to make the test pass
- “refactor” the code until it conforms to the simplicity criteria
- repeat, “accumulating” unit tests over time

`https://www.agilealliance.org/`

TDD: Expected Benefits

- many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort
- the same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases
- although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of "internal" or technical quality, for instance improving the metrics of cohesion and coupling

`https://www.agilealliance.org/`

TDD: Common Pitfalls 1/2

Typical individual mistakes include:

- forgetting to run tests frequently
- writing too many tests at once
- writing tests that are too large or coarse-grained
- writing overly trivial tests, for instance omitting assertions
- writing tests for trivial code, for instance accessors

https://www.agilealliance.org/

TDD: Common Pitfalls 2/2

Typical individual mistakes include:

- partial adoption – only a few developers on the team use TDD
- poor maintenance of the test suite – most commonly leading to a test suite with a prohibitively long running time
- abandoned test suite (i.e. seldom or never run) – sometimes as a result of poor maintenance, sometimes as a result of team turnover

`https://www.agilealliance.org/`

TDD: Signs of Use

- “code coverage” is a common approach to evidencing the use of TDD; while high coverage does not guarantee appropriate use of TDD, coverage below 80% is likely to indicate deficiencies in a team’s mastery of TDD

- version control logs should show that test code is checked in each time product code is checked in, in roughly comparable amounts

https://www.agilealliance.org/

Skill Levels: Beginner

- able to write a unit test prior to writing the corresponding code
- able to write code sufficient to make a failing test pass

`https://www.agilealliance.org/`

Skill Levels: Intermediate

- practices “test driven bug fixing”: when a defect is found, writes a test exposing the defect before correction
- able to decompose a compound program feature into a sequence of several unit tests to be written
- knows and can name a number of tactics to guide the writing of tests (for instance “when testing a recursive algorithm, first write a test for the recursion terminating case”)
- able to factor out reusable elements from existing unit tests, yielding situation-specific testing tools

`https://www.agilealliance.org/`

Skill Levels: Advanced

- able to formulate a “roadmap” of planned unit tests for a macroscopic features (and to revise it as necessary)
- able to “test drive” a variety of design paradigms: object-oriented, functional, event-drive
- able to “test drive” a variety of technical domains: computation, user interfaces, persistent data access. . .

`https://www.agilealliance.org/`

Plan

Brief Description

- A CMake-based buildsystem is organized as a set of high-level logical targets.
- Each target corresponds to an executable, or a library, or is a custom target that contains custom commands.
- Dependencies between targets are expressed in the generation system to determine the order of generation and the rules for regeneration when changes are made.
- CMake commands are written in CMake Language and included in CMakeLists.txt files or with the .cmake extension.

Resources

- Detailed description: `https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html`
- CMake Language is described in `https://cmake.org/cmake/help/latest/manual/cmake-language.7.html`

Folder Structure

```
...
└─ tutorial-step1
   ├─ CMakeLists.txt
   ├─ TutorialConfig.h.in
   └─ src
      └─ tutorial.cxx
```

Description

- CMakeLists.txt - includes information about targets and their dependencies
- src - folder that includes source files
- TutorialConfig.h.in - file based on which the TutorialConfig.h header file will be generated

CMakeLists.txt

```
cmake_minimum_required (VERSION 2.6)
project (Tutorial)
# The version number (here 1.0).
set (Tutorial_VERSION_MAJOR 1)
set (Tutorial_VERSION_MINOR 0)

# configure a header file to pass some of the CMake settings
# to the source code
configure_file (
  "${PROJECT_SOURCE_DIR}/TutorialConfig.h.in"
  "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  )

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
include_directories("${PROJECT_BINARY_DIR}")

# add the executable
add_executable(Tutorial tutorial.cxx)
```

https://cmake.org/cmake-tutorial/

Variables and Commands Description 1/2

project (<project-name>) - set the project namei

Tutorial - project name

Tutorial_VERSION_MAJOR, Tutorial_VERSION_MINOR - project version given as <major>.<minor>

configure_file (<input> <output>) - copy the file <input> into <output>

PROJECT_SOURCE_DIR - variable that stores the name of the folder with project sources

$PROJECT_SOURCE_DIR - variable value

PROJECT_BINARY_DIR - the folder where the project will be built

Variables and Commands Description 2/2

include_directories() - add the given directories to those the compiler uses to search for include files. Relative paths are interpreted as relative to the current source directory

add_executable(<name> source1 [source2 . . .)] - adds an executable target called <name> to be built from the source files listed in the command invocation

TutorialConfig.h.in

```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

src/tutorial.cxx

```
// A simple program that computes the square root of a number
#include "TutorialConfig.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  if (argc < 2) {
    fprintf(stdout, "%s Version %d.%d\n", argv[0],
            Tutorial_VERSION_MAJOR,
            Tutorial_VERSION_MINOR);
    fprintf(stdout, "Usage: %s number\n", argv[0]);
    return 1;
  }
  double inputValue = atof(argv[1]);
  double outputValue = sqrt(inputValue);
  fprintf(stdout, "The square root of %g is %g\n",
          inputValue, outputValue);
  return 0;
}
```

Commands Description (partial)

```
cmake -H<PROJECT_SOURCE_DIR>
      -B<PROJECT_BINARY_DIR>
```

- create the folder `<PROJECT_BINARY_DIR>` , where the files needed for building process will be generated

```
cmake -build<dir>
```

- generate the tree with the binaries; `<dir>` is the folder where this tree will be generated

A full description can be found at
`https://cmake.org/cmake/help/v3.0/manual/cmake.1.html#manual:cmake(1)`

Demo Mac OS

cmake-step1-mac

Demo Windows Command Line

cmake-step1-windows

Demo Visual Studio

cmake-step1-vs

Plan

Testing Frameworks fro C++

- Boost Test Library `https://www.boost.org/doc/libs/1_66_0/libs/test/doc/html/index.html`
- CppUnit `http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html`
- CUTE (C++ Unit Testing Easier) `http://cute-test.com/`
- Google C++ Mocking Framework `https://github.com/google/googletest/tree/master/googlemock`
- Google Test `https://github.com/google/googletest/tree/master/googletest`
- Microsoft Unit Testing Framework for C++ `https://msdn.microsoft.com/en-us/library/hh598953.aspx`
- ...

Six Stepd to Follow

- Step 1: Initialize the project
- Step 2: Incorporate the Google Test project
- Step 3: Add the source files to the project
- Step 4: Add tests
- Step 5: Complete the files with the test configuration information
- Step 6: Build and execute the project

Example 1/2

We assume that we have a function that tests the primality of an integer, stored in the isprime.cc file.

Example inspired from
`https://github.com/google/googletest/blob/master/googletest/samples/sample1_unittest.cc`

Example 1/2

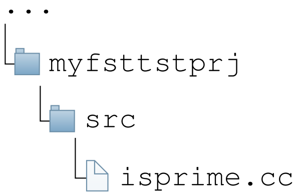
```
bool IsPrime(int n) {
  // Trivial case 1: small numbers
  if (n <= 1) return false;

  // Trivial case 2: even numbers
  if (n % 2 == 0) return n == 2;
  // Now, we have that n is odd and n >= 3.
  // Try to divide n by every odd number i, starting from 3
  for (int i = 3; ; i += 2) {
    // We only have to try i up to the square root of n
    if (i > n/i) break;
    // Now, we have i <= n/i < n.
    // If n is divisible by i, n is not prime.
    if (n % i == 0) return false;
  }
  // n has no integer factor in the range (1, n),
  // and thus is prime.
  return true;
}
```

Step 1: Initialize the project

1.1 creates a <proj> folder for the project

1.2 creates a subfolder <proj>/src, in houses they deposit the sources

1.3 add sources in <proj>/src (if any)



Step 2: Incorporate the Google Test project

2.1 add the CMakeList.txt.in file to <proj>

2.2 add the googletest.cmake file in <proj>

```
...
└ myfsttstprj
  ├ CMakeLists.txt.in
  ├ googletest.cmake
  └ src
    └ isprime.cc
```

CMakeList.txt.in file

It is a template for CMake.txt.

```
cmake_minimum_required(VERSION 2.8.2)

project(googletest-download NONE)

include(ExternalProject)
ExternalProject_Add(googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG master
    SOURCE_DIR "${CMAKE_BINARY_DIR}/googletest-src"
    BINARY_DIR "${CMAKE_BINARY_DIR}/googletest-build"
    CONFIGURE_COMMAND ""
    BUILD_COMMAND ""
    INSTALL_COMMAND ""
    TEST_COMMAND ""
)
```

Commands/Parameters Description

project (<project-name> NONE) - NONE means skipping languages (C and Cxx are allowed by default)

include (<file | module>) - load and execute commands from the given file / module as a parameter

ExternalProject - creates a custom target to generate a project in an external tree

ExternalProject_Add () - creates a custom target to guide downloading, updating, configuring, building, installing and testing the steps of an external project

googletest.cmake file

Includes commands that download and unpack Google Test at setup time.

```
configure_file(CMakeLists.txt.in googletest-download/CMakeLists.txt)
execute_process(COMMAND "${CMAKE_COMMAND}" -G "${CMAKE_GENERATOR}" .
    WORKING_DIRECTORY "${CMAKE_BINARY_DIR}/googletest-download" )
execute_process(COMMAND "${CMAKE_COMMAND}" --build .
    WORKING_DIRECTORY "${CMAKE_BINARY_DIR}/googletest-download" )
# Prevent GoogleTest from overriding our compiler/linker options
# when building with Visual Studio
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)

# Add googletest directly to our build. This adds
# the following targets: gtest, gtest_main, gmock
# and gmock_main
add_subdirectory("${CMAKE_BINARY_DIR}/googletest-src"
                 "${CMAKE_BINARY_DIR}/googletest-build")
# The gtest/gmock targets carry header search path
# dependencies automatically when using CMake 2.8.11 or
# later. Otherwise we have to add them here ourselves.
if(CMAKE_VERSION VERSION_LESS 2.8.11)
    include_directories("${gtest_SOURCE_DIR}/include"
                        "${gmock_SOURCE_DIR}/include")
endif()
```

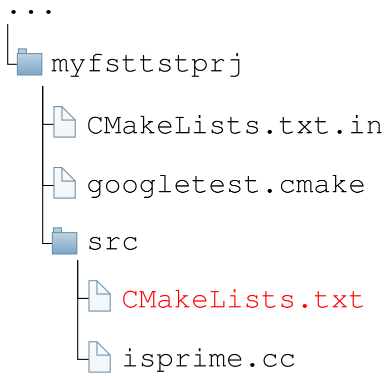
Commands/Parameters Description

execute_process - runs the given sequence of commands so that the standard output of each process is connected to the standard input of the next. A single standard error "pipe" is used for all processes. In the example, we have only one command in each execute_process.

set(<variable> <value>) - - sets the variable <variable> to the value <value>

Step 3: Add the source files to the project

3.1 creates a CMakeList.txt file in <proj>/src and in each of its subfolders
3.2 add in each CMakeList.txt commands with information about the targets included in the subfolder



src/CMakeLists.txt file

Each subfolder must have a CMakeList.txt file to know how it is used in generation.
Here is that from src/ folder:

```
add_library(myfunctions "")

target_sources(
    myfunctions
    PRIVATE
        isprime.cc
    PUBLIC
        ${CMAKE_CURRENT_LIST_DIR}/myfsttstprj.h
    )

target_include_directories(
    myfunctions
    PUBLIC
        ${CMAKE_CURRENT_LIST_DIR}
    )
```

Commands/Parameters Description

```
add_library(<name> [STATIC | SHARED | MODULE]
                                         [EXCLUDE_FROM_ALL]
        source1 [source2 ...])
```

– adds a library to the target <name>

```
target_sources(<target>
           <INTERFACE|PUBLIC|PRIVATE> [items1...]
          [<INTERFACE|PUBLIC|PRIVATE> [items2 ...]  ...])
```

– creates a target <target>, which has to be created with add_executable() or add_library() command

```
target_include_directories(<target> [SYSTEM] [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

– specifies the folders include used to compile <target>.

Step 4: Add tests

4.1 creates a <proj>/tests subfolder

4.2 create test files in <proj>/tests (e.g., one for each class)

4.3 creates a CMakeList.txt file in <proj>/tests

4.4 includes in <proj>/tests a main.cc file with the main() function running the tests

Step 4: Our Example

```
...
|
myfsttstprj
  CMakeLists.txt.in
  googletest.cmake
  src
    CMakeLists.txt
    isprime.cc
  tests
    CMakeLists.txt
    isprime_unittest.cc
    main.cc
```

Adding the tests

- It is best to have the tests in a separate subfolder; for our example this is ./tests
- tests are defined using the TEST macro
- the tests are grouped into test cases

```
// Tests some trivial cases.
TEST(IsPrimeTest, Trivial) {
  EXPECT_FALSE(IsPrime(0));
  EXPECT_FALSE(IsPrime(1));
  EXPECT_TRUE(IsPrime(2));
  EXPECT_TRUE(IsPrime(3));
}
```

- include the tests in a file, for example, ./tests/isprime-unittest.cc
- more details about writing the tests can be found at `https://github.com/google/googletest/blob/master/googletest/docs/primer.md`

tests/CMakeList.txt File

```
add_executable(
    unit_tests
    isprime_unittest.cc
)

target_link_libraries(
    unit_tests
    gtest_main
    myfunctions
)

add_test(
  NAME
    unit
  COMMAND
    ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL_BINDIR}/unit_tests
)
```

Commands/Parameters Description

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
                [EXCLUDE_FROM_ALL]
                source1 [source2 ...])
```

– adds an executable target, named <name>, to be generated from the sources mentioned in the command

```
target_link_libraries(<target> [item1 [item2 [...]]]
                      [[debug|optimized|general] <item>] ...)
```

– links a target to a library add_executable() or add_library() command

```
add_test(NAME <name> COMMAND <command> [<arg>...]
         [CONFIGURATIONS <config>...]
         [WORKING_DIRECTORY <dir>])
```

– adds a test to be executed

Function Running all Tests (main.cc)

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

int main(int argc, char** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Step 5: Complete the files with the test configuration information

5.1 add the CMakeList.txt file to the <proj> folder
5.2 add the <proj>Config.h.in file to the <proj> folder

```
...
myfsttstprj
    CMakeLists.txt
    CMakeLists.txt.in
    googletest.cmake
    myfsttstprjConfig.h.in
src
    CMakeLists.txt
    isprime.cc
tests
    CMakeLists.txt
    isprime_unittest.cc
    main.cc
```

<proj>/CMakeList.txt file

```
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
project(myfsttstprj CXX)

# First part is similar to the previous project

# C++11
set(CMAKE_CXX_STANDARD 11)

# place binaries and libraries according to GNU standards
include(GNUInstallDirs)
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/${CMAKE_INSTALL

# we use this to get code coverage
if(CMAKE_CXX_COMPILER_ID MATCHES GNU)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fprofile-arcs -ftest-cove
endif()

include(./googletest.cmake)
add_subdirectory(src)
enable_testing()
add_subdirectory(tests)
```

Commands/Parameters Description

GNUInstallDirs - modul care definește folderele standard GNU

CMAKE_ARCHIVE_OUTPUT_DIRECTORY - where the ARCHIVE targets are generated

CMAKE_INSTALL_LIBDIR - where the GNU object-code librrary are found

CMAKE_LIBRARY_OUTPUT_DIRECTORY - where the LIBRARY targets are generated

CMAKE_RUNTIME_OUTPUT_DIRECTORY - where theRUNTIME targets are generated

CMAKE_INSTALL_BINDIR - where the binaries are generated

enable_testing() - allow tests for the current folder and its subfolders

Step 6: Build and execute the project

220/5000 6.1 Configuring the project with the command cmake -H. -Bbuild in the <proj> folder

6.2 Building the project with the command cmake –build. in the build subfolder

6.3 execution of unit tests with the command ./bin/unit_tests

Demo Mac OS

gtest-v1-mac

Demo Visual Studio

gtest-v1-vs

Adding a Demo Executable

Add

```
add_executable(demo src/demo.cc)
```

in the file `<proj>/CMakeList.txt`

and the demo file demo.cc

in the folder `<proj>/src/`

Demo Visual Studio

gtest-v2-vs

Plan

First Step 1/3

- From Wikipedia:

In mathematics, the greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For example, the gcd of 8 and 12 is 4.

1. The definition already gives us a test, which we include in the file ./tests/gcd-unittest.cc

```
TEST(GcdTest, Positive) {
  EXPECT_EQ(4, Gcd(8,12));
}
```

2. add the file gcd-unittest.cc in ./tests/CMakeList.txt
3. add in ./src/myfsttstprj.h the prototype of Gcd:

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

First Step 2/3

4. add to the ./src/ folder the gcd.cc file in which we write the empty GCD function:

   ```
   int Gcd(int a, int b) {
     // nothing
   }
   ```

5. add the file gcd.cc in ./src/CMakeList.txt and comment isprime_unittest.cc (for optimisation)
6. build the binaries
7. Ignore the warning

First Step 2/3

8. test:

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from GcdTest
[ RUN      ] GcdTest.Positive
.../myfsttstprj/tests/gcd_unittest.cc:88: Failure
Expected equality of these values:
  4
  Gcd(8,12)
    Which is: 1
[  FAILED  ] GcdTest.Positive (0 ms)
[----------] 1 test from GcdTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] GcdTest.Positive

 1 FAILED TEST
```

First Refinement 1/3

1. add the Euclid algorithm:

```
int Gcd(int a, int b) {
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

First Refinement 2/3

2. build and test:

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[----------] 1 test from GcdTest (0 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

First Refinement 3/3

3. add more positive tests:

```
TEST(GcdTest, Positive) {
  EXPECT_EQ(4, Gcd(8,12));
  EXPECT_EQ(7, Gcd(28,21));
  EXPECT_EQ(1, Gcd(23,31));
  EXPECT_EQ(1, Gcd(48,17));
}
```

4. build and test:

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[----------] 1 test from GcdTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

Second Refinement 1/3

1. add trivial tests

```
TEST(GcdTest, Trivial) {
  EXPECT_EQ(18, Gcd(18,0));
  EXPECT_EQ(24, Gcd(0,24));
  EXPECT_EQ(19, Gcd(19,19));
  EXPECT_EQ(0, Gcd(0,0));    // undefined
}
```

2. build and generate

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[ RUN      ] GcdTest.Trivial
^C
```

Oooops! The execution does not terminate ...

Second Refinement 2/3

3. proceed by elimination and see that Gcd(18,0) is the problem
4. analyze the problem:

```
int Gcd(int a, int b) {
  while (a != b)
    if (a > b)
      a = a - b; // if b == 0, a is not modified!!!
    else
      b = b - a;
  return a;
}
```

5. fix it:

```
int Gcd(int a, int b) {
  if (b == 0) return a;
  if (a == 0) return b;
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

Second Refinement 3/3

6 build and test it:

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[ RUN      ] GcdTest.Trivial
[       OK ] GcdTest.Trivial (0 ms)
[----------] 2 tests from GcdTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms tot
[  PASSED  ] 2 tests.
```

Third Refinement 1/3

1. add negative tests

```
TEST(GcdTest, Negative) {
  EXPECT_EQ(6, Gcd(18, -12));
  EXPECT_EQ(4, Gcd(-28, 32));
  EXPECT_EQ(1, Gcd(-29, -37));
}
```

2. build and generate

```
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
...
.../gcd_unittest.cc:104: Failure
Expected equality of these values:
  6
  Gcd(18, -12)
    Which is: -2147483638
^C
```

Oooops! The execution does not terminate again ...

Third Refinement 2/3

Back to the documentation:
"$d \mid a$ if and only if $d \mid -a$. Thus, the fact that a number is negative does not change its list of positive divisors relative to its positive counterpart. . . .
Therefore, GCD(a, b) = GCD(|a|, |b|) for any integers a and b, at least one of which is nonzero."

3. include the case when the numbers are negative:

```
int Gcd(int a, int b) {
  if (b == 0) return a;
  if (a == 0) return b;
  if(a < 0) a = -a;
  if(b < 0) b = -b;
  while (a != b)
    if (a > b)
      a = a - b;
    else
      b = b - a;
  return a;
}
```

Third Refinement 3/3

4. build and test it:

```
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[ RUN      ] GcdTest.Trivial
[       OK ] GcdTest.Trivial (0 ms)
[ RUN      ] GcdTest.Negative
[       OK ] GcdTest.Negative (0 ms)
[----------] 3 tests from GcdTest (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 3 tests.
```

Plan

What Is a Test Fixture

- Allows the use of the same object configuration for multiple tests
- It is recommended for testing classes
- The steps to follow are:
  1. Derive a class from :: testing :: Test.
  2. Inside the class, state any objects you intend to use.
  3. If necessary, write a default constructor or a SetUp() function to prepare the objects for each test. A common mistake is to write SetUp() as Setup(), with a small u - don't let this happen.
  4. If necessary, write a destructor or TearDown() function to free up the resources allocated in SetUp().
  5. If necessary, define methods to allow test sharing.

Further Info

https://github.com/google/googletest/blob/master/googletest/docs/FAQ.md

TEST_F() Macro

- it must be used whenever we have "test fixture" type objects
- template:

```
TEST_F(test_case_name, test_name) {
 ... test body ...
}
```

- the first parameter must be the name of the "test fixture" class
- you must first define a test fixture class before using it in a TEST_F()

How TEST_F() Works

For each test defined with TEST_F(), Google Test will:

1. create a new “test fixture” at runtime
2. initialize with SetUp ()
3. run the test
4. free memory by calling TearDown()
5. delete "test fixture".

Keep in mind that different tests in the same test have different test objects, and Google Test always deletes one test item before creating the next one.

Google Test does not reuse the same "fixture test" for multiple tests. Any changes that a test makes to the "test fixture" do not affect other tests.

Example: Class under Test

Sample3 from
`https://github.com/google/googletest/tree/master/googletest/samples`

```
template <typename E> // E is the element type.
class Queue {
 public:
  Queue();
  void Enqueue(const E& element);
  E* Dequeue(); // Returns NULL if the queue is empty.
  size_t size() const;
  ...
};
```

Example: Test Fixture Class

```
class QueueTestSmpl3 : public ::testing::Test {
 protected:
  virtual void SetUp() {
    q1_.Enqueue(1);
    q2_.Enqueue(2);
    q2_.Enqueue(3);
  }

  // virtual void TearDown() {}

  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
};
```

The TearDown() method is not required because there is nothing to clean after a test.

Example: Tests

```
TEST_F(QueueTestSmpl3, DefaultConstructor) {
  // You can access data in the test fixture here.
  EXPECT_EQ(0u, q0_.Size());
}

TEST_F(QueueTestSmpl3, Dequeue) {
  int *n = q0_.Dequeue();
  EXPECT_TRUE(n == NULL);

  n = q1_.Dequeue();
  ASSERT_TRUE(n != NULL);
  EXPECT_EQ(1, *n);
  EXPECT_EQ(0u, q1_.Size());
  delete n;

  n = q2_.Dequeue();
  ASSERT_TRUE(n != NULL);
  EXPECT_EQ(2, *n);
  EXPECT_EQ(1u, q2_.Size());
  delete n;
}
```

What Happen When a Test is Executed

Google Test builds a QueueTestSmpl3 object (let’s call it t1).

- t1.SetUp () initializes t1.
- The first test (DefaultConstructor) is run over t1.
- t1.TearDown () cleans up after the test (if any).
- t1 is destroyed.

The above steps are repeated with another QueueTest object, in this case the Dequeue test.

Preparing the Tests

- CMakeList.txt and googletest.cmake files are defined in the same way as in the previous project.
- The exception is the ./src folder, which no longer requires the CMakeList.txt file, for this particular case, because it only includes the sample3-inl.h file (in which the Queue class is described with the inline implementation of the methods).
- The header file sample3-inl.h is explicitly included in sample3_unittest.cc, because the QueueTestSmpl3 class uses instances of the Queue class

Tests Running

```
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from QueueTestSmpl3
[ RUN      ] QueueTestSmpl3.DefaultConstructor
[       OK ] QueueTestSmpl3.DefaultConstructor (0 ms)
[ RUN      ] QueueTestSmpl3.Dequeue
[       OK ] QueueTestSmpl3.Dequeue (0 ms)
[ RUN      ] QueueTestSmpl3.Map
[       OK ] QueueTestSmpl3.Map (0 ms)
[----------] 3 tests from QueueTestSmpl3 (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 3 tests.
```

Conclusion

- TDD, like any other software development concept or method, requires practice.
- The more you use TDD, the easier TDD becomes.
- Don’t forget to keep the tests simple.
- The tests, which are simple, are easy to understand and easy to maintain.
- Tools like Google Test, Google Mock, CppUnit, JustCode, JustMock, NUnit and Ninject are important and help facilitate the practice of TDD.
- But it is good to know that TDD is a practice and a philosophy that goes beyond the use of tools.
- Experience with tools and frameworks is important, the skills acquired will inspire confidence in the application developer.
- But it is good to know that the tools should not be the center of attention.
- Equal time must be given to the idea of "test first".

Demo Visual Studio

cmake-sample3-vs