```
/*
 * File:   MyWindow.cpp
 * Author: io447
 *
 * Created on 21 September 2015, 12:36 PM
 */
#include <QFileDialog>
#include <QApplication>
#include <QMessageBox>
#include <iostream>
#include <string>
#include <QDebug>
#include <boost/lexical_cast.hpp>
#include <QStandardItemModel>

#include "MyWindow.h"
#include "FileArchiver.h"
#include "FileRecord.h"
#include "FileLib.h"

MyWindow::MyWindow() {
    widget.setupUi(this);

    widget.saveCurrentBttn->setEnabled(false);
    widget.retrieveVersionBttn->setEnabled(false);
    widget.setReferenceBttn->setEnabled(false);
    widget.showCommentBttn->setEnabled(false);

    //connect SelectFile() with selectFileBttn
    connect(widget.selectFileBttn, SIGNAL(clicked()), this, SLOT(SelectFile()));
//connect saveCurrentBttn with SaveCurrent()
connect(widget.saveCurrentBttn, SIGNAL(clicked()), this, SLOT(SaveCurrent()));
//connect RetrieveVersion() with retrieveVersionBttn which will open RetrieveForm
    connect(widget.retrieveVersionBttn, SIGNAL(clicked()), this, SLOT(RetrieveVersion()));

//connect setReferenceBttn with SetReferenceVersion() to delete unnecessary file versions
connect(widget.setReferenceBttn, SIGNAL(clicked()), this, SLOT(SetReferenceVersion()));
//connect ShowComment() with showCommentBttn
connect(widget.showCommentBttn, SIGNAL(clicked()), this, SLOT(ShowComment()));
}

MyWindow::~MyWindow() {
}

//file selection
void MyWindow::SelectFile() {
    //declare new select file dialog
    QFileDialog dialog(this);
    //set mode to existing file
    dialog.setFileMode(QFileDialog::ExistingFile);
    //set view mode to detail
    dialog.setViewMode(QFileDialog::Detail);
```

```
    QStringList fileNames;

    if (dialog.exec() == QDialog::Rejected)
    {
        return;
    }

    fileNames = dialog.selectedFiles();

    //Display name of file as chosen by user
    if(!fileNames.isEmpty())
    {
        fileName = fileNames[0];
        widget.selectFilePath->setText(fileName);
    }

    //convert to std string
    std::string stdFileName;
    stdFileName = fileName.toStdString();

    //for now not catching exception bad_alloc
    FilePtr currentPath = new FileArchiver;

    //If a record already exists
    if(!currentPath->Exists(stdFileName))
    {
        CreateFirstVersion(stdFileName);
    }

    RetrieveVersionDataForFile();
}

void MyWindow::SaveCurrent()
{
    FileRecord fileRec(fileName.toStdString());

    if(fileRec.IsChanged())
    {
        AddNewVersion(fileName.toStdString());
        RetrieveVersionDataForFile();
    }
    else
    {
        QMessageBox msgBox;

        msgBox.setWindowTitle("No save required.");
        msgBox.setText("No modifications since last version. No save necessary.");
        msgBox.setStandardButtons(QMessageBox::Ok);
        msgBox.exec();
    }
}

void MyWindow::ShowComment()
```

```
{
        QModelIndexList indexes = widget.tableView->selectionModel()->selectedRows();

        if(!(indexes.size() > 0))
        {
            QMessageBox msgBox(QMessageBox::Information, "Error",
                    "Please select the file version", QMessageBox::Ok, 0);

            msgBox.exec();
            return;

        }

        if(indexes.size() > 0)
        {
            //get version number of selected record
            QVariant data = indexes[0].data(0);
            //retrieve version

            VersionRecord selectedVersion(fileName.toStdString(), data.toInt());
    if(selectedVersion.IsValid())
            {

QMessageBox msgBox(QMessageBox::Information, "Comment for selected version",
QString(selectedVersion.GetComment().c_str()), QMessageBox::Ok, 0);

                msgBox.exec();
            }
        }
}

void MyWindow::SetReferenceVersion()
{
        QModelIndexList indexes = widget.tableView->selectionModel()->selectedRows();

        if(!(indexes.size() > 0))
        {
            QMessageBox msgBox(QMessageBox::Information, "Error",
                    "Please select the file version", QMessageBox::Ok, 0);

            msgBox.exec();
            return;

        }


        if(indexes.size() > 0)
        {
            QMessageBox msgBox(QMessageBox::Question, "Set reference version",
                    "Are you sure you want to purge these records?", QMessageBox::Yes |
QMessageBox::No, 0);
            msgBox.setDefaultButton(QMessageBox::No);
```

```
            if(msgBox.exec() == QMessageBox::Yes)
            {
                FileRecord fileRec(fileName.toStdString());
                const QAbstractItemModel *myModel = indexes[0].model();

                for(int i = myModel->data(myModel->index(0, 0)).toInt(); i <
myModel->data(myModel->index(indexes[0].row(), 0)).toInt(); ++i)
                {
                    fileRec.GetVersion(i).PurgeVersion();
                }

                // Repopulate the table.
                RetrieveVersionDataForFile();
            }
        }
}

void MyWindow::CreateFirstVersion(std::string fileName)
{
    getCommentWindow = new GetCommentForm();
    QString comm;

    if(getCommentWindow->exec() == QDialog::Rejected)
    {
        return;
    }

    comm = getCommentWindow->GetComment();

    std::string commentStd = comm.toStdString();
    FileRecord fileRec;

    fileRec.CreateFile(fileName, commentStd);
}

void MyWindow::AddNewVersion(std::string fileName)
{
    getCommentWindow = new GetCommentForm();
    QString comm;

    if(getCommentWindow->exec() == QDialog::Rejected)
    {

        return;
    }

    comm = getCommentWindow->GetComment();

    std::string commentStd = comm.toStdString();
    FileRecord fileRec(fileName);

    comm = getCommentWindow->GetComment();
    fileRec.AddNewVersion(fileName, commentStd);
```

```
}

void MyWindow::RetrieveVersionDataForFile()
{
    FileRecord fileRec(fileName.toStdString());

    if(fileRec.GetNumberOfVersions() == 0)
    {
        return;
    }

    QStandardItemModel *myModel = new QStandardItemModel(fileRec.GetNumberOfVersions(), 3,
this);
    myModel->clear();

    myModel->setHorizontalHeaderItem(0, new QStandardItem(QString("Version #")));
    myModel->setHorizontalHeaderItem(1, new QStandardItem(QString("Date")));
    myModel->setHorizontalHeaderItem(2, new QStandardItem(QString("Size")));

    vector<VersionRecord> versionRecs = fileRec.GetAllVersions();
    unsigned int currentRow = 0;

    for(vector<VersionRecord>::iterator it = versionRecs.begin(); it != versionRecs.end(); ++it)
    {
        myModel->setItem(currentRow, 0, new
QStandardItem(QString(boost::lexical_cast<string>(it->GetVersionNumber()).c_str())));
        myModel->setItem(currentRow, 1, new
QStandardItem(QString(FileLib::GetFormattedModificationDate(fileRec.GetFilename()).c_str())));
        myModel->setItem(currentRow, 2, new
QStandardItem(QString(boost::lexical_cast<string>(it->GetSize()).c_str())));

        ++currentRow;
    }

    widget.tableView->setModel(myModel);
    widget.tableView->resizeColumnsToContents();
    widget.tableView->setEditTriggers(QAbstractItemView::NoEditTriggers);
    widget.tableView->setSelectionBehavior(QAbstractItemView::SelectRows);
    widget.tableView->setSelectionMode(QAbstractItemView::SingleSelection);

widget.saveCurrentBttn->setEnabled(true);
    widget.retrieveVersionBttn->setEnabled(true);
    widget.showCommentBttn->setEnabled(true);
    widget.setReferenceBttn->setEnabled(true);

    widget.tableView->show();
}

void MyWindow::RetrieveVersion()
{
    QString directory;
    QString outFilename;
```

```
    QModelIndexList indexes = widget.tableView->selectionModel()->selectedRows();

    if(!(indexes.size() > 0))
    {
        QMessageBox msgBox(QMessageBox::Information, "Error",
                    "Please select the version to be retrieved", QMessageBox::Ok, 0);

        msgBox.exec();
        return;

    }

    retrieveWindow = new RetrieveForm;
//execute RetrieveForm and details of where retrieved file will be placed

    if(retrieveWindow->exec() == QDialog::Accepted)
    {
        directory = retrieveWindow->GetDirectory();
        outFilename = retrieveWindow->GetOutputFilename();
    }

//convert data from RetrieveForm to full file output path
std::string fullOutputPath;

fullOutputPath += directory.toStdString();
fullOutputPath += "/";
fullOutputPath += outFilename.toStdString();

//retrieve version
        if(indexes.size() > 0)
        {
            VersionRecord selectedVersion(fileName.toStdString(), indexes[0].row() + 1);
            selectedVersion.GetFileData(fullOutputPath);
        }
}
```

```
/*
 * File:   ProjectConstants.h
 * Author: philipedwards
 *
 * Created on 19 September 2015, 11:01 PM
 */

#ifndef PROJECTCONSTANTS_H
#definePROJECTCONSTANTS_H

#include <string>
#include <fstream>
#include <iostream>

#define DEBUG_LOGGING
#define DEBUG_LOG_TO_FILE

#define NIXON_SNAKE

const int MURMUR_SEED_1 = 23455;
const int MURMUR_SEED_2 = 2086235969;
const int FILENAME_LENGTH(767);

const int BLOCK_SIZE = 4000;

void logToFile(std::string message);
void log(std::string message);

const std::string COMPRESSION_WORK_PATH = "./temp/";

#endif/* PROJECTCONSTANTS_H */
```

```
/*
 * File:   RetrieveForm.h
 * Author: io447
 *
 * Created on 21 September 2015, 4:45 PM
 */


#ifndef _RETRIEVEFORM_H
#define_RETRIEVEFORM_H

#include "ui_RetrieveForm.h"

class RetrieveForm : public QDialog {
    Q_OBJECT
public:
    RetrieveForm();
    virtual ~RetrieveForm();
public slots:
    void SelectDirectory();
    void SetFileName();
QString GetDirectory();
QString GetOutputFilename();
private:
    Ui::RetrieveForm widget;
QString directoryPath;
QString outputFilename;

};

#endif/* _RETRIEVEFORM_H */
```

```
/*
 * File:   RetrieveForm.cpp
 * Author: io447
 *
 * Created on 21 September 2015, 4:45 PM
 */

#include <QFileDialog>
#include <QApplication>
#include <iostream>
#include <string>

#include "RetrieveForm.h"

RetrieveForm::RetrieveForm() {
    widget.setupUi(this);

//select directory button
    connect(widget.pushButtonDirectory, SIGNAL(clicked()), this, SLOT(SelectDirectory()));

   //ok | cancel buttons
    connect(widget.buttonBox, SIGNAL(accepted()), this, SLOT(SetFileName()));

    widget.buttonBox = new QDialogButtonBox(QDialogButtonBox::Ok
                                    | QDialogButtonBox::Cancel);

    connect(widget.buttonBox, SIGNAL(accepted()), this, SLOT(accept()));

    connect(widget.buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
}

RetrieveForm::~RetrieveForm() {
}

void RetrieveForm::SelectDirectory()
{
    //declare new select file dialog
    QFileDialog dialogDir(this);
    //set mode to existing file
    dialogDir.setFileMode(QFileDialog::Directory);
    //set view mode to detail
    dialogDir.setViewMode(QFileDialog::Detail);

    QStringList dirNames;
    //QString fileName;
    if (dialogDir.exec())
    {
        //dialog.selectFile(fileName);
         dirNames = dialogDir.selectedFiles();
    }

    //Display name of directory as chosen by user
    QString dirName;
```

```
    if(!dirNames.isEmpty())
    {
        dirName = dirNames[0];
        widget.lineEditDirectory->setText(dirName);
    }

directoryPath = dirName;

}
void RetrieveForm::SetFileName(){

    outputFilename = widget.lineEditFilename->text();

}

QString RetrieveForm::GetDirectory()
{
return directoryPath;
}

QString RetrieveForm::GetOutputFilename()
{
return outputFilename;
}
```

```
/*
 * File:   TestUtilities.h
 * Author: philipedwards
 *
 * Created on 21 September 2015, 8:41 AM
 */

#ifndef TESTUTILITIES_H
#defineTESTUTILITIES_H

#include <string>
#include <mysql_connection.h>
#include <mysql_driver.h>

void createFile(unsigned int seed, std::string filename, int length);
void appendFile(int seed, std::string filename, int length);

void DropTables();
void CreateTables();

void ExecuteSQLFile(std::string path);

bool ExecuteUpdateStatement(sql::Connection* dbcon, std::string sqlstatement);

void CommitFileWithOneVersion(std::string path);
void CommitFileWithTwoVersions();

bool GenerateFilesAndCommitVersionsAndVerifyRetrieval(std::string path, unsigned int size,
unsigned int numVersions);

void RunTestCommitFileOneVersion();
void RunTestCommitFileOneVersionRetrieve();
void RunTestCommitFileWithTwoVersionsRetrieveBoth();

void RunTestPurge();

#endif/* TESTUTILITIES_H */
```

```
#include <string>
#include "DBConnector.h"
#include "TestUtilities.h"
#include "ProjectConstants.h"
#include <fstream>
#include "FileRecord.h"
#include "MurmurHash3.h"

#include <cppconn/driver.h>
#include <cppconn/exception.h>
#include <cppconn/resultset.h>
#include <cppconn/statement.h>
#include "boost/lexical_cast.hpp"

#include <cstdlib>

using namespace std;

void createFile(unsigned int seed, string filename, int length)
{
srand(seed);

ofstream outFile(filename.c_str());

if(outFile.is_open())
{
while(length > 0)
{
length--;
char num = rand() % 256;
outFile.put(num);
}
}
outFile.close();
}

void appendFile(int seed, string filename, int length)
{
srand(seed);

ofstream outFile(filename.c_str(), ios::app);

if(outFile.is_open())
{
while(length > 0)
{
length--;
char num = rand() % 256;
outFile.put(num);
}
}
}
```

```
void DropTables()
{
ExecuteSQLFile("sql/DropTables.sql");
}
void CreateTables()
{
ExecuteSQLFile("sql/CreateTables.sql");
}

void ExecuteSQLFile(string path)
{
sql::Connection* dbcon = DBConnector::GetConnection();
dbcon->setSchema("redsquare");

if(dbcon == NULL)
{
log("Failed to open database connection");
return;
}

ifstream inFile(path.c_str());
if(!inFile.is_open())
{
log("Failed to open sql file");
delete dbcon;
dbcon = NULL;
return;
}

while(!inFile.eof())
{
string statementstring;
getline(inFile,statementstring);
if(statementstring.empty() == false)
{
log(statementstring);
ExecuteUpdateStatement(dbcon, statementstring);
}
}

//inFile.close();
//delete dbcon;
dbcon = NULL;
}

bool ExecuteUpdateStatement(sql::Connection* dbcon, std::string sqlstatement)
{
sql::Statement* stmt = dbcon->createStatement();
bool bSuccess = false;
try
{
bSuccess = stmt->executeUpdate(sqlstatement);
dbcon->commit();
```

```
}
catch (sql::SQLException &e)
{
cout << "ERROR: " << endl;
cout << e.what() << endl;
cout << e.getErrorCode() << endl;
cout << e.getSQLState() << endl;
}
delete stmt;
return bSuccess;
}

void CommitFileWithOneVersion(string path)
{
FileRecord newFile;
bool bSuccess = newFile.CreateFile(path, "initial version");

string logMessage = "The result was: ";
if(bSuccess)
{
logMessage += "Success!";
}
else
{
logMessage += "The nixon snake has failed";
}

log(logMessage);
}

void CommitFileWithTwoVersions()
{
FileRecord newFile;
string path = "testData/testFile.dat";
createFile(25, path, 200000);
bool bSuccess = newFile.CreateFile(path, "initial version");

//check that we can't add the same version
if(bSuccess)
{
log("Trying to add same version");
bSuccess = !newFile.AddNewVersion(path, "same version");
if(bSuccess == false)
{
log("ERROR: was able to add the same version");
}
}

//check that adding a new version works
if(bSuccess)
{
log("Trying to commit a new version");
appendFile(25, path, 200);
```

```
bSuccess = newFile.AddNewVersion(path, "Version 2");
}

string logMessage = "The result was: ";
if(bSuccess)
{
logMessage += "Success!";
}
else
{
logMessage += "The nixon snake has failed";
}

log(logMessage);
}

bool GenerateFilesAndCommitVersionsAndVerifyRetrieval(std::string path, unsigned int size,
unsigned int numVersions)
{
FileRecord myRecord;
bool bSuccess = true;

for(unsigned int i = 0; i < numVersions; i++)
{
if(bSuccess == false)
{
break;
}
string currentpath = path + "." + boost::lexical_cast<string>(i);
createFile(i * 200, currentpath, size);
if(i == 0)
{
bSuccess = myRecord.CreateFile(currentpath, "initial version");
if(bSuccess == false)
{
log("ERROR: Failed to create new file: " + currentpath);
}
}
else
{
log("Trying to add new version");
if(myRecord.IsValid())
{
bSuccess = myRecord.AddNewVersion(currentpath, "Version: " + boost::lexical_cast<string>(i) );
if(bSuccess == false)
{
log("ERROR: Failed to create new version: " + currentpath);
}
}
}


}
```

```
if(bSuccess)
{
string originalName = path + ".0";
for(unsigned int i = 0; i < numVersions; i++)
{
string currentpath = path + "." + boost::lexical_cast<string>(i) + ".ret";
VersionRecord currentRecord(originalName, i + 1);
unsigned int retrievedHash = 0;

if(currentRecord.IsValid())
{
currentRecord.GetFileData(currentpath);
MurmurHash3_x86_32_FromFile(currentpath, MURMUR_SEED_1, &retrievedHash);
if(retrievedHash != currentRecord.GetHash())
{
log("HASHES DID NOT MATCH! " + currentpath);
bSuccess = false;
break;
}
else
{
log("Hashes matched. Retrieval worked correctly");
}
}
else
{
bSuccess = false;
break;
}
}
}

return bSuccess;
}

void RunTestCommitFileOneVersion()
{
//DropTables();
//CreateTables();
CommitFileWithOneVersion("MurmurHash3.cpp");
}

void RunTestCommitFileWithTwoVersionsRetrieveBoth()
{
//DropTables();
//CreateTables();

string path = "testData/testFile.dat";
string original = "testData/testFile.dat.orig";
string retrievedLatest = "testData/retrievedLatest.dat";
string retrievedOriginal = "testData/retrievedOriginal.dat";

CommitFileWithTwoVersions();
```

```
createFile(25, path + ".orig", 200000);

VersionRecord originalVersion(path, 1);
originalVersion.GetFileData(retrievedOriginal);

VersionRecord latestVersion(path, 2);
latestVersion.GetFileData(retrievedLatest);



unsigned int hash1 = 0;
unsigned int hash2 = 0;
unsigned int hash3 = 0;
unsigned int hash4 = 0;
MurmurHash3_x86_32_FromFile(path, MURMUR_SEED_1, &hash1);
MurmurHash3_x86_32_FromFile(original, MURMUR_SEED_1, &hash2);
MurmurHash3_x86_32_FromFile(retrievedLatest, MURMUR_SEED_1, &hash3);
MurmurHash3_x86_32_FromFile(retrievedOriginal, MURMUR_SEED_1, &hash4);

if(hash1 == hash3 && hash2 == hash4)
{
log("Hashes match. File successfully retrieved");
}
else
{
log("Error. Hashes do not match. File not retrieved correctly");
}
}

void RunTestCommitFileOneVersionRetrieve()
{
//DropTables();
//CreateTables();

/*
string fileinpath = "MurmurHash3.cpp";
string fileoutpath = "MurmurHash3.cpp.ret";
/*/
string fileinpath = "testData/nixon.jpg";
string fileoutpath = "testData/nixonout.jpg";
//*/

CommitFileWithOneVersion(fileinpath);
VersionRecord newRec(fileinpath, 1);
newRec.GetFileData(fileoutpath);



unsigned int hash1 = 0;
unsigned int hash2 = 0;
MurmurHash3_x86_32_FromFile(fileinpath, MURMUR_SEED_1, &hash1);
MurmurHash3_x86_32_FromFile(fileoutpath, MURMUR_SEED_1, &hash2);

if(hash1 == hash2)
```

```
{
log("Hashes match. File successfully retrieved");
}
else
{
log("Error. Hashes do not match. File not retrieved correctly");
}
}

void RunTestPurge()
{
    string path = "testData/testFile.dat";
    FileRecord existingFile(path);

    if (!existingFile.IsValid())
    {
        log("FileRecord is not valid.");
        return;
    }

    log("Purging file.");
    existingFile.PurgeOldVersions(0);

    return;
}
```

```
/*
 * File:   main.cpp
 * Author: philipedwards
 *
 * Created on 16 September 2015, 11:53 AM
 */

#include <QApplication>
#include "ProjectConstants.h"
#include "FileRecord.h"
#include "FileLib.h"

#include <string>

#include "TestUtilities.h"
#include "FileArchiver.h"
#include "boost/lexical_cast.hpp"

//main window header
#include "MyWindow.h"

using namespace std;

int main(int argc, char *argv[]) {
// initialize resources, if needed
// Q_INIT_RESOURCE(resfile);


QApplication app(argc, argv);

        FileLib::SetupWorkingDirectories();
//DropTables();
//CreateTables();
//GenerateFilesAndCommitVersionsAndVerifyRetrieval("testData/testDataagain.dat", 20000,100);

        /*


RunTestCommitFileOneVersionRetrieve();
//RunTestCommitFileWithTwoVersionsRetrieveBoth();

FileArchiver test;
vector<FileRecord> files = test.GetFiles();

for(unsigned int i = 0; i < files.size(); i++)
{
log("File found in database: " + files[i].GetFilename() + " " +
boost::lexical_cast<string>(files[i].GetCurrentVersionNumber()));
}
        */
    //RunTestPurge();

// create and show your widgets here
```

```
        MyWindow win;
        win.show();
return app.exec();

//return 0;
}
```

```
#include "ProjectConstants.h"

void logToFile(std::string message)
{
#ifdef DEBUG_LOG_TO_FILE
std::ofstream outFile("debuglog.log", std::ios::app);
if(outFile.is_open())
{
outFile << message << std::endl;
outFile.close();
}
#endif // DEBUG_LOG_TO_FILE
}

void log(std::string message)
{
#ifdef DEBUG_LOGGING
std::cout << message << std::endl;
logToFile(message);
#endif //DEBUG_LOGGING
}
```

```
#ifndef VERSIONRECORD_H
#define VERSIONRECORD_H

#include <string>
class VersionRecord
{
public:

// Constructor
VersionRecord();

// Constructor
VersionRecord(std::string filename, unsigned int versionNumber);

// Destructor
~VersionRecord();

// Returns the id of a version
unsigned int GetVersionId();

// Returns the version number
unsigned int GetVersionNumber();

// Returns the size
unsigned int GetSize();

// Returns the hash of the
unsigned int GetHash();

        // Returns the modification time.
        unsigned int GetModificationTime();

        // Returns a formatted string of the modification time/date.
        std::string GetFormattedModificationTime();

//public members for transfer of record to/from persistent storage - the function signatures
will depend on the persistance mechanism that is chosen
bool CreateVersion(std::string keyFilename, std::string pathFilename, unsigned int
currentVersion, unsigned int newHash, std::string newComment);

// Returns the comment on the version
std::string GetComment();

// Returns true if the data in Version is usable
bool IsValid();

bool GetFileData(std::string fileOutPath);

void PurgeVersion();

unsigned int RetrieveSizeFromDisk(std::string path);

protected:
```

```
//retrieves the record information from the database
bool RetrieveVersionRecordFromDB(std::string inFilename, unsigned int versionNumber);
//updates the FileRecord in the database;
bool UpdateRecordInDB();
void Init();

bool InsertVersionIntoDB(std::string keyFileName);

bool InsertBlocks(std::string zipPath);

//The version identifier - generated primary key
unsigned int VersionID;

std::string Filename;

//The version number of this version
unsigned int VersionNumber;
//the length of this version in bytes
unsigned int Size;

unsigned int Time;
unsigned int FileModificationTime;
std::string Comment;

//the has of the entire version of the file
unsigned int Hash;
//database connection
sql::Connection* dbcon;

bool bIsValid;

private:

};

#endif
```

```
#include <iostream>
#include <fstream>

#include "mysql_connection.h"
#include "mysql_driver.h"

#include <cppconn/driver.h>
#include <cppconn/exception.h>
#include <cppconn/resultset.h>
#include <cppconn/statement.h>
#include <cppconn/prepared_statement.h>



#include "VersionRecord.h"
#include "ProjectConstants.h"
#include "DBConnector.h"
#include "MurmurHash3.h"
#include "boost/lexical_cast.hpp"

#include <streambuf>
#include <istream>
#include <ctime>

#include "CompressUtils.h"
#include "FileLib.h"

using namespace std;

//this struct is copied from this forum post:
// http://stackoverflow.com/a/7782037
struct membuf : std::streambuf
{
membuf(char* begin, char* end)
{
this->setg(begin, begin, end);
}
};

// Constructor, prepares for use
VersionRecord::VersionRecord()
{
Init();
}

// Destructor, sets poit for connection to null
VersionRecord::~VersionRecord()
{
dbcon = NULL;
}

// Constructor, tries to retrieve a record from the db
VersionRecord::VersionRecord(std::string filename, unsigned int versionNumber)
```

```
{
Init();
RetrieveVersionRecordFromDB(filename, versionNumber);
}

// Sets class members to initial values
void VersionRecord::Init()
{
dbcon = DBConnector::GetConnection();
VersionID = 0;
VersionNumber = 0;
Size = 0;
Time = 0;
FileModificationTime = 0;
Hash = 0;
bIsValid = false;
}

// Tries to retrieve a version record from the database
bool VersionRecord::RetrieveVersionRecordFromDB(std::string inFilename, unsigned int
versionNumber)
{
// Prepare for failure
bIsValid = false;

// Setup statement
sql::PreparedStatement *pstmt = dbcon->prepareStatement("select * from Version where filename=?
and version=?");
sql::ResultSet *rs;

// Try to run Query
try
{
pstmt->setString(1,inFilename);
pstmt->setInt(2,versionNumber);
rs = pstmt->executeQuery();

// Output Results
while(rs->next())
{
//count = rs->getUInt(1);
VersionID = rs->getUInt("id");
Filename = rs->getString("filename");
VersionNumber = rs->getUInt("version");
//TODO: change to getInt64?
Size = rs->getUInt("size");
Time = rs->getUInt("time");
FileModificationTime = rs->getUInt("filemodtime");
Comment = rs->getString("comment");
Hash = rs->getUInt("hash");
bIsValid = true;
}
```

```
}
catch (sql::SQLException &e)
{
log("ERROR: ");
log(e.what());
log(e.getErrorCode());
log(e.getSQLState());
}

// Clean up
delete rs;
delete pstmt;
rs = NULL;
pstmt = NULL;

// Return result
return bIsValid;
}

// Updates a version record in the db
bool VersionRecord::UpdateRecordInDB()
{
sql::Statement *stmt = dbcon->createStatement();
bool bSuccess = true;

if(stmt == NULL)
{
//failed to get a connection to the database
bSuccess = false;
}

if(IsValid() == false)
{
bSuccess = false;
}

try
{
//create file record
if(bSuccess)
{
//beginning of statement
string sqlstatement = "update Version set ";
//curhash
sqlstatement += "filename = \"" + Filename + "\", ";
sqlstatement += "size = " + boost::lexical_cast<string>(Size) + ", ";
sqlstatement += "time = " + boost::lexical_cast<string>(Time) + ", ";
sqlstatement += "filemodtime = " + boost::lexical_cast<string>(FileModificationTime) + ", ";
sqlstatement += "comment = \"" + Comment + "\", ";
sqlstatement += "version = " + boost::lexical_cast<string>(VersionNumber) + ", ";
sqlstatement += "hash = " + boost::lexical_cast<string>(Hash);
//end of statement
```

```
sqlstatement += " where id = " + boost::lexical_cast<string>(VersionID) + ";";

log(sqlstatement);

bSuccess = stmt->executeUpdate(sqlstatement);
}
}
catch (sql::SQLException &e)
{
log("ERROR: ");
log(e.what());
log(e.getErrorCode());
log(e.getSQLState());
}

delete stmt;
return bSuccess;
}

unsigned int VersionRecord::GetVersionId()
{
return VersionID;
}

unsigned int VersionRecord::GetVersionNumber()
{
return VersionNumber;
}

unsigned int VersionRecord::GetSize()
{
return Size;
}

unsigned int VersionRecord::GetHash()
{
return Hash;
}

unsigned int VersionRecord::GetModificationTime()
{
        return FileModificationTime;
}



string VersionRecord::GetFormattedModificationTime()
{
    char buffer[80];
    time_t fileTime = FileModificationTime;

    strftime(buffer, 80, "%F %T", localtime(&fileTime));
    return string(buffer);
}
```

```
bool VersionRecord::InsertVersionIntoDB(string keyFilename)
{
	// Prepare for sql statements
	bool bSuccess = false;
	const char* insertVersion = "insert into Version(filename, version, hash, filemodtime,
size, time, comment) values (?, ?, ?, ?, ?, ?, ?)";
	const char* selectVersion = "select id from Version where hash=?";
	sql::PreparedStatement *pstmt = NULL;
	sql::ResultSet *rs = NULL;

	// Try to insert record and get the id
	try
	{
		pstmt = dbcon->prepareStatement(insertVersion);
		pstmt->setString(1, keyFilename);
		pstmt->setInt64(2, VersionNumber);
		pstmt->setInt64(3, Hash);
		pstmt->setInt64(4, FileModificationTime);
		pstmt->setInt64(5, Size);
		pstmt->setInt64(6, Time);
		pstmt->setString(7, Comment);

		bool bNewVersionMade = pstmt->executeUpdate();

		if (bNewVersionMade == false)
		{
			return false;
		}

		// Run Query
		pstmt = dbcon->prepareStatement(selectVersion);
		pstmt->setInt64(1, Hash);
		rs = pstmt->executeQuery();

		// Output Results
		while(rs->next())
		{
			VersionID = rs->getUInt(1);
		}
		bSuccess = true;

		if(bSuccess)
		{
			RetrieveVersionRecordFromDB(keyFilename, VersionNumber);
			if(!IsValid())
			{
				log("Failed to retrieve version record from database. This
version was not created correctly");
				bSuccess = false;
			}
		}
	}
```

```
        catch (sql::SQLException &e)
        {
                log("ERROR: ");
                log(e.what());
                log(e.getErrorCode());
                log(e.getSQLState());
                log("Failed to create version in table Version");
                bSuccess = false;
        }

        pstmt->close();
        pstmt = NULL;
        delete rs;
        rs = NULL;
        return bSuccess;
}

bool VersionRecord::CreateVersion(string keyFilename, string pathFilename, unsigned int
currentVersion, unsigned int newHash, string newComment)
{
bool bSuccess = true;

// Set all variables to instance of class
FileModificationTime = FileLib::GetModifiedDate(pathFilename);
Time = time(0);
Size = RetrieveSizeFromDisk(pathFilename);
Comment = newComment;
VersionNumber = currentVersion;
Hash = newHash;

// Create a new version
bSuccess = InsertVersionIntoDB(keyFilename);

if(bSuccess)
{
// Clean temp folder just incase
zipRemoveZip();

// Copy file to temp folder
zipCopyContents(pathFilename);

// Compress file in temp folder
zipCompress();

// Create Zip path
string zipPath = "./temp/data.gz";

// Create Blocks In Database
bSuccess = InsertBlocks(zipPath);
if(!bSuccess)
{
log("Failed to insert blocks for this version");
}
```

```
// Update
UpdateRecordInDB();

// Clean up temp folder
zipRemoveZip();
}
else
{
log("Failed to create a record in the database for this version");
}

return bSuccess;
}

bool VersionRecord::InsertBlocks(string zipPath)
{
bool bSuccess = true;

// Open File
ifstream ins(zipPath.c_str());
if (!ins.good())
{
log("Failed to open file. Cannot create version");
bSuccess = false;
}

sql::Statement *stmt = dbcon->createStatement();
if(bSuccess)
{
ins.seekg(0,ios::end);
int bytesRemaining = ins.tellg();
ins.seekg(0,ios::beg);

try
{
unsigned int versionIndex = 0;
char block[BLOCK_SIZE];
while (!ins.eof() && bytesRemaining > 0)
{
// Get Block
int blockSize = 0;
if(bytesRemaining > BLOCK_SIZE)
{
blockSize = BLOCK_SIZE;
}
else
{
blockSize = bytesRemaining;
}
bytesRemaining -= blockSize;
ins.read((char*)block, blockSize);
log("block size is " + boost::lexical_cast<string>(blockSize));
```

```
if (blockSize == 0)
{
break;
}

// Hash 1
unsigned int hash1 = 0;
MurmurHash3_x86_32(block, blockSize, MURMUR_SEED_1 , &hash1);

// Hash 2
unsigned int hash2 = 0;
MurmurHash3_x86_32(block, blockSize, MURMUR_SEED_2 , &hash2);

// Query DB Hash 1 in table Blocks
unsigned int blockId = 0;

// Run Query
sql::ResultSet *rs = stmt->executeQuery("select id from Block where hash1 = " +
boost::lexical_cast<string>(hash1));

// Output Results
while(rs->next())
{
blockId = rs->getUInt(1);
}

delete rs;

// If hash 1 already exists
if (blockId != 0)
{
log("Hash is not equal to zero");
// Query DB Hash 2 in table Blocks
unsigned int result;

// Run Query
sql::ResultSet *rs = stmt->executeQuery("select id from Block where hash2 = " +
boost::lexical_cast<string>(hash2) + " and id = " + boost::lexical_cast<string>(blockId));

// Output Results
while(rs->next())
{
result = rs->getUInt(1);
}

delete rs;

// If hash 2 matches the same id as
if (result != 0)
{
// Use existing block
bSuccess = stmt->executeUpdate("insert into VtoB(versionid, blockid, versionindex) values (" +
boost::lexical_cast<string>(this->VersionID) + ", " + boost::lexical_cast<string>(blockId) + ",
```

```
" + boost::lexical_cast<string>(versionIndex++) + ")");
}
else
{
// Create a new block
sql::PreparedStatement *pstmt = dbcon->prepareStatement("insert into Block(hash1, hash2, data)
values (?,?,?)");
pstmt->setUInt(1,hash1);
pstmt->setUInt(2,hash2);
membuf sbuf(block, block + blockSize);
istream in(&sbuf);
pstmt->setBlob(3, &in);
bSuccess = pstmt->executeUpdate();
delete pstmt;
bSuccess = stmt->executeUpdate("commit");

// Link block with VtoB
bSuccess = stmt->executeUpdate("insert into VtoB(versionid, blockid, versionindex) values (" +
boost::lexical_cast<string>(VersionID) + ", " + boost::lexical_cast<string>(blockId) + ", " +
boost::lexical_cast<string>(versionIndex++) + ")");
dbcon->commit();
}
}
else
{
log("Hash equals zero ");
// Create a new block
sql::PreparedStatement *pstmt = dbcon->prepareStatement("insert into Block(hash1, hash2, data)
values (?,?,?)");
pstmt->setUInt(1,hash1);
pstmt->setUInt(2,hash2);
membuf sbuf(block, block + blockSize);
istream in(&sbuf);
pstmt->setBlob(3, &in);
bSuccess = pstmt->executeUpdate();
delete pstmt;

// Run Query
dbcon->commit();
int i = 0;
bool bFound = false;
while (bFound == false && i < 100)
{
i++;
string sqlstatement = "select id from Block where hash1 = " + boost::lexical_cast<string>(hash1)
+ " and hash2 = " + boost::lexical_cast<string>(hash2);
log(sqlstatement);
sql::ResultSet *rs1 = stmt->executeQuery(sqlstatement);

if(rs1->next() == false)
{
log("Failed to find block that was just committed");
bSuccess = false;
```

```
}
else
{
bSuccess = true;
bFound = true;
blockId = rs1->getUInt(1);
}
delete rs1;
}

if(bSuccess)
{
log("blockId = " + boost::lexical_cast<string>(blockId));


// Link block with VtoB
string sqlstatement = "insert into VtoB(versionid, blockid, versionindex) values (" +
boost::lexical_cast<string>(VersionID) + ", " + boost::lexical_cast<string>(blockId) + ", " +
boost::lexical_cast<string>(versionIndex++) + ")";
log(sqlstatement);
bSuccess = stmt->executeUpdate(sqlstatement);
dbcon->commit();
}
}
}
}
catch (sql::SQLException &e)
{
log("ERROR: ");
log(e.what());
log(e.getErrorCode());
log(e.getSQLState());
bSuccess = false;
}
}

ins.close();

delete stmt;
stmt = NULL;

return bSuccess;
}


// Returns the size of a file passed to it
unsigned int VersionRecord::RetrieveSizeFromDisk(string path)
{
ifstream ins(path.c_str());

if (!ins.good())
{
log("Failed to open file. Cannot get file size.");
```

```
}

int fileSize = 0;
if(ins.good())
{
ins.seekg (0, ios::end);
fileSize = ins.tellg();
ins.seekg (0, ios::beg);
}

return fileSize;
}

std::string VersionRecord::GetComment()
{
return Comment;
}

bool VersionRecord::IsValid()
{
return bIsValid;
}

bool VersionRecord::GetFileData(std::string fileOutPath)
{
bool bSuccess = IsValid();
sql::Statement *stmt = dbcon->createStatement();

if(stmt == NULL)
{
//failed to get a connection to the database
bSuccess = false;
}

ofstream outFile("./temp/data.gz");

if(outFile.is_open() == false)
{
log("Unable to open file to write on disk");
bSuccess = false;
}

try
{
//create file record
if(bSuccess)
{
string sqlstatement = "select blockid from VtoB where versionid = " +
boost::lexical_cast<string>(VersionID) + " order by versionindex ASC";

log(sqlstatement);

sql::ResultSet *rs = stmt->executeQuery(sqlstatement);
```

```
int blocksRetrieved = 0;
while(rs->next() && bSuccess)
{
blocksRetrieved++;
log("retrieved block");

unsigned int blockid = rs->getUInt("blockid");

//for all block records, fetch block, write to disk
string blockretsql = "select data from Block where id = " +
boost::lexical_cast<string>(blockid);
sql::ResultSet *rs2 = stmt->executeQuery(blockretsql);

if(rs2->next())
{
istream* data = rs2->getBlob("data");
char outbuf[BLOCK_SIZE];
int blobsize = data->readsome(outbuf, BLOCK_SIZE);
outFile.write(outbuf, blobsize);
}
else
{
bSuccess = false;
}

delete rs2;
rs2 = NULL;
}

if(blocksRetrieved == 0)
{
bSuccess = false;
}

delete rs;
rs = NULL;
}



}
catch (sql::SQLException &e)
{
log("ERROR: ");
log(e.what());
log(e.getErrorCode());
log(e.getSQLState());
bSuccess = false;
}

outFile.close();

zipUncompressTo(fileOutPath);
```

```
delete stmt;
stmt = NULL;
return bSuccess;
}

void VersionRecord::PurgeVersion()
{
    // Catch invalid version
    if (!bIsValid)
        return;

    sql::Statement *stmt = dbcon->createStatement();

    try
    {
        // Delete records from VtoB
        stmt->executeUpdate("delete from VtoB where versionid = " +
boost::lexical_cast<string>(VersionID));

        // Clean data Blocks up
        // This will show which blocks need to be deleted "select id from Block where id not in
(select b.id from Block b join VtoB v on b.id = v.blockid);"
        stmt->executeUpdate("delete b from Block b left join VtoB v on v.blockid = b.id where
v.blockid is null");

        // Remove the version record
        stmt->executeUpdate("delete from Version where id = " +
boost::lexical_cast<string>(VersionID));

    }
    catch (sql::SQLException e)
    {
        log("ERROR: ");
        log(e.what());
        log(e.getErrorCode());
        log(e.getSQLState());
    }

    delete stmt;

    // Record no longer exists, it becomes invalid
    bIsValid = false;
}
```

```
/*
 * File:   backendtests.h
 * Author: philipedwards
 *
 * Created on 30/09/2015, 6:40:09 PM
 */

#ifndef BACKENDTESTS_H
#defineBACKENDTESTS_H

#include <cppunit/extensions/HelperMacros.h>

class backendtests : public CPPUNIT_NS::TestFixture {
    CPPUNIT_TEST_SUITE(backendtests);

    //CPPUNIT_TEST(testMethod);
    //CPPUNIT_TEST(testFailedMethod);
    CPPUNIT_TEST(hashFileTest);
    CPPUNIT_TEST(commitRetrieveTest);
    CPPUNIT_TEST(purgeTest);

    CPPUNIT_TEST_SUITE_END();

public:
    backendtests();
    virtual ~backendtests();
    void setUp();
    void tearDown();

private:
    //void testMethod();
    //void testFailedMethod();

    void hashFileTest();
    void commitRetrieveTest();
    void purgeTest();

    bool GenerateFilesAndCommitVersionsAndVerifyRetrieval(std::string path, unsigned int size,
unsigned int numVersions);
};

#endif/* BACKENDTESTS_H */
```

```
/*
 * File:   backendtests.cpp
  * Author: philipedwards
 *
 * Created on 30/09/2015, 6:40:09 PM
 */

#include "backendtests.h"
#include "MurmurHash3.h"
#include "ProjectConstants.h"
#include "TestUtilities.h"
#include "FileArchiver.h"
#include "FileRecord.h"
#include "VersionRecord.h"
#include "FileLib.h"
#include "boost/lexical_cast.hpp"

#include <string>
using namespace std;


CPPUNIT_TEST_SUITE_REGISTRATION(backendtests);

backendtests::backendtests()
{
}

backendtests::~backendtests()
{
}

void backendtests::setUp()
{
        FileLib::SetupWorkingDirectories();
DropTables();
CreateTables();
}

void backendtests::tearDown()
{
}

bool backendtests::GenerateFilesAndCommitVersionsAndVerifyRetrieval(std::string path, unsigned
int size, unsigned int numVersions)
{
FileRecord myRecord;
bool bSuccess = true;

for(unsigned int i = 0; i < numVersions; i++)
{
if(bSuccess == false)
{
break;
```

```
}
string currentpath = path + "." + boost::lexical_cast<string>(i);
createFile(i * 200, currentpath, size);
if(i == 0)
{
bSuccess = myRecord.CreateFile(currentpath, "initial version");
if(bSuccess == false)
{
log("ERROR: Failed to create new file: " + currentpath);
CPPUNIT_ASSERT_MESSAGE("ERROR: Failed to create new file: " + currentpath,false);
}
}
else
{
log("Trying to add new version");
if(myRecord.IsValid())
{
bSuccess = myRecord.AddNewVersion(currentpath, "Version: " + boost::lexical_cast<string>(i) );
if(bSuccess == false)
{
log("ERROR: Failed to create new version: " + currentpath);
CPPUNIT_ASSERT_MESSAGE("ERROR: Failed to create new version: " + currentpath,false);
}
}
}


}

if(bSuccess)
{
string originalName = path + ".0";
for(unsigned int i = 0; i < numVersions; i++)
{
string currentpath = path + "." + boost::lexical_cast<string>(i) + ".ret";
VersionRecord currentRecord(originalName, i + 1);
unsigned int retrievedHash = 0;

if(currentRecord.IsValid())
{
currentRecord.GetFileData(currentpath);
MurmurHash3_x86_32_FromFile(currentpath, MURMUR_SEED_1, &retrievedHash);
if(retrievedHash != currentRecord.GetHash())
{
log("HASHES DID NOT MATCH! " + currentpath);
CPPUNIT_ASSERT_MESSAGE("HASHES DID NOT MATCH! " + currentpath,false);
bSuccess = false;
break;
}
else
{
log("Hashes matched. Retrieval worked correctly");
}
}
```

```
else
{
bSuccess = false;
CPPUNIT_ASSERT_MESSAGE("Invalid version record retrieved",false);
break;
}
}
}

return bSuccess;
}

void backendtests::hashFileTest()
{
string file1 = "testData/hashFile1.dat";
string file2 = "testData/hashFile2.dat";

//hash files of different lengths

createFile(24000, file1, 14000);
createFile(2888000, file2, 24000);

unsigned int hash1 = 0;
unsigned int hash2 = 0;

MurmurHash3_x86_32_FromFile(file1, MURMUR_SEED_1, &hash1);
MurmurHash3_x86_32_FromFile(file2, MURMUR_SEED_1, &hash2);

//if the hashes match this is an error
if(hash1 == hash2)
{
CPPUNIT_ASSERT_MESSAGE("Files of different lengths", false);
}

//hash files of the same length
createFile(24000, file1, 14000);
createFile(28000, file2, 14000);

MurmurHash3_x86_32_FromFile(file1, MURMUR_SEED_1, &hash1);
MurmurHash3_x86_32_FromFile(file2, MURMUR_SEED_1, &hash2);

if(hash1 == hash2)
{
CPPUNIT_ASSERT_MESSAGE("Files of the same length", false);
}



//hash the same file with different seeds
MurmurHash3_x86_32_FromFile(file1, MURMUR_SEED_1, &hash1);
MurmurHash3_x86_32_FromFile(file1, MURMUR_SEED_2, &hash2);
if(hash1 == hash2)
{
CPPUNIT_ASSERT_MESSAGE("Same file with different seeds", false);
```

```
}
}

void backendtests::commitRetrieveTest()
{
bool bSuccess;
bSuccess = GenerateFilesAndCommitVersionsAndVerifyRetrieval("testData/20VersionFile.dat", 30000,
20);
CPPUNIT_ASSERT_MESSAGE("20 version file had an unknown error", bSuccess);

bSuccess = GenerateFilesAndCommitVersionsAndVerifyRetrieval("testData/30VersionFile.dat",
100000, 30);
CPPUNIT_ASSERT_MESSAGE("30 version file had an unknown error", bSuccess);

bSuccess = GenerateFilesAndCommitVersionsAndVerifyRetrieval("testData/40VersionFile.dat", 80000,
40);
CPPUNIT_ASSERT_MESSAGE("40 version file had an unknown error", bSuccess);
}

void backendtests::purgeTest()
{
    std::string  path = "testData/purgeFile.dat";
    unsigned int size = 24000;
    unsigned int numVersions = 5, numVersionsToKeep = 2;
    bool bSuccess = GenerateFilesAndCommitVersionsAndVerifyRetrieval(path, size, numVersions);

    // GenerateFilesAndCommitVersionsAndVerifyRetrieval() will check equality of hashes on
generation.
    CPPUNIT_ASSERT(bSuccess);

    FileRecord fileRec(path + ".0");

    CPPUNIT_ASSERT(fileRec.IsValid());
    fileRec.PurgeOldVersions(numVersionsToKeep);

    vector<VersionRecord> versionRecs = fileRec.GetAllVersions();

    // Check that the expected number of versions are the actual number of versions
(post-purge).
    CPPUNIT_ASSERT(numVersionsToKeep == versionRecs.size());

    unsigned int hash;
    std::string testPath = path + ".ret";

    for(vector<VersionRecord>::iterator it = versionRecs.begin(); it != versionRecs.end(); ++it)
    {
        it->GetFileData(testPath);
        MurmurHash3_x86_32_FromFile(testPath, MURMUR_SEED_1, &hash);

        CPPUNIT_ASSERT(hash == it->GetHash());
    }
}
```

```
/*
 * File:   backendtestrunner.cpp
 * Author: philipedwards
 *
 * Created on 30/09/2015, 6:40:10 PM
 */

#include <cppunit/BriefTestProgressListener.h>
#include <cppunit/CompilerOutputter.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/TestResult.h>
#include <cppunit/TestResultCollector.h>
#include <cppunit/TestRunner.h>

int main()
{
// Create the event manager and test controller
CPPUNIT_NS::TestResult controller;

// Add a listener that colllects test result
CPPUNIT_NS::TestResultCollector result;
controller.addListener(&result);

// Add a listener that print dots as test run.
CPPUNIT_NS::BriefTestProgressListener progress;
controller.addListener(&progress);

// Add the top suite to the test runner
CPPUNIT_NS::TestRunner runner;
runner.addTest(CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest());
runner.run(controller);

// Print test in a compiler compatible format.
CPPUNIT_NS::CompilerOutputter outputter(&result, CPPUNIT_NS::stdCOut());
outputter.write();

return result.wasSuccessful() ? 0 : 1;
}
```

```
/*
 * File:   FileLibTester.h
 * Author: jjc224
 *
 * Created on 23/09/2015, 4:24:16 PM
 */

#ifndef FILELIBTESTER_H
#defineFILELIBTESTER_H

#include <cppunit/extensions/HelperMacros.h>

class FileLibTester : public CPPUNIT_NS::TestFixture
{
    CPPUNIT_TEST_SUITE(FileLibTester);

    CPPUNIT_TEST(testAppendPath);
    CPPUNIT_TEST(testGetFilename);
    CPPUNIT_TEST(testGetPath);
    CPPUNIT_TEST(testNormalize);
    CPPUNIT_TEST(testSplitPath);

    CPPUNIT_TEST_SUITE_END();

public:
    /*
    FileLibTester();
    virtual ~FileLibTester();
    void setUp();
    void tearDown();
    */

private:
    void testAppendPath();
    void testGetFilename();
    void testGetHash();
    void testGetModifiedDate();
    void testGetPath();
    void testNormalize();
    void testSplitPath();

    // Helper functions (not in FileLib).
    std::string generateUnormalizedPath(const std::string elements[], const std::size_t SIZE);
};

#endif/* FILELIBTESTER_H */
```

```
/*
 * File:   FileLibTester.cpp
 * Author: jjc224
 *
 * Created on 23/09/2015, 4:24:17 PM
 */

#include "FileLibTester.h"
#include "FileLib.h"
#include <climits>

CPPUNIT_TEST_SUITE_REGISTRATION(FileLibTester);

void FileLibTester::testAppendPath()
{
    const std::size_t NUM_PATH_ELEMENTS_1 = 2;
    const std::size_t NUM_PATH_ELEMENTS_2 = 3;

    std::string pathElements1[NUM_PATH_ELEMENTS_1] = {"/test", "path1"};
    std::string pathElements2[NUM_PATH_ELEMENTS_2] = {"another", "path2", "test"};

    std::string   path1   =   generateUnormalizedPath(pathElements1,   NUM_PATH_ELEMENTS_1);
    std::string   path2   =   generateUnormalizedPath(pathElements2,   NUM_PATH_ELEMENTS_2);
    std::string expectedResult = "/test/path1/another/path2/test/";
    FileLib fileLib;
    std::string result = fileLib.AppendPath(path1, path2);

    CPPUNIT_ASSERT(result == expectedResult);
}

void FileLibTester::testGetFilename()
{
    const std::size_t NUM_PATH_ELEMENTS = 2;
    std::string pathElements[NUM_PATH_ELEMENTS] = {"/some", "path"};

    std::string file = "file.ext";
    std::string path = generateUnormalizedPath(pathElements, NUM_PATH_ELEMENTS) + file;
    FileLib fileLib;
    std::string result = fileLib.GetFilename(path);
    std::string expectedResult = file;

    CPPUNIT_ASSERT(result == expectedResult);
}

void FileLibTester::testGetPath()
{
    const std::size_t NUM_PATH_ELEMENTS = 5;
    std::string pathElements[NUM_PATH_ELEMENTS] = {"/path", "to", "file.ext"};

    std::string path = generateUnormalizedPath(pathElements, NUM_PATH_ELEMENTS);
    FileLib fileLib;
    std::string result = fileLib.GetPath(path);
    std::string expectedResult;
```

```
        // This produces a string of the form "/some/path/to/a/file".
    for(std::size_t i = 0; i < NUM_PATH_ELEMENTS -  1; ++i)   // The -1 is so that we don't
include the filename.
        expectedResult += pathElements[i] + "/";

    CPPUNIT_ASSERT(result == expectedResult);
}

void FileLibTester::testNormalize()
{
    const std::size_t NUM_PATH_ELEMENTS = 5;
    std::string pathElements[NUM_PATH_ELEMENTS] = {"/some", "path", "to", "a", "file"};

    std::string path = generateUnormalizedPath(pathElements, NUM_PATH_ELEMENTS);
    FileLib fileLib;
    std::string result = fileLib.Normalize(path);

    std::string expectedResult;

    // This produces a string of the form "/some/path/to/a/file".
    for(std::size_t i = 0; i < NUM_PATH_ELEMENTS; ++i)
        expectedResult += pathElements[i] + "/";

    CPPUNIT_ASSERT(result == expectedResult);
}

void FileLibTester::testSplitPath()
{
    const std::size_t NUM_PATH_ELEMENTS = 5;
    std::string pathElements[NUM_PATH_ELEMENTS] = {"C:", "some", "test", "path", "file.ext"};
    std::vector<std::string> expectedResult;
    std::string path = generateUnormalizedPath(pathElements, NUM_PATH_ELEMENTS);

    for(std::size_t i = 0; i < NUM_PATH_ELEMENTS; ++i)
        expectedResult.push_back(pathElements[i]);

    FileLib fileLib;
    std::vector<std::string> result = fileLib.SplitPath(path);

    CPPUNIT_ASSERT(result == expectedResult);
}

// Helper functions (not in FileLib).
// To be tweaked (this is just a quick test).
std::string FileLibTester::generateUnormalizedPath(const std::string elements[], const
std::size_t SIZE)
{
    std::string path, currPath;
    std::string delimiters[2] = {"/", "\\"};
    const int MAX_DELIMETER_SIZE = 2;

    // This produces a non-normalized path of the form:
```

```
    // elements[0]\elements[1]//elements[2]\\\elements[3]////...
    for(std::size_t i = 0; i < SIZE; ++i)
    {
        currPath = elements[i] + std::string(delimiters[(i + 1) % 2]);
        path.append(currPath);
    }

    return path;
}
```

```
/*
 * File:   CompressUtils.h
 * Author: thomas
 *
 * Created on 26 September 2015, 4:34 PM
 */

#ifndef COMPRESSUTILS_H
#defineCOMPRESSUTILS_H

#include <string>


// Creates a copy of a file to the temp folder
void zipCopyContents(std::string path);

// Compresses a file given a path to a file
void zipCompress();

// Uncompresses a zip to the specified location path
void zipUncompressTo(std::string path);

// Removes all zips from the temp folder
void zipRemoveZip();



#endif/* COMPRESSUTILS_H */
```

```
#include <iostream>
#include <cstdlib>
#include <string>
#include "CompressUtils.h"
#include "ProjectConstants.h"

using namespace std;



// VersionRecord::CreateVersion
// VersionRecord::GetFileData



// Compresses a file given a path to a file
void zipCompress()
{
string command = "gzip -9 ";
command.append(COMPRESSION_WORK_PATH);
command.append("data");
system(command.c_str());
}

// Uncompresses a zip to the specified location path
void zipUncompressTo(string path)
{
string command = "gunzip -c ";
command.append(COMPRESSION_WORK_PATH);
command.append("data.gz > ");
command.append(path);
system(command.c_str());
zipRemoveZip();
}

// Creates a copy of a file to the temp folder
void zipCopyContents(string path)
{
string command = "cp ";
command.append(path);
command.append(" ");
command.append(COMPRESSION_WORK_PATH);
command.append("/data");
system(command.c_str());
}

// Removes all files from temp folder
void zipRemoveZip()
{
string command = "rm ";
command.append(COMPRESSION_WORK_PATH);
command.append("data.gz -f");
system(command.c_str());
}
```

```
/*
 * File:   DBConnector.h
 * Author: philipedwards
 *
 * Created on 16 September 2015, 12:36 PM
 */

#ifndef DBCONNECTOR_H
#defineDBCONNECTOR_H

#include <mysql_connection.h>
#include <mysql_driver.h>

class DBConnector
{
public:
    DBConnector();
    virtual ~DBConnector();
    static sql::Connection* GetConnection();
private:

};

#endif/* DBCONNECTOR_H */
```

```
/*
 * File:   DBConnector.cpp
 * Author: philipedwards
 *
 * Created on 16 September 2015, 12:36 PM
 */

#include <string>
#include <fstream>

#include <mysql_connection.h>
#include <mysql_driver.h>

#include <cppconn/driver.h>
#include <cppconn/exception.h>
#include <cppconn/resultset.h>
#include <cppconn/statement.h>

#include "boost/lexical_cast.hpp"

#include "DBConnector.h"
#include "ProjectConstants.h"


using namespace std;

static sql::Connection* dbcon = NULL;
static sql::Driver* driver = NULL;
static bool bInitialised = false;

string decrypt(string s)
{
for (unsigned int i = 0; i < s.size(); i++)
{
s[i] = s[i] - 1 - i%2;
}

return s;
}

DBConnector::DBConnector()
{

}

DBConnector::~DBConnector()
{

}

sql::Connection* DBConnector::GetConnection()
{
//sql::Connection* dbcon = NULL;
```

```
//sql::Driver* driver = NULL;
if(bInitialised != false && dbcon != NULL)
{
return dbcon;
}

// Get data for connection from file
string host;
string dbname;
string user;
string pw;

ifstream ins("db.txt");
getline(ins, dbname);
getline(ins, user);
getline(ins, pw);
getline(ins, host);
ins.close();

//*
host = decrypt(host);
dbname = decrypt(dbname);
user = decrypt(user);
pw = decrypt(pw);

/*/

    host = "127.0.0.1";
    dbname = "redsquare";
    user = pw = "root";

log("host: " + host);
log("user: " + user);
log("pw: " + pw);
//*/

// Connect to database
try
{
driver = get_driver_instance();
dbcon = driver->connect(host, user, pw);
dbcon->setSchema(dbname);
bInitialised = true;
}
catch (sql::SQLException&e)
{
log("ERROR: ");
log(e.what());
log(boost::lexical_cast<string>(e.getErrorCode()));
log(e.getSQLState());
}

//dbcon->setTransactionIsolation(sql::TRANSACTION_READ_UNCOMMITTED);
```

```
return dbcon;
}
```

```
/*
 * File:   FileArchiver.cpp
 * Author: Thomas Nixon
 *
 * Created on 19 September 2015, 3:59 PM
 */

#ifndef FILEARCHIVER_H
#define FILEARCHIVER_H

#include "mysql_connection.h"
#include "mysql_driver.h"
#include <string>
#include <vector>

#include "FileRecord.h"

class FileArchiver
{

protected:
sql::Connection* dbcon;
public:
// Constructor
FileArchiver();

// Destructor
~FileArchiver();

// Checks if a file exists already
bool Exists(std::string filename);

// Gets a file record for a filename
FileRecord GetFile(std::string filename);

// Returns the number of versions on a file
int GetNumVersions(std::string filename);

// Adds a new file to the database
bool AddFile(std::string filename, std::string comment);

// Adds a new version to a file in the database
bool AddVersion(std::string filename, std::string comment);

// Returns all the files in the database as a vector
std::vector<FileRecord> GetFiles();

// Returns all the filename stored in the database
std::vector<std::string> GetFileNames();

// Detect all file changes and return the ones that have changed
std::vector<FileRecord> DetectChangedRecords();
};
```

```
#endif
```

```
// File: FileArchiver.cpp
// Author: Thomas Nixon

#include "mysql_connection.h"
#include "mysql_driver.h"

#include <cppconn/driver.h>
#include <cppconn/exception.h>
#include <cppconn/resultset.h>
#include <cppconn/statement.h>

#include <string>
#include <iostream>
#include <sstream>
#include <fstream>
#include "FileArchiver.h"
#include "DBConnector.h"
#include "FileRecord.h"
#include "VersionRecord.h"
#include "ProjectConstants.h"
#include "boost/lexical_cast.hpp"

#include "MurmurHash3.h"

using namespace std;

// Constructor
FileArchiver::FileArchiver()
{
dbcon = DBConnector::GetConnection();
}

// Destructor
FileArchiver::~FileArchiver()
{
//dbcon->close();
//delete dbcon;
dbcon = NULL;
}

// Checks if a file exists already
bool FileArchiver::Exists(std::string filename)
{
FileRecord myFile(filename);
return myFile.IsValid();
}

// Gets a file record for a filename
FileRecord FileArchiver::GetFile(std::string filename)
{
FileRecord myFile(filename);
return myFile;
}
```

```
// Returns the number of versions on a file
int FileArchiver::GetNumVersions(std::string filename)
{
FileRecord myFile(filename);
return myFile.GetNumberOfVersions();
}

// Adds a new file to the database
bool FileArchiver::AddFile(string filename, string comment)
{
FileRecord myFile;
return myFile.CreateFile(filename, comment);
}

// Adds a new version to a file in the database
bool FileArchiver::AddVersion(string filename, string comment)
{
FileRecord myFile(filename);
if(myFile.IsValid())
{
return myFile.AddNewVersion(filename, comment);
}

return false;
}

// Returns all the files in the database as a vector
vector<FileRecord> FileArchiver::GetFiles()
{
vector<string> filenames = GetFileNames();
vector<FileRecord> files;

for(unsigned int i = 0; i < filenames.size(); i++)
{
FileRecord myRec(filenames[i]);
if(myRec.IsValid())
{
files.push_back(myRec);
}
}

return files;
}

// Returns all the filename stored in the database
vector<string> FileArchiver::GetFileNames()
{
vector<string> filenames;
string sqlstatement = "select filename from File";

try
{
```

```
// Run Query
sql::Statement *stmt = dbcon->createStatement();
sql::ResultSet *rs = stmt->executeQuery(sqlstatement);

// Output Results
while(rs->next())
{
string filename = rs->getString("filename");
filenames.push_back(filename);
}

delete rs;
rs = NULL;
delete stmt;
stmt = NULL;
}
catch (sql::SQLException &e)
{
log("ERROR: ");
log(e.what());
log(boost::lexical_cast<string>(e.getErrorCode()));
log(e.getSQLState());
}

return filenames;
}

// Detect all file changes and return the ones that have changed
vector<FileRecord> FileArchiver::DetectChangedRecords()
{
vector<FileRecord> records = GetFiles();
vector<FileRecord> changed;

for(unsigned int i = 0; i < records.size(); i++)
{
FileRecord& myFile = records[i];
if(myFile.IsValid() && myFile.IsChanged())
{
changed.push_back(myFile);
}
}

return changed;
}
```

```
// Author: Joshua Coleman (jjc224).

#ifndef FILELIB_H
#defineFILELIB_H

#include <vector>
#include <string>

class FileLib
{
public:
static std::vector<std::string> SplitPath(std::string path);
static std::string Normalize(std::string path);
static std::string GetPath(std::string path);
static std::string GetFilename(std::string path);
static time_t  GetModifiedDate(std::string path);                        // Returns modification
date in seconds.
                static std::string  GetFormattedModificationDate(std::string path);      //
Returns a user-friendly date for the GUI.
static std::string AppendPath(std::string &path1,  std::string path2);   // Appends path2 onto
path1.
static unsigned int GetHash(std::string path);
                static void SetupWorkingDirectories();
};

#endif/* FILELIB_H */
```

```
// Author: Joshua Coleman (jjc224).

#include "FileLib.h"
#include "MurmurHash3.h"
#include "ProjectConstants.h"
#include  <cstdlib>                                     // For
FileLib::SetupWorkingDirectories(): command execution (system()).
#include  <boost/algorithm/string/classification.hpp>   // For FileLib::SplitPath(): string
searching (boost::is_any_of()) to assist splitting.
#include <boost/algorithm/string/split.hpp>// For FileLib::SplitPath(): string splitting
(boost::split()).
#include <boost/algorithm/string.hpp>// For FileLib::Normalize(): string trimming
(boost::algorithm::trim()).
#include  <boost/regex.hpp>                             // For FileLib::Normalize(): substring
replacement through regular expressions.
#include  <boost/filesystem.hpp>                        // For FileLib::GetModificationDate():
uses boost::filesystem::last_write_time() to obtain modification date.

using namespace std;

vector<string> FileLib::SplitPath(string path)
{
vector<string> splittedPath;

path = Normalize(path);
boost::split(splittedPath, path,  boost::is_any_of("/"), boost::token_compress_on);   // Split
path by forward slash ("/") into vector.

        // There will be a null-byte in the last string if the path ends in the delimiter.
        // So, remove it.
        if(splittedPath.back().empty())
        {
            splittedPath.pop_back();
        }

return splittedPath;
}

// Replaces one or more backslashes and two or more forward slashes with a single forward slash.
string FileLib::Normalize(string path)
{
boost::regex re("\\\\+|//+");
path = boost::regex_replace(path, re, "/", boost::match_default | boost::format_all);

return path;
}

string FileLib::GetPath(string path)
{
path = Normalize(path);
size_t lastSlashIndex = path.find_last_of("/");

if(lastSlashIndex != string::npos)
```

```
{
path =  path.substr(0, lastSlashIndex);   // Whatever is up to the final slash must be the path.
}

return path;
}

string FileLib::GetFilename(string path)
{
return SplitPath(path).back();
}

time_t FileLib::GetModifiedDate(string path)
{
time_t modificationTime;
path = Normalize(path);

try
{
modificationTime = boost::filesystem::last_write_time(path.c_str());
}
catch(const boost::filesystem::filesystem_error &e)
{
cerr << "FileLib::GetModifiedDate(): error accessing file '" << path << "' (" <<
e.code().message() << ")." << endl;
}

return modificationTime;
}

string FileLib::GetFormattedModificationDate(string path)
{
        char buffer[80];
        time_t fileTime = GetModifiedDate(path);

        strftime(buffer, 80, "%F %T", localtime(&fileTime));
        return string(buffer);
}

string FileLib::AppendPath(string &path1, string path2)
{
path1.append(path2);
        path1 = Normalize(path1);

        return path1;
}

unsigned int FileLib::GetHash(string path)
{
unsigned int hash;

MurmurHash3_x86_32_FromFile(path, MURMUR_SEED_1, &hash);
return hash;
```

```
}

void FileLib::SetupWorkingDirectories()
{
        const  string  unitTestPath     = "./testData";
        const string tempRetrievalPath = "./temp";

        const  string  clearUnitTestPath      = "rm -rf " + unitTestPath;
        const  string  clearTempRetrievalPath = "rm -rf " + tempRetrievalPath;
        const  string  createUnitTestPath     =  "mkdir  " + unitTestPath;
        const string createTempRetrievalPath =  "mkdir " + tempRetrievalPath;

system(clearUnitTestPath.c_str());
system(clearTempRetrievalPath.c_str());
system(createUnitTestPath.c_str());
system(createTempRetrievalPath.c_str());
}
```

```
#ifndef FILERECORD_H
#define FILERECORD_H

#include <string>
#include <vector>
#include "DBConnector.h"

#include "VersionRecord.h"

class FileRecord
{
public:
FileRecord();
~FileRecord();
FileRecord(std::string filename);

//Creates the file record on the data
bool CreateFile(std::string filename, std::string newComment);

VersionRecord GetVersion(unsigned int versionNum);

std::vector<VersionRecord> GetAllVersions();

void PurgeOldVersions(int numberOfVersionsToKeep);

// Returns the number of versions this file has
int GetNumberOfVersions();

unsigned int GetCurrentVersionNumber();

//Ensures there is a valid corresponding record in the database
bool IsValid();

bool GetVersionFileContents(unsigned int versionNumber, std::string fileOutPath);

// Gets the full file path
std::string GetFilename();

// Gets the length of a specific version of the file
unsigned int GetVersionSize(unsigned int versionNumber);

// Friendly function for adding a new file version
// Returns false if the version has not changed
bool AddNewVersion(std::string NewFileVersionPath, std::string newComment);

//returns true if the file on disk has been modified
//returns false if the file is the same
//returns false if the file does not exist on disk
bool IsChanged();

protected:
//retrieves the record information from the database
```

```
bool RetrieveFileRecordFromDB(std::string inFilename);
//called by constructors to perform common functionality
void Init();
//updates the FileRecord in the database;
bool UpdateRecordInDB();

// The full path name of the file
std::string Filename;

// The hash of the current version of the file
unsigned int CurrentVersionHash;

// The current revision number
unsigned int CurrentVersion;

// the number of versions this file has had
unsigned int NumberOfVersions;

// The last modified time of the file
int ModifiedTime;

// The primary ID of the file in the datastore
unsigned int FileID;

// Adds a new version of the file
// Will fail if the hash of the filebuffer data is the same as the current version hash



//uses murmur3 to get a 32 bit hash of the full file
unsigned int GetHashOfFileBuffer(int FileLength, const char* FileBuffer);

sql::Connection* dbcon;

bool bIsValid;
};

#endif
```

```
#include "FileRecord.h"
#include "MurmurHash3.h"
#include <stdint.h>

#include <fstream>

#include <cppconn/driver.h>
#include <cppconn/exception.h>
#include <cppconn/resultset.h>
#include <cppconn/statement.h>

#include <boost/filesystem.hpp>

#include "DBConnector.h"
#include "ProjectConstants.h"

#include <boost/lexical_cast.hpp>

using namespace std;

FileRecord::FileRecord()
{
Init();
}

void FileRecord::Init()
{
dbcon = DBConnector::GetConnection();


Filename            = "";
CurrentVersionHash  = 0;
CurrentVersion = 0;
NumberOfVersions    = 0;
ModifiedTime        = 0;

bIsValid = false;
}

FileRecord::~FileRecord()
{
if(dbcon != NULL)
{
//dbcon->close();
}
//delete dbcon;
dbcon = NULL;
}

FileRecord::FileRecord(std::string filename)
{
Init();
```

```
RetrieveFileRecordFromDB(filename);
}

bool FileRecord::CreateFile(string filename, string newComment)
{
sql::Statement *stmt = dbcon->createStatement();
bool bSuccess = true;

if(stmt == NULL)
{
//failed to get a connection to the database
bSuccess = false;
}

bIsValid = false;
try
{
//create file record
if(bSuccess)
{
//beginning of statement
string sqlstatement = "insert into File(filename, curhash, curversion, numversions) values(";
//filename
sqlstatement += "\"" + filename + "\"" + ", ";
//curhash
sqlstatement += boost::lexical_cast<string>(CurrentVersionHash) + ", ";
//curversion
sqlstatement += boost::lexical_cast<string>(CurrentVersion) + ", ";
//numversions
sqlstatement += boost::lexical_cast<string>(NumberOfVersions);
//end of statement
sqlstatement += ");";

log(sqlstatement);

bSuccess = stmt->executeUpdate(sqlstatement);
dbcon->commit();
}



//retrieve record from DB
if(bSuccess)
{
log("Retrieving record from db");
RetrieveFileRecordFromDB(filename);
bSuccess = IsValid();
}

if(bSuccess)
{
log("Adding new version");
bSuccess = AddNewVersion(Filename, newComment);
```

```
}
}
catch (sql::SQLException &e)
{
log("ERROR: ");
log(e.what());
log(e.getErrorCode());
log(e.getSQLState());
bSuccess = false;
}

delete stmt;
return bSuccess;
}

unsigned int FileRecord::GetCurrentVersionNumber()
{
return CurrentVersion;
}

bool FileRecord::UpdateRecordInDB()
{
sql::Statement *stmt = dbcon->createStatement();
bool bSuccess = true;

if(stmt == NULL)
{
//failed to get a connection to the database
bSuccess = false;
}

if(IsValid() == false)
{
bSuccess = false;
}

try
{
//create file record
if(bSuccess)
{
//beginning of statement
string sqlstatement = "update File set ";
//curhash
sqlstatement += "curhash = " + boost::lexical_cast<string>(CurrentVersionHash) + ", ";
//curversion
sqlstatement += "curversion = " + boost::lexical_cast<string>(CurrentVersion) + ", ";
//numversions
sqlstatement += "numversions = " + boost::lexical_cast<string>(NumberOfVersions);
//end of statement
sqlstatement += " where filename = \"" + Filename + "\";";

log(sqlstatement);
```

```
bSuccess = stmt->executeUpdate(sqlstatement);
}
}
catch (sql::SQLException &e)
{
log("ERROR: ");
log(e.what());
log(e.getErrorCode());
log(e.getSQLState());
bSuccess = false;
}

delete stmt;
return bSuccess;
}

VersionRecord FileRecord::GetVersion(unsigned int versionNum)
{
VersionRecord newVersion(Filename, versionNum);
return newVersion;
}

vector<VersionRecord> FileRecord::GetAllVersions()
{
vector<VersionRecord> allVersions;

for(unsigned int i = 0; i < NumberOfVersions; i++)
{
VersionRecord newVersion = GetVersion(i + 1);

if(newVersion.IsValid())
{
allVersions.push_back(newVersion);
}
}

return allVersions;
}

void FileRecord::PurgeOldVersions(int numberOfVersionsToKeep)
{
for(unsigned int i = 0; i <= NumberOfVersions - numberOfVersionsToKeep; i++)
{
VersionRecord purgeVersion = GetVersion(i);
if(purgeVersion.IsValid())
{
            log("Actual Purge Record No: " + boost::lexical_cast<string>(i));
purgeVersion.PurgeVersion();
}
}
}
```

```
int FileRecord::GetNumberOfVersions()
{
return NumberOfVersions;
}

//Ensures there is a valid corresponding record in the database
bool FileRecord::IsValid()
{
return bIsValid;
}

bool FileRecord::GetVersionFileContents(unsigned int requestedVersionNumber, string fileOutPath)
{
VersionRecord requestedVersion = GetVersion(requestedVersionNumber);
if(requestedVersion.IsValid())
{
return requestedVersion.GetFileData(fileOutPath);
}
return false;
}

std::string FileRecord::GetFilename()
{
return Filename;
}

unsigned int FileRecord::GetVersionSize(unsigned int versionNumber)
{
VersionRecord version = GetVersion(versionNumber);
if(version.IsValid())
{
return version.GetSize();
}
else
{
log("Invalid version, cannot retrieve version size");
return 0;
}
}



bool FileRecord::AddNewVersion(string NewFileVersionPath, string newComment)
{
bool bSuccess = true;
unsigned int newHash = 0;

log("Attempting to add new version from file: " + NewFileVersionPath);

if(boost::filesystem::exists(NewFileVersionPath) == false)
{
log("ERROR: File does not exist");
bSuccess = false;
}
```

```
if(bSuccess)
{
MurmurHash3_x86_32_FromFile(NewFileVersionPath, MURMUR_SEED_1, &newHash);

log("Hash generated for new version = " + boost::lexical_cast<string>(newHash));

//fail if hash matches existing
if(NumberOfVersions > 0 && CurrentVersionHash == newHash)
{
log("New version hash is no different. File is unchanged");
bSuccess = false;
}
}

//Add new version
VersionRecord newVersion;
if(bSuccess)
{
log("Adding new version");
bSuccess = newVersion.CreateVersion(Filename, NewFileVersionPath, CurrentVersion + 1, newHash,
newComment);
}

//Update version details
if(bSuccess)
{
log("New version added");
NumberOfVersions += 1;
CurrentVersion = newVersion.GetVersionNumber();
CurrentVersionHash = newVersion.GetHash();
bSuccess = UpdateRecordInDB();
}

return bSuccess;
}

bool FileRecord::IsChanged()
{
if(IsValid() == false)
{
return false;
}

if(boost::filesystem::exists(Filename) == false)
{
return false;
}

unsigned int fileHash;
MurmurHash3_x86_32_FromFile(Filename, MURMUR_SEED_1, &fileHash);

if(fileHash == CurrentVersionHash)
```

```
{
return false;
}
return true;
}

unsigned int FileRecord::GetHashOfFileBuffer(int FileLength, const char* FileBuffer)
{
uint32_t out;
MurmurHash3_x86_32(FileBuffer, FileLength, 10000, &out);
return out;
}

bool FileRecord::RetrieveFileRecordFromDB(string inFilename)
{
bIsValid = false;
try
{
// Run Query
sql::Statement *stmt = dbcon->createStatement();
sql::ResultSet *rs = stmt->executeQuery("select * from File where filename = '" + inFilename +
"'");

// Output Results
while(rs->next())
{
//count = rs->getUInt(1);
Filename = rs->getString("filename");
CurrentVersionHash = rs->getUInt("curhash");
CurrentVersion = rs->getUInt("curversion");
NumberOfVersions = rs->getUInt("numversions");
bIsValid = true;
}

delete rs;
rs = NULL;
delete stmt;
stmt = NULL;
}
catch (sql::SQLException &e)
{
log("ERROR: ");
log(e.what());
log(e.getErrorCode());
log(e.getSQLState());
}

return bIsValid;
}
```

```
/*
 * File:   GetCommentForm.h
 * Author: io447
 *
 * Created on 21 September 2015, 4:35 PM
 */

#ifndef _GETCOMMENTFORM_H
#define _GETCOMMENTFORM_H

#include "ui_GetCommentForm.h"

class GetCommentForm : public QDialog{
    Q_OBJECT
public:
    GetCommentForm();
    virtual ~GetCommentForm();
public slots:
    void SetComment();
    QString GetComment();

private:
    Ui::GetCommentForm widget;
    QString comment;
};

#endif/* _GETCOMMENTFORM_H */
```

```
/*
 * File:   GetCommentForm.cpp
 * Author: io447
 *
 * Created on 21 September 2015, 4:35 PM
 */
#include <QFileDialog>
#include <QApplication>
#include <iostream>
#include <string>

#include "GetCommentForm.h"

GetCommentForm::GetCommentForm() {
    widget.setupUi(this);

    connect(widget.buttonBoxComment, SIGNAL(accepted()), this, SLOT(SetComment()));
    connect(widget.buttonBoxComment, SIGNAL(rejected()), this, SLOT(reject()));


}

GetCommentForm::~GetCommentForm() {
}

void GetCommentForm::SetComment()
{
    comment = widget.textGetCommentForm->toPlainText();
}

QString GetCommentForm::GetComment()
{
    return comment;
}
```

```
//-----------------------------------------------------------------------------
// MurmurHash3 was written by Austin Appleby, and is placed in the public
// domain. The author hereby disclaims copyright to this source code.

#ifndef _MURMURHASH3_H_
#define _MURMURHASH3_H_

#include <string>

//-----------------------------------------------------------------------------
// Platform-specific functions and macros

// Microsoft Visual Studio

#if defined(_MSC_VER) && (_MSC_VER < 1600)

typedef unsigned char uint8_t;
typedef unsigned int uint32_t;
typedef unsigned   int64 uint64_t;

// Other compilers

#else   // defined(_MSC_VER)

#include <stdint.h>

#endif // !defined(_MSC_VER)

//-----------------------------------------------------------------------------

void  MurmurHash3_x86_32 ( const void * key, int len, uint32_t seed, void * out );

void MurmurHash3_x86_32_FromFile( std::string filepath, uint32_t seed, void * out );

void MurmurHash3_x86_128 ( const void * key, int len, uint32_t seed, void * out );

void MurmurHash3_x64_128 ( const void * key, int len, uint32_t seed, void * out );

//-----------------------------------------------------------------------------

#endif // _MURMURHASH3_H_
```

```
//-----------------------------------------------------------------------------
// MurmurHash3 was written by Austin Appleby, and is placed in the public
// domain. The author hereby disclaims copyright to this source code.

// Note - The x86 and x64 versions do _not_ produce the same results, as the
// algorithms are optimized for their respective platforms. You can still
// compile and run any of them on any platform, but your performance with the
// non-native version will be less than optimal.

#include "MurmurHash3.h"

#include <string>
#include <fstream>
using namespace std;

//-----------------------------------------------------------------------------
// Platform-specific functions and macros

// Microsoft Visual Studio

#if defined(_MSC_VER)

#define FORCE_INLINE    forceinline

#include <stdlib.h>

#define  ROTL32(x,y)     _rotl(x,y)
#define  ROTL64(x,y)     _rotl64(x,y)

#define BIG_CONSTANT(x) (x)

// Other compilers

#else   // defined(_MSC_VER)

#define FORCE_INLINE inline attribute ((always_inline))

inline uint32_t rotl32 ( uint32_t x, int8_t r )
{
return (x << r) | (x >> (32 - r));
}

inline uint64_t rotl64 ( uint64_t x, int8_t r )
{
return (x << r) | (x >> (64 - r));
}

#define  ROTL32(x,y)     rotl32(x,y)
#define  ROTL64(x,y)     rotl64(x,y)

#define BIG_CONSTANT(x) (x##LLU)

#endif // !defined(_MSC_VER)
```

```
//-----------------------------------------------------------------------------
// Block read - if your platform needs to do endian-swapping or can only
// handle aligned reads, do the conversion here

FORCE_INLINE uint32_t getblock32 ( const uint32_t * p, int i )
{
return p[i];
}

FORCE_INLINE uint64_t getblock64 ( const uint64_t * p, int i )
{
return p[i];
}

//-----------------------------------------------------------------------------
// Finalization mix - force all bits of a hash block to avalanche

FORCE_INLINE uint32_t fmix32 ( uint32_t h )
{
h ^= h >> 16;
h *= 0x85ebca6b;
h ^= h >> 13;
h *= 0xc2b2ae35;
h ^= h >> 16;

return h;
}

//----------

FORCE_INLINE uint64_t fmix64 ( uint64_t k )
{
k ^= k >> 33;
k *= BIG_CONSTANT(0xff51afd7ed558ccd);
k ^= k >> 33;
k *= BIG_CONSTANT(0xc4ceb9fe1a85ec53);
k ^= k >> 33;

return k;
}

//-----------------------------------------------------------------------------

void MurmurHash3_x86_32 ( const void * key, int len,
uint32_t seed, void * out )
{
const uint8_t * data = (const uint8_t*)key;
const int nblocks = len / 4;

uint32_t h1 = seed;

const uint32_t c1 = 0xcc9e2d51;
```

```
const uint32_t c2 = 0x1b873593;

//----------
// body

const uint32_t * blocks = (const uint32_t *)(data + nblocks*4);

for(int i = -nblocks; i; i++)
{
uint32_t k1 = getblock32(blocks,i);

k1 *= c1;
k1 = ROTL32(k1,15);
k1 *= c2;

h1 ^= k1;
h1 = ROTL32(h1,13);
h1 = h1*5+0xe6546b64;
}

//----------
// tail

const uint8_t * tail = (const uint8_t*)(data + nblocks*4);

uint32_t k1 = 0;

switch(len & 3)
{
case 3: k1 ^= tail[2] << 16;
case 2: k1 ^= tail[1] << 8;
case 1: k1 ^= tail[0];
k1 *= c1; k1 = ROTL32(k1,15); k1 *= c2; h1 ^= k1;
};

//----------
// finalization

h1 ^= len;

h1 = fmix32(h1);

*(uint32_t*)out = h1;
}

void MurmurHash3_x86_32_FromFile( string filepath, uint32_t seed, void * out )
{
int len = 0;
ifstream ins(filepath.c_str());

if(ins.is_open() == false)
{
//could not open the file
```

```
return;
}

//get file length
ins.seekg(0, ios::end);
len = ins.tellg();
ins.seekg(0, ios::beg);

//const uint8_t * data = (const uint8_t*)key;
const int nblocks = len / 4;

uint32_t h1 = seed;

const uint32_t c1 = 0xcc9e2d51;
const uint32_t c2 = 0x1b873593;

//----------
// body

//const uint32_t * blocks = (const uint32_t *)(data + nblocks*4);

for(int i = -nblocks; i; i++)
{
uint32_t k1;
//ins >> k1;
ins.get((char*)&k1, 4);
//ins.read(&k1, 4);

k1 *= c1;
k1 = ROTL32(k1,15);
k1 *= c2;

h1 ^= k1;
h1 = ROTL32(h1,13);
h1 = h1*5+0xe6546b64;
}

//----------
// tail

const int tailLength = len - (nblocks * 4);
uint8_t* tail = new uint8_t[tailLength];
for( int i = 0; i < tailLength; i++)
{
tail[i] = ins.get();
}

uint32_t k1 = 0;

switch(len & 3)
{
case 3: k1 ^= tail[2] << 16;
case 2: k1 ^= tail[1] << 8;
```

```
case 1: k1 ^= tail[0];
k1 *= c1; k1 = ROTL32(k1,15); k1 *= c2; h1 ^= k1;
};

//----------
// finalization

h1 ^= len;

h1 = fmix32(h1);

*(uint32_t*)out = h1;
}

//-----------------------------------------------------------------------------

void MurmurHash3_x86_128 ( const void * key, const int len,
uint32_t seed, void * out )
{
const uint8_t * data = (const uint8_t*)key;
const int nblocks = len / 16;

uint32_t h1 = seed;
uint32_t h2 = seed;
uint32_t h3 = seed;
uint32_t h4 = seed;

const uint32_t c1 = 0x239b961b;
const uint32_t c2 = 0xab0e9789;
const uint32_t c3 = 0x38b34ae5;
const uint32_t c4 = 0xa1e38b93;

//----------
// body

const uint32_t * blocks = (const uint32_t *)(data + nblocks*16);

for(int i = -nblocks; i; i++)
{
uint32_t k1 = getblock32(blocks,i*4+0);
uint32_t k2 = getblock32(blocks,i*4+1);
uint32_t k3 = getblock32(blocks,i*4+2);
uint32_t k4 = getblock32(blocks,i*4+3);

k1 *=  c1;  k1 = ROTL32(k1,15); k1 *= c2; h1 ^= k1;

h1 = ROTL32(h1,19); h1 += h2; h1 = h1*5+0x561ccd1b;

k2 *=  c2;  k2 = ROTL32(k2,16); k2 *= c3; h2 ^= k2;

h2 = ROTL32(h2,17); h2 += h3; h2 = h2*5+0x0bcaa747;

k3 *= c3; k3  = ROTL32(k3,17); k3 *= c4; h3 ^= k3;
```

```
h3 = ROTL32(h3,15); h3 += h4; h3 = h3*5+0x96cd1c35;

k4 *=  c4;  k4 = ROTL32(k4,18); k4 *= c1; h4 ^= k4;

h4 = ROTL32(h4,13); h4 += h1; h4 = h4*5+0x32ac3b17;
}

//----------
// tail

const uint8_t * tail = (const uint8_t*)(data + nblocks*16);

uint32_t k1 = 0;
uint32_t k2 = 0;
uint32_t k3 = 0;
uint32_t k4 = 0;

switch(len & 15)
{
case 15: k4 ^= tail[14] << 16;
case 14: k4 ^= tail[13] << 8;
case 13: k4 ^= tail[12] << 0;
 k4 *=  c4;  k4 = ROTL32(k4,18); k4 *= c1; h4 ^= k4;

case 12: k3 ^= tail[11] << 24;
case 11: k3 ^= tail[10] << 16;
case 10: k3 ^= tail[ 9] << 8;
case  9: k3 ^= tail[ 8] << 0;
 k3 *=  c3;  k3 = ROTL32(k3,17); k3 *= c4; h3 ^= k3;

case  8: k2 ^= tail[ 7] << 24;
case  7: k2 ^= tail[ 6] << 16;
case  6: k2 ^= tail[ 5] << 8;
case  5: k2 ^= tail[ 4] << 0;
 k2 *=  c2;  k2 = ROTL32(k2,16); k2 *= c3; h2 ^= k2;

case  4: k1 ^= tail[ 3] << 24;
case  3: k1 ^= tail[ 2] << 16;
case  2: k1 ^= tail[ 1] << 8;
case  1: k1 ^= tail[ 0] << 0;
 k1 *=  c1;  k1 = ROTL32(k1,15); k1 *= c2; h1 ^= k1;
};

//----------
// finalization

h1 ^= len; h2 ^= len; h3 ^= len; h4 ^= len;

h1 += h2; h1 += h3; h1 += h4;
h2 += h1; h3 += h1; h4 += h1;

h1 = fmix32(h1);
```

```
h2 = fmix32(h2);
h3 = fmix32(h3);
h4 = fmix32(h4);

h1 += h2; h1 += h3; h1 += h4;
h2 += h1; h3 += h1; h4 += h1;

((uint32_t*)out)[0] = h1;
((uint32_t*)out)[1] = h2;
((uint32_t*)out)[2] = h3;
((uint32_t*)out)[3] = h4;
}

//-----------------------------------------------------------------------------

void MurmurHash3_x64_128 ( const void * key, const int len,
const uint32_t seed, void * out )
{
const uint8_t * data = (const uint8_t*)key;
const int nblocks = len / 16;

uint64_t h1 = seed;
uint64_t h2 = seed;

const uint64_t c1 = BIG_CONSTANT(0x87c37b91114253d5);
const uint64_t c2 = BIG_CONSTANT(0x4cf5ad432745937f);

//----------
// body

const uint64_t * blocks = (const uint64_t *)(data);

for(int i = 0; i < nblocks; i++)
{
uint64_t k1 = getblock64(blocks,i*2+0);
uint64_t k2 = getblock64(blocks,i*2+1);

k1 *=  c1;  k1 = ROTL64(k1,31); k1 *= c2; h1 ^= k1;

h1 = ROTL64(h1,27); h1 += h2; h1 = h1*5+0x52dce729;

k2 *=  c2;  k2 = ROTL64(k2,33); k2 *= c1; h2 ^= k2;

h2 = ROTL64(h2,31); h2 += h1; h2 = h2*5+0x38495ab5;
}

//----------
// tail

const uint8_t * tail = (const uint8_t*)(data + nblocks*16);

uint64_t k1 = 0;
uint64_t k2 = 0;
```

```
switch(len & 15)
{
case 15: k2 ^= ((uint64_t)tail[14]) << 48;
case 14: k2 ^= ((uint64_t)tail[13]) << 40;
case 13: k2 ^= ((uint64_t)tail[12]) << 32;
case 12: k2 ^= ((uint64_t)tail[11]) << 24;
case 11: k2 ^= ((uint64_t)tail[10]) << 16;
case 10: k2 ^= ((uint64_t)tail[ 9]) << 8;
case  9: k2 ^= ((uint64_t)tail[ 8]) << 0;
 k2 *=  c2;  k2 = ROTL64(k2,33); k2 *= c1; h2 ^= k2;

case  8: k1 ^= ((uint64_t)tail[ 7]) << 56;
case  7: k1 ^= ((uint64_t)tail[ 6]) << 48;
case  6: k1 ^= ((uint64_t)tail[ 5]) << 40;
case  5: k1 ^= ((uint64_t)tail[ 4]) << 32;
case  4: k1 ^= ((uint64_t)tail[ 3]) << 24;
case  3: k1 ^= ((uint64_t)tail[ 2]) << 16;
case  2: k1 ^= ((uint64_t)tail[ 1]) << 8;
case  1: k1 ^= ((uint64_t)tail[ 0]) << 0;
 k1 *=  c1;  k1 = ROTL64(k1,31); k1 *= c2; h1 ^= k1;
};

//----------
// finalization

h1 ^= len; h2 ^= len;

h1 += h2;
h2 += h1;

h1 = fmix64(h1);
h2 = fmix64(h2);

h1 += h2;
h2 += h1;

((uint64_t*)out)[0] = h1;
((uint64_t*)out)[1] = h2;
}

//-----------------------------------------------------------------------------
```

```
/*
 * File:   MyWindow.h
 * Author: io447
 *
 * Created on 21 September 2015, 12:36 PM
 */

#ifndef _MYWINDOW_H
#define_MYWINDOW_H

#include "ui_MyWindow.h"
#include "GetCommentForm.h"
#include "RetrieveForm.h"
using namespace std;


class FileArchiver;
class FileRecord;
typedef FileArchiver* FilePtr;
class TableModel;


class MyWindow : public QMainWindow {
    Q_OBJECT
public:
    MyWindow();
    virtual ~MyWindow();
public slots:
    //select file
    void SelectFile();
    void SaveCurrent();
    void ShowComment();
    void CreateFirstVersion(std::string fileName);
    void AddNewVersion(std::string fileName);
    void RetrieveVersionDataForFile();
    void RetrieveVersion();
    void SetReferenceVersion();
private:
    Ui::MyWindow widget;
    GetCommentForm * getCommentWindow;
    RetrieveForm * retrieveWindow;
    TableModel * tablemodel;
    QString fileName;
};

#endif/* _MYWINDOW_H */
```