UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

**FACULTATEA DE INFORMATICĂ**



LUCRARE DE LICENŢĂ

**Review Manager**

propusă de

***Olaru Gabriel***

**Sesiunea:** *iulie, 2016*

Coordonator ştiinţific

**Asist. Dr. Vasile Alaiba**

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

**FACULTATEA DE INFORMATICĂ**

**Review Manager**

***Olaru Gabriel***

**Sesiunea:** *iulie, 2016*

Coordonator ştiinţific

**Asist. Dr. Vasile Alaiba**

DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA

DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul „*Review Manager*” este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

* toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
* reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
* codul sursă, imaginile etc. preluate din proiecte *open*-*source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
* rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi,

Absolvent *Gabriel Olaru*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că nu sunt de acord ca Lucrarea de licență cu titlul „*Review Manager*”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de testetc. ) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, nu sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” Iași să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi,

Absolvent *Gabriel Olaru*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Cuprins

[Introducere 7](#_Toc454382229)

[Context 7](#_Toc454382230)

[Motivație 7](#_Toc454382231)

[Obiectiv 8](#_Toc454382232)

[Contribuții 8](#_Toc454382233)

[Structura lucrării 8](#_Toc454382234)

[Capitolul I. Tehnologii folosite 9](#_Toc454382235)

[QT 9](#_Toc454382236)

[De ce Qt? 9](#_Toc454382237)

[Signals and Slots 10](#_Toc454382238)

[Management-ul resurselor 10](#_Toc454382239)

[Server Qt 10](#_Toc454382240)

[QtcpServer 10](#_Toc454382241)

[PostgreSQL 11](#_Toc454382242)

[Design Pattern-uri 11](#_Toc454382243)

[Singleton [3] 12](#_Toc454382244)

[Bridge 12](#_Toc454382245)

[Fire de execuție 12](#_Toc454382246)

[Sistemul de operare Windows7/10 12](#_Toc454382247)

[Concluzii 13](#_Toc454382248)

[Capitolul II. Arhitectura aplicației 14](#_Toc454382249)

[Server 14](#_Toc454382250)

[Baza de date 15](#_Toc454382251)

[Tabele 15](#_Toc454382252)

[Funcții 17](#_Toc454382253)

[Client 18](#_Toc454382254)

[Capitolul III. Detalii de implementare 23](#_Toc454382255)

[Server 23](#_Toc454382256)

[Baza de date 27](#_Toc454382257)

[Client 30](#_Toc454382258)

[Capitolul IV. Șcenarii de utilizare 44](#_Toc454382259)

[1. Logarea 44](#_Toc454382260)

[2. MainWindow 45](#_Toc454382261)

[3. CreateReview 45](#_Toc454382262)

[4. EditReview 46](#_Toc454382263)

[5. ViewReview 47](#_Toc454382264)

[6. ViewSession 48](#_Toc454382265)

[7. SetModification 48](#_Toc454382266)

[8. Statistici 49](#_Toc454382267)

[9. QuickMenu 50](#_Toc454382268)

[Configurare și cerințe 50](#_Toc454382269)

[Cerințe 50](#_Toc454382270)

[Concluzii 51](#_Toc454382271)

[Bibliografie 52](#_Toc454382272)

# Introducere

Aceasta este o temă propusă de către CONTINENTAL AUTOMOTIVE ROMANIA S.R.L, realizată în întregime de către mine, în perioada în care am lucrat ca „work and student” în cadrul companiei.

## Context

Calitatea este un concept care se utilizează în toate domeniile vieții, însă care prezintă un puternic caracter subiectiv. Un portofoliu de produse și servicii de calitate este motivul principal care determină clientul să se fidelizeze față de companie și să facă recomandări, altor parteneri de afaceri.

Referitor la definiția calității din standardul SR ISO 8402 este util de menționat că termenul „calitate” nu ar trebui utilizat în interpretarea greșită de „grad de excelență”, într-un sens comparativ sau pentru evaluări în sens cantitativ, deoarece aceste semnificații satisfac numai o parte a definiției. Pe de altă parte, „gradul de excelență” este o definiție vagă, deoarece excelența este alt termen dificil de definit. [[1]](#footnote-1)

Calitatea produselor și serviciilor a fost identificată ca fiind unul dintre cei mai importanți factori în atingerea succesului de către o companie[[2]](#footnote-2). Ea devansează factori precum: experiența, termene de livrare, preț, service sau avantaje tehnice.

## Motivație

De-a lungul timpului, odată cu evoluția tehnologiilor, calitatea joacă un rol important în acest domeniu IT. Dezvoltarea constantă a companiilor pe o piata concurențială nu se poate face făra un produs sau un serviciu care să aducă reale satisfacții utilizatorului final (un produs sau un serviciu de calitate).

Încrederea în capacitatea de a produce „calitate” a devenit premisa de bază pentru o colaborare stransă între firme, corporații, angajați și clienți.

Un alt factor care a stat la baza acesteia, a fost dorința de a învăța un framework nou, cum ar fi Qt-ul, dar și pasiunea pentru programare fiind prilejul perfect pentru a duce la bun sfarșit acest lucru.

Pe piața aplicațiilor nu există foarte multe tool-uri de acest gen care să-ți ofere o interfață ușor de folosit, intuitivă, prietenoasă cu utilizatorul și care să facă management-ul fișierelor.

## Obiectiv

Îmi propun să realizez o aplicație care să îmbine în cel mai bun mod cele menționate anterior, să ofere utilizatorului prilejul de a-și îmbunătăți cunoștințele, modul în care lucrează, calitatea serviciilor lui și totodată ocazia de a împărtăși cu ceilalți colegi cunoștințele sale.

## Contribuții

Am realizat o aplicație care îndeplinește toate cerințele și atinge scopurile propuse. În consecința am creat un program care are două moduri: offline în care utilizatorii care sunt participanți la review, pot analiza fișierele uploadate dar și mod „online” în care creatorul review-ului poate organiza diferite sesiuni în care se pot discuta eventualele greșeli sau îmbunătățiri.

Din punct de vedere al arhitecturii, aplicația este formată dintr-un client scris în Qt și doua servere. Unul fiind serverul PostgreSql și celălalt este scris în Qt folosind QtcpSocket și QtcpServer pentru a facilita comunicarea între cele doua entități.

## Structura lucrării

Această lucrare conține patru capitole și anume:*Tehnologiile folosite, Arhitectura Aplicației, Detalii de implementare* și *Șcenarii de utilizare.* Cu ajutorul lor, doresc să evidențiez necesitatea acestei aplicații pentru dezvoltarea servicilor și perfecționarea lor.

# Capitolul I. Tehnologii folosite

Tehnologii folosite pentru dezvoltarea programului „Review Manager” :

* Sistemul de operare Windows7/10
* Server Qt
* Qt
* QtcpServer
* QtcpSocket
* PostGreSQL
* Design Pattern-uri
* Fire de execuție

## QT

Qt este o suita inter-platformă de dezolvare folosită atât pentru crearea programelor cu interfață grafică cât și pentru programe fără interfață, cum sunt serverele. Cele mai cunoscute utilizări ale QT-ului sunt : browserul web Opera, Google Earth, Skype, Qtopia.

Începând cu versiunea 4, Qt este disponibil pentru toate platformele sub o licență duală, GPL și o licență proprietară.

## De ce Qt?

Qt completează deficiențele limbajului C++ asigurând un mediu plăcut de dezvoltare și un framework prietenos cu programatorul. Conține o documentație foarte bună cu exemple relevante.

Qt are la bază un set de biblioteci numit QtCore care acoperă un set de funcționalități de bază, dupa cum urmează:

* un sistem GUI, care este unul dintre punctele forte ale QT-ului.
* event loop și sistem de evenimente ușor de folosit
* Template-uri, cum ar fi QList, QVector, QMap; acestea oferă o alternativă perfectă la container-ele STL
* Un sistem model-view
* Clase de management al resurselor

Există o serie de pachete bogate în facilităti din diferite domenii, printre care :

* procesare XML
* unit testing
* OpenGL
* Manipulare SVG
* Comunicare D-Bus

### Signals and Slots

Semnalele și sloturile oferă o alternativă la callback-uri, fiind losely-couple. Asta înseamnă că obiectul nu este conștient de cine este conectat la dânsul. Un semnal este emis atunci când un eveniment trebuie anunțat. Un semnal este o funcție făra corp iar slot-urile sunt funcții care sunt apelate ca răspuns la semnal.

Conexiunea se realizează prin urmatorul apel QObject::connect(). Pentru a putea folosi acestă alternativă programul trebuie să respecte următoarele aspecte: fiecare clasă trebuie să moștenească Q\_Object și să conțină macro-ul Q\_OBJECT.

### Management-ul resurselor

Mecanismele pentru acest astepect sunt: Ownership Hierachy și Implicit sharing.

Ownership Hierachy se ocupă de distrugerea obiectelor iar implicit sharing este un mod sigur de a lucra cu pointeri.

## Server Qt

Majoritatea comunicărilor între procese folosesc modelul client-server. În acest model unul dintre cele doua procese, adică clientul, se conectează la celălalt proces, și anume server pentru a-i cere anumite informții . Este important de menționat faptul că procesul client, trebuie să cunoască adresa server-ului pe când serverul nu trebuie să știe adresa clientului apriori stabilirii conexiunii între cei doi.

Qt oferă posibilitatea de a scrie cod foarte eficient cu un nivel de abstractizare ridicat, prin urmare, pentru a dezvolta o aplicație performantă, care să poată rula pe orice sistem de operare, un server scris în Qt ar putea fi o soluție foarte bună.

## QtcpServer

QtcpServer are la bază protocolul TCP (Transmission Control Protocol). Acesta efectuează o conexiune full duplex intre două puncte, folosind mecanismul Three-way handshaking. Acest mecanism presupune trimiterea unui pachet inițial apoi se așteaptă confirmarea de primire de la destinatar în vederea trimiterii următorului pachet. Dacă confirmarea nu este primită într-un anumit interval de timp prestabilit, se retransmite pachetul.

Datorită conexiunii full-duplex, în momentul în care o aplicație nu mai are date de trimis, TCP va închide conexiunea doar într-o singură direcție.

## PostgreSQL

PostgresSQL [1] este un sistem de baze de date relaționale și este distribuit sub licență open source. Acesta permite folosirea limbajelor procedurale pentru a executa blocuri de cod direct în serverul bazei de date. Se pot folosi pentru a crea funcții definite de utilizator (subrutine, trigerre, agregate și funcții fereastră) sau pentru a crea blocuri ad hoc „DO”. Oferă o bază de date complexă cu posibilitatea de a crea tabele multiple, indici, trigger-e, view-uri. Postgresql este folosit în general pentru baze de date de dimensiuni mari, el excelând la acest capitol după cum spun urmatoarele statistici:

* Dimensiunea maximă a bazei de date: nelimitat
* Dimensiunea maximă a unei tabele: 32 TB
* Dimensiunea maximă a unei înregistrări: 1, 6 TB
* Dimensiunea maximă a unui câmp: 1 GB
* Număr maxim de înregistri într-o tabelă: nelimitat
* Număr maxim de coloane într-o tabelă: 250 - 1600 în funcție de tipul coloanelor

## Design Pattern-uri

Design pattern-urile [2] reprezintă soluții generale și reutilizabile ale unei probleme comune în design-ul software. Un design pattern este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei, nu o bucată de cod ce poate fi aplicata direct. În general pattern-urile orientate pe obiect arată relațiile și interacțiunile dintre clase sau obiecte, fără a specifica însă forma finală a claselor sau a obiectelor implicate.

Acestea sunt de 3 tipuri

* “Gang of Four” patterns
  + Creational Patterns
  + Structural Patterns -
  + Behavioural Patterns

### Singleton [3]

Pattern-ul Singleton este utilizat pentru a restricționa numărul de instanțieri ale unei clase la un singur obiect, deci reprezintă o metodă de a folosi o singură instanță a unui obiect în aplicație.

### Bridge

Decuplează o abstracție de implementările sale.

* Abstraction
  + defineste interfața abstracției
* RefinedAbstraction
  + defineste o specializare a abstracției
* Implementor
  + defineste interfața implementării
  + nu trebuie sa coincidă cu cea a abstracției
* ConcreteImplA, ConcreteImplB
  + definesc implementările concrete ale abstracției

## Fire de execuție

Un fir de execuție, numit și thread, este o secvență de instrucțiuni care poate fi executată concurent cu o altă secvență, într-un mediu multithreading, în timp ce partajează aceeași zonă de memorie. Aceste fire de execuție rulează în paralel pentru a crește eficiența programelor. În sistemele cu procesoare multiple sau cu nuclee multiple, firele de execuție rulează în același timp pe procesoare sau nuclee diferite. Pentru procesoarele cu un singur nucleu, sistemul împarte timpul de execuție între thread-uri. [4]

## Sistemul de operare Windows7/10

Windows este un sistem de operare produs de Microsoft pentru PC-uri bazat pe platformele IA-32 și x86-64. [[3]](#footnote-3)

Acesta a cucerit o foarte mare parte a pieții, fiind la ora actuală cel mai răspândit sistem de operare (SO) din lume. Se estimează că astăzi peste 91 % din calculatoarele de tip Personal Computer (PC-uri cu procesor de la compania Intel, sau compatibil cu Intel, ca de exemplu de la firma AMD) rulează Windows. Cu toate acestea, celelalte sisteme de operare precum Linux, MacOS, FreeBSD, NetBSD, Solaris, HP-UX etc. au recuperat în ultimii ani o parte a pieței, reducând astfel monopolul aproape total al companiei Microsoft, care în 1999 avea o cotă de piață de peste 95 %.

## Concluzii

Întrucât intenționez dezvoltarea unei aplicații cât mai performante posibile, care să ruleze eficient, am ales tehnologiile mai sus menționate datorită avantajelor pe care acestea le prezintă.

La nivelul serverului am ales să folosesc QtcpServer și QtcpSocket pentru siguranța oferită din punct de vedere al trasnferurilor de mesaje între client și acesta. Totodată această conexiune este mai rapida și mai eficientă. Pentru stocarea datelor, bazele de date PostgreSQL, au fost alegerea perfectă, datorită volumului mare de date pe care acestea il pot reține în tabele.

La nivel de client, am ales să folosesc tot Qt datorită bibliotecilor oferite, management-ul resurselor, sistemului GUI dar și pentru avent loop-uri (semnale și sloturi).

# Capitolul II. Arhitectura aplicației

În continuare urmează să prezint arhitectura aplicației dar și modul în care comunică și sunt îmbinate părți importante ale programului.

Arhitectura aplicației este de tip client-server. Pentru a oferi o mai bună operabilitate au fost create două servere. Un server PostGreSql care se ocupă de toate cererile pentru lucru cu baza de date și un server scris în Qt pentru lucrul cu fișiere, astfel nu se îngreunează procesele și nu există o dependența foarte mare între cele două.

## Server

Pentru a obține o aplicație usor de înțeles, modificat, înteținut și testat, la nivelul serverului am încercat să folosesc cât mai multe principii de programare orietată pe obiecte.

Astfel, clasele încapsulează datele, ascund reprezentarea și sunt ușor de refolosit. De asemenea, am încercat ca responsabilitățile să fie alocate în așa mod încât coreziunea să rămână ridicată. O coeziune ridicată presupune ca responsabilitățile unui element din sistem sunt înrudite și concentrate în jurul aceluiași concept. Sarcinile sunt și ele împărțite în așa fel încât cuplarea rămâne slabă. Cuplarea slabă presupune dependențe puține între clase, impact scăzut în sistem la schimbarea unei clase și potențial ridicat de refolosire astfel încât programul este flexibil.

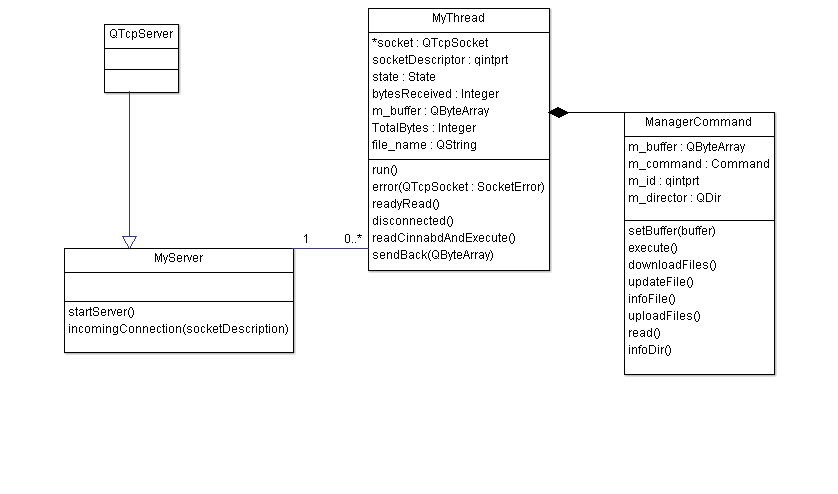
Acest server se ocupă exclusiv cu management-ul fișierelor după cum urmează:

* Descărcare proiecte
* Incărcare fișiere
* Vizualizare conținut folder
* Vizualizare conținut fișier

În Fig. 1 se poate vedea structura server-ului care este formată din 4 clase și anume: Clasa MyServer, fiind cea principală, pornește serverul și ascultă cererile de conexiune. Clasa MyThread se ocupă de conexiunea clienților și comunicarea cu aceștia. Ultima clasă, CommandManager procesează cererile venite din partea clienților și le execută.

Serverul lucrează cu un utilitar extern, și anume, 7zip. Acesta este folosit de către CommandManager la arhivarea și dezarhivarea folderelor, acest process fiind de ajutor la versionarea fișierelor și a folderelor precum și pentru scrierea în buffer. Acest program este folosit și pe partea de client în procesul de export al fișierelor xlsx dar și în momentul când se creează un review nou iar utilizatorul încarcă fișierele de care are nevoie.

Fig. 1 Diagramă de clase Server



## Baza de date

Aceasta este formată din paisprezece tabele, cu structurile din urmatoarele imagini. (Fig. 2, Fig. 3).

### Tabele

Tabela REVIEW este cea care reține toate informațiile despre review-ul propriu-zis iar tabela REVIEW\_SESSION care stochează informațiile despre o sesiune dintr-un Review.

O altă tabela importantă este FINDINGS, unde se rețin informațiile legate de un anumit finding cum ar fi id-ul sesiunii în care a fost găsit, severitatea, linia precum și numele acestuia.

USERS conține datele tuturor utilizatorilor, cum ar fi numele, parola, locația, funcția, email-ul iar tabela REVIEW\_PARTICIAPANTS reține toți userii care participă la un review cât și efortul depus de aceștia.

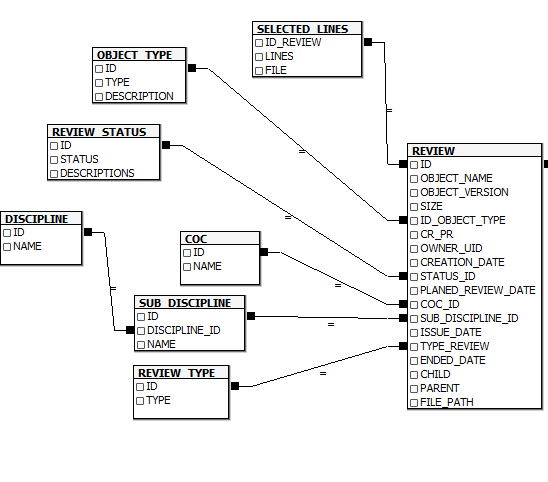
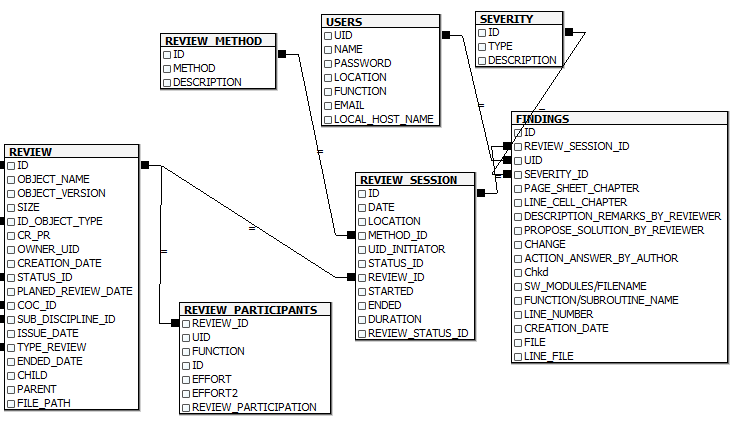


Fig. 3 Diagramă baze de date

Fig. 2 Diagramă bază de date

Restul tabelelor (SEVERITY, REVIEW\_METHOD, OBJECT\_TYPE, REVIEW\_STATUS, COC, SUB\_DISCIPLINE, REVIEW\_TYPE, DISCIPLINE) au o singură întrebuințare și anume să memoreze un singur tip de dată, cea din componența numelui lor . Totodată, ajută și la o mai bună înțelegere a bazei de date precum și fluidizarea acesteia.

### Funcții

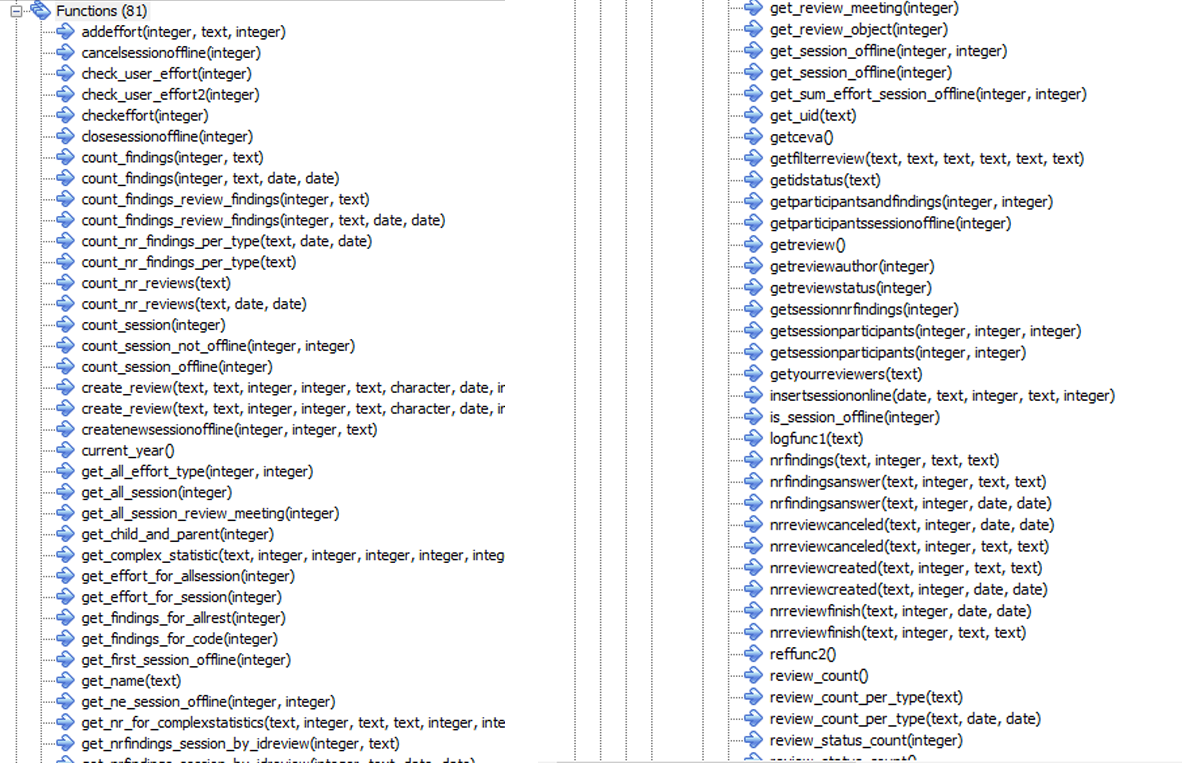
 Funcțiile joacă un rol foarte important în tot acest proces. Prin intermediul lor, PostgreSQL, dintr-un sistem de baze de date relaționele devine un server cu capacități extinse, multe procesări de date realizându-se și în interiorul serverului de baze de date. În Fig. 4 se pot vedea unele funcții care realizează acest lucru:

Fig. 4 Funții bază de date

## Client

În ceea ce privește arhitectura clientului, aceasta este formată din mai multe module. Acestea au rolul de a ușura atât munca dezvoltatorului cât și cea a utilizatorului. Fiecare fereastra deschisă constituie un nou modul.

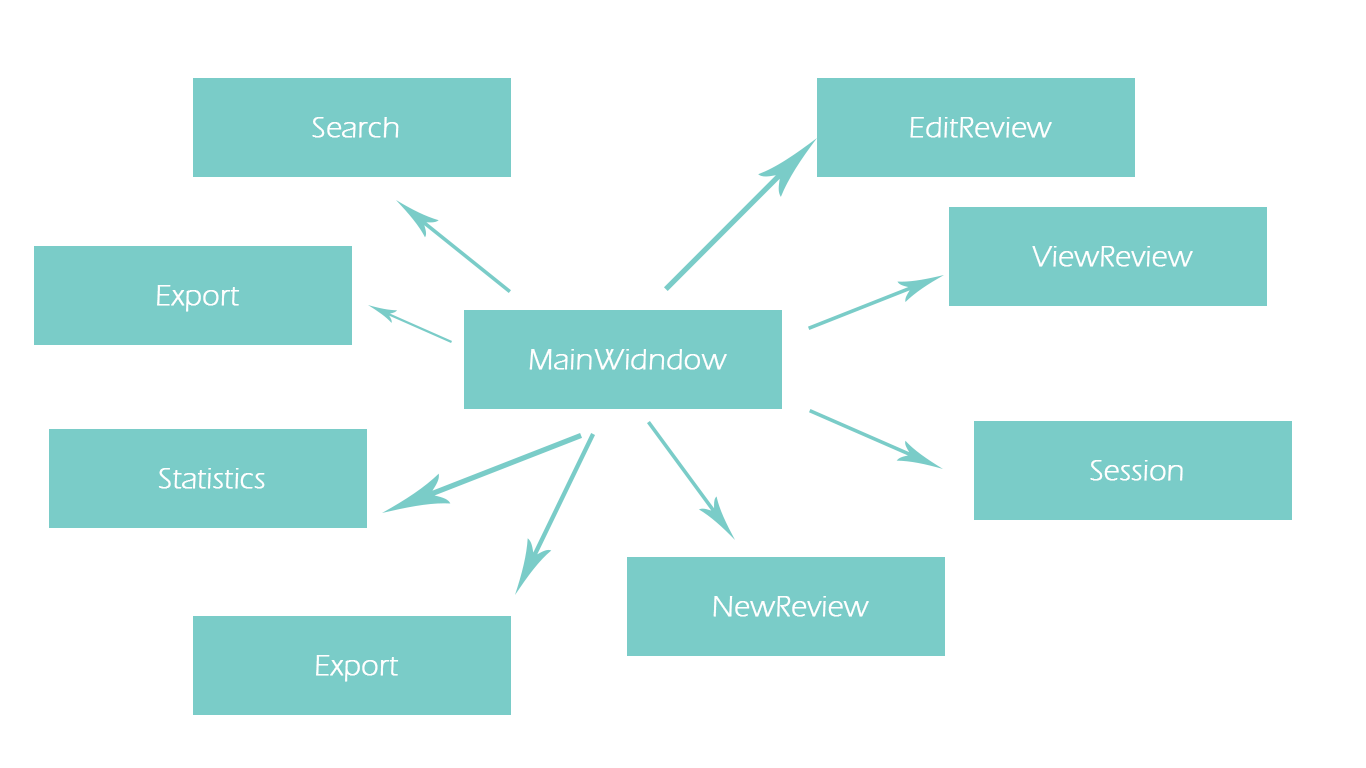
* **MainWindow** 
  + este fereastra principala din care se pot accesa majoritatea funcționalităților : NewReview, EditReview, Search, ViewReview, Export, Statistics, Session (Fig. 5)

Fig. 5 Funcționalități MainWindow

* **Session**
* Oferă posibilitatea utilizatorului să creeze pe langă sesiunea offline, un alt tip de sesiune, la alegere, din următoarea listă : *4-Eyes*, *Walthrough*, *Inspection* sau *Intensive Inspection*.
* **Export**
  + Acest modul (Fig. 6 ) face exportul informațiilor din baza de date, despre un anumit review, în format xslx, astfel realizând un raport detaliat.

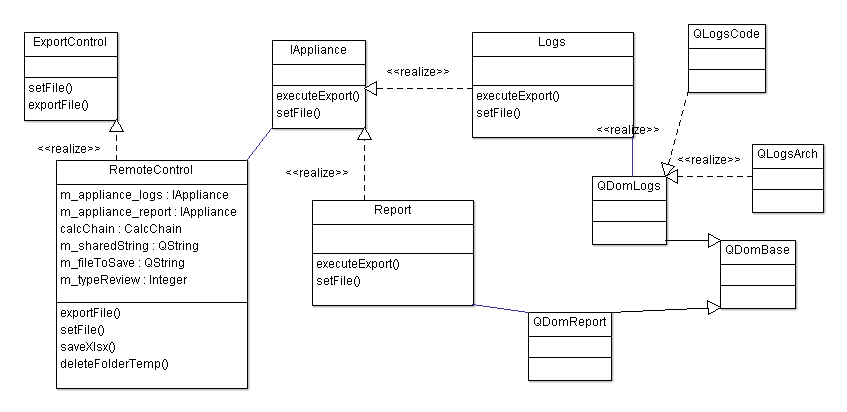


Fig. 6 Diagramă de clase Export

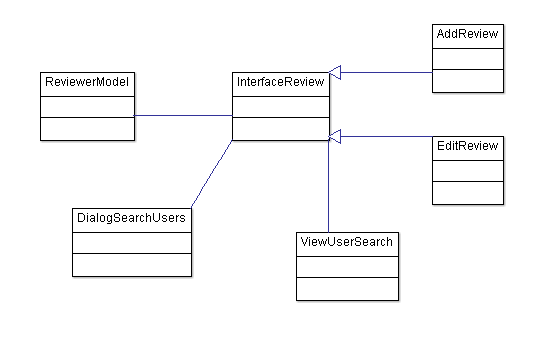
* **InterfaceReview**
  + Este clasa moștenită de NewReview și EditReview. Aceasta realizează crearea sau editarea unui review prin intermediul celor două clase menționate anterior.

Fig. 7Diagramă de clase InterfaceReview

* **Search**
  + Aici utilizatorul poate să caute un review, după anumite criterii dar totodata, poate să-și customizeze fereastra principala.
  + SearchApi este interfața implementată de catre SearchApiView care se ocupă cu cautările fară imput-uri iar SearchApiInput se ocupă de căutările care conțin parametri.

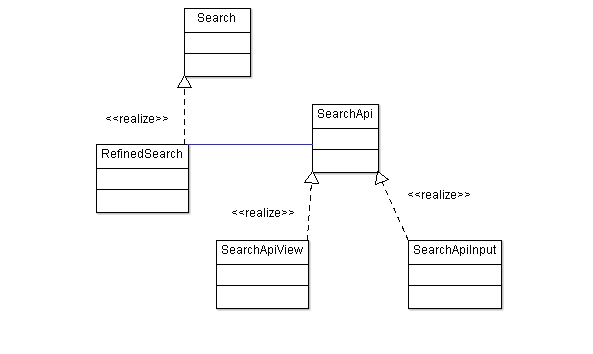


Fig. 8 Diagramă de clase Search

* **ViewReview** 
  + Aceasta este clasa care expune interfața în care utilizatorul poate să vizualizeze detalii și statistici despre review-ul curent dar și să deschidă sesiunile create.
  + În Fig. 9, sunt expuse clasele care sunt folosite pentru a facilita următoarele functii :
    - Deschiderea unui review
    - Adăugarea efortului depus
    - Inserarea, editarea și ștergerea unui finding

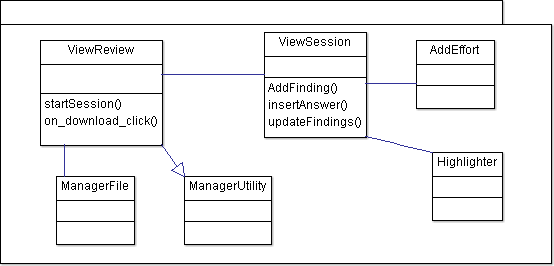


Fig. 9 Diagrame de clase ViewReview

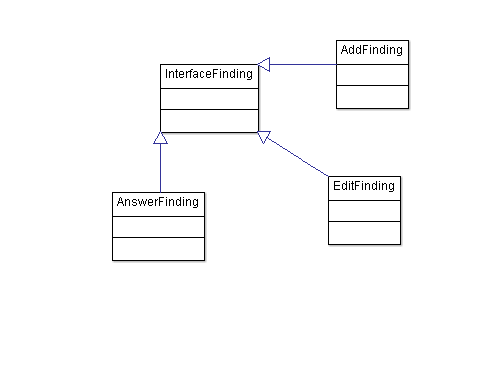
* **InterfaceFinding**

Fig. 10 Diagramă de clase InterfaceFinding

Clasa aceasta realizează crearea sau editarea unui finding. Ea conține două clase care o moștenesc, astfel interfața grafică este refolosită.

**ManagerUtility**

Acest modul este unul foarte important deoarece prin intermediul lui, aplicația comunică cu serverul bazei de date și tot prin intermediul lui primește notificări. Structura acestuia este detaliata în Fig. 11.

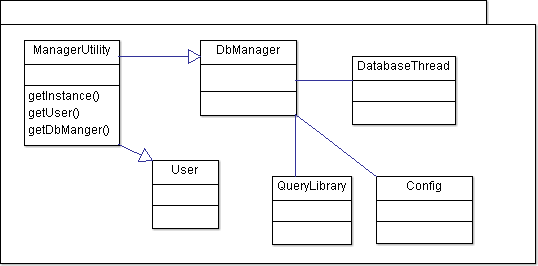


Fig. 11 Diagramă de clase ManagerUtility

Clasa *DbManager* pregatește query-urile, după care le introduce în coada din *DbThread*. Aceastea sunt executate în ordinea venirii lor. Clasa *Config* citește datele de conectare la baza de date iar *QueryLibrary* deține o serie de query-uri predefinite.

# Capitolul III. Detalii de implementare

În urmatorele pagini vom prezenta câteva detalii de implementare

## Server

Acesta va fi un proces pornit permanent care va facilita lucrul cu fișiere. Clasa *MyServer* moștenește QtcpServer având posibilitatea să creeze o adresă, unde clienții se pot conecta, după cum urmează:

|  |
| --- |
| void MyServer::startServer()  {  int port = 1234;  if(!this->listen(QHostAddress::Any, port))  {  qDebug() << "Could not start server";  }  else  {  qDebug() << "Listening to port " << port << ". . . ";  }  } |

Listing 1 StartServer

. Cand un client dorește să se conecteze, se va apela funcția *incomingConnection,* care creează un fir de execuție, astfel facilitând lucrul simultan cu mai mulți clienti conectați:

|  |
| --- |
| void MyServer::incomingConnection(qintptr socketDescriptor)  {  qDebug() << socketDescriptor << " Connecting. . . ";  MyThread \*thread = new MyThread(socketDescriptor, this);  connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));  thread->start();  } |

Listing 2 incomingConnection

În momentul în care thread-ul este pornit, se creează un QtcpSocket prin intermediul căruia se va realiza comunicarea și se setează starea INIT. Cand clientul trimite o comandă în server, se va apela slot-ul readyRead() care citește tot mesajul. Atunci când a terminat cu succes citirea din socket, se creează *ManagerCommand* care va executa comanda clientului și va intoarce rezultatul dorit de acesta.

|  |
| --- |
| QByteArray ManagerCommand::execute()  {  QByteArray returnValue;  switch (m\_command) {  case UPLOAD:  returnValue=uploadFiles();  break;  case DOWNLOAD:  returnValue= downloadFiles();  break;  case INFOFILE:  returnValue= infoFile();  break;  case INFODIR:  returnValue= infoDir();  break;  case UPDATE:  break;  default:  break;  }  return returnValue;  } |

Listing 3 Execuția comenzii trimise de client

Pentru trimiterea mesajelor de la client la server și invers, este necesar ca inițial să fie trimis numarul total de biți ai mesajului. Pentru citirea și scrierea mesajelor se vor folosi funcțiile readyRead()(Listing 4) respectiv sendBack() (Listing 5), cea din urmă primind ca parametru, bufferul gata procesat.

|  |
| --- |
| void MyThread::readyRead()  {  bytesReceived += (int)socket->*bytesAvailable*();  QByteArray buffer=socket->readAll();  int pos;  switch(state)  {  case INIT:  pos=buffer. indexOf(";");  TotalBytes=buffer. left(pos). toInt();  buffer. remove(0, pos+1);  m\_buffer. append(buffer);  state=READ\_ALL;  break;  case READ\_ALL:  m\_buffer. append(buffer);  break;  }  if (bytesReceived == TotalBytes)  {  state=INIT;  bytesReceived=0;  readCommandAndExecute();  TotalBytes=0;  m\_buffer. clear();  }  } |

Listing 4 Citirea serverului din buffer

Listing 5 Scrierea serverului în buffer

|  |
| --- |
| void MyThread::sentBack(QByteArray buffer)  {  qDebug()<<"sendt back";  socket->write(buffer);  } |

Procesarea fișierelor se realizează cu ajutorul programului *7zip*, care arhivează sau dezarhivează în funcție de nevoie, conținutul folderelor după care acesta va fi scris într-un buffer. Una dintre funcțiile care demonstrează cele spuse, este *downloadFile():*

|  |
| --- |
| QByteArray ManagerCommand::downloadFiles()  {  int idReview=read(). toInt();  QString folderName=read(). toString();  QDir dir(FOLDER\_FILE+QString::number(idReview)+"/"+folderName);  qDebug()<<"dirpath="<<dir. path()<<"\n";  QProcess \*proc=new QProcess();  QString m\_fileToSave=FOLDER\_TEMP+ folderName+". zip";  QString command="\""+m\_director. absolutePath()+"/7z/7za. exe\" a -tzip -r \""+m\_fileToSave+"\" "+"\"+FOLDER\_FILE+QString::number(idReview)+"/"+folderName+"/\*. \*\"";  qDebug()<<command;  proc->start(command);  proc->waitForFinished(60000);  QFile file( fileToSave );  file. open(QIODevice::ReadOnly);  TotalBytes=file. size();  QByteArray returnVariable;  if(TotalBytes<1e+9)  {  QByteArray mydata=file. readAll();  QString auxbytes=QString::number(TotalBytes)+";"+QString::number(DOWNLOAD)+";";  TotalBytes+=auxbytes. size();  QString total=QString::number(TotalBytes)+";"+QString::number(DOWNLOAD)+";";;  returnVariable. append(total);  returnVariable. append(mydata);  }  file. remove();  return returnVariable;  } |

Listing 6 Funția de downloadFile a serverului

## Baza de date

PostgreSQL oferă posibilitatea unui client să creeze și să se aboneze la un canal, prin care serverul poate să trimită notificări acestuia, atunci când în baza de date are loc o acțiune. Funcția *notify\_trigger()* demonstrează acest lucru prin urmatorul cod parțial:

|  |
| --- |
| CREATE OR REPLACE FUNCTION public. notify\_trigger()  RETURNS trigger AS  $BODY$  DECLARE  id bigint;  id\_table integer;  name\_table text;  author text;  severity bigint;  id\_session bigint;  duration interval;  id\_review bigint;  session\_type text;  effort integer;  BEGIN  IF TG\_TABLE\_NAME='REVIEW' THEN  IF TG\_OP='INSERT' OR TG\_OP='UPDATE' THEN  id:=NEW. "ID";  PERFORM pg\_notify('change\_'|| 'REVIEW', json\_build\_object('table', TG\_TABLE\_NAME, 'type', TG\_OP, 'id', id)::text);  END IF;  END IF;  IF TG\_TABLE\_NAME='FINDINGS' THEN  IF TG\_OP = 'INSERT' OR TG\_OP = 'UPDATE' THEN  id := NEW. "ID";  author:=NEW. "UID";  severity:=NEW. "SEVERITY\_ID";  SELECT "REVIEW\_SESSION\_ID" INTO id\_table FROM "FINDINGS" WHERE "ID"=id;  ELSE  id = OLD. "ID";  author=OLD. "UID";  severity=OLD. "SEVERITY\_ID";  id\_table:=OLD. "REVIEW\_SESSION\_ID";  END IF;    PERFORM pg\_notify('change\_'|| 'REVIEW\_SESSION'||'\_'||id\_table, json\_build\_object('table', TG\_TABLE\_NAME, 'type', TG\_OP, 'id', id, 'author', author, 'severity', severity)::text);  SELECT "REVIEW\_ID" INTO id\_review FROM "REVIEW\_SESSION" WHERE "ID"=id\_table;  RAISE NOTICE 'Calling cs\_create\_job(%)', id\_table;  PERFORM pg\_notify('change\_'|| 'REVIEW\_FINDINGS'||'\_'||id\_review, json\_build\_object('table', TG\_TABLE\_NAME, 'type', TG\_OP, 'id\_session', id\_table, 'severity', severity)::text);  PERFORM pg\_notify('change\_'|| 'SESSION\_ONLINE\_FINDINGS'||'\_'||id\_review, json\_build\_object('table', TG\_TABLE\_NAME, 'type', TG\_OP, 'id', id)::text);  END IF;  RETURN NULL;  END;  $BODY$  LANGUAGE plpgsql VOLATILE |

Listing 7 Triggerul din baza de date

În codul anterior când una dintre condiții este îndeplinită, se trimite o notificare prin intermediul funcției *pg\_notify*. Primul parametrul este numele canalului, urmat de un mesaj, în cazul nostru, un json\_*object* care conține diferite informații, cum ar fi: id-ul tabelei, severitatea unui finding.

Funcțiile joacă și ele un rol important în eficientizarea întregului proces, astfel clientul trebuie doar să cunoască numele funcției dorite și parametri acestuia . În acest mod se micșoreză șansa unei greșeli. De exemplu în funcția *get\_child\_and\_parent(integer),* clientul nu trebuie să execute nenumărate interogări, acestea fiind realizate de catre server, după cum urmează:

|  |
| --- |
| CREATE OR REPLACE FUNCTION public. get\_child\_and\_parent(id\_review integer)  RETURNS SETOF tree\_node AS  $BODY$  DECLARE  r tree\_node%rowtype;  v\_id integer;  v\_status\_id integer;  BEGIN  SELECT "CHILD" INTO v\_id FROM "REVIEW" WHERE "ID"=id\_review;  r. id\_review:=v\_id;  SELECT "STATUS\_ID" INTO v\_status\_id FROM "REVIEW" WHERE "ID"=v\_id;  IF v\_status\_id=2 OR v\_status\_id=1 THEN  r. status:=1;  ELSE  r. status:=0;  END IF;  return next r;    SELECT "PARENT" INTO v\_id FROM "REVIEW" WHERE "ID"=id\_review;  r. id\_review:=v\_id;  SELECT "STATUS\_ID" INTO v\_status\_id FROM "REVIEW" WHERE "ID"=v\_id;  IF v\_status\_id=2 OR v\_status\_id=1 THEN  r. status:=1;  ELSE  r. status:=0;  END IF;  return next r;  return;  END;  $BODY$  LANGUAGE plpgsql VOLATILE  COST 100  ROWS 1000; |

Listing 8 Funția care realizeză căutarea familiei de review-uri

Funcția din Listing 8, caută toți copii și parinții review-ului cât și statusul acestora. În cazul în care, interogarea nu returnează nimic, *tree\_node* va luat valoarea null.

## Client

Partea de client a fost dezvoltată în Qt, fiind o aplicație care rulează pe sistemul de operare Windows.

Pentru conexiunea la serverul bazei de date *PostgreSQL*, am creat *ManagerUtility* iar pentru conexiunea la serverul ce se ocupă cu lucrul fișierelor am creat *ManagerFile*. În amebele cazuri se utilizează pattern-ul Singleton, pentru ca această conexiune sa fie unică și să poată fi accesată de oriunde.

**Autentificare**

La intrarea în aplicație, a utilizatorului, se incarcă datele din fișierul de configurare pentru conexiunea la baza de date după care se afișează form-ul de loghin în care utilizatorul este rugat să introducă uid-ul, parola și să-și aleagă proiectul pe care lucrează. La apăsarea butonului Login, se crează conexiunea la baza de date după care se apelează funcția Login. Dacă acesta a introdus datele corecte, aplicația va deschide fereastra principala unde vor fi încarcate elementele ce țin de interfață. În caz contrat, se va invoca funcția *addConnectionLogin* pentru a se repeta pasul anterior.

Listing 9 Login-ul și verificarea acestuia

|  |
| --- |
| if(ManagerUtility::getInstace()->getDbManager()->setConfig())  {  LoginForm \*longinForm=new LoginForm(ManagerUtility::getInstace()->getDbManager()->getProjectsName(), m\_error\_login);  if (longinForm->exec() != QDialog::Accepted)  {  okFlag=0;  }  else  {  if( ManagerUtility::getInstace()->getDbManager()->connectDb(longinForm->getProject()))  {  if(ManagerUtility::getInstace()->getDbManager()->loginUser(longinForm->getUser(), longinForm->getPassword())==0)  {  m\_error\_login=1;  QMetaObject::invokeMethod(this, "addConnectionLogin"); |

**InterfaceReview**

Această clasă conține o interfață grafică, care este moștenita de catre *NewReview* și *EditReview*.

În momentul când se apelează constructorul clase *NewReview* sau *EditReview*, se vor executa o serie de apeluri consecutive asupra bazei de date. Pentru a nu bloca interfața, aceste funcții se execută asincron, rezultatul fiind furnizat sloturilor, în momentul în care se emite semnalul conectat la acestea.

|  |
| --- |
| ManagerUtility::getInstace()->getDbManager()->getAllSeverity();  ManagerUtility::getInstace()->getDbManager()->getAllStatus();  ManagerUtility::getInstace()->getDbManager()->getAllMethod(); |

Listing 10 Extragerea informațiilor din baza de date

După ce interfața a fost creată utilizatorul trebuie să completeze datele necesare și să adauge persoanele care vor participa la review. Pentru o interfață mai intuitivă am creat *ModelComboBox* care reține rolul user-ului în review într-un combobox în care se vor concatena atribuțiile acestuia.

|  |
| --- |
| ModelComboBox \* roleComboBox=new ModelComboBox(1);  Reviewer \*reviewer=new Reviewer(model->record(0). value(0). toString(). trimmed(), email, roleComboBox, INSERT);  connect(roleComboBox, SIGNAL(currentTextChanged(QString)), reviewer, SLOT(handleSelectRole(QString)));  m\_reviewerModel. addReviewer(reviewer);  ui->tableWidget\_sessionParticiapants->setCellWidget(row, 4, roleComboBox); |

Listing 11 Adăugare user în tableview

La apăsarea butonului Create, se execută o serie de verificări, pentru toate imputurile obligatorii. În urmatorul pas se apelează funcția *insertReview*, din *DbManager*. Când acesta a fost inserat cu succes se apelează slotul *handleSignal\_insertUpdateReviewin* în care se adaugă în baza de date userii care vor participa. Am folosit metoda aceasta deoarece aveam nevoie să primesc id-ul review-ului înainte de a adăuga participanții.

În cazul în care se editeaza review-ul, la început se vor extrage din baza de date toți userii care sunt asignați acestui review. Ei vor fi memorați într-un *TableWidget*. Atunci când se apeleaza *editReview* se verifică starea fiecarui reviewer astfel încât, dacă acesta era participant dar a fost trecut în starea *DELETE*, va fi șters iar daca are starea *INSERT*, va fi introdus în baza de date. Toate aceste stări sunt furnizate de modelul *ReviewerModel* care are ca item un *Reviewer*.

**Export**

Aici am folosit design pattern-ul *Command*. Am creat un obiect *RemoteControl* care reține parametri pentru comandă. Acesta anunța obiectul comandă (*Logs*, *Report*) prin execuția metodei *exportFile().*

|  |
| --- |
| void RemoteControl::exportFile(QString nameFile)  {  deleteFolderTemp(m\_director. absolutePath()+FOLDER\_TEMP);  if(m\_appliance\_report!=NULL && m\_appliance\_logs!=NULL)  {  m\_fileToSave=nameFile;  QString tempFolder=createTempFolder();  setFile(tempFolder);  m\_appliance\_report->executeExport();  m\_appliance\_logs->executeExport();  saveXlsx();  }  } |

Listing 12Funcția de export

În interiorul acestei funcții se crează un folder temporar în care se vor introduce toate fișierele necesare și care vor urma să fie editate. În funcție de tipul obiectului, funcția *executeExport(),* realizează sarcini diferite.

Obiectul *Logs* va încărca din baza de date toate finding-urile și în functie de tipul review-ului va crea un obiect *QdomLogs* și va apela *createLogs().*  Aceasta va scrie toate datele în fisierul xml într-un anumit format.

Pentru obiectul *Report* se vor executa mai mulți pași întrucât este mai complex. Aici avem nevoie de mai multe informații pe care le vom extrage cu ajutorul urmatoarelor funcții:

* get\_object\_review(m\_idReview);
* get\_review\_meeting(m\_idReview);
* get\_review\_preparation(m\_idReview);

Fiecare functie în parte are asociat un slot care va fi apelat în momentul în care ne este furnizat rezultatul interogării și se vor executa funcțiile din *QdomReport*:

* m\_dom->createObjectReviewReport(model);
* m\_dom->createReviewPreparation(model);
* m\_dom->createMeeting(model);
* m\_dom->save();

În momentul terminarii execuției celor două apeluri ale funcției *executeExport()*, se va salva fișierul xlsx în următorul mod: se va arhiva întreg folderul temporar creat la inceput, se salvează în locațioa dorită de user iar în final va fi șters din folderul temporar pentru a nu ocupa spațiu.

Arhivarea și dezarhivarea fișierului de tip xlsx se realizează cu ajutorul programului 7zip. Pentru a lucra cu un fișier temaplate, cu formatul xlsx am adoptat următorul mod:

* dezarhivez acest fișier, el la origine fiind o arhivă, iar din interiorul lui avem nevoie de folder-ul worksheet . Acesta conține două fișiere „sheet1. xml” și „sheet2. xml” pe care ulterior le vom edita .
* Pentru a putea introduce date în cele două fișiere, trebuie sa facem legatura între acestea cu ajutorul fișierului „sharedString. xml”. Acesta atribuie fiecarui string memorat în el, un numarul unic. Numarul acesta trebuie introdus în sheet1 sau sheet2.
* Pentru a nu strica lanțul de comandă, trebuie să mai edităm un fișier, și anume „calcChain. xml”.
* Toate acestea se fac pentru a putea utiliza șablonul pentru fiecare tip de review în parte și astfel se evită crearea de la zero
* În final se reface structura

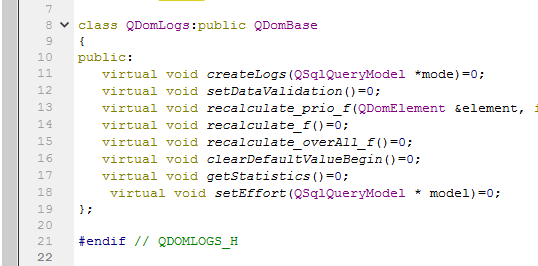
Pentru a lucra cu fișierul xml am creat interfața QdomBase care este implementată de catre QdomLogs și QdomReport. In Fig. 12 se poate vedea o listare a operațiilor

Fig. 12 Lista de operații posibile a clasei QDomBase

**SetModificication**

Pentru ca utilizatorul să poată evidenția schimbările aduse de el asupra fișierelor am creat această clasă. Aici este încărcat arborele directorului în care se află fișierele review-ului.

Atunci când se selecteză un fișier, se face request pentru conținutul acestuia și se afișează în view. Pentru a nu îngreuna traficul de date, în momentul încarcării fișierului, utilizatorul nu mai poate selectat alt fișier, fiind necesar finalizarea execuției.

|  |
| --- |
| void SetModification::selectionChangedSlot(const QItemSelection & const QItemSelection &  {    const QModelIndex index = ui->m\_listFile->selectionModel()->currentIndex();  QString pathRoot=index. parent(). data(Qt::DisplayRole). toString();  QString selectedText = index. data(Qt::DisplayRole). toString();  if(ceva. compare("file")==0)  {  m\_fileSelected=pathRoot+"/"+selectedText;  ui->m\_numeFisierDeschis->setText(selectedText);  qDebug()<<m\_fileSelected;  ui->m\_listFile->setEnabled(false);  connect(ManagerFile::getInstace(), SIGNAL(signalInfoFile(QByteArray)), this, SLOT(signal\_handlerInfoFile(QByteArray)));  ManagerFile::getInstace()->infoFile(m\_idReview, pathRoot, selectedText);  } |

Listing 13 Prealurea conținutului unui fișier din baza de date

Utilizatorul poate evidenția liniile de cod ținând click stânga pe cod și trăgând în jos până în punctul dorit. Astfel se crează o selecție de culoare galbenă. După care trebuie să apese click dreapta pe mouse și să dea add(), selecția devenind verde. În momentul acesta se introduce în *m\_lines\_selected*, care este un *QMap<Qstring, Qvector<Qstring>>*, numele fișierulului asupra caruia sa realizat selecția împreună cu punctele de început si sfârșit a acesteia.

Totodată în *m\_lines\_selected\_from\_db* sunt introduse fișierele și liniile selectate asupra cărora se va face split după delimitatorul „-”, după care se setează cursorului, poziția de start și de final, selecția având culoarea roșie.

|  |
| --- |
| void SetModification::selectArea(QString line, Qt::GlobalColor color)  {  QStringList list=line. split("-");  QTextCursor c = ui->m\_contentFile->textCursor();  c. setPosition(QString(list. at(0)). toInt());  c. setPosition(QString(list. at(1)). toInt(), QTextCursor::KeepAnchor);  ui->m\_contentFile->setTextCursor(c);  QTextCharFormat tf;  tf = ui->m\_contentFile->currentCharFormat();  tf. setBackground(QBrush((Qt::GlobalColor)color));  ui->m\_contentFile->setCurrentCharFormat(tf);  QTextCursor cu= ui->m\_contentFile->textCursor();  cu. setPosition(0);  ui->m\_contentFile->setTextCursor(cu);  } |

Listing 14 Funcția selectarea()

Dacă user-ul se răzgândește, acesta poate apăsa butonul reset care va anula selecția facută anterior.

|  |
| --- |
| void SetModification::on\_pushButton\_reset\_clicked()  {  if(m\_lines\_selected. contains(m\_fileSelected))  {  clearFromReset(m\_fileSelected);  m\_lines\_selected[m\_fileSelected]. clear();  }  } |

Listing 15 Funcția de resetare a selecției

**ViewReview**

Atunci când utilizatorul apasă butonul *Open Review* se va deschide fereastra *ViewReview*. La început se vor conecta toate semnalele cu baza de date și butoanele din view.

|  |
| --- |
| void ViewReview::connectAction()  {  logFile(LOG);  connect(ui->m\_table\_allSession, SIGNAL(customContextMenuRequested(QPoint)), this, SLOT(showContextMenu(const QPoint&)));  QObject::connect(ManagerUtility::getInstace()->getDbManager(), SIGNAL(sendReceiveSignal(const QString&, QSqlDriver::NotificationSource, const QVariant &)), this, SLOT(receiveSignal(const QString &, QSqlDriver::NotificationSource, const QVariant &)));  QObject::connect(ui->pushButton\_statSession, SIGNAL(clicked()), this, SLOT(startSessionClick()));  QObject::connect(ui->pushButton\_viewSession, SIGNAL(clicked()), this, SLOT(viewSessionClick()));  QObject::connect(m\_viewRightClick, SIGNAL(triggered()), this, SLOT(viewSessionClick()));  connect(ManagerFile::getInstace(), SIGNAL(signalClose()), this, SLOT(signal\_handlerCloseServer()));  QObject::connect(ui->pushButton\_openParent, SIGNAL(clicked()), this, SLOT(openParent()));  QObject::connect(ui->pushButton\_openChild, SIGNAL(clicked()), this, SLOT(openChild()));  } |

Listing 16 Conectare semnale și sloturi

În cazul în care user-ul este cel ce a creat acest review iar review-ul înca este în starea CREATED atunci acesta va avea un buton Set pentru a porni sesiunea offline. În caz contrar în view va apărea butonul Open Session.

|  |
| --- |
| void ViewReview::connectSubscribers()  {  m\_channel\_session="change\_REVIEW\_"+QString::number(m\_idReview);  m\_channel\_review\_finding="change\_REVIEW\_FINDINGS\_"+QString::number(m\_idReview);  m\_channel\_session\_participants="change\_REVIEWER\_SESSION\_"+QString::number(m\_idReview);  ManagerUtility::getInstace()->getDbManager()->subScriebToNotification(m\_channel\_session);  ManagerUtility::getInstace()->getDbManager()->subScriebToNotification(m\_channel\_review\_finding);  ManagerUtility::getInstace()->getDbManager()->subScriebToNotification(m\_channel\_session\_participants);  } |

Pentru a primi notificări de la baza de date se vor crea canale, prin care se vor trimite mesaje. Pentru a nu trimite notificări care nu sunt necesare, altui user care are deschisă aceeași fereastră dar pentru un alt review, se creează un canalul special, cu id-ul review-ului în coadă, astefel se vor putea diferenția.

Listing 17 Crearea canalului de comunicație cu baza de date

**ViewSession**

În momentul în care user-ul apasă butonul Open Session din *ViewReview* se va deschide fereastra View Session. În constructorul acesteia se setează UI-ul, se conecteză semnalele la sloturi și se aduc din baza de date toate informațiile necesare. Totodată se cere serverului, informații despre structura directorului în care se află fișierele review-ului, acestea fiind introduse ulterior într-un *TreeView*.

Fiecare finding adăugat poate să fie asociat unei linii dintr-un fișier sau nu. În cazul în care este asociat, atunci cand user-ul va da click pe finding, automat se va prelua numele fișierului din tabel, se va face request la server pentru conținut și se va afișa în view, linia la care se află finding-ul, aceasta fiind evidențiată.

Când utilizatorul dorește să adauge un finding cu o linie specifică, atunci el trebuie să evidențieze acea linie prin apăasarea butonul click stânga după care să apese butonul click dreapta și să apeleze funcția add(). După inserare, view-ul se va updata automat, finding-ul fiind evidențiat prin culoarea verde.

|  |
| --- |
| void DbManager::updateReview(QVector<QString> review)  {  QString query="select update\_review(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";  QVector<QString>vector;  for(int i=0;i<review. size();i++)  {  vector. push\_back(review[i]);  }  m\_dbThread->setQuery(query, vector, &handleSignal\_updateReview);  } |

**DbManager**

Listing 18 Funcția de update a Review-ului

Prin intermediul acestei clase se poate comunica cu baza de date. Atunci când o funcție este apelată, se pregatește query-ul, se construiește vectorul din care se va face bind la valori și daca este cazul se va furniza și funcția de callback. Toate acestea se vor trimite ca parametri la DbThread prin funcția setQuery. La finalizarea cererii, se va apela callback-ul dat ca parametru, din care se va emite un semnal . Acest semnal a fost conectat anterior la un slot al apelantului. Astfel cel care a apelat funcția va primi rezultatul dorit. Folosind această metodă, nu este necesar ca DbManagerul să cunoasca cealaltă entitate.

**DbThread**

|  |
| --- |
| void DbThread::check\_stackForQuery()  {  if(!m\_run)  {  if(m\_stackQuery. size()>0 && m\_stackSlot. size()>0)  {  m\_query\_string=m\_stackQuery[0];  m\_slot=m\_stackSlot[0];  m\_bind=m\_stackBind[0];  m\_stackQuery. pop\_front();  m\_stackSlot. pop\_front();  m\_stackBind. pop\_front();  m\_run=true;  qDebug()<<"DbThread::start";  this->start();  }  }  } |

DbManager-ul apelează funcția setQuery(), care introduce fiecare parametru într-o coadă, după care verifică starea thread-ului prin funcția check\_stackForQuery() (Listing 19). Daca thread-ul nu rulează, atunci se setează variabilele *m\_slot, m\_bind, m\_query\_string* cu cele din coadă și se dă start() la thread. În caz contrar, se așteaptă finalizarea acestuia.

Listing 19 Funcția check\_stackForQuery

În funcția *run()* se verifică daca conexiunea la baza de date este deschisă, în caz contrat se emite semnalul *disconnectDb()*. După acest pas thread-ul pornește execuția query-ului. La finalizarea acestuia se va returna un *QSqlQueryModel* în care se va afla rezultatul, prin intermediul callback-ului, furnizat funcției *setQuery().*

|  |
| --- |
| void DbThread::*run*()  {  qDebug()<<"DbThread::run";  qRegisterMetaType<callback\_fct>("callback\_fct");  if(m\_db. isOpen())  {  qDebug()<<"DbThread::database is open";  }  else  {  qDebug()<<"DbThread::database is not open";  emit disconnectDb();  }  m\_db. transaction();  QSqlQuery \* query=getQuery();  query->prepare(m\_query\_string);  for(int i=0;i<m\_bind. size();i++)  {  qDebug()<<"bind value";  query->bindValue(i, m\_bind[i]);  }  if(query->exec())  {  qDebug()<<"query succesful";  qDebug()<<query->size();  }  else  {  query->lastError();  }  m\_db. commit();  QSqlQueryModel \* model=new QSqlQueryModel;  model->setQuery(\*query);  if(m\_slot!=0)  {  qDebug()<<"DbThread::emit signal";  emit finish(model, m\_slot);  }  delete query;  m\_bind. clear();  qDebug()<<"DbThread::finish run";  m\_run=false;  check\_stackForQuery();  } |

Listing 20 Funcția run din DbThread

Tot prin intermediul acestei clase se poate înregistra canalul dorit și să te abonezi la el. În Listing 21 se pot vedea cele doua funcții .

|  |
| --- |
| void DbThread::subScriebToNotification(QString channel)  {  logFile(LOG);  qDebug()<<"subscribe ToNotification="<<channel;  m\_run=true;  if(!m\_db. driver()->*subscribeToNotification*(channel))  {  qDebug( qPrintable(m\_db. driver()->lastError(). databaseText()) );  }  m\_run=false;  check\_stackForQuery();  }  void DbThread::unListenChannel(QString channel)  {  qDebug()<<"Unsubscribe";  m\_run=true;  if(!m\_db. driver()->*unsubscribeFromNotification*(channel))  {  qDebug( qPrintable(m\_db. driver()->lastError(). databaseText()) );  }  m\_run=false;  check\_stackForQuery();  } |

Listing 21 Funcțtiile de abonare și dezabonare de la canlul bazei de date

Atunci când baza de date trimite o notificare, se apelează slot-ul receiver (Listing 23) conectat la semnalul notification (Listing 22).

|  |
| --- |
| QObject::connect(m\_db. driver(), SIGNAL(notification(const QString&, QSqlDriver::NotificationSource, const QVariant &)), this, SLOT(receiver(const QString&, QSqlDriver::NotificationSource, const QVariant &)) ); |

Listing 22Conectare semnal

|  |
| --- |
| void DbThread::receiver(const QString &s, QSqlDriver::NotificationSource d, const QVariant &b)  {  logFile(LOG);  qDebug()<<s<<d<<b;  QMap<QString, QVariant>map;  if(b. toString(). isEmpty())  {  map. insert("type", "signal");  }  else  {  toMap(b, map);  }  emit sendReceiveSignal(s, d, map);  } |

Listing 23 Slotul care se apeleză la notifcări

Pentru a trimite aplicației mesajul din baza de date, acesta este inserat într-un obiect de tip *QMap*, pentru ca informațiile să fie accesate mai ușor.

**InterfaceFinding**

Această clasă ajută la adăugarea finding-urilor. În funcție de tipul review-ului sunt schimbate în view, laber-urile *label\_pageSheet, label\_lineCell, label\_propose* . Dacă userul nu a completat corect toate câmpurile, atunci va fi semnalizat prin setarea focusului pe *textlabel-ul* necompletat.

**Search**

Pentru crearea acestui modul am folosit design pattern-ul *Bridge*. Am dorit sa avem două clase care au aceeași structură dar care să opereze diferit, astfel să evităm legarea permanentă dintre clasa abstractă și implementarea sa.

Clasa *Search* este clasa abstractă care defintește interfața iar clasa *RedefineSearch* definește o specializare a interfeței, și anume searchBy(). *SearchApi* joacă rolul implementatorului abstract definind interfața iar SearchApiInput și SearchApiView o implementează propriu-zis.

# Capitolul IV. Șcenarii de utilizare

În acest capitol vom prezenta șcenariile de utilizare posibile ale aplicației, precum și imagini care să ajute la o mai bună înțelegere.

## Logarea

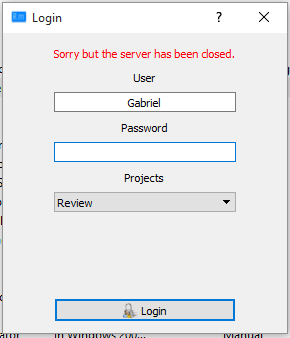
Pentru a se loga, utilizatorul trebuie să introducă uid-ul, parola și să selecteze proiectul dorit.

Fig. 13 Login , server închis

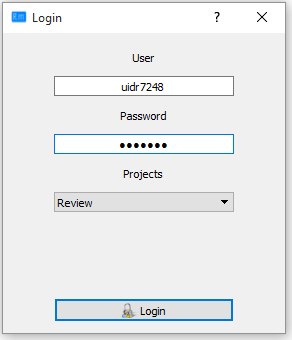


Fig. 14 Login

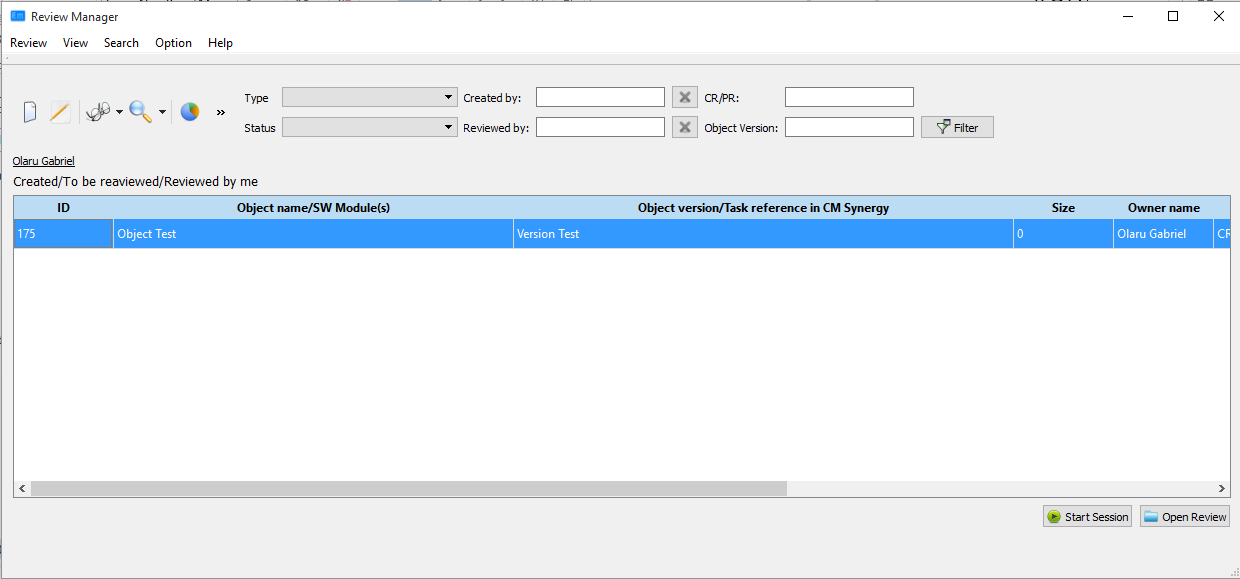
În cazul în care serverul este închis se va afișa un mesaj de eroare, în caz contrat programul va deschide fereastra principlă. Fig. 15

Fig. 15 Fereastra principală

## MainWindow

Fereastra MainWindow (Fig. 15) conține un tabel în care sunt afișate review-urile în funcție de ce tip de view a selectat utilizatorul. În caz că userul selectează un review creat de el, atunci în partea dreaptă jos, va apărea un buton Start Session, altfel va fi afișat doar cel cu Open Review.

Atunci când utilizatorul dă click pe Open Review se va deschide review-ul selectat. În cazul în care el nu a selectat nici un review, butoanele vor fi gray out.

În partea de sus, fereastra conține un meniu cu următoarele opțiuni:

* Review: Create Review, Exit
* View: de aici se poate seta ce tipuri de review-uri să apară în tabelul principal
* Search: oferă posiblitatea utilizatorului să caute review-uri după anumite criterii
* Option: user-ul poate schimba tema aplicației în funcție de preferințe sau să apeleze funcția export.

Sub acesta, se află un meniul rapid pe care îl vom detalia în secțiunea [*QuickMenu*](#_QuickMenu).

## CreateReview

Fig. 16 Fereastra Create review

În această etapa, user-ul trebuie să completeze câmpurile Object Name, Object Version, CR/PR, Object type și să încarce fișierele ce urmează să fie revizuite. Prin apăsarea butonului Add, se pot adăuga utilizatori noi iar în dreptul lor va apărea un combobox, în care se poate seta rolul acestora.

Fiecare Review, trebuie să aiba minim un Reviewer[[4]](#footnote-4), Moderator[[5]](#footnote-5) și un Minute Keeper[[6]](#footnote-6), altfel utilizatorul nu va putea creea review-ul.

## EditReview

Fig. 23 Fereastra de editare a review-ului

Acest lucru se realizează prin selectarea review-ului din fereastra principală și selectarea opțiunii Edit. View-ul este identic cu cel de la NewReview doar că aici datele sunt completate, utilizatorul având datoria de a edita participanții la review dar și status-ul.

Sunt anumite criterii pe care un Review trebuie sa le respecte:

* Dacă este în starea *Created* sau *Ongoing* atunci acesta va putea fi trecut în *Canceled*, *Finished* sau *Finished follow up needed*
* Dacă din *Ongoing* este trecut în starea *Finished follow up needed* atunci automat acest review se va închide iar un alt review cu aceleași informații se va crea, cel din urmă fiind fiul precedentului.
* Dacă este în starea *Canceled* sau *Finished*, acesta nu va putea fi editat

## ViewReview

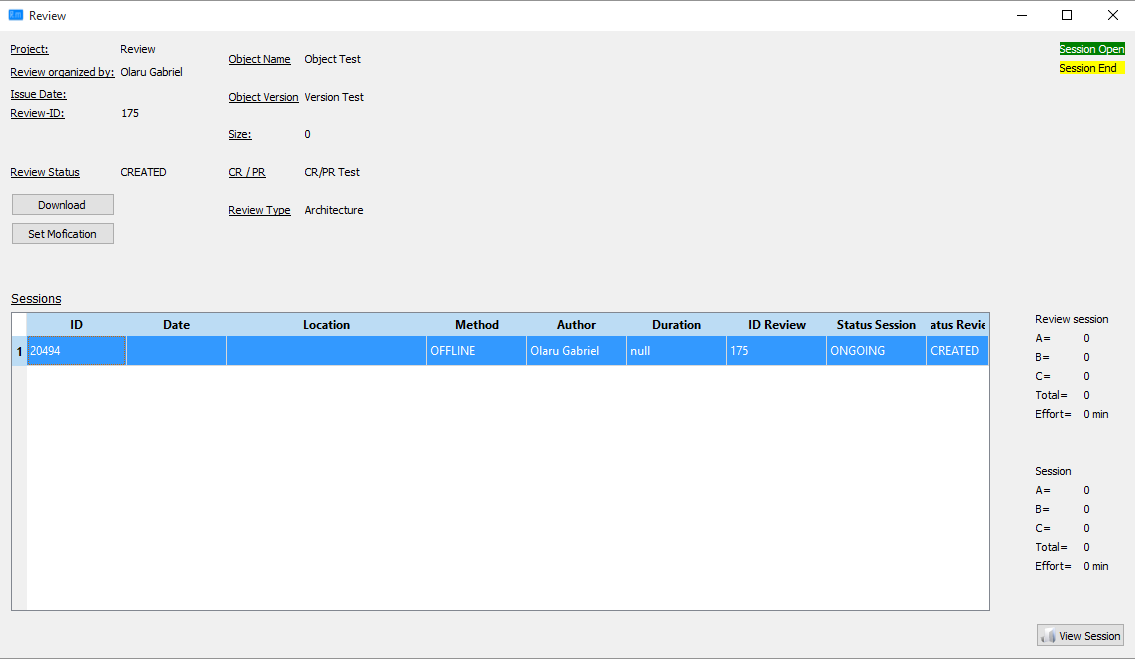


Fig. 17 Fereastra de vizualizare a review-ului

Dacă user-ul este autorul review-ului iar acesta nu are sesiunea offline deschisă, atunci în view va apărea în locul butonul ViewSession, butonul Start.

Odată deschisă sesiunea offline, ceilalți participanți la review, vor putea adăuga finding-uri. Tot din acest view, se pot descărca fișierele review-ului sau se pot adăuga marcaje pentru a evidenția modificările aduse fișierelor.

## ViewSession

Fig. Fereastra de vizualizare a sesiunii

În cazul în care user-ul ce a deschis sesiunea este participant la review, va avea posibilitatea să adauge finding-uri. Acest lucru se poate realiza în două moduri:

* Selectează linia din fișierul dorit, dupa care apasă click dreapta selectând optiunea Add.
* Apasă butonul *Add Finding* din dreapta jos a ferestrei. În acest mod se adaugă un finding mai general

În partea stângă sus se află lista participanților iar în partea de jos, lista fișierelor cât și conținutul acestora.

## SetModification

Dacă utilizatorul dorește să evidențieze modificările aduse fișierelor, are această posibilitate. El trebuie să selecteze liniile din fișier cu ajutorul mouse-ului și să dea *Add*. În momentul acesta, datele sunt salvate temporar, ulterior prin apăsarea butonul *Ok*, acestea vor fi salvate în baza de date. În cazul în care a comis vreo greșeală sau dorește altă selecție poate apăsa pe butonul *Reset*, care va șterge datele temporare.

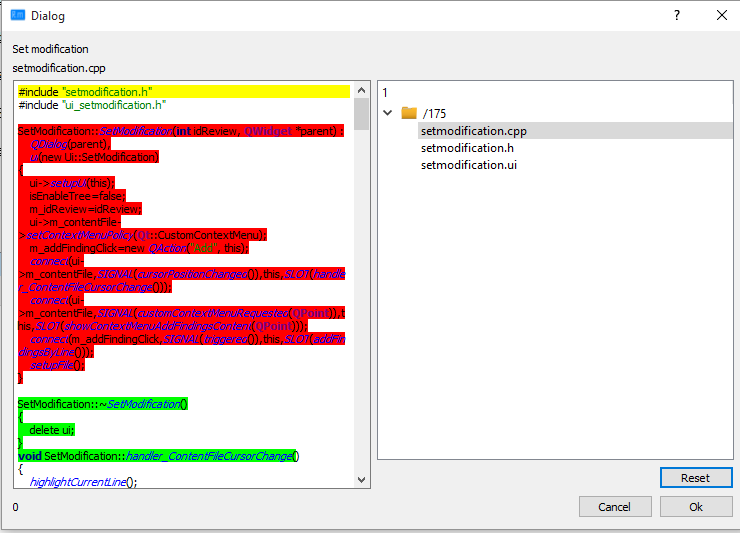
Liniile colorate cu roșu sunt cele din baza de date, iar liniile colorate cu verde sunt liniile adăugate în sesiunea curentă.

Fig. 19 Adăugarea modificărilor în baza de date

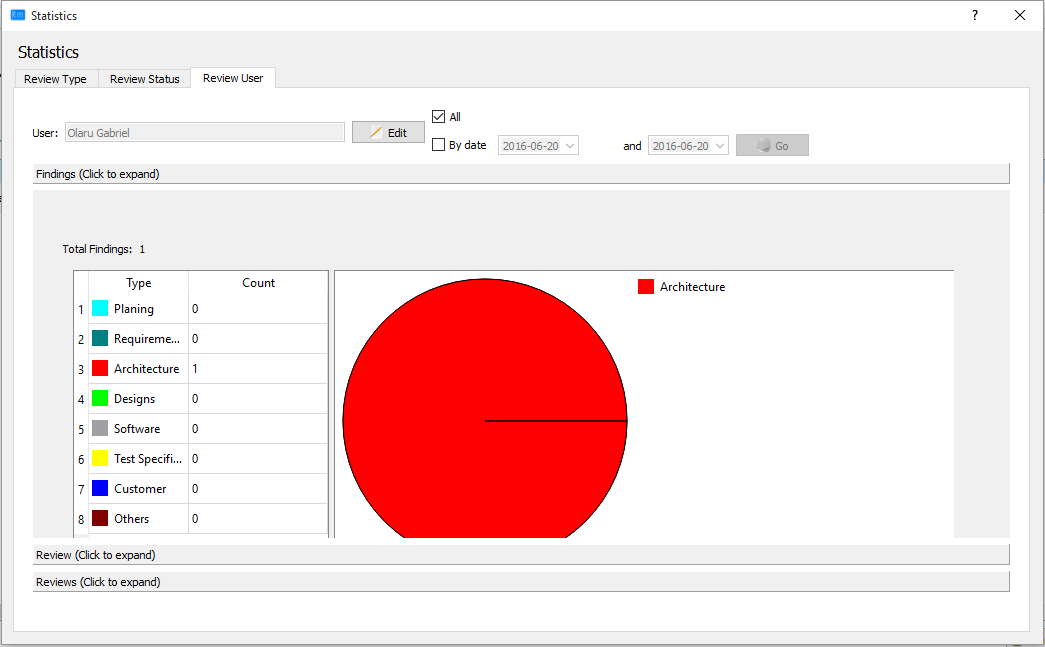


Fig. 20 Statistici

## Statistici

Aici (Fig. 20) utilizatorul poate să-și țină o evidența la câte review-uri a participat în funcție de:

* Tip
* Status

Totodată poate să vadă și numarul de finding-uri pentru fiecare tip de review.

## QuickMenu

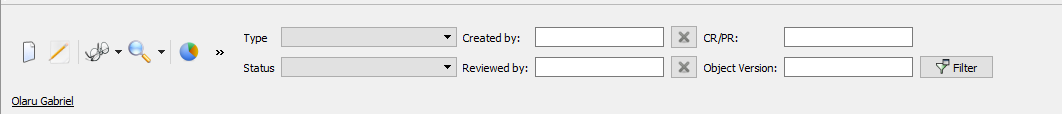


Fig. 21 Meniu rapid

Acesta oferă user-ului posibilitatea de a accesa unele funcții mai rapid dar totodată poate să-și construiască un query care să-i filtreze review-urile afișate în tabelul din view-ul principal.

# Configurare și cerințe

## Cerințe

* Windows 7/8/8. 1/10
* Microsoft Visual C++ 2013 Redistributable

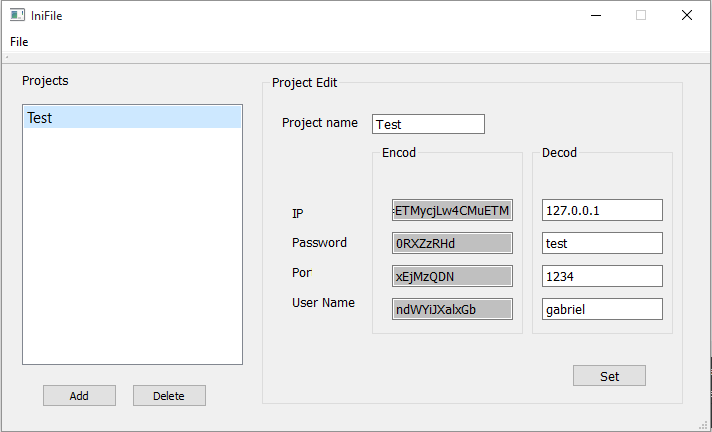


Fig. 22 Programul de configurare a fișierului cu extensia ini

Pentru a configura aplicația la baza ta de date, trebuie să editezi fișiserul ini din folderul unde se află aceasta. Editarea se realizează cu programul *IniFile.exe* care condifică datele necesare.

Au fost folosite două librării, una pentru a codifica stringu-uri și cealaltă pentru decodifica, acestea fiind incărcate dinamic atât în acest program cât și în clientul *Review Manager*.

# Concluzii

Consider că Review Manager și-a indeplinit cu succes obiectivele propuse și anume, crearea unei aplicații în care utilizatorul are ocazie să-și testeze și dezvolte aptitudinile de programare. Acest lucru este posibil prin faptul că el are ocazia să vadă codul altor persoane din care ar putea învăța lucruri noi sau să-și vadă propriile greșeli.

Pentru realizarea acestui proiect, a trebuit să sincronizez perfect comunicarea dintre client și server, deoarece unele fișiere pot avea dimensiuni mari în timpul uploadării sau descărcării . Thread-urile din client, pentru comunicarea cu baza de date dar și pentru update-ul view-ului, au fost un mic impediment, peste care am reușit să trec cu succes. Ele facilitând buna funcționare a aplicației.

Crearea întregii arhitecturi a fost o provocare deoarece sunt multe module de îmbinat dar acest lucru m-a ajutat să-mi consolidez cunoștințele despre toate tehnologiile folosite și să învăț altele noi.

Pe viitor aplicația ar putea fi îmbunătățită prin următoarele aspecte : posibilitatea de a integra *Git-ul* [[7]](#footnote-7)și *Jira*[[8]](#footnote-8). Astfel se va putea obține o mai bună versionare a fișierelor dar și un management mai bun al task-urilor. De asemenea ar putea fi creat un wrapper [[9]](#footnote-9)pentru a muta aplicația pe web.

# Bibliografie

|  |  |
| --- | --- |
| [1] | S. Juba, A. Vannahme și A. Volkov, Learning PostgreSQL, November 2015. |
| [2] | E. Gamma, R. Heml și R. Johnson, Elements of Reusable Object-Oriented Software, KevinZhang. |
| [3] | P. D. LUCANU, „Introduction to OOP using C++,” [Interactiv]. Disponibil la: https://sites.google.com/site/fiicoursepoo/curriculum/. |
| [4] | „Multithreading in C++,” [Interactiv]. Disponibil la: https://www.tutorialcup.com/cplusplus/multithreading.htm. |
| [5] | R. Hanmer, Pattern-Oriented Software Architecture For Dummies. |
| [6] | „Documentație oficilă QT,” [Interactiv]. Disponibil la: http://doc.qt.io/qt-5/. |
| [7] | „Design Patterns Explained Simply,” [Interactiv].  Disponibil la: https://sourcemaking.com/design-patterns-ebook?utm\_expid=19254892.1A9Al5XlTO\_jQBAtz3tw.1&utm\_referrer=  https%3A%2F%2Fsites.google.com%2Fsite%2Ffiicoursepoo%2Fcurriculum. |

1. https://ro. wikipedia. org/wiki/Calitate [↑](#footnote-ref-1)
2. http://www. managementconcept. ro/calitate. php [↑](#footnote-ref-2)
3. https://ro.wikipedia.org/wiki/Microsoft\_Windows [↑](#footnote-ref-3)
4. Reviewer= user care trebuie să gasească finding-uri [↑](#footnote-ref-4)
5. Moderator=user-ul care moderează rewier-ul [↑](#footnote-ref-5)
6. Minute Keeper= user care ține evidența timpului [↑](#footnote-ref-6)
7. Git=program de versionare a fișiererol [↑](#footnote-ref-7)
8. Jira=program folosit de echipele care folosesc agile [↑](#footnote-ref-8)
9. Wrapper=calsă care incapsulează funcționalitățile altei clase [↑](#footnote-ref-9)