*UNIVERSITATEA ȘTEFAN CEL MARE SUCEAVA*

FACULTATEA DE INGINERIE ELECTRICĂ ȘI ȘTIINȚA CALCULATOARELOR

PROGRAMUL DE STUDII CALCULATOARE

PROIECT DE DIPLOMĂ

Testarea unitară în C++ - Implementarea unei aplicații de teste

Îndrumător,

Conf.univ.dr.ing. Schipor Andrei-Ovidiu

Student,

Pamparău Cristian

SUCEAVA, 2018

Cuprins

[1. Introducere 4](#_Toc518114715)

[1.1. Descrierea problemei 4](#_Toc518114716)

[1.2. Argument utilizare Linux, QtCreator, CMake 4](#_Toc518114717)

[2. Tehnologii folosite 7](#_Toc518114718)

[2.1. CMake 7](#_Toc518114719)

[2.1.1. Introducere. Ce este CMake? 7](#_Toc518114720)

[2.1.2 Deschiderea unui proiect în QtCreator 8](#_Toc518114721)

[2.1.3 Comenzi folosite 10](#_Toc518114722)

[3. Biblioteci utilizate 13](#_Toc518114723)

[3.1. Google Protocol Buffers 13](#_Toc518114724)

[3.1.1. Generalități 13](#_Toc518114725)

[3.1.2. Detalii tehnice de folosire 13](#_Toc518114726)

[3.1.3. Detalii de compilare în C++ 15](#_Toc518114727)

[3.2. MySql-Connector-C 17](#_Toc518114728)

[3.2.1. Generalități 17](#_Toc518114729)

[3.2.2. Argument dialect SQL 17](#_Toc518114730)

[3.2.3. Detalii de compilare în C++ 17](#_Toc518114731)

[3.3. Google Test. Google Mock 19](#_Toc518114732)

[3.3.1. Generalități 19](#_Toc518114733)

[3.3.2. Detalii tehnice de folosire 19](#_Toc518114734)

[3.3.3. Detalii de compilare în C++ 21](#_Toc518114735)

[4. Detalii de implementare 22](#_Toc518114736)

[4.1. Scrierea de cod testabil 22](#_Toc518114737)

[4.1.1. Introducere 22](#_Toc518114738)

[4.1.2. Elemente netestabile. Fișierul 23](#_Toc518114739)

[4.1.3. Elemente netestabile. Baza de date 25](#_Toc518114740)

[4.2. Scrierea testelor unitare 28](#_Toc518114741)

[4.2.1. Introducere 28](#_Toc518114742)

[4.2.2. Scrierea testelor utilizând framework-ul GoogleTest 29](#_Toc518114743)

[4.2.3. Scrierea testelor utilizând framework-ul GoogleMock 31](#_Toc518114744)

[5. Manual de utilizare 33](#_Toc518114745)

[5.1. Compilarea proiectului 33](#_Toc518114746)

[6. Concluzii 34](#_Toc518114747)

[6.1. Analiza obiective propuse 34](#_Toc518114748)

[6.2. Direcții viitoare de extindere și dezvoltare 35](#_Toc518114749)

[7. Bibliografie 37](#_Toc518114750)

# 1. Introducere

## 1.1. Descrierea problemei

În revista „Today Software Magazine”, Alexandru Bolboacă afirma că „testarea unitară se referă la scrierea unor bucăți de cod, denumite cod de testare, care validează codul de producție. Testarea majorității aplicației devine așadar automată” [1]. Pornind de la acest adevăr, aș vrea să ne imaginăm pentru câteva momente faptul că un grup de programatori au lucrat o perioadă de 3 luni la un produs inedit cu un mare succes pe piață. Oricât de mare ar fi impactul aplicației, dacă echipa nu va extinde aplicația cu noi funcționalități, se poate ajunge în situația în care clienții să aleagă concurența. Din nefericire, confirmarea completă de către echipa de testare poate dura mult, neglijând timpul necesar rezolvării bug-urilor găsite. Prin urmare, este aproape imposibil ca produsul să fie livrat la timp, complet testat. Și totuși, dacă echipa ar putea valida întregul proiect în câteva minute și nu în săptămâni sau luni? O soluție ar fi să apelăm la conceptul de testare unitară (unit testing).

Lucrarea de față este o continuare a lucrării „Testarea unitară în C++ - Implementarea unui proiect testabil” și își propune să abordeze testarea unitară a aplicației ce a stat la baza lucrării anterior amintite, pornind de la un executabil ce presupune rularea unui set de teste unitare.

## 1.2. Argument utilizare Linux, QtCreator, CMake

Concret, în limbaj natural, proiectul pentru care s-au scris unit teste constă într-o aplicație de tip server care pentru un *request* primit, trimite la client un *response*. *Request*-urile primite sunt în formatul XML sau Protobuf, cu convenția că primul caracter este fie X (pentru XML), fie P (pentru Protobuf). În momentul în care un mesaj este primit pe socket, primul pas este acela de a stabili protocolul. Mai apoi, se va decide tipul *request*-ului. În prezent, aplicația suportă și a fost proiectată să „înțeleagă” doar *request*-ul de autentificare, însă design-ul proiectului permite extinderea tipurilor de *request*-uri, de exemplu, *request*-ul de trimitere a unui e-mail în numele clientului. Serverul creat nu va procesa niciun tip de request decât în situația în care clientul a primit un răspuns pozitiv la *request*-ul de autentificare. Orice alt scenariu diferit de acesta va duce la închiderea conexiunii clientului, în unele situații fără ca acesta să fi primit un răspuns.

*Request*-ul de autentificare conține un token codificat AES folosind cheia „graduation\_key” (stocat codificat în baza de date) și presupune verificarea acestuia în baza de date, identificând astfel și *user*-ul corespunzător token-ului primit. Orice *request* primit trebuie să aibă un identificator unic (*id*), identificator care va fi conținut de răspunsul ce se va creea pentru acesta.

La pornirea aplicației, se vor citi parametrii din fișierul de configurare, fișier al cărui titlu va fi preluat din funcția ce inițializează parametrii cu valori implicite (*host*-ul implicit este 127.0.0.1, portul implicit pentru conexiunea la baza de date este 3306 etc.) În situația în care nu există fișierul de configurare pe disc, acesta va fi creat și populat cu parametri impliciți, iar serverul va fi pornit ascultând pentru conexiuni, într-u buclă infinită. Pentru fiecare client care se va conecta, se va crea un fir de execuție (*thread*).

Dezvoltată în limbajul C++, sub Linux, utilizând QtCreator drept IDE și gestionată prin CMake, aplicația permite compilarea și respectiv creerea a niciunui executabil, a executabilului ce reprezintă serverul, sau e executabilului de teste unitare.Orice combinație din cele de mai sus este posibilă, datorită opțiunilor din CMake. Indiferent ce executabil se va creea, codul sursă comun celor două exeuctabile va fi compilat tot timpul și se va genera pentru acesta o librărie statică ce va fi link-editată în momentul compilării executabilelor. Librăria statică denumită „graduation\_project\_library” va avea drept dependințe bibliotecile *rapidxml*, *mysql* și respectiv *google protobuf*. În același fel, executabilul de teste unitare va avea o dependință suplimentară, și anume, *gtest/gmock*.

Am ales să implementez acest proiect utilizând Linux (Debian) drept sistem de operare în primul rând pentru că sunt destul de obișnuit cu dezvoltarea aplicațiilor în C++ pe acest sistem de operare. Experiența personală mi-a dovedit faptul că majoritatea sistemelor sau aplicațiilor de tip server, back-end rulează pe Linux și pot fi manipulate și menținute mult mai ușor, de exemplu, cu ajutorul scripturilor de bash planificate prin *cron*[[1]](#footnote-1), utilitarul fiind disponibil doar pe sisteme ce rulează Linux [2].

Un alt motiv pentru care am optat pentru o distribuție de Linux este faptul că aplicația de tip server va rula tot pe o distribuție de Linux și datorită posibilităților de explorare a unui fișier *core dumped*, de exemplu, generat de un *Segmentation Fault*. De asemenea, utilizarea *sanitizer*-elor drept posibilitate de depanare a unor crash-uri se poate realiza, urmări, analiza și investiga dintr-un terminal, mai ales dacă output-ul unui program este redirectat către un fișier, fișier ce va fi apoi analizat cu funcții de comandă și analiză specifice Linux, cum ar fi *grep*, *awk* etc. De cele mai multe ori, aplicațiile de tip server rulează pe mașini unde nu avem la dispoziție o interfață grafică pentru sistem de operare și nici nu putem deschide un proiect pentru debug și investigații dintr-un mediu de programare. Cele mai multe investigații se vor realiza prin comenzile de bash pentru editarea fișierelor, precum nano, vim etc. Tocmai din acest motiv, proiectul de față nu a fost programat să fie compilabil și pe Windows.

În ceea ce privește mediul de dezvoltare și tehnologia de gestionare a fișierelor și de compilare folosită, am ales să descriu proiectul (fișierele sursă, pași și parametri de compilare și respectriv *linke-ditare*) prin scripturi de CMake dat fiind faptul că se apropie de felul în care se pot descrie comenzi sub Linux. De asemenea, se pot trata și specifica fișierele de compilare sau parametri diferiți pentru sisteme de operare diferite (Mac, Windows, Linux), asigurând astfel o portabilitate destul de mare pentru proiecte. În același timp, cu ajutorul CMake se pot crea diferite tipuri de makefile-uri, precum Xcode pentru proiecte MacOS, proiecte Visual Studio pentru Windows etc.

Pentru a genera și a deschide un proiect în QtCreator, trebuie să se deschidă fișierul de configurare simple plasate în directorul codului sursă (numite fișiere CMakeLists.txt). Fiecare proiect conține un singur fișier de configurare, fișier care poate include la rândul lui alte fișiere cu extensia *cmake* fie pentru compilarea unor module, fie pentru rezolvarea dependințelor externe. Opțiunea pentru QtCreator vine în urma obișnuinței și utilizării acestuia foarte frecvent, pe de o parte, dar și pentru că toate setările și configurările proiectului depind minimal de acesta și foarte mult de CMake, pe de altă parte.

Așa după cum am amintit anterior, token-ul din *request*-ul de autentificare este verificat într-o bază de date pentru a se confirma faptul că cel care încearcă să se conecteze la server este un client cunoscut. În acest sens, am optat pentru MySQL drept server de bază de date în primul rând pentru că este *open-source*, fiind instalat prin managerul de pachete din Linux (pachetele *mariadb-server-10.1* și repectiv *mysql-workbench* pentru interfața grafică). Totodată, în momentul de față aplicația nu necesită multă cantitate de memorie pentru a funcționa eficient, în baza de date stocându-se informații relativ puține. Un alt motiv pentru care am făcut această alegere este dat de faptul că multe dintre cele mai mari organizații din lume, printre care Facebook, Google, Adobe se bazează pe MySQL pentru a economisi resurse temporale și financiare.

# 2. Tehnologii folosite

## 2.1. CMake

### 2.1.1. Introducere. Ce este CMake?

Conform documentației oficiale, CMake este o familie de instrumente *open-source*, multi-platformă destinate să construiască, să testeze și să împacheteze software-ul [3]. Acestă tehnologie este folosită pentru a controla procesul de gestionare și compilare a software-ului utilizând simple fișiere de configurare independente de platformă și compilatoare. CMake este capabil să genereze mediu de compilare nativ pentru compilarea codului sursă, crearea de biblioteci, generarea de așa-numitele *wrappere* și construirea executabilelor în combinații arbitrare. De asemenea, tehnologia suportă generarea de librării statice și dinamice. O altă caracteristică interesantă a tehnologiei în discuție este aceea că generează un fișier cache conceput pentru a fi utilizat cu un editor grafic. De exemplu, atunci când CMake rulează, acesta poate localiza fișiere, biblioteci și executabile și poate intâlni directive directe de construire. Aceste informații sunt adunate în cache, care pot fi modificate de programator înainte de generarea fișierelor native compilate.

CMake a fost proiectat să suporte ierarhii de directoare complexe și aplicații dependente de mai multe biblioteci. De exemplul, această tehnologia suportă proiectele constând din mai multe seturi de instrumente (adică biblioteci), unde fiecare set de instrumente poate conține mai multe directoare, iar aplicația depinde de acestea la care se adaugă codul scris de programator. Tehnologia se poate, de asemenea, ocupa de situații în care executabilele trebuie să fie construite(*build-uite*) pentru a genera codul care apoi este compilat și legat într-o aplicație finală.

Utilizarea tehnologiei este simplă. Procesul de build este controlat prin crearea a unul sau mai multe fișiere CMakeLists.txt în fircare director (inclusiv subdirectoarelor) care alcătuiesc un proiect. Fiecare fișier CMakeLists.txt conține una sau mai multe comenzi. Fiecare coamdă are forma *COMMAND (args ....)*, unde *COMMAND* este numele unei comenzi și *args* reprezintă lista de argumente separate prin spațiu. CMake pune la dispoziție o multitudine de comenzi predefinite, însă se pot crea defini și alte comenzi. În plus, programatorul poate adăuga și alte generatoare de *makefile*-uri pentru o combinație specială de compilatoare sau sisteme de operare.

Suita de instrumente CMake a fost creată de Kitware ca răspuns la necesitatea unui mediu puternic de construcție cross-platform pentru proiecte open-source precum ITK și VTK.

Un exemplu minimal de fișier CMakeLists.txt care consturiește un fișier executabil cu denumirea „Exemplu” pe baza unui singur fișier executabil „main.cpp” este prezentat în fig.2.1.

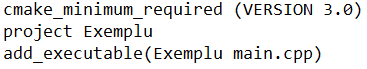


fig.2.1. Exemplu fișier CMakeLists.txt

### 2.1.2 Deschiderea unui proiect în QtCreator

Așa după cum am menționat în secțiunea anterioară, pentru compilarea unui proiect este necesar cel puțin un fișier cu numele *CMakeLists.txt*. Dat fiind faptul că lucrarea de față are la bază un proiect ce folosește biblioteci externe, dar și construirea a două executabile/module distincte ce se pot rula independent, pentru a fi cât mai ușor de înțeles, am ales ca acest fișier să conțină fișiere/scripturi *.cmake* organizate într-o structură de directoare pe care o vom detalia în continuare. Astfel, în directorul principal al proiectului, am creat directorul *cmake*, director ce conține la rândul său un director *lib*, trei fișiere pentru compilarea/construirea unei librării statice și respectiv a două executabile (*build\_library.cmake, build\_server.cmake, build\_unit\_tests.cmake*), precum și fișierul *dependencies.cmake*, fișier ce va enumera bibliotecile externe care vor fi descărcate (copiate) și compilate pentru a fi utilizate în cele trei fișiere anterior amintite. În acest sens, directorul *lib* conține fișiere de tipul *resolve\_<library\_name>.cmake*, unde *library\_name* reprezintă numele bibliotecii pe care acesta o va aduce în directorul de build, o va compila cu parametrii specificați în fișier și va genera calea pentru include, locația dar și numele bibliotecii care va fi generată și legată la executabilele finale. Un ultim fișier aflat în directorul *lib* este *ExternalProject.cmake*, fișier pus la dispoziție de către CMake care va conține atributul *SOURCE\_SUBDIR* al comenzii *ExternalProject\_Add* folosit pentru compilarea bibliotecii Google Protocol Buffers.

Proiectul a fost dezvoltat utilizând QtCreator versiunea 4.4.0. bazat pe versiunea Qt 5.7.1. Pentru deschiderea proiectului, se va alege opțiunea *File -> Open File or Project.* Pe ecran va apărea o fereastră cu titlul *Open File* unde se va accesa directorul proiectului, se va alege fișierul *CMakeLists.txt*, apoi se va efectua click pe butonul *Open*. După acest pas, pe ecran va apărea o fereastră similară cu cea din figura fig.2.2.

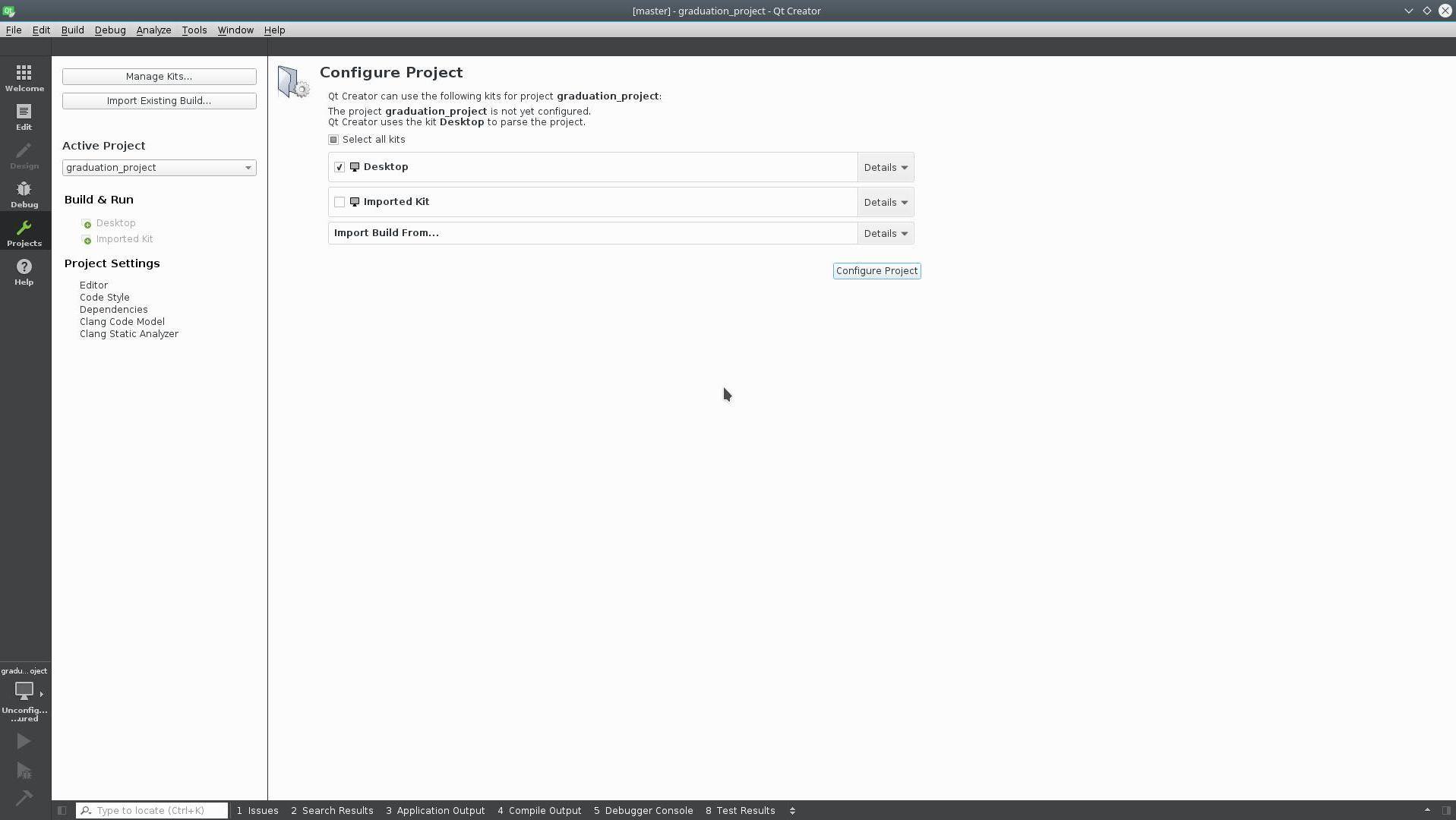


fig.2.2. Prima deschidere a proiectului în QtCreator 4.4.0.

După cum se observă, în partea stângă există un buton denumit *Manage Kits* unde vor exista opțiuni precum *Environement, Text Editor, C++, Debugger, Analyzer, Designer, Devices*, cel mai important fiind *Build&Run*. În această secțiune, în submeniul *Kits,* există prestabilit Kit-ul *Manual-Desktop(default).* Un astfel de kit este definit de caracteristici dintre care cele mai importanți sunt : numele kit-ului (în cazul de față *Desktop*), tipul dispozitivului (posibile variante : *Desktop*, *Android Device*, *QNX* *Device* și respectiv *Generic Linux Device*, dar în cazul de față este selectată ultima opțiune), compilatorul folosit (detectat automat din sistem locația compilatorului de C și respectiv C++ din */usr/bin*), debugger-ul folosit (detectat automat *System GDB* din /usr/bin/gdb), instrumentul CMake (detectat automat *System CMake* din /usr/bin/cmake), generatorul CMake precum și configurările CMake.

În ferestra stângă, în mod automat, este completată caseta text *Build directory* ca fiind un alt director în același director cu locația proiectului. Numele directorului recomandat de QtCreator are sintaxa : *build-<Numele\_proiectului\_detectat\_automat>-<Numele kit-ului>-<Tipul de build>,* unde tipul de build este una din următoarele variante : Debug, Default, Minimum Size Release, Release, Release with Debug Information. În mod automat este ales tipul *Default*. Astfel, în cazul de față, directorul recomandat are denumirea *build-graduation\_project-Desktop-Default.* Sub această fereastră, va apare o casetă cu perechi chei-valoare de setări și opțiuni ale proiectului (cele implicite și respectiv cele detectate automat din fișierele de CMake încărcate). După alegerea unui kit, la un click pe opțiunea *Edit* din partea stângă, va apărea următoarea fereastră.

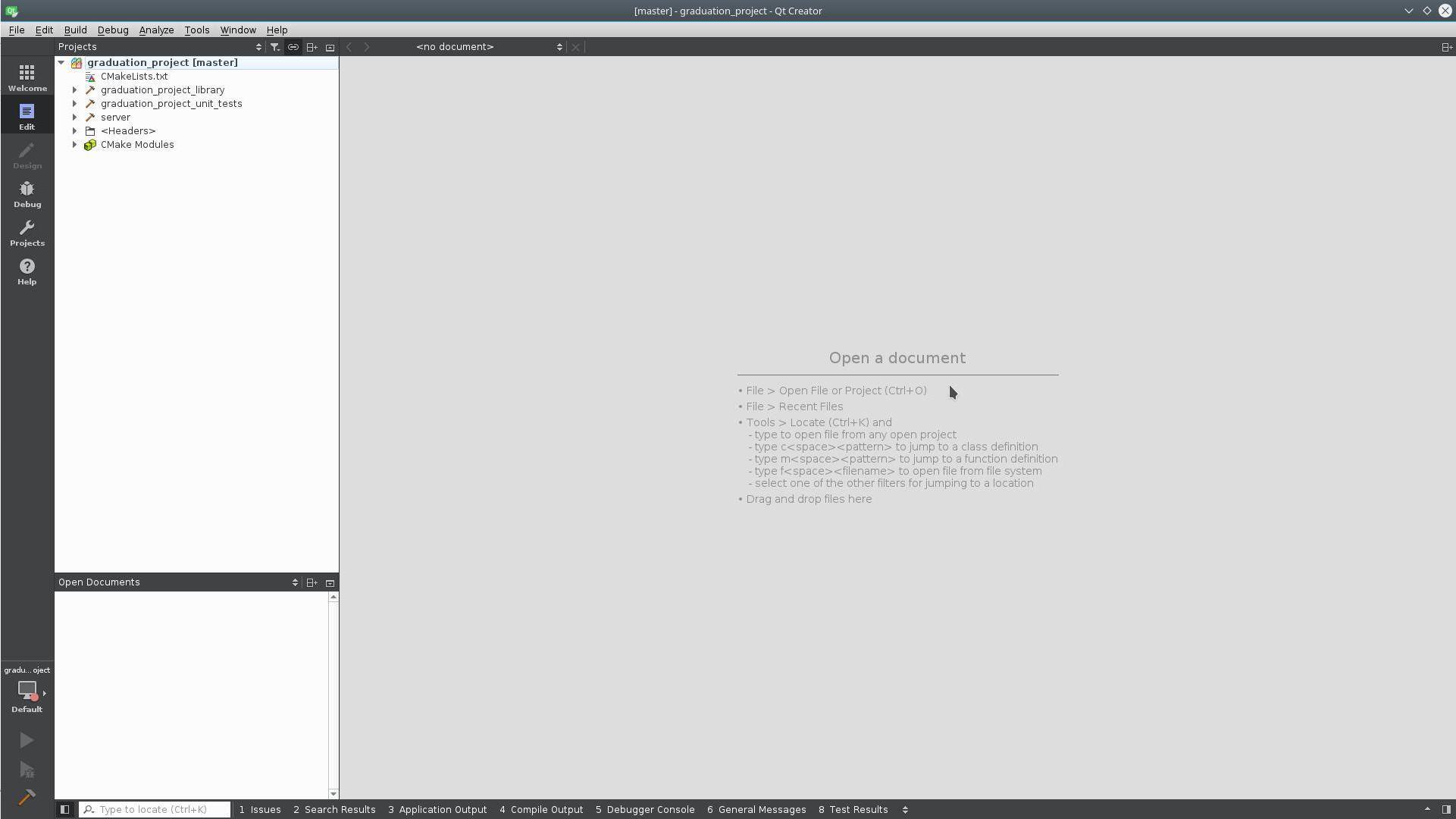


fig.2.3. Fereastra apărută la analiza fișierelor cmake cu succes.

Este important de precizat faptul că după selectarea kit-ului, QtCreator va începe analiza fișierului CMakeLists.txt (și a fișierelor incluse de către acesta). Doar în situația în care sintaxa cmake este respectată, iar pe disc există toate fișierele specificate în aceste scripturi, pe ecran va fi afișată o imagine asemănătoare ca cea din figura anterioară. Altfel, va fi afișat doar numele proiectului, branch-ul de pe care a fost încărcat (master) și fișierul CMakeLists.txt. După rezolvarea tuturor erorilor și a conflictelor detectate, la o nouă rulare a comenzii cmake vor apărea directoarele cu fișierele sursă. Un director este asociat fie unui nume de executabil/nume de bibliotecă sau a unui namespace.

### 2.1.3 Comenzi folosite

În continuare, voi prezenta pe scurt principalele comenzi CMake utilizate.

* cmake\_minimum\_required(VERSION versiune\_folosită) – Setează versiunea minimă pe care proiectul o va folosi. Dacă versiunea curentă a CMake instalat pe sistem este mai mică decât valoarea specificată cu această comandă, se va opri procesarea proiectului și se va raporta o eroare de tipul „CMake <versiunea\_din\_comanda> or higher is required. You are running version <versiunea\_instalată\_pe\_sistem>”. În cazul proiectului de față, a fost utilizată versiunea 3.1.
* project(<nume > [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]] [LANGUAGES <language-name>...]) - Setează numele, versiunea și activează un limbaj pentru întregul proiect. Numele proiectului poate fi referit în scripturile cmake prin variabila ${PROJECT\_NAME}
* option(<nume\_opțiune> <descrierea\_opțiunii> <valoare\_inițală>) – Oferă o opțiune pe care programatorul o poate seta pe ON/OFF (valoarea poate fi una dintre acestea două). Opțiunile folosite în proiect sunt *build\_unit\_tests* și respectiv *build\_server*. În funcție de valorile acestor două opțiuni, se vor incluse sau nu fișierele ce vor rezolva dependințele modulelor, se vor adăuga sau fișiere pentru a fi compilate și se vor lega diferite librării. Acestă opțiune poate fi condiția logică a unei instrucțiuni if CMake

*if (nume\_optiune)*

*# comenzi de CMake de executat*

*endif(nume\_optiune)*

* include\_directories(<calea\_directorului>) – Adaugă directorul <calea\_directorului> acelora pe care compilatorul le va folosi pentru a căuta fișiere incluse în fișiere sursă. Exemplu : include\_directories(src)
* include(<file/module>) – Încarcă și rulează comenzi CMake dintr-un fișier sau modul specificat.Exemplu : include(cmake/lb/resolve\_mysql.cmake)
* set(<variable> <value>) – Setează variabila <variable> la valoarea <value>. Exemplu : set(protobuf\_library\_name libprotobuf.a)
* add\_dependencies(<target> [<target-dependency>]...) – Adaugă <target>-ului dependințele care urmează în lista de argumente separate prin spațiu. Exemplu : add\_dependencies(graduation\_project\_library mysql).
* add\_library(<nume> [STATIC | SHARED | MODULE] [EXCLUDE\_FROM\_ALL] source1 [source2 ...]) - Crează și adaugă o librărie (static/shared/module) la proiect pe baza fișierelor sursă specificate.
* add\_executable(<nume> [WIN32] [MACOSX\_BUNDLE] [EXCLUDE\_FROM\_ALL] source1 [source2 ...]) – Crează și adaugă un executabil la proiect utilizând fișierele specificate.
* target\_link\_libraries(<target> [item1 [item2 [...]]] [[debug|optimized|general] <item>] ...) – Leagă un target (bibliotecă, executabil) la una sau mai multe biblioteci.
* message([<mode>] "message to display" ...) – Afișează un mesaj pentru utilizator
* ExternalProject\_Add(<nume> <lista\_de\_comenzi>) – Creează un target (de exemplu o bibliotecă) specializat care va descărca (copia), actualiza,configura, construi(build), instala sau chiar testa un proiect extern. Cele mai uzuale comenzi sunt :

1. URL – Calea completă a sursei de unde va fi adus proiectul. Acesta poate fi calea completă a unui fișier de pe disc, caz în care se va efectua o copiere, sau o adresă web, caz în care se va efectua o descărcare a surselor.
2. SOURCE\_DIR – Directorul unde se vor afla fișierele sursă ce vor fi compilate
3. DINARY\_DIR – Directorul unde se dorește a se realiza compilarea și creearea executabilelor.
4. CMAKE\_ARGS – În cazul în care target-ul este un alt proiect ce se va compila cu CMake, aici se vor specifica comenzile principale de rulate a acestuia, cum ar fi CMAKE\_BUILD\_TYPE (Debug/Release) sau alte opțiuni definite de respectivul proiect.
5. BUILD\_IN\_SURCE – Poate fi ON/OFF și semnifică faptul că SOURCE\_DIR coincide cu BINARY\_DIR. De regulă, această opțiune se utilizează pentru bibliotecile header-only.

# 3. Biblioteci utilizate

## 3.1. Google Protocol Buffers

### 3.1.1. Generalități

Protocol Buffers (referit în continuare protobuf) reprezintă un mecanism Google independent de limbaj, independent de platformă, extensibil pentru serializarea datelor structurate (similar cu XML, dar mai mic, mai rapid și mai simplu) [4]. Programatorul trebuie să își definească o singură dată modul în care datele să fie structurate, apoi se poate utiliza codul sursă generat special pentru a scrie și citi datele către și de la o varietate de fluxuri de date și utilizând o varietate de limbaje de programare. De asemenea, se poate actualiza structura de date fără a influența negativ programele implementate care au fost compilate cu structura anterioară.

În ceea ce privește avantajele acestei bilioteci pentru serializarea datelor structurate, spre deosebire de XML, protobuf este mai simplu, este de la 3 la 10 ori mai mic, de la 20 la 100 de ori mai rapid, conferă ambiguitate redusă și generează clase de acces la date care sunt ușor de folosit.

### 3.1.2. Detalii tehnice de folosire

Modul de funcționare al bibliotecii presupune că programatorul să își definească felul în care informațiile vor fi structurate prin definirea mesajelor protobuf în unul sau mai multe fișiere cu extensia *proto*. Fiecare mesaj protobuf este o înregistrare logică mică a informațiilor, care conține o serie de perechi nume-valoare. În continuare se va prezenta un exemplu de fișier proto care va defini un mesaj ce va conține informațiile despre o persoană.

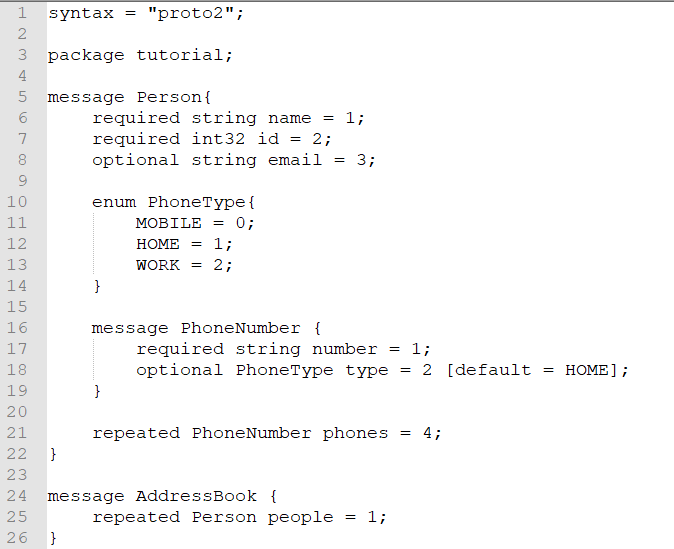


fig.3.1. Exemplu de fișier proto

Așa după cum se poate observa, formatul mesajului este foarte simplu – fiecare tip de mesaj are unul sau mai multe câmpuri numerotate unic și fiecare câmp are un nume și un tip de valoare, unde valorile pot fi numere (întreg sau în virgulă mobilă), booleani, șiruri de caractere, octeți de date bruți, sau chiar (ca în figura de mai sus) alte mesaje protobuf, permițând astfel o structurare de date ierarhică. De asemenea, se pot specifica faptul ca unele câmpuri să fie opționale, altele obligatorii, iar altele să fie repetate.

După definirea tipurilor de mesaje, se va executa compilatorul protobuf pentru limbajul de programare dorit care va genera automat clasele de acces la date. Acestea, odată generate, vor oferi accesori simpli pentru fiecare câmp (cum ar fi *name()* și *set\_name()* ), precum și metodele de serializare/procesare a întregii structuri în/din octeți de date bruți. Deci, dacă limbajul de programare ales este C++, exemplul de mai sus va genera o clasă numită *Person*. Apoi, se va putea utiliza această clasă în aplicația programatorului pentru a popula, serializa și a prelua mesaje protobuf *Person.* Codul care s-ar putea scrie (în cadrul spațiului de nume *tutorial* – detalii în secțiunea următoare) este descris în figura fig.3.2.

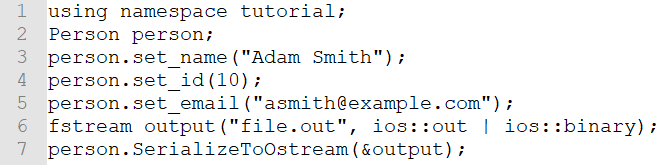


fig.3.2. Exemplu de utilizare al clasei protobuf generate – serializare

Apoi, mai târziu se va putea citi mesajul scris anterior în felul următor:

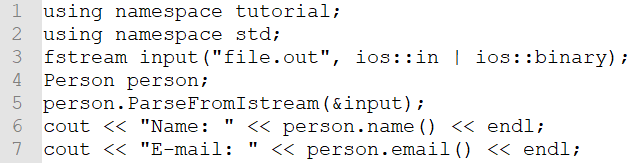


fig.3.3 Exemplu de utilizare al clasei protobuf generate – procesare

Așa cum am menționat, programatorul poate, de asemenea, să adauge câmpuri noi unui mesaj din fișierul proto. Vechile binare vor ignora pur și simplu noul câmp în momentul procesării. Deci, dacă un protocol de comunicații folosește protobuf ca format de date, acesta se poate extinde fără probleme.

### 3.1.3. Detalii de compilare în C++

După cum se observă în fig.3.1, fișierul proto începe cu o declarație a pachetului care ajută la prevenirea conflictelor de numire între diferite proiecte. În C++, clasele generate vor fi plasate într-un spațiu de nume care se potrivește cu numele pachetului.

În pasul următor, pe baza fișierului proto vor trebui generate clasele de care programatorul va avea nevoie pentru a citi și scrie mesajele *AddressBook*. Pentru a face acest lucru, va trebui rulat compilatorul *protoc* al protobuf pe fișierul *.proto*. Pentru aceasta, se va descărca pachetul pentru instalarea *protoc*. Următorul pas îl constituie rularea compilatorului *protoc* cu specificarea directorului sursă (locația fișierelor sursă ale proiectului – dacă nu se va furniza o valoare, se va utiliza directorul curent), directorul destinație (locația unde se dorește a se genera fișierele) și calea spre fișierul *.proto* . În cazul de față, se va rula următoarea comandă:



fig.3.4. Exemplu de generare a fișierelor protobuf

Deoarece limbajul folosit este C++, s-a utilizat opțiunea *–cpp\_out*. Comanda anterior rulată, va genera în directorul specificat două fișiere:

* *addressbook.pb.h*, fișierul antet care declară clasele utilizate;
* *addressbook.pb.cc,* ce va conține implementarea claselor declarate.

În fișierul *resolve\_google\_protobuf.cmake* care rezolvă dependința bibliotecii în discuție, se vor genera fișierele protobuf daca acestea nu există, sau dacă există și sunt vide. Practic, dacă se va dori adăugarea unui nou mesaj în fișierul proto, fie se vor șterge fișierele *\*.pb.\**, fie se va ștegre conținutul din aceste fișiere, după care se va rula din nou comanda cmake.

Comanda *ExternalProject\_Add* ce este folosită pentru protobuf folosește o serie de parametri. Un primul parametru este numit *URL*, care indică spre un director ce face parte din codul sursă al proiectului; astfel, la o primă compilare a proiectului se va efectua o copie a surselor în locatia SOURCE\_DIR, în locul unei descărcări.

În ceea ce privește argumentele cmake pentru compilarea subproiectului protobuf s-a optat pentru o compilare în modul Release, CMAKE\_PREFIX\_INSTALL indicând spre BINARY\_DIR și dezactivarea rulării testelor protobuf.

Un ultim parametru important de specificat pentru această comandă este SOURCE\_SUBDIR care indică spre numele subdirectorului din directorul clonat ce va conține fișierul CMakeLists.txt. Acest parametru se poate folosi grație importării fișierului *ExternalProject.cmake*, fișier pus la dispoziție de CMake.

Așa după cum am menționat anterior, tot în fișierul resolve pentru protobuf se verifică dacă fișierele *\*.pb.\** există și sunt vide, sau dacă nu există. Acest lucru s-a făcut prin comanda *file* care verifica existența și/sau dimensiunea unui fișier existent pe disc, dar și prin comanda *add\_custom\_comand*. Pentru această comandă, s-a specificat faptul că executarea va avea loc POST\_BUILD (după ce librăria protobuf a fost compilată) și respectiv comanda similară celei din fig.3.4.

## 3.2. MySql-Connector-C

### 3.2.1. Generalități

Software-ul MySQL oferă un server de baze de date SQL (Structured Query Language) foarte rapid, multithreaded, multi-utilizator și robust. Serverul MySQL este destinat sistemelor de producție cu sarcină critică, grele, precum și pentru integrarea software-ului dezvoltat în masă. MySQL este o marcă înregistrată a Oracle Corporation și a afiliaților săi [5].

MySQL este un sistem de gestiune a bazelor de date operaționale , fiind cel mai popular SGBD open-source la ora actuală. Frecvent utilizat în combinație cu limbajul de programare PHP, se pot dezvolta diverse tipuri de aplicații scrise în multe limbaje de programare dat fiind faptul că există multe scheme API disponibile. Limbajele de programare care suportă conectori MySQL sunt: C, C++, C#, Java, Perl, PHP, Python etc.

În ceea ce privește administrarea bazei de date se poate folosi linia de comandă, sau prin descărcare, o interfață grafică : MySQL Administrator, MySQL Query Browser, MySQL Workbench. În cadrul proiectului de față, s-a utilizat MySQL Workbench.

### 3.2.2. Argument dialect SQL

Dat fiind faptul că proiectul a fost dezvoltat sub Linux (Debian), datorită bunelor practici în ceea ce privește utilizarea unei distribuții de Linux, am fost nevoit să aleg un server de bază de date care să poată fi instalat cu ajutorul managerului de pachete. În acest sens, pentru server am instalat pachetul *mariadb-server-10.1*, iar pentru interfața grafică am fost nevoit să instalez pachetul *mysql-workbench.*

Un alt argument al utilizării acestui dialect SQL a fost, pe lângă popularitatea acestuia, faptul că multe dintre cele mai mari organizații din lume (Facebook, Google, Adobe, Alcatel Lucent, Zappos) se bazează pe MySQL pentru a economisi timp și bani prin alimentarea site-urilor Web de mare volum, a sistemelor critice pentru afaceri și a software-ului ambalat.

### 3.2.3. Detalii de compilare în C++

MySQL pune la dispoziție, așa după cum am văzut anterior, un set de conectori pentru diferite limbaje de programare. Acești conectori sunt biblioteci compilate, arhivate. Din acest motiv, în sursele proiectului de unde se face clonarea (copierea) dependințelor, în directorul *mysql*, există un director *lib* ce conține biblioteca *libmysqlclient.a*, bibliotecă deja compilată. În acest sens, fișierul *resolve\_mysql.cmake* nu va conține decât calea pentru fișierele de inclus (include\_path: *mysql/include*) și respectiv calea bibliotecii ce se va folosi la link-editarea executabilului final (*mysql/lib/libmysqlclient.a*). Directorul *lib* conține de asemenea biblioteca și cu extensia *.so*, astfel încât să se poată asigura portabilitatea proiectului pe diferite sisteme de operare.

## 3.3. Google Test. Google Mock

### 3.3.1. Generalități

Google Test (referit în continuare gtest) este o bibliotecă ce oferă suport pentru testarea unitară pentru limbajul de programare C++ dezvoltat de Google [6]. Testele scrise folosind *gtest* se scriu prin a se defini afirmații (macro-uri de testare prefixate cu ASSERT, EXPECT), care verifică dacă o anumită condiție este adevărată sau nu. Rezultatul afirmației poate fi succes, eșec nefatal (programul continuă și rulează și testele rămase de executat) sau eșec fatal (programul se oprește și nu mai rulează celelalte teste). Dacă apare un eșec fatal, atunci scopul curent este întrerupt; altfel, programul continuă în mod normal (eșecul fatal este returnat de mulțimea de macro-uri ce sunt prefixate cu ASSERT, în timp ce eșecul nefatal este cauzat de macro-urile prefixate cu EXPECT).

Testele sunt alcătuite din o mulțime de afirmații care verifică comportamentul codului testat. Spunem despre un test că nu reușește (*fails*) în situația în care acesta se blochează sau are o afirmație nereușită; în caz contrar, testul reușește (*passed*, *succeed*).

Un caz de testare conține unul sau mai multe teste. Este recomandat ca testele să fie grupate în cazuri de testare care reflectă structura codului testat. În situația în care mai multe teste într-un caz de testare necesită partajarea anumitor obiecte și funcții, acestea se pot pune într-o clasă de testare (test fixture, TEST\_F). În final, spunem faptul că un program de teste unitare poate conține unul sau mai multe cazuri de test.

Google Mock (referit în continuare gmock) este o extensie a Google Test pentru scrierea și utilizarea claselor de machete C++ [7]. Printre specificațiile bibliotecii amintim faptul că *gmock* permite crearea claselor mock triviale utilizând simple macro-uri, suportă o serie bogată de „potrivitori” ( din engleză *matchers*) și acțiuni, ocupându-se de așteptări (macro-uri de testare prefixate cu EXPECT) neordonate, parțial ordonate sau complet ordonate.

### 3.3.2. Detalii tehnice de folosire

În ceea ce privește modul de folosire al bibliotecilor *gtest* și respectiv *gmock* vom menționa faptul că un executabil ce rulează teste unitare este format din două componente: colecție de fișiere antet/sursă ce conțin scenarii/cazuri de testare (modul cum se vor scrie acestea, politici și bune practici vor fi dezvoltate în capitolul următor) și un fișier sursă principal (poate fi *main.cpp)* al cărui conținut este recomandat și va fi redat în figura fig.3.5.

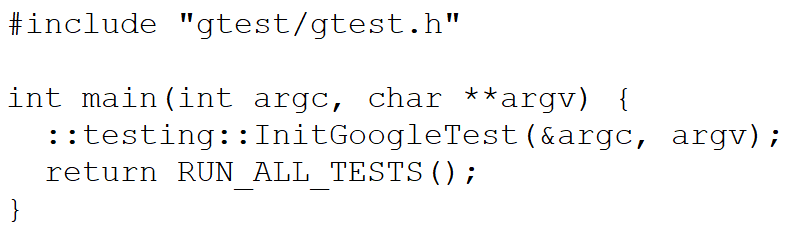


fig.3.5. Fișierul main (recomandat) pentru teste unitare

Funcția *InitGoogleTest* analizează linia de comandă pentru flag-urile suportate de Google Test și elimină toate flag-urile necunoscute. Acest lucru permite programatorului să controleze comportamentul unui program de testare prin diferite opțiuni. Important este faptul că această metodă trebuie apelată înainte de a apela metoda *RUN\_ALL\_TESTS* dacă opțiunile specificate în linia de comandă se doresc a fi luate în considerate.

Așa după cum s-a menționat anterior, programatorul își poate personaliza comportamentul testelor prin specificarea în linia de comandă a diferitelor opțiuni. În continuare se vor prezenta o serie de astfel de opțiuni.

1. *--gtest\_death\_test\_style*. Acest parametru poate avea una din valorile *fast* sau *threadsafe.* Dacă valoarea parametrului este prima specificată, atunci declarația testului de ieșire este executată imediat. Altfel, procesul execută din nou testul așa cum a fost invocat inițial, însă va adăuga *flag*-uri suplimentare pentru a determina testul unic considerat a fi problema.
2. *--gtest\_list\_tests*. Suprascrie toate opțiunile și afișează cazurile de testare și numele fiecărui test.
3. *--gtest\_filter.* Rulează testele ce se vor specifica prin acest parametru. Un test se poate specifica prin *caz\_test*:*nume\_test*. Însă, acest parametru poate primi și valori precum *caz\_test*.\* (ceea ce înseamnă că se vor rula toate testele din cazul de test denumit *caz\_test*), *caz\_test*.\*-*nume\_test* (se vor rula toate setele din cazul de test denumit *caz\_test*, cu excepția testului *nume\_test).*

Lista tuturor opțiunilor ce pot fi executate se poate afișa rulând executabilul testelor unitare cu parametrul *–help*.

### 3.3.3. Detalii de compilare în C++

În cadrul proiectului *googletest* al utilizatorului GitHub *google* există două directoare pentru *gtest* și respectiv *gmock*. Astfel, compilarea dependinței de testare implică generarea a două biblioteci, *libgmock.a* și respectiv *libgtest.a .* În acest sens, se va folosi drept URL (parametru al comenzii *ExternalProject\_Add* ) locația din cadrul proiectului ce indică spre directorul sub-proiectului *googletest,* locație ce face parte din proiectul lucrării de față.

Dat fiind faptul că proiectul *gtest* nu este header-only, acesta va fi compilat, fiind necesară legarea acestuia la proiectul principal. În acest sens, vor trebui setate valori pentru SOURCE\_DIR și respectiv BINARY\_DIR. Valoarea parametrului BINARY\_DIR este *${external\_projects\_base\_dir}/Install/googletest*, în timp ce locația pentru SOURCE\_DIR este *${external\_projects\_base\_dir}/Source/googletest.*

Un ultim aspect important de precizat pentru compilarea acestor două biblioteci îl constituie modul de compilare, ce are loc în modul Release, lucrul specificat prin parametrul CMAKE\_ARGS cu valoarea *-DCMAKE\_BUILD\_TYPE=Release.* Evident, dacă vor fi specificate manual căile pentru include și căile pentru biblioteci, comanda *ExternalProject\_Add* nu va fi executată, lucru ce se întâmplă în cazul tuturor dependințelor.

# 4. Detalii de implementare

## 4.1. Scrierea de cod testabil

### 4.1.1. Introducere

În esență, un test unitar (unit test) este o metodă care instanțiază o mică parte dintr-o aplicație și verifică comportamentul acesteia independent de alte părți. Proiectanții de software ar trebui să țină seama de acest aspect și să contureze o arhitectură generală care să permită folosirea acelorași nume de metode, cu același tip returnat și cu aceleași argumente atât în executabilul pentru teste, cât și în executabilul final, cel real. Pentru a putea realiza acest lucru, este important de stabilit întâi care sunt entitățile, conceptele „problemă” astfel încât, în final, aplicația ce va rula unit testele să ruleze independent și să nu trimită anumite *request*-uri la o bază de date, la sistemul de operare pentru ora sau data curentă, să citească din fișier și așa mai departe. Astfel, ne punem problema codului duplicat și trebuie să stabilim clasele, entitățile din program care vor fi comune atât în codul pentru teste, cât și în codul „real”. Având în vedere aceste noțiuni, în cadrul proiectului de față, conceptele ce comunică cu exteriorul aplicației sunt: conceptul de fișier, conceptul de bază de date și respectiv conceptul de socket. Soluția la care s-a apelat a fost transformarea acestor concepte în noțiuni și elemente testabile, mai puțin socket-ul. Acest element din urmă nu se poate testa cu ajutorul testelor unitare, el putând fi testat prin testele de integrare[[2]](#footnote-2) (*integration tests*).

Vom denumi în continuare drept element testabil o colecție de spații de nume corespunzătoare unei anumite entități care asigură testabilitatea codului prin scrierea unei interfețe și prin două, sau chiar trei tipuri de implementări: implementare reală (folosită în cadrul executabilului ce va produce serverul), implementare mock (folosită în scenariile de unit teste ce vor folosi biblioteca *gmock*) și respectiv implementare fake (folosită în scenariile ce vor avea drept scop folosirea bibliotecii *gtest*, printr-o implementare ce va simula conceptul real cu ajutorul unor structuri de date). Cu toate că în cadrul limbajului C++ nu există conceptul de interfață așa cum îl întâlnim în limbajul Java, se va folosi în continuare acest termen cu specificarea faptului că această clasă va conține interfața publică a entității ce se dorește a fi testabilă. În unele cazuri, interfața poate conține și anumiți membri sau metode ce vor fi implementate, nefiind metode virtuale pure. În cazul implementării mock, clasa va fi alcătuită dintr-o serie de macro-uri puse la dispoziție de biblioteca *gmock*, specificând parametrii importanți ce definesc o metodă : tipul returnat, numele metodei, numărul de argumente precum și tipul acestora. Pe de altă parte, implementarea fake, așa după cum s-a menționat anterior, cuprinde seria de metode impuse de interfață și folosește intern anumite structuri de date (vectori, string-uri, pointeri la char etc) cu ajutorul cărora se va simula comportamentul real al funcției din implementarea reală. De exemplu, în cazul conceptului de fișier, în implementarea reală se vor folosi, evident, apeluri pe *fstream*. O metodă de a simula scrierea/citirea în/din fișier cu scrierea/citirea în/din o instanță de *fstream* este utilizarea unei instanțe de *stringstream.* În acest fel, se vor analiza toate operațiile suportate de interfața publică și se vor găsi implementări oarecum similare pentru operațiile care nu sunt identice. Prin operații identice înțelegem acele operații care se pot aplica direct asupra ambelor instanțe (de fstream și respectiv de *stringstream*), precum aplicarea operatorului <<. Există însă și operații, ca de exemplu metoda *open* (pentru *fstream*), care nu sunt implementate în clasa *std::stringstream*. Cu toate acestea, putem să ne gândim la faptul că deschiderea fișierului poate fi similară cu alocarea unui spațiu de memorie al instanței de *stringstream*. În sfârșit, după cum se poate deduce ușor, implementarea reală va efectua apeluri directe pe instanțe ce vor comunica cu exteriorul aplicației.

După identificarea elementelor „problemă” și transformarea lor în elemente testabile, metodele puse la dispoziție de acestea trebuie să fie apelate prin interfața publică. Prin urmare, în orice clasă ce se vor folosi instanțele elementelor testabile, constructorii acestora trebuie să primească drept parametri referințe ale interfețelor, salvându-se apoi în câmpuri ale clasei. Așa după cum se observă, scrierea de cod testabil poate face uneori ca proiectarea codului să fie greoaie, necesitând o analiză mult mai minuțioasă. Totodată, instanțele reale ale conceptelor (implementarea reală) vor trebui injectate din constructor în constructor, de la cel mai înalt nivel până la clasele cele mai de jos din arhitectura claselor ce vor folosi respectivele noțiuni.

### 4.1.2. Elemente netestabile. Fișierul

Unul dintre cerințele testelor unitare spune despre faptul că acestea trebuie să ruleze foarte repede (la nivel de milisecunde), dat fiind faptul că într-un proiect major pot exista mii de teste. În consecință, este recomandat ca acestea să nu comunice cu exteriorul aplicației. În secțiunea anterioară s-a explicat termenul de element testabil sau netestabil. Așa după cum am precizat, noțiunea de fișier este un element, o entitate ce nu se încadreză în acest aspect al testelor unitare. Din acest motiv, s-a explicat modalitatea prin care fișierul poate fi transformat în entitate sau element testabil. Astfel, în cadrul spațiului de nume *graduation\_project* s-a creat un alt spațiu de nume *file*. În cadrul acestuia din urmă se vor implementa toate funcționalitățile pentru fișiere ce vor fi utilizate mai departe în proiect. Pentru o ușoară înțelegere a codului și implicit a proiectului, am ales să organizez fiecare tip de implementare într-un spațiu de nume distinct. În acest sens am definit patru spații de nume cu titlu sugestiv: *interf, mock, fake, real.* În fiecare din acest spațiu de nume există un fișier antet și un fișier sursă, toate având numele de *file* (*file.hpp* și respectiv *file.cpp*)*.*

Un aspect important de precizat este faptul că fiecărui spațiu de nume îi corespunde fizic un director pe disc, în timp ce fiecărei clase îi corespunde fizic un set de fișiere (antet și sursă). Acest lucru face mai ușor de înțeles și de urmărit arhitectura proiectului atunci când acesta este deschis în mediul de programare utilizat, QtCreator.

Tot legat de modul de proiectare și bune practici în scrierea codului, precizez faptul că fiecare fișier antet utilizează drept header guard locația fișierului în structura de directoare și fișiere de pe disc, sau, conform paragafului anterior, „locația” clasei ce va fi definită în arhitectura claselor. Dacă, de exemplu, vom avea clasa *file* de tip interfață, urmează de aici că această clasă va fi declarată într-un fișier *file.hpp* din directorul *interf*, director care face parte la rândul lui din cadrul directorului *file,* iar acesta din urmă este plasat în directorul principal al proiectului, *graduation\_project.* Astfel, clasa va face parte din spațiul de nume *graduation\_project::file::interf.* De aici, deducem faptul că acest fișier antet va avea header guard-ul GRADUATION\_PROJECT\_FILE\_INTERF\_FILE\_HPP, valoare pentru directivele de preprocesare *#ifndef* și respectiv *#define*.

Conținutul interfeței publice asociată conceptului de fișier, precum și arhitectura acestui concept poate fi urmărită și analizată în figura fig.4.1.

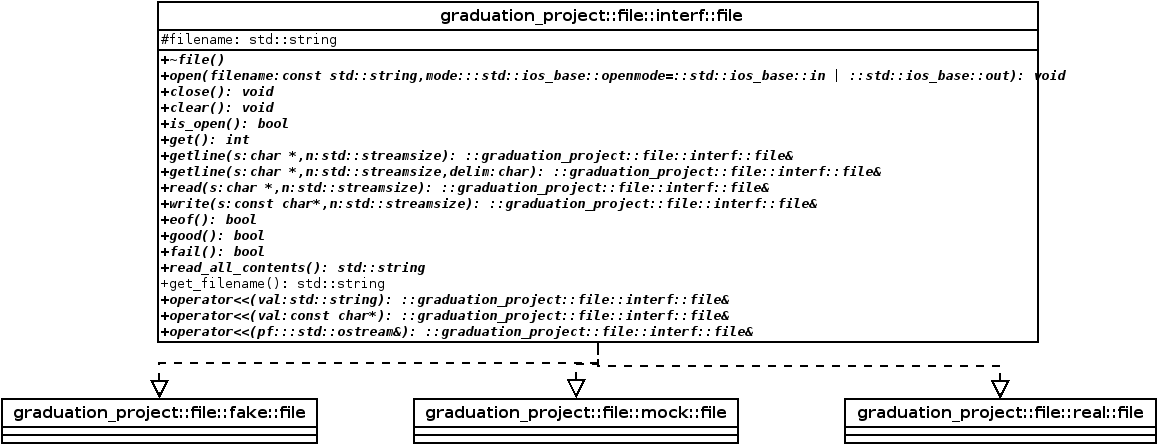


fig.4.1. Diagrama de clase pentru lucrul cu fișiere

Diagrama din figura de mai sus prezintă, așa după cum menționasem anterior, doar interfața publică pentru lucrul de fișiere. Este de la sine înțeles faptul că cele trei clase ce extind *interf::file* vor adăuga corp metodelor pur virtuale ce le moștenesc. Din acest motiv, dar și pentru a evita redundanța informațiilor s-a specificat numele complet al claselor ce extind interfața.

### 4.1.3. Elemente netestabile. Baza de date

În continuare se va prezenta un alt element ce violează paradigma testelor unitare prin utilizarea acestuia în cadrul unei aplicații de testare, și anume, baza de date. Așa după cum am prezentat în capitolul 2, s-a folosit MySQL drept sistem de gestiune a bazelor de date, iar în cod, s-a utilizat *mysql-connector-c*. Astfel, în cadrul directorului (spațiului de nume) *database* vor exista ca și în cazul lucrului cu fișiere, cele patru spații de nume (*interf*, *fake*, *mock*, *real*). Pe lângă acestea, a fost nevoie de câteva clase auxiliare. O astfel de clasă este *database\_connection\_info*, clasă ce integrează setul de parametri și caracteristici ce vor fi utilizați pentru crearea unei conexiuni la baza de date. Pe lângă clasa *exception* particularizată în cazul conceptului de bază de date, s-a creat clasa *quick\_connection*, clasă ce înglobează apelarea anumitor metode din biblioteca utilizată, la un alt nivel, denumit în continuare layer 2.

Primul layer, așa cum l-am definit anterior, reprezintă nivelul de bază al bazei de date, nivel ce presupune efectuarea de apeluri 1:1 cu biblioteca de mysql. Altfel spus, toate metodele ce se vor implementa vor fi de fapt apelearea metodelor cu același nume din biblioteca de mysql. Pentru că, de exemplu, crearea unei simple conexiuni la baza de date presupune scrierea a câtorva linii de cod, ne-am propus să extindem implementarea adăugand un alt nivel, layer 2 așa cum l-am numit. În cadrul acestuia, metodele cu același nume din mysql vor include și anumite verificări, înglobând una sau mai multe metode din biblioteca mysql. Pentru o înțelegere mai bună, se va prezenta în figura de mai jos un exemplu de implementare a metodei *connect* în layer-ul 1, cât și în layer-ul 2.

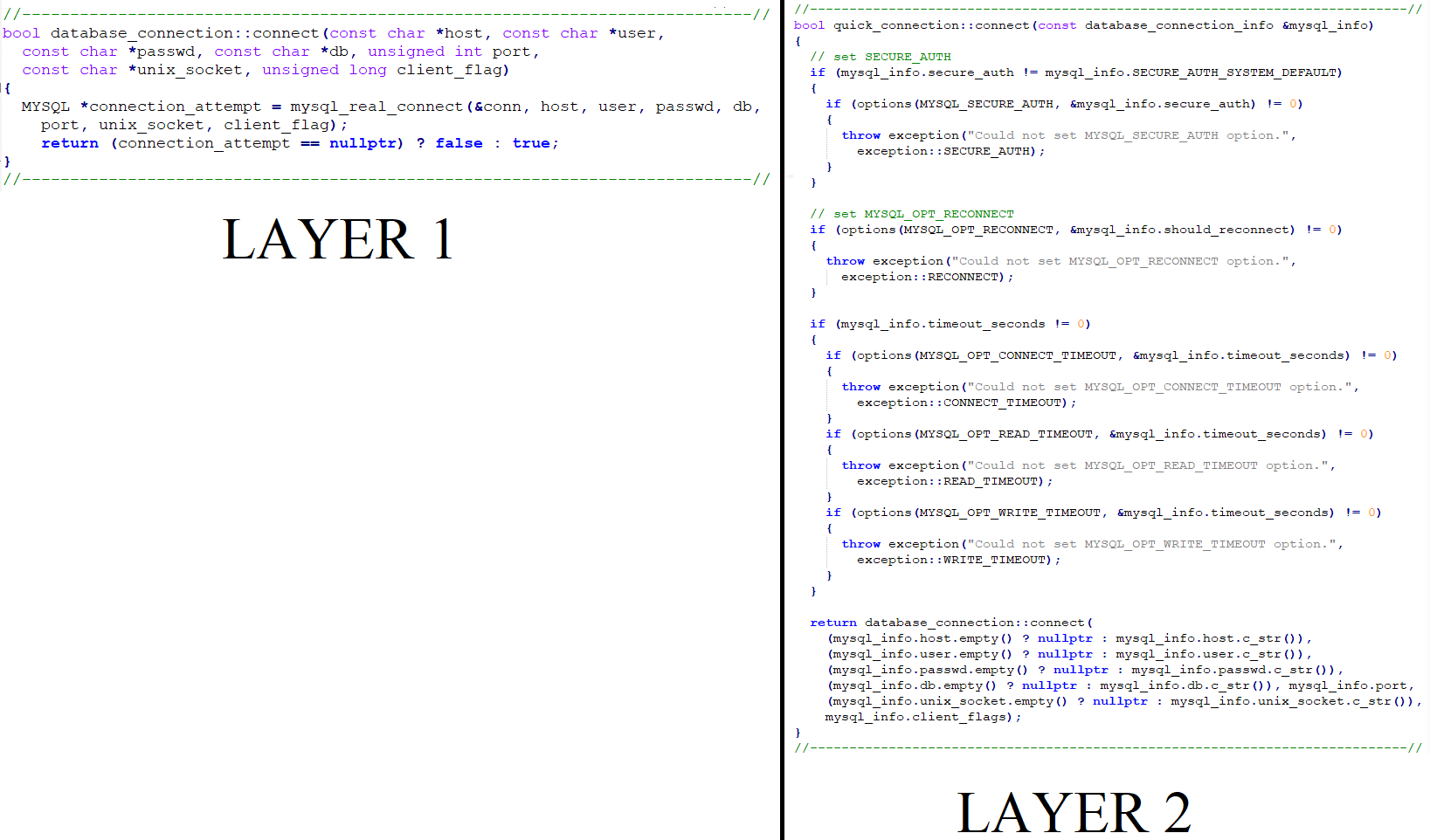


fig.4.2. Metoda connect implementată în layer 1 și în layer 2

Așa după cum se observă, utilizarea layer-ului 2 a fost realizată pentru ca programatorul să utilizeze funcționalitatea conexiunii într-un mod mult mai organizat și ușor de înțeles. Este important de precizat faptul că layer-ul 1 este alcătuit din clasele *database\_connection,* în timp ce clasele *database\_system, result\_set* și respectiv *quick\_connection* alcătuiesc layer-ul 2.

Tot din motivul de a ușura modul în care programatorul va folosi aceste clase au fost gândite și clasele *cell* și *row*, din spațiul de nume *result.* Așa după cum sugerează și numele claselor, acestea s-au dorit a fi asemănătoare cu rezultatele pe care SGBD-ul le poate returna în cazul unei interogări. De exemplu, dacă se va efectua o interogare pentru a afla numărul de înregistrări dintr-un tabel (funcția *count)*, va fi returnată o înregistrare formată dintr-o singură celulă (*cell)*. Dacă, în schimb, se vor selecta toate înregistrările dintr-un tabel (*Select \* )*, rezultatul întors va fi un set de rânduri (*row*).

4.1.4. Elemente netestabile. Socket-ul

Una dintre definițiile utilizate pentru conceptul de socket spune faptul că acesta este „un punct de conexiune într-o reţea TCP\IP” iar „când două programe aflate pe două calculatoare în rețea doresc să comunice, fiecare din ele utilizează un socket” [9]. Conform teoriei testelor unitare, *socket*-urile nu pot fi transformate în elemente testabile așa cum se întâmplă în cazul lucrului cu fișiere. În principal, partea dintr-un program ce se ocupă cu comunicația în rețea nu se testează cu ajutorul acestor tipuri de teste. Acest lucru se datorează și faptului că pentru o aplicație de tip server, de exemplu, programatorul sau echipa de dezvoltare vor trebui să ruleze și teste de performanță, să ruleze serverul în condiții de stres prin crearea mai multor conexiuni server-client. Prin teste de performanță putem înțelege, în primul rând, numărul de clienți (în strânsă legătură cu numărul de fire de execuție deschise pentru un client, în funcție de felul cum a fost proiectată aplicația) cu care serverul poate comunica la un anumit moment. Un alt criteriu ce se va urmări prin aceste teste de performanță (aceste teste pot intra în categoria testelor de integrare, denumite în literatura de specialitate din limba engleză *integration tests*) este consumul de memorie și procesor ce se observa, de exemplu, prin comanda *top* într-un terminal de Linux. În sfârșit, după cum se observă, testarea claselor ce alcătuiesc și definesc conceptul de *socket* nu intră în cadrul discuției noastre, ce au drept scop testele unitare. Cu toate acestea, codul curent folosește funcții din standard-ul limbajului C/C++ și a fost testat prin conectarea propriu-zisă a unui client la o instanță de server deschisă pe un anumit port în cadrul aceluiași IP sau a unui IP extern.

## 4.2. Scrierea testelor unitare

### 4.2.1. Introducere

Pentru a scrie un test unitar este nevoie, în primul rând de o metodă ce se dorește a fi testată. Pornind de la această metodă, în funcție de complexitatea acesteia, se poate scrie fie un singur test în cazul căreia se vor testa mai multe situații limită ale funcției sau așa numitele cazuri de testare[[3]](#footnote-3) sau se poate scrie câte un test pentru fiecare caz al fiecărei funcții. Astfel, putem spune că se pot scrie teste per metode sau teste per scenarii. Dacă ținem cont de faptul că un test unitar ar trebui să dureze maxim *1-2 ms* pentru a se determina dacă a trecut cu succes sau nu, dar și de faptul că un programator începător în testele unitare poate să folosească fără deosebire așteptări cu eșec fatal sau nu (ceea ce va fi periculos în situația în care un test va conține trei scenarii, iar în primul scenariu s-au introdus așteptări cu eșec fatal deoarece în cazul unei așteptări eronate, celelalte două scenarii nu se vor executa niciodată, testul încheindu-se cu eșec), recomandarea autorului este scrierea testelor unitare per scenarii.

Un alt aspect al testelor unitare îl constituie existența claselor de teste. În momentul de față nu există standardizări ale regulilor de scriere a testelor unitare, ci obiectivul principal care guvernează modul de implementare este rapiditatea testelor. Există totuși o serie de bune practici ce sunt utilizate în scrierea testelor unitare [10] și conform cărora o clasă se recomandă să fie testată prin interfața ei publică, adică doar metodele publice să fie testate. Există însă și metodele *protected*/*private* ce alcătuiesc clasa. Și în acest punct părerile sunt împărțite în sensul că, de regulă, metodele *protected/private* sunt apelate în metodele publice; metodele publice, la rândul lor, fiind testate, se poate considera implicit testat comportamentul metodelor *protected/private.* Acest lucru nu este neapărat adevărat atunci când logica metodelor este complexă și se dorește pentru cât mai multe cazuri. Un scenariu de test pentru o metodă publică poate implica multe scenarii pentru metodelor ce sunt folosite în cadrul respectivei metode, lucru ce va duce la o înțelegere mai grea a scenariilor de test imbricate. Tocmai din acest motiv, s-au introdus clasele de testare ce sunt vizibile în teste prin macro-ul *TEST\_F* (*test fixtures*). De regulă, o clasă de test va extinde clasa *::testing::Test* din biblioteca *gtest* și respectiv clasa pentru care se vor scrie teste. O clasă de teste poate avea constructori/destructori sau metode echivalente pentru aceștia, și anume, metodele virtuale *virtual void SetUp()* și respectiv *virtual void TearDown()*. Aceste metode au un comportament similar constructorilor/destructorilor în sensul că au rolul de a pregăti obiectele obiectele pentru fiecare test și sunt apelate automat.

Pentru scrierea efectivă a testelor se pot utiliza două *framework*-uri: *GoogleTest* și/sau *GoogleMock*. Testele ce au fost scrise folosind primul *framework* poartă denumirea de teste *fake*, în timp ce testele scrise folosind cel de-al doilea *framework* poartă denumirea de teste *mock*. În continuare se vor trata diferit cele două tipuri de teste, cu mențiunea că în cadrul aplicației ce stă la baza prezentei lucrări s-a încercat scrierea de teste atât *mock* cât și *fake*, atât cât situația și logica metodelor/claselor a permis acest lucru.

### 4.2.2. Scrierea testelor utilizând framework-ul GoogleTest

Testele unitare ce au fost scrise utilizând acest *framework* presupun, așa după cum am văzut în cazul tuturor testelor unitare, viteză mare de execuție, adică testul scris nu trebuie să comunice cu resurse externe aplicației. Pentru exemplificare, se va prezenta fișierul drept resursă externă.

În primul rând, orice test unitar scris cu *framework*-ul în discuție este alcătuit din două părți. Prima parte presupune crearea obiectelor și apelarea metodei ce se dorește a fi testată, iar a doua parte constă din o serie de așteptări care verifică corectitudinea metodei și prin care programatorul va putea să verifice dacă metoda scrisă de acesta se comportă așa după cum a fost proiectată. Testele *fake* se evidențiază prin faptul acestea trebuie să conțină în interiorul lor structuri de date ce vor simula resursa externă pe care metoda reala le va folosi. În cazul nostru, operațiile de scriere/citire în/din fișier le vom simula cu operațiile de scriere/citire în/din o instanță de *stringstream*. După cum se observă, acest lucru constituie de fapt dificultatea testelor *fake* în sensul că în unele situații este destul de greu să găsim o structură de date care să simuleze cel mai bine realitatea. Din acest motiv au apărut testele *mock*. Însă, pentru exemplificare se va prezenta în continuare o metodă din aplicație ce a testată atât *fake* cât și *mock*. Corpul metodei poate fi vizualizat în figura următoare.

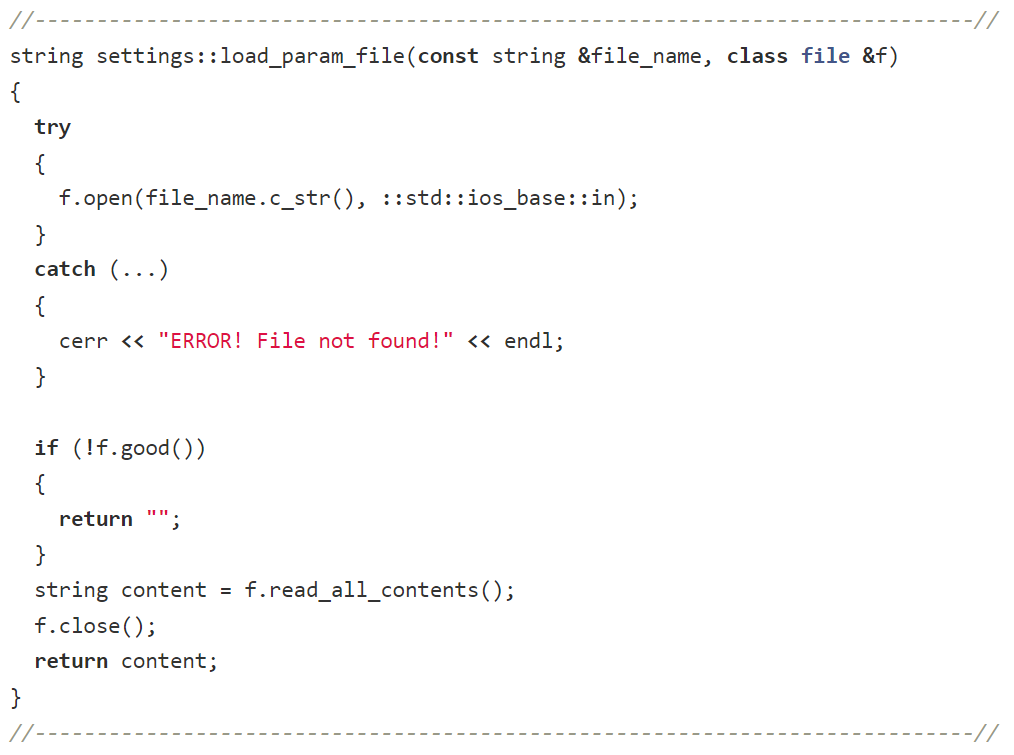


fig. 4.3. Metoda load\_param\_file a clasei settings

Așa după cum se observă, metoda *load\_param\_file* încearcă (*try-catch*) deschiderea fișierului cu numele primit ca parametru, după care citește tot conținutul fișierului (metoda *read\_all\_contents*) pe care îl întoarce ca *string*. Pentru această metodă se va putea analiza testul *fake* din figura următoare.

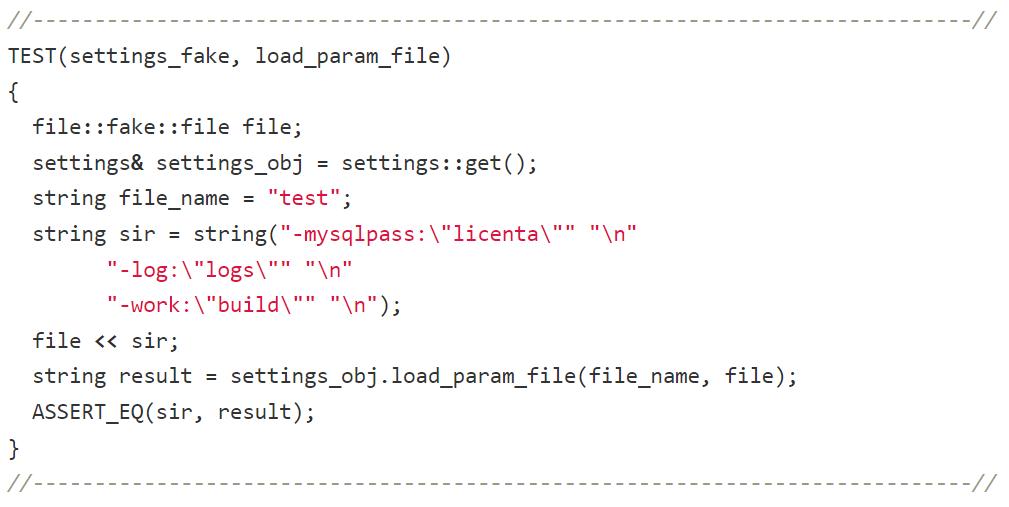


fig.4.4. Test *fake* pentru metoda *settings::load\_param\_file*

Dacă se va analiza figura 4.4. se va observa faptul că primele 7 linii constituie prima parte a testului, adică crearea obiectului *fake,* crearea unicului obiect al clasei *settings* (clasa *settings* este o clasă *singleton*), definirea numelui fișierului, a conținutului fișierului după care „scrierea” conținutului în fișierul *fake*, urmând ca în final conținutul „fișierului” să fie stocat în variabila *result*. A doua parte a testului constă din compararea cu eșec imediat (ASSERT) a conținutului inițial cu conținutul întors de metoda *load\_param\_file*. Dacă, din orice motiv, cele două șiruri de caractere nu erau identice, metoda *load\_param\_file* se considera a fi scrisă greșit, ceea ce ar fi necesitat investigarea acesteia sau rularea metodei pas cu pas.

### 4.2.3. Scrierea testelor utilizând framework-ul GoogleMock

Prima diferență între testele scrise cu acest framework (teste *mock*) este la nivel de structură. În acest sens, cele două părți ale unui test sunt inversate, adică prima parte a testului constă în crearea obiectelor necesare și în definirea tuturor așteptărilor ce vor genera și vor defini corpul metodei de testat. O altă diferență constă în faptul că așteptările nu verifică doar egalitatea sau aruncarea sau nu a unor excepții, ci verifică (sau definesc) așteptările la nivelul apelurilor funcțiilor. Funcțiile ce se doresc a fi apelate (*EXPECT\_CALL)* trebuie să aibă „implementare” *mock*, adică să fie definite de *macro*-uri GoogleMock de genul *MOCK\_METHOD<X>*, <X> reprezintă numărul de parametri ai funcției (pentru detalii, se pot analiza din directorul codului sursă *src*, conținutul spațiilor de nume *mock* ). A doua parte a testului constă din apelarea efectivă a metodei ce se dorește a fi testată. Dacă în corpul funcției testate există anumite funcții *mock* ce nu au fost specificate în secțiunea așteptărilor, iar comportamentul acesteia nu este definitoriu, *GoogleMock* ca genera un avertisment la rularea testelor prin care va indica faptul că respectiva metodă nu a fost specificată în secțiunea așteptărilor, mesajul de avertisment terminându-se prin specificarea că *Gmock* nu recomandă specificarea așteptării corespunzătoare metodei doar pentru a se evita avertismentul.

O particularitate importantă a testelor *mock* este dată de faptul că macro-urile *EXPECT\_CALL* permit definirea comportamentului prin specificarea numărului de apelări a respective metode ( *Times(X)* – X este numărul ce indică de câte ori metoda va fi apelată; dacă nu se specifică de câte ori să se apeleze funcția, *GoogleMock* va ține cont de celelalte așteptări și va calcula automat numărul de apelări ale metodei), tipul returnat o singură dată (*WillOnce*), tipul returnat de mai multe ori (*WillRepedeatly*), ordinea în care se vor verifica așteptările (instanțierea unui obiect al clasei *Sequence*, după care se va specifica *InSequence* cu parametrul obiectului anterior creat) etc. Un test *mock* poate pica dacă orice aspect ce vizează metoda testată nu este respectat, adică poate pica dacă se apelează o anumită metodă de trei ori, când de fapt era de așteptat să se apeleze o singură dată, dacă se specifică să se returneze un anumit rezultat și în realitate se returnează altceva etc. Pentru metoda din fig.4.4. se va prezenta în figura următoare un test *mock* pentru o bună înțelegere a noțiunilor anterior prezentate.



fig.4.5. Test *mock* pentru metoda *settings::load\_param\_file*

# 5. Manual de utilizare

## 5.1. Compilarea proiectului

În ceea ce privește compilarea proiectului, dacă se dorește ca acest lucru să se realizeze din QtCreator, atunci după deschiderea cu succes a proiectului, nu trebuie decât să fie fie alese executabilele ce se vor compila din meniul Edit->Build (prin bifarea opțiunilor *build\_server* și/sau *build\_unit\_tests*). Ultimul pas în procesul de compilare îl constituie efectuarea unui click pe butonul de Build aflat în colțul din stânga jos, așa cum se poate observa în fig.2.3.

Pentru compilarea proiectului din linie de comandă, se recomandă în primul rând crearea unui director numit, de exemplu, *build* în afara directorului proiectului (*graduation\_project*) utilizând comanda *mkdir.* Apoi, odată ce suntem în acest director nou creat, se va rula comanda cmake cu anumiti parametri ce vor fi exemplificați în cele ce urmează.

1. Locația fișierului *CMakeLists.txt*, de exemplu, în cazul de față, locația va fi *../graduation\_project/*
2. Tipul de compilare (Debug/Release). Acest parametru este opțional și poate fi specificat prin opțiunea de cmake -DCMAKE\_BUILD\_TYPE. După specificarea acestui parametru, se va introduce semnul egal urmat de una din cele două valori amintite pentru tipul de compilare (Debug, Release).
3. Opțiuni definite de programator în cadrul fișierelor cmake. În cazul de față avem două opțiuni: *build\_server* și respectiv *build\_unit\_tests*. În mod implicit aceste opțiuni sunt active, setate pe ON în fișierul *CMakeLists.txt.* Dacă se dorește dezactivarea unei comenzi, acest lucru se poate face prin opțiunea *-Dnume\_opțiune=0* sau *-Dnume\_opțiune=OFF*, unde *nume\_opțiune* reprezintă toate opțiunile definite de programator.

Un exemplu de comandă completă de compilare în modul Debug, cu dezactivarea compilării unit testelor, de exemplu, poate fi vizualizată în fig.5.1.



fig.5.1. Compilarea proiectului(server) în mod Debug

# 6. Concluzii

## 6.1. Analiza obiective propuse

În debutul celor două lucrări complementare s-au enunțat o serie de obiective pe care aplicațiile cele două să le atingă. În continuare se va face o scurtă reluare a acestora pentru a observa dacă acestea au fost atinse sau nu.

Compus din două executabile, proiectul de față constă în aplicația de tip server și aplicația ce înglobează o serie de teste unitare ce verifică segmente din proiect. Pentru aplicația de tip server ne-am dorit să realizăm un server funcțional ce va deservi doar primul pas, cel de autentificare. Serverul comunică cu clienții utilizând protocolul TCP/IP prin *request-response*-uri scrise fie în formatul XML, fie în formatul Protocol Buffers (Protobuf). Odată ce serverul a fost notificat prin evenimentul *on\_first\_message* de un prim mesaj sosit pe socket, acesta va extrage primul caracter pentru a determina protocolul (X pentru XML, P pentru Probobuf), așa după cum a fost stabilit la început. Odată identificat protocolul, primul caracter este șters, iar conținutul mesajului brut este dat spre parsare pentru a genera și formata *request*-ul. *Request*-ul de autentificare conține un *token* criptat AES folosind cheia ‘*graduation\_key*’și este analizat și trimis la o bază de date pentru ca, pe baza existenței acestuia să se determine *user*-ul ce s-a conectat la server. O autentificare ce nu se efectuează cu succes duce la închiderea conexiunii cu clientul de către server. Dacă clientul s-a autentificat cu succes, serverul mai poate închide sesiunea în următoarele condiții: se primește un nou *request* de autentificare, se primește un *request* ce nu este suportat de server, se primește un text aleator, ce nu este în formatul cunoscut de server.

În dorința de a încerca să corectăm și să validăm cât mai mult cod din proiect, au fost scrise 41 de teste unitare, acoperind o gamă variată de scenarii și condiții de test. Testele au fost scrise utilizând biblioteci C++ de testare dezvoltate de către Google.

După o analiză atentă a lucrării de față, precum și a lucrării „Teste unitare în C++ - Implementarea unei aplicații testabile” se va observa ușor faptul că toate obiectivele propuse au fost atinse. Mai mult decât atât, citirorului i-a fost oferit suport pentru a înțelege cât mai bine tehnologiile utilizate, importanța lor, dar și modul de a compila și executa ambele aplicații.

## 6.2. Direcții viitoare de extindere și dezvoltare

În secțiunea anterioară (din cadrul lucrării complementatre) s-a prezentat un caz de utilizare real, iar pentru implementarea acestuia, un prim pas, așa după cum s-a afirmat, ar fi adăugarea de suport pentru mai multe tipuri de *request*-uri (și implicit tipuri de *response*-uri), ceea ce presupune în primul rând extinderea diagramelor *request*-*response*, adăugarea mai multor tipuri de mesaje protobuf și nu în ultimul rând suplimentarea codului curent.

În secțiunea Manualului de utilizare (din cadrul lucrării complementare) s-au prezentat două tipuri de *output*-uri corespunzătoare serverului și respectiv clientului. Dat fiind faptul că cel mai probabil serverul va rula în background (rularea in background se poate face cu comanda *nohup* : nohup ./server& ) sau într-o sesiune de tmux, deci nimeni nu va putea sa urmărească ceea ce acesta va printa, o importanță aparte o au fișierele de log. Managementul fișierelor sau a fișierului de log va fi tratat separat într-un alt subiect însă, pentru aceasta, drept dezvoltare a proiectului se va folosi biblioteca *log4cxx* (ce poate fi instalată sub Linux prin pachetul *liblog4cxx-dev*). Această bibliotecă este soluția cea mai bună în urma cercetărilor întreprinse pe această ramură de autor până în prezent, este foarte complexă și permite definirea modului de logare printr-un fișier de configurare *XML*. Fișierul de configurare va permite definirea unui număr de *logger*-e (fișiere de log), nivele de severitate a mesajelor logate, timpul generării mesajului, conținutul mesajului, dimensiunea mesajului, *thread*-ul din care a fost generat mesajul etc.

Având la dispoziție unul sau mai multe fișiere de log, se va putea merge mai departe cu dezvoltarea aplicației prin utilizarea tehnologiei *Kibana*, tehnologie ce permite, conform documentației oficiale, vizualizarea datelor Elasticsearch și navigarea în Stack-ul Elastic. Practic, această tehnologie este o modalitate de a vizualiza și analiza fișierele de log cu ajutorul unei interfețe grafice. Desigur, folosind sistemul de operare Linux, analiza și filtrarea unor fișiere se poate realiza și cu ajutorul unor comenzi specifice, precum *grep, find, egrep, awk* etc. Vizualizarea datelor folosind *Kibana* se va realiza pornind de la un fișier de configurare ce va defini Elasticsearch-ul, un fișier de configurare *yml* precum și datele de intrare, sau fișierele de log.

Un ultim lucru ce se poate adăuga implementării curente este crearea unei aplicații de tip client ce va stres-testa serverul, realizând testele de performanță, testele de integrare (*Integration tests*). La acest moment, ideea de bază a aplicației client este ca acesteia să i se puna la dispoziție un director ce conține un număr de X fișiere, fiecare fișier fiind un nou client al serverului. Aplicația va citi toate *request*-urile din fișiere și le va trimite la server, simulând *request*-uri ce pot proveni de la o aplicație reală. Dat fiind faptul că serverul suportă și *request*-uri în formatul Protobuf, o parte din aplicație va fi responsabilă de prelucrarea *request*-urilor în format XML prin transformarea acestora în *request*-uri în format Protobuf ce le va expedia serverului, testând în acest fel și *request*-urile în format Protobuf.

# 7. Bibliografie

1. *Din uneltele artizanului software: Unit testing, Alexandru Bolboacă*  <https://www.todaysoftmag.ro/article/377/din-uneltele-artizanului-software-unit-testing/> (resursă online accesată în Aprilie 2018)
2. *Resursă online accesată în Iunie 2018:* <https://www.linode.com/docs/tools-reference/tools/schedule-tasks-with-cron/>
3. *Documentația oficială CMake*: <https://cmake.org/> (resursă online accesată în Aprilie 2018)
4. *Documentația oficială Protocol Buffers, Google:* <https://developers.google.com/protocol-buffers/?hl=ro/> (resursă online accesată în Mai 2018)
5. *Documentația oficială MySQL:* <https://dev.mysql.com/doc/refman/8.0/en/introduction.html> (resursă online accesată în Mai 2018)
6. *Documentația oficială Google Test:* <https://github.com/google/googletest/> (resursă online accesată în Mai 2018)
7. *Documentația oficială Google Mock:* <https://github.com/google/googletest/blob/master/googlemock/README.md> (resursă online accesată în Mai 2018)
8. *Resursă online accesată în iunie 2018:* <http://softwaretestingfundamentals.com/integration-testing/>
9. *Suport de laborator pentru disciplina „Sisteme de control distribuit”, Universitatea Tehnică Cluj-*Napoca: <http://control.aut.utcluj.ro/scd/lab2/laborator2_1.htm> (resursă online accesată în Iunie 2018)
10. *Resursă online accesată în iunie 2018:* <http://developer.mantidproject.org/UnitTestGoodPractice.html/>

1. cron – utilitar clasic găsit pe sistemele Linux pentru a executa sarcini la intervale sau momente prestabilite. [↑](#footnote-ref-1)
2. Teste de integrare - constituie un nivel al testării software în care unitățile individuale sunt combinate și testate ca grup. Scopul acestui nivel de testare este acela de a expune defectele în interacțiunea dintre unitățile integrate [8]. [↑](#footnote-ref-2)
3. *Cazurile de testare* (*test-cases*) pot să facă parte dintr-un singur scop dacă metoda folosește structuri de date ce pot fi reinițializate prin operații simple (de exemplu, metoda care verifică dacă un șir de caractere este palindrom sau nu, necesită doar o operație de atribuire pentru a seta un nou șir de caractere ca date de intrare), sau pot fi separate în mai multe scopuri în situația în care pentru o anumită metodă este nevoie de un constructor care să primească diferiți parametri în funcție de scenariul de test. [↑](#footnote-ref-3)