Programare orientata- obiect (POO) utilizand C++

Testare

Dorel Lucanu

Cuprins

- "Test-driven development" (TDD)
- “Unit testing frameworks”
- Google Test Framework
- Crearea unei funcții cu abordarea TDD
- “Test fixtures”
- Concluzii

From

SOFTWARE ENGINEERING TECHNIQUES Report on a conference sponsored by the NATO SCIENCE COMMITTEE Rome, Italy, 27th to 31st October 1969 (http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF):

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

- "Test-driven development" (TDD) - definitia data de Agile Alliance (https://www.agilealliance.org/)

TDD - definitie

- "Test-driven development" (TDD) - se referă la un stil de programare în care trei activități sunt strâns legate între ele: codificarea, testarea (sub formă de de scriere de “unit tests”) și proiectarea (sub formă de refactorizare).
- Acesta poate fi descris succint de următoarele reguli:
  - scrie un “unit test” care descrie o funcționalitate a programului
  - executați testul, care ar trebui să eșueze deoarece programul nu are această funcționalitate
  - scrieți “suficient" cod, cel mai simplu posibil, pentru a trece testul
  - "refactorizați" codul până când acesta se conformează criteriilor de simplitate
  - repetați pașii de mai sus, acumulând “unit tests” în timp

TDD - beneficii

- reduceri semnificative ale ratelor defectelor, cu costul unei creșteri moderate a efortului de dezvoltare inițială
- cheltuieli generale sunt mai mult decât compensate de o reducere a efortului în fazele finale ale proiectelor
- deși cercetările empirice nu au reușit până acum să confirme acest lucru, experții veterani spun că TDD conduce la îmbunătățirea calităților de proiectare în cod și, în general, la un grad mai ridicat de "calitate internă" sau tehnică, de exemplu îmbunătățirea măsurătorilor coeziunii și cuplării

TDD - Greselile tipice individuale

- se uită frecvent rularea testelor
- scrierea de prea multe teste simultan
- scrierea testelor care sunt prea mari sau cu granulație grosieră
- scrierea de teste prea triviale
- scrierea de teste pentru cod trivial, de exemplu metode de tip “get” sau “set”

TDD - Greșeli tipice ale echipei

- adoptare parțială - doar câțiva dezvoltatori din echipa folosesc TDD
- întreținerea necorespunzătoare a suitei de teste - cel mai frecvent duce la o suită de teste cu un timp de funcționare extrem de prohibitiv
- suită de teste abandonată - uneori ca rezultat al unei întrețineri necorespunzătoare, uneori ca rezultat al fluctuației echipei

TDD - Origini

- 1976: publicația "Software Reliability" a lui Glenford Myers, afirmă ca "axiomă" că "a developer should never test their own code" (Dark Ages of Developer Testing)
- 1990: testarea este dominată de tehnica "black box", în particular sub forma de instrumente de testare de tip "capture and replay"
- 1991: crearea framework-ului de testare de la Taligent (similar cu SUnit)
- 1994: Kent Beck scrie framework-ul de testare SUnit pentru Smalltalk
- 1998: un articol despre Extreme Programming menționează că "we usually write the test first"
- 1998 to 2002: conceptul "Test First" este elaborat în "Test Driven"
- 2000: apare o tehnică nouă bazată pe “Mock Objects”
- 2003: a apărut cartea "Test Driven Development: By Example" by Kent Beck (accesibilă online: https://www.eecs.yorku.ca/course_archive/2003-04/W/3311/sectionM/case_studies/money/KentBeck_TDD_byexample.pdf)

Utilizare

- “code coverage" este o abordare comună care demonstrează utilizarea TDD;
  - o acoperire înaltă nu garantează neapărat o utilizare adecvată a TDD, dar
  - o acoperire sub 80% este susceptibilă să indice deficiențele în stăpânirea TDD de către echipă
- log-urile de la sistemele de “control version” ar trebui să arate că testele sunt verificate în fiecare moment în care codul de produs este încărcat în sistem, în cantități comparabile

TDD - nivelul de începător

- e capabil să scrie un “unit test” înainte de a scrie codul corespunzător
- e capabil să scrie cod suficient pentru a face să eșueze un test

TDD - nivelul intermediar

- aplică practici de "fixare a bug-urilor testate": atunci când se găsește un defect, scrie un test expunând defectul înainte de corectare
- are capabilitatea de a descompune o funcționalitate compusă a programului într-o secvență de mai multe teste unitare care urmează să fie scrise
- cunoaște și poate numi un număr de tactici care să ghideze scrierea testelor (de exemplu "atunci când testează un algoritm recursiv, mai întâi scrie un test pentru cazul de terminare a recursiei")
- e capabil să evalueze elementele refolosibile din testele unității existente, generând instrumente de testare specifice situației
-

TDD - nivelul avansat

- e capabil să formuleze o "foaie de parcurs" a testelor unitare planificate pentru caracteristicile macroscopice (și să o revizuiască după cum este necesar)

- e capabil să aplice o varietate de paradigme de design pentru testare: orientate spre obiecte, funcționale, “drive-events”

- e capabil să "testeze" o varietate de domenii tehnice: calcul, interfețe utilizator, acces la date persistente ...

- Framework-uri de testare

Framework-uri de testare pentru 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

- …

- Google Test

Google Test - Principii

- Testele trebuie să fie independente și repetabile.
- Testele ar trebui să fie bine organizate și să reflecte structura codului testat.
- Testele trebuie să fie portabile și reutilizabile.
- Când testele nu reușesc, trebuie să furnizeze cât mai multe informații despre problemă.
- Cadrul (framework-ul) de testare trebuie să elibereze scrierea de teste de celelalte activități și să se permită concentrarea numai pe conținutul testului.
- Execuția testelor trebuie să fie rapidă.

Google Test - Concepte de bază

- aserțiune (assertion) - declarații care verifică dacă o condiție este adevărată
  - Rezultatul aserțiunii poate fi un succes, un eșec nonfatal sau un eșec fatal.
  - Dacă apare un defect fatal, acesta întrerupe execuția funcției curente; altfel programul continuă în mod normal.

- Testele folosesc aserțiuni pentru a verifica comportamentul codului testat.
  - Dacă un test se blochează sau are o aserțiune eșuată, atunci acesta eșuează (fails); în caz contrar, reușește (succeeds).

Google Test - Concepte de bază

- Un caz de testare (test case) conține unul sau mai multe teste.
  - Testele trebuie grupate în cazuri de testare care reflectă structura codului testat.
  - Atunci când mai multe teste într-un caz de testare necesită partajarea obiectelor și subrutinelor obișnuite, le puteți pune într-o clasă de testare.

- Un program de testare (test program) poate conține mai multe cazuri de testare.

- "Test fixture" - permite să reutilizați aceeași configurație de obiecte pentru mai multe teste diferite.

Google Test - Instalare cu CMake

- Google Test vine cu un script de instalare CMake (CMakeList.txt), care poate fi folosit pe mai multe platforme ("C" înseamnă platformă “cross-platform”).
- Dacă nu aveți deja instalat CMake, îl puteți descărca gratuit de la http://www.cmake.org/.

- CMake funcționează generând fișiere “make” native care pot fi utilizate în mediul compilator ales de dumneavoastră.

- Google Test poate fi instalat ca un proiect independent sau poate fi încorporat într-un CMake existent pentru a construi un alt proiect.

- Vom prezenta varianta a doua, cu Google Test încorporat

Intermezzo - CMake

- Un sistem de generare de binare (buildsystem) bazat pe CMake este organizat ca un set de ținte logice (logical targets) la nivel înalt.
- Fiecare țintă corespunde unui executabil sau unei biblioteci sau este o țintă personalizată ce conține comenzi personalizate.
- Dependențele dintre ținte sunt exprimate în sistemul de generare pentru a determina ordinea de generare și regulile pentru regenerare atunci când se fac schimbări.
- O descriere detaliata a conceptelor CMake poate fi găsită la adresa https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html
- Comenzile CMake sunt scrise în "CMake Language" și incluse în fișiere CMakeLists.txt sau cu extensia .cmake.
- CMake Language este descris în documentul de la adresa https://cmake.org/cmake/help/latest/manual/cmake-language.7.html

Intermezzo - CMake - exemplu

- Primul exemplu din Tutorial (https://cmake.org/cmake-tutorial/)
- conținut folder:

```
tutorial-step1$ ls
  CMakeLists.txt
  src
  TutorialConfig.h.in
tutorial-step1$ ls src/
  tutorial.cxx
tutorial-step1$
```

- CMakeLists.txt - include informații despre ținte și dependențele dintre acestea
- src - folder care include fisierele sursa
- TutorialConfig.h.in - fisier pe baza caruia va fi generat fișierul antet TutorialConfig.h

Intermezzo - CMake - exemplu

- fișierul 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)
```

- fișierul TutorialConfig.h.in

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

Explicarea variabilelor si comenzilor din fisiere

- project (<project-name>) - seteaza numele proiectului

  Tutorial - numele proiectului

- Tutorial_VERSION_MAJOR, Tutorial_VERSION_MINOR - versiunea proiectului, care e de forma <major>.<minor>

- configure_file (<input> <output>) - copie fișierul <input> în fișierul <output>

- PROJECT_SOURCE_DIR - variabila care memorează calea la folderul sursa a proiectului (dat de cea mai recentă comanda project())
- ${PROJECT_SOURCE_DIR} - valoarea variabilei

- PROJECT_BINARY_DIR - folderul în care va fi generat proiectul
-

Explicarea variabilelor si comenzilor din fisiere

- include_directories() - Aadugă folderele date ca parametru la cele pe care compilatorul le utilizează pentru a căuta fișierele menționate în “include”

- add_executable(<name> source1 [source2 …]) - adaugă o țintă executabilă numită <name>, care urmează să fie generată din fișierele sursă enumerate în invocarea comenzii.
  - <name> corespunde numelui țintă logic și trebuie să fie unic la nivel global în cadrul unui proiect.
  - numele real al fișierului executabil generat este construit pe baza convențiilor platformei native (cum ar fi <name> .exe sau doar <name>).

- Indexul cu toți termenii CMake:
  https://cmake.org/cmake/help/v3.0/genindex.html

Intermezzo - CMake - exemplu

- fișierul 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;
}
```

Intermezzo - CMake - exemplu

- procesul de generare
- … si executia programului

```
$ cmake -H. -Bbuild
...
$ ls
 CMakeLists.txt
 build
 src
 TutorialConfig.h.in
$ cd build
$ ls
 CMakeCache.txt
 cmake_install.cmake
 Makefile
 CMakeFiles
 TutorialConfig.h
$ cmake --build .
...
$ ls
 CMakeCache.txt
 Makefile
 TutorialConfig.h
 CMakeFiles
 Tutorial
 cmake_install.cmake
$ ./Tutorial 12
The square root of 12 is 3.4641
```

Explicarea comenzilor

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

creează folderul <PROJECT_BINARY_DIR> și-l populează cu fișierele necesare generării executabilelor țintă pe platforma nativă

- cmake --build<dir> - genereaza arborele cu binarele proiectului; <dir> este folderul în care va fi generat
  - în acest exemplu este creat un singur binar, Tutorial

- descrierea comenzii cmake poate fi gasita la adresa

https://cmake.org/cmake/help/v3.0/manual/cmake.1.html#manual:cmake(1)

Încorporarea lui Google Test la un proiect

- presupunem ca avem o funcție care testează primalitatea unui număr întreg, memorată în fișierul isprime.cc (exemplu inspirat din https://github.com/google/googletest/blob/master/googletest/samples/sample1_unittest.cc)

```
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;
}
```

Încorporarea lui Google Test la un proiect

- includem isprime.cc în subfolderul ./src
- creăm mai întâi un șablon de fișier CMake.txt, memorat în fișierul CMakeLists.txt.in, care să genereze Google Test ca proiect extern. CMakeLists.txt.in are următorul conținut:

```
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 ""
)
```

Explicarea comenzilor/parametrilor

- project (<project-name> NONE) - NONE înseamnă că se sare peste limbaje (implicit sunt permise C si Cxx)
- include(<file|module>) - încarcă și execută comezile din fișierul/modulul dat ca parametru
- ExternalProject - crează o țintă personalizată pentru a genera un proiect într-un arbore extern
- ExternalProject_Add() - creează o țintă personalizată pentru a conduce descărcarea, actualizarea, configurarea, construirea, instalarea și testarea pașilor unui proiect extern

Încorporarea lui Google Test la un proiect

- creăm un fișier googletest.cmake, care include comenzi ce descarcă și despachetează Google Test la momentul configurării. googletest.cmake are următorul conținut:

```
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()
```

Explicarea comenzilor/parametrilor

- execute_process - rulează secvența de comenzi dată astfel încât ieșirea standard a fiecărui proces conectată la intrarea standard a următorului. Un singur “pipe” standard de eroare este utilizat pentru toate procesele. În exemplu, avem câte o singură comandă.
- set(<variable> <value>) - setează variabila <variable> pe valoarea <value>
-

Crearea fișierului CMakeList.txt în subfolderul ./src

- fiecare subfolder trebuie să aibă un fișier CMakeList.txt, pentru a ști cum este utilizat la generare
- pentru subfolderul ./src din exemplul nostru CMakeList.txt are următorul conținut:

```
add_library(myfunctions "")

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

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

Explicarea comenzilor/parametrilor

- add_library(<name> [STATIC | SHARED | MODULE]
  [EXCLUDE_FROM_ALL]
  source1 [source2 ...]) - adaugă o bibliotecă la ținta <name>
- target_sources(<target>
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] …]) - adaugă surse la o ținta. <target> trebuie să fi fost creat de o comandă precum add_executable () sau add_library ()
- target_include_directories(<target> [SYSTEM] [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] …]) - specifică directoarele “include” sau țintele utilizate la compilarea țintei <target>
-

Adăugarea de teste

- E bine ca testele să fie într-un subfolder separat; pentru exemplul nostru acesta este ./tests
- testele sunt definite cu ajutorul macroului TEST
- testele sunt grupate în cazuri de test

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

- includem testele într-un fișier, de exemplu, ./tests/isprime-unittest.cc
- mai multe detalii despre scrierea testelor se găsesc în https://github.com/google/googletest/blob/master/googletest/docs/Primer.md

Crearea fișierului CMakeList.txt în subfolderul ./tests

- include comenzi pentru crearea binarelor de testare
- pentru exemplul nostru are următorul conținut

```
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
)
```

Explicarea comenzilor/parametrilor

- add_executable(<name> [WIN32] [MACOSX_BUNDLE]
  [EXCLUDE_FROM_ALL]
  source1 [source2 ...]) - adaugă o țintă execuatbilă, cu numele <name>, ce urmează a fi generate din sursele menționate în comandă
- target_link_libraries(<target> [item1 [item2 [...]]]
  [[debug|optimized|general] <item>] ...) - leagă (link) o țintă la o bibliotecă
- add_test(NAME <name> COMMAND <command> [<arg>...]
  [CONFIGURATIONS <config>...]
  [WORKING_DIRECTORY <dir>]) - adaugă un test la proiect ce urmează a fi executat cu ctest

Adăugarea funcției care rulează toate testele

- în folderul ./tests/ se include un fișier main.cc cu funcția main() ce rulează testele

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

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

Crearea fișierului CMakeList.txt la nivel de proiect

- este pus în folderul proiectului
- pentru exemplul nostru are următorul conținut

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

# First part is similar to the previous proect

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

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

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

Explicarea comenzilor/parametrilor

- GNUInstallDirs - modul care definește folderele standard GNU
- CMAKE_ARCHIVE_OUTPUT_DIRECTORY - unde vor fi puse țintele ARCHIVE atunci când sunt generate
- CMAKE_INSTALL_LIBDIR - unde se găsesc bibliotecile GNU cod-obiect
- CMAKE_LIBRARY_OUTPUT_DIRECTORY - unde cor fi puse țintele LIBRARY atunci când vor fi generate
- CMAKE_RUNTIME_OUTPUT_DIRECTORY - unde se pun țintele RUNTIME atunci când vor fi generate
- CMAKE_INSTALL_BINDIR - unde se pun țintele executabile (binarele) atunci când vor fi generate
- enable_testing() - permite testarea pentru folderul curent și subfoldere

Generarea de binare și testare

- următoarele comenzi sunt date în folderul proiectului

```
$ cmake -H. -Bbuild
...
$ cd build
$ cmake --build .
...
$ ./bin/unit_tests
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from IsPrimeTest
[ RUN      ] IsPrimeTest.Negative
[       OK ] IsPrimeTest.Negative (0 ms)
[ RUN      ] IsPrimeTest.Trivial
[       OK ] IsPrimeTest.Trivial (0 ms)
[ RUN      ] IsPrimeTest.Positive
[       OK ] IsPrimeTest.Positive (0 ms)
[----------] 3 tests from IsPrimeTest (0 ms total)

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

Un test care eșuează

- adăugăm în fișierul isprime_unittest.cc:

```
// A failling test
TEST(IsPrimeTest, Failing) {
  EXPECT_FALSE(IsPrime(7));
}
```

- generăm binare și testăm

```
$ cmake --build .
…
$ ./bin/unit_tests
Running main() from gtest_main.cc
[==========] Running 4 tests from 1 test case.
...
[ RUN      ] IsPrimeTest.Failing
/Users/dlucanu/Documents/cursuri/poo/2017-2018/Curs-10-exemple/
firstproj-w-tests/tests/isprime_unittest.cc:106: Failure
Value of: IsPrime(7)
  Actual: true
Expected: false
[  FAILED  ] IsPrimeTest.Failing (1 ms)
...

 1 FAILED TEST
```

Adăugarea unui demo la proiect

- adăugăm fișierul demo.cc în subfolderul ./src

```
#include "firstprojwtestsConfig.h"
#include "firstprojwtests.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],
firstprojwtests_VERSION_MAJOR,
            firstprojwtests_VERSION_MINOR);
    fprintf(stdout, "Usage: %s number\n", argv[0]);
    return 1;
  }
  int inputValue = atoi(argv[1]);
  bool outputValue = IsPrime(inputValue);
  if (outputValue)
    fprintf(stdout, "The number %d is prime.\n", inputValue);
  else
    fprintf(stdout, "The number %d is not prime.\n", inputValue);
  return 0;
}
```

Adăugarea unui demo la proiect

- adăugarea următoarei linii în CMakeList.txt:

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

- generarea de binare și executarea demo-ului

```
$ cmake --build .
...
$ ls bin/
isprimetest unit_tests
$ ./bin/isprimetest 17
The number 17 is prime.
$
```

- Crearea unei funcții cu abordarea TDD

GCD utilizând TDD

- 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.

- Defiiția ne dă deja un test, pe care îl includem în fișierul ./tests/gcd-unittest.cc:

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

- Atenție: fișierul gcd-unittest.cc trebuie adăugat în CMakeList.txt din ./tests/
- Adăugăm în ./src/firstprojwtests.h prototipul (signatura) funcției Gcd:

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

- adăugăm în folderul ./src fișierul gcd.cc în care scriem funcția Gcd vidă

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

- Atenție: fișierul gcd.cc trebuie adăugat în CMakeList.txt din ./src/ și comentăm linia cu isprime_unittest.cc (pt minimizare)
- Generăm binarele:

```
$ cmake -H. -Bbuild
...
$ cd build
$ cmake --build .
[ 17%] Built target isprimetest
Scanning dependencies of target myfunctions
[ 23%] Building CXX object src/CMakeFiles/myfunctions.dir/gcd.cc.o
/Users/dlucanu/Documents/cursuri/poo/2017-2018/Curs-10-exemple/
firstproj-w-tests/src/gcd.cc:6:1: warning: control
      reaches end of non-void function [-Wreturn-type]
}
^
1 warning generated.
```

GCD utilizând TDD

- Ingorăm deocamdată avertismentul
- testăm

```
$ ./bin/unit_tests
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
/Users/dlucanu/Documents/cursuri/poo/2017-2018/Curs-10-exemple/
firstproj-w-tests/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
```

GCD utilizând TDD

- adăugăm algoritmul lui Euclid la corpul funcției Gcd:

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

- generăm binarele și testăm:

```
$ cmake --build .
...
$ ./bin/unit_tests
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.
```

GCD utilizând TDD

- Adăugăm mai multe teste pozitive

```
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));
}
```

- generăm și testăm

```
$ cmake --build .
...
$ ./bin/unit_tests
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.
```

GCD utilizând TDD

- adăugăm cazurile triviale

```
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
}
```

- generăm și testăm

```
$ cmake --build .
...
$ ./bin/unit_tests
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! Se pare că un test rulează la infinit…

GCD utilizând TDD

- Prin eliminare observăm că Gcd(18,0) rulează la infinit
- Analizăm sursa erorii

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

- Corectăm

```
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;
}
```

GCD utilizând TDD

- generăm și testăm

```
$ cmake --build .
...
$ ./bin/unit_tests
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 total)
[  PASSED  ] 2 tests.
```

GCD utilizând TDD

- Adăugăm teste negative

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

- generăm și testăm

```
$ cmake --build .
...
$ ./bin/unit_tests
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
...
/Users/dlucanu/Documents/cursuri/poo/2017-2018/Curs-10-exemple/
firstproj-w-tests/tests/gcd_unittest.cc:104: Failure
Expected equality of these values:
  6
  Gcd(18, -12)
    Which is: -2147483638
^C
```

- Ooops again! Un test eșuat și altul nu se termină!

GCD utilizând TDD

- Ne documentăm:

d | a if and only if d | −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.

- Adăugăm în program cazurile când numerele pot fi și 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;
}
```

GCD utilizând TDD

- generăm și testăm

```
$ cmake --build .
...
$ ./bin/unit_tests
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.
$
```

- Toate testele au trecut! Probabil programul este corect …

- TEST Fixtures

Ce este un “test fixture” (trusă de testare)

- Permite utilizarea acleiași configurații de obiecte pentru mai multe teste

- Pașii de urmat sunt:

1. Derivați o clasă din :: testing :: Test.
2. În interiorul clasei, declarați orice obiecte pe care intenționați să le utilizați.
3. Dacă este necesar, scrieți un constructor implicit sau o funcție SetUp () pentru a pregăti obiectele pentru fiecare test. O greșeală obișnuită este de a scrie SetUp () ca Setup (), cu u mic - nu lăsați să se întâmple asta.
4. Dacă este necesar, scrieți o funcție destructor sau TearDown () pentru a elibera resursele alocate în SetUp ().
5. Dacă este necesar, definiți metode pentru a permite partajarea testelor.

Constructor/Desctructor sau SetUp/TearDown?

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

- Google Test nu reutilizează același obiect de testare în mai multe teste. Pentru fiecare TEST_F, Google Test
  - va crea un obiect “test fixture” nou,
  - va apela imediat SetUp (),
  - va rula corpul de testare,
  - va apela TearDown () și
  - apoi va șterge imediat obiectul dispozitivului de testare.

- Când aveți nevoie să scrieți logica de setare (setup) și de rupere (tear-down) la fiecare test, aveți posibilitatea să alegeți între utilizarea constructorului de testare / destructor sau SetUp () / TearDown ().

Constructor/Desctructor - beneficii

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

- Prin inițializarea unei variabile membre în constructor, avem opțiunea de a face const, ceea ce ajută la prevenirea modificărilor accidentale ale valorii sale și face testele mai clare.
- În cazul în care trebuie să derivăm o subclasă din clasa “test fixture”, constructorul din subclasă apelează mai întâi constructorul clasei de bază, iar destructorul din subclasa apelează ulterior destructorul clasei de bază.
- Cu SetUp () / TearDown (), o subclasă poate face greșeala de a uita să apeleze clasa de bază "SetUp () / TearDown () sau să le apeleze la momentul incorect.

Cazuri (rare) când se utilizează SetUp()/TearDown()

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

- Dacă operația de “tear-down” ar putea arunca o excepție, trebuie să folosiți TearDown (), deoarece aruncarea de excepții într-un destructor duce la un comportament nedefinit.
- Rețineți că multe biblioteci standard (cum ar fi STL) pot arunca atunci când excepțiile sunt activate în compilator. Prin urmare, ar trebui să preferați TearDown () dacă doriți să scrieți teste portabile care funcționează cu sau fără excepții.
- Macrourile pentru aserțiuni aruncă o excepție când se specifică flagul --gtest_throw_on_failure. Prin urmare, nu ar trebui să utilizați aserțiunile Google Test într-un destructor dacă intenționați să efectuați testele cu acest flag.
- Într-un constructor sau un distrugător, nu puteți efectua apeluri de funcții virtuale pentru aceste obiecte. (Puteți apela o metodă declarată ca virtuală, dar va fi legată static.) De aceea, dacă trebuie să apelați o metodă care va fi înlocuită într-o clasă derivată, trebuie să utilizați SetUp () / TearDown ().

macroul TEST_F()

- trebuie utilizat ori de câte ori avem obiecte de tip “test fixture”
- format

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

- primul parametru trebuie să fie numele clasei “test fixture”
- trebuie să definiți mai întâi o clasă “test fixture” înainte de a o utiliza într-o TEST_F ()

Ce se întâmplă în spatele unui TEST_F()

Pentru fiecare test definit cu TEST_F (), Google Test va:

- crea un nou “test fixture” (o nouă trusă de testare) în timpul de execuție
- inițializa cu SetUp ()
- rula testul
- elibera memoria apelând TearDown ()
- șterge “test fixture”.

Rețineți că diferitele teste din același test au obiecte de testare diferite, iar Google Test șterge întotdeauna un element de testare înainte de a-l crea pe următorul.

Google Test nu reutilizează același “test fixture” pentru mai multe teste. Orice schimbări pe care le face un test la “test fixture”, acestea nu afectează alte teste.

Exemplu: calsa care se testează

- Vom considera exemplul Sample3 din https://github.com/google/googletest/tree/master/googletest/samples
- Clasa aflată sub test implementează o listă FIFO

```
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;
  ...
};
```

Exemplu: clasa “test fixture”

```
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_;
};
```

- metoda TearDown() nu e necesară deoarece nu e nimic de curățat după un test

Exemplu: teste

```
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;
}
```

Exemplu: ce se întâmplă la rularea unui test

- Google Test construiește un obiect QueueTestSmpl3 (să-l numim t1).
- t1.SetUp() inițializează t1 .
- Primul test (DefaultConstructor) este rulat peste t1 .
- t1.TearDown() curăță după terminarea testului.
- t1 este distrus.
- Pașii de mai sus sunt repetați cu alt obiect QueueTest, în acest caz testul Dequeue.

Pregătirea testelor

- Fișierele CMakeList.txt și googletest.cmake sunt definite la fel ca la proiectul precedent
- Excepție face olderul ./src, pentru acesta nu mai e necesar fisșierul CMakeList.txt deoarece include doar fișierul sample3-inl.h (în care clasa Queue este descrisă cu implementarea inline a metodelor)
- fișierul antet sample3-inl.h este inclus explicit în sample3_unittest.cc, deoarece clasa QueueTestSmpl3 utilizează instanțe ale clasei Queue

Rularea testelor

```
$ cmake -H. -Bbuild
...
$ cd build
$ cmake --build .
...
$ ./bin/unit_tests
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.
```

Concluzii

- TDD, la fel ca oricare alt concept sau metodă de dezvoltare software, necesită practică.
- Cu cât folosiți mai mult TDD, cu atât devine mai ușor TDD.
- Nu uitați să păstrați testele simple.
- Testele care sunt simple, sunt ușor de înțeles și ușor de întreținut.
- Instrumente precum Google Test, Google Mock, CppUnit, JustCode, JustMock, NUnit și Ninject sunt importante și ajută la facilitarea practicării TDD.
- Dar e bine de știut că TDD este o practică și o filozofie care depășește utilizarea instrumentelor.
- Experiența cu instrumente și framework-uri este importantă, abilitățile dobândite vor inspira încrederea în dezvoltator de aplicații.
- Dar e bine de știut că nu instrumentele ar trebui să fie în centrul de atenție.
- Trebuie acordat timp egal ideii de "test first”.