**СОДЕРЖАНИЕ**

1 ПОСТАНОВКА ЗАДАЧИ………………………………………… 5

1.1 Выбор логических структур для хранения данных…….. 5

2 ВЫБОР СРЕДЫ РАЗРАБОТКИ………………………………….. 6

3 СТРУКТУРА ВХОДНЫХ И ВЫХОДНЫХ ДАННЫХ………. 8

4 ДИАГРАММА КЛАССОВ…………………………………………... 9

5 ОПИСАНИЕ КЛАССОВ…………………………………………….. 11

6 БЛОК-СХЕМА АЛГОРИТМА……………………………………… 15

6.1 Метод bool MoveFigure( Cell& nextCell )…………………….. 15

6.2 Функция int EnterMsgLoop( bool ( \*ptr\_display )…………….. 16

7 АЛГОРИТМЫ ПО ШАГАМ………………………………………... 17

7.1 Метод void Draw( )…………………………………………….. 17

7.2 Метод bool CheckMove(Cell& currentCell, Cell& nextCell)…. 18

8 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ……………………………. 20

ПРИЛОЖЕНИЕ А. ЛИСТИНГ ПРОГРАММЫ……………………... 25

**ВВЕДЕНИЕ**

С++ - компилируемый строго типизированный язык программирования общего назначения. Название С++ происходит от С, в котором унарный оператор ++ обозначает инкремент переменной. Своего рода – «инкрементированный СИ». При создании С++ стремились сохранить совместимость с языком С, потому большинство программ созданных на С, работают и на С++ в силу идентичного синтаксиса.

Данный язык является языком высокого уровня с большой модульностью и, вследствие, широкой областью применения. Существует большое количество реализаций языка С++, как бесплатных, так и коммерческих и для различных видов платформ.

Редактор QT – это кроссплатформенный инструментарий на языке программирования С++, однако, имеющий привязки к таким языкам как PYTHON, RUBY, JAVA, PHP и другие.

Данный ретранслятор позволяет запускать написанное с его помощью ПО в большинстве современных ОС путем простой компиляции программы для каждой ОС без изменения исходного кода. Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая библиотеками для работы с сетью, базами данных и XML.

Редактор прост и удобен, интуитивно понятен людям, имеющим опыт работы с С++. Полностью объектно-ориентированный, легко расширяется, включает в себя библиотеки, содержащие мощный инструментарий, например слоты, сигналы.

Удобством также является наличие в QT собственного графического редактора, ориентированного на создание объектов интерфейса, виджетов «на ходу», напрямую размещая их на экране. При этом начальные данные виджетов автоматически переносятся в код. Инструментарий позволяет объединять элементы интерфейса воедино, вызывать дополнительные рабочие окна. Также, большим плюсом является наличие качественной и доступной документации, позволяющей быстро и просто учиться работать с редактором.

Редактор баз данных SQlite – прост, автономен и понятен. Не требует большого дискового пространства, позволяет быстро и просто создавать таблицы баз данных, определять тип данных, вносимых в таблицы, делать настройки безопасности.

**1 ПОСТАНОВКА ЗАДАЧИ**

Разработать компьютерное приложение учета строительных материалов при помощи языка С++.

К реализации предъявляются следующие требования:

1. Свободный ввод-вывод информации в программу

2. Наличие простого и понятного интерфейса пользователя

1.1 **Выбор логических структур для хранения данных**

По своей сути, программа учета строительных материалов является ни чем иным, как перечень с большим количеством связанных строк. Большую роль играет безопасность хранения, четкость и безошибочность извлечения, редактирования и помещения данных в программе. Среди возможных вариантов является:

1. создание стека данных. Минусы – сложности управления и сортировки информации, ограниченная скорость доступа к ячейкам.

2. запись в файл. Минусы – структурированность массивов данных, сложности разделения информации,

3. создание базы данных. Является оптимальным вариантом, поскольку структурированность данных в базах данных является первостепенной задачей, скорость и простота доступа к ячейкам проще и быстрее, нежели у предыдущих пунктов.

Ввиду того, что базы данных изначально создавались для хранения большого количества информации структурного типа, с возможностью быстрого доступа и редактирования, в дальнейшем будем использовать именно базы данных. Для чего будем использовать условно-бесплатную программу SQLite. Условно бесплатной она является потому, как позволяет создать простую базу данных, однако количество строк и столбцов в ней не превышает 20, что не мешает вести дальнейшую разработку. В дальнейшем при использовании разрабатываемого приложения, можно воспользоваться расширенной версией, в которой лимит по хранимым строкам снимается.

**2 ВЫБОР СРЕДЫ РАЗРАБОТКИ**

Для реализации приложения была выбрана бесплатная программа QT Creator. Она является бесплатным интерпретатором языка С++ с широким функционалом и удобной графической составляющей. QT Creator имеет встроенные библиотеки для работы с базами данных, что также говорит в ее пользу.

Основная задача QT Creator – упростить разработку приложения с помощью фреймворка QT на разных платформах. Поэтому среди возможностей, присущих любой среде разработки, есть и специфичные, такие как отладка приложений на QML и отображение в отладчике данных из контейнеров QT.

В QT организовано автодополнение, в том числе ключевых слов, введенных в стандарте С++11, подсветка кода, возможность задания стилей выравнивания, отступов, подстановки скобок, что упрощает работу и делает код визуально более понятным и воспринимаемым.

Реализован ряд возможностей при работе с сигнатурами методов, а именно:

1. автогенерация пустого тела метода после его обновления

2. возможность автоматически изменить сигнатуру метода в

определении, если она была изменена в объявлении и наоборот

3. возможность автоматически поменять порядок следования аргументов.

Архитектура:

Функция main по сути своей — вызов загрузчика плагинов, которые и реализуют всю функциональность. Поэтому все компоненты QtCreator так или иначе связаны с плагинами[[7]](https://ru.wikipedia.org/wiki/Qt_Creator#cite_note-7).

**ЕxtensionSystem** представляет интерфейс для разработки плагинов, обеспечивает загрузку плагинов и базовое взаимодействие. Содежит в себе класс-интерфейс IPlugin, от которого должны наследоваться все плагины в программе.

**Aggregation** содержит функции и классы, предназначенные для объединения нескольких компонентов в единое целое.

**Utils** — библиотека утилит, которые могут вызывать плагины.

**QmlJS** — библиотека для поддержки QML и JavaScript.

**3 СТРУКТУРА ВХОДНЫХ**  **И ВЫХОДНЫХ ДАННЫХ**

Входные данные:

QSqlDatabase – экземпляр класса QSqlDatabase, обеспечивает соединение QT с таблицами баз данных

QSqlTableModel – позволяет редактировать данные в таблицах баз данных

Выходные данные:

QDataWidgetMapper – образует связи между секциями данных в модели и виджетом

QAbstractItemModel – образует стандартный абстрактный интерфейс для моделей классов

QWidget – базовый класс для всех объектов пользовательского интерфейса

QSortFilterProxyModel – класс подключающий поддержку сортировки и фильтрации данных между моделью и изображаемой информацией

**4 ТАБЛИЦА SQL**

Для проекта, была создана таблица SQLite с 5 вводимыми полями. Шестым полем является невидимый идентификатор строки. Наименования столбцов:

0. Номер строки в базе данных (невидимый)

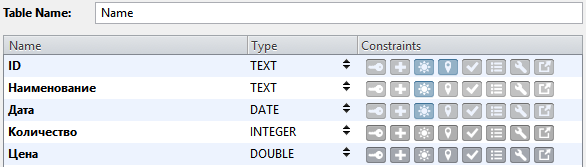
1. ID – уникальный серийный номер товара. Выставлены параметры – обязательный для заполнения, уникальный, текстовый.

2. Наименование - наименование вносимого товара. Выставленный параметр – обязателен для заполнения, текстовый

3. Дата - дата внесения товара в базу данных. Выставленный параметр – обязательный для заполнения, дата.

4. Количество – количество товара находящегося на складе. Выставленный параметр – целое число

5. Цена – цена за единицу товара. Выставленный пареметр – дробное число. Данный выбор может быть спорным, однако на данный момент известно о скором переходе на монеты и новую валюту. Параметр легко настраивается**.**

****

Название таблицы базы данных – NAME. В дальнейшем, под таким именем к ней будет обращаться QT. Добавление колонок в базу данных осуществляется нажатием кнопки “+” на главном окне:

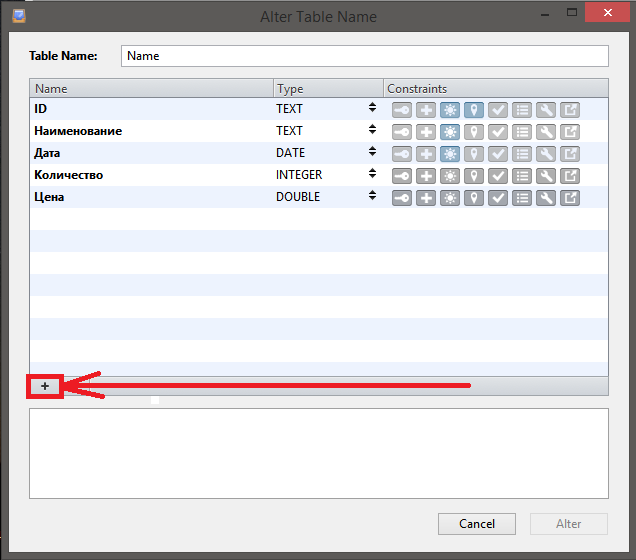
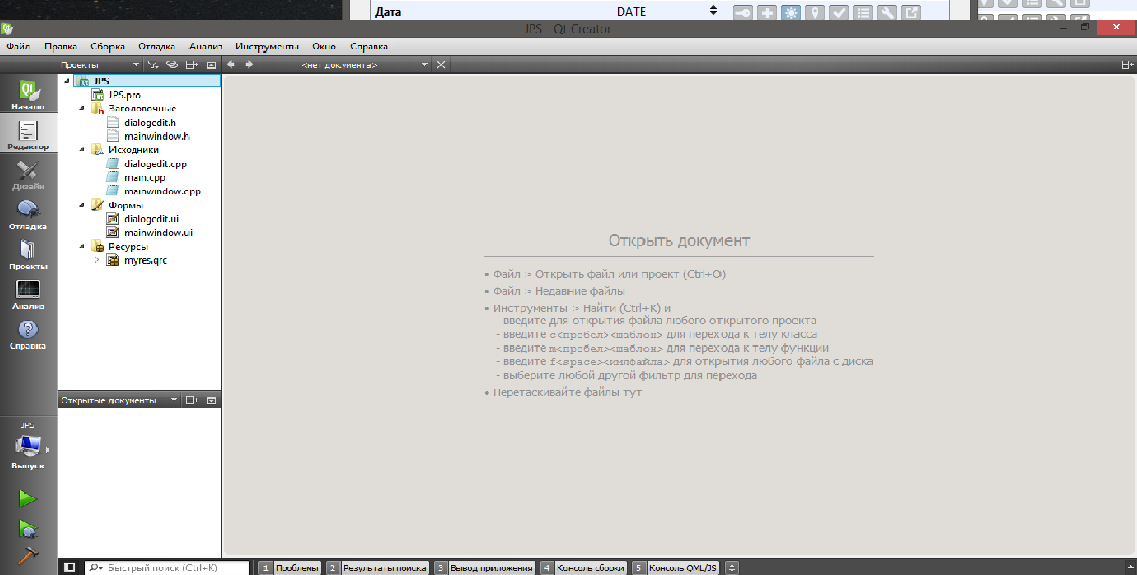


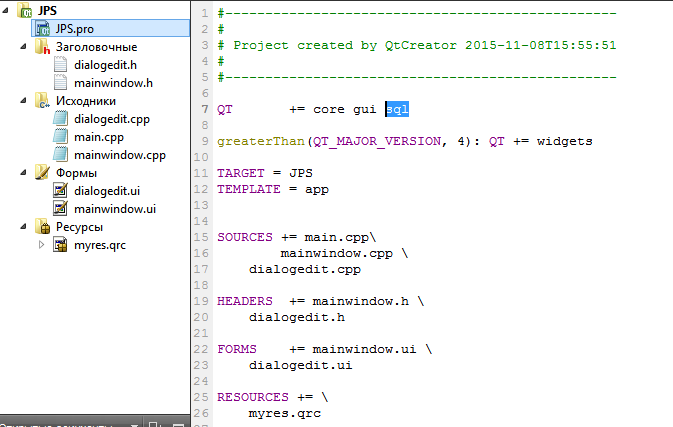
Таблица сохраняется в обычный файл с уже прописанными в нем данными. Файл с таблицей сохранен под именем “mydatabase.sqlite”

РАБОТА С ИНТЕРФЕЙСОМ

Главное окно

Для создания интерфейса будущей программы, а также совместно написания кода работы программы, используется QT Creator

Для начала, нужно научить QT «видеть» базу данных. Для этого выбираем основной файл проекта, находим строку QT += core gui и добавляем в конце sql

4

Объявляем #include <QSqlDatabase>

Обьявляем в private

QSqlDatabase mydb;

Объявляем таблицу и задаем параметры

mydb=QSqlDatabase::addDatabase("QSQLITE");

mydb.setDatabaseName("C:/JPS/mydatabase.sqlite");

//путь к таблице БД

if (!mydb.open())

//вывод в статусбар подключилось ли к БД

ui->label->setText("Failed to open the database");

else

ui->label->setText("Database is connected");

model = new QSqlTableModel(this, mydb);

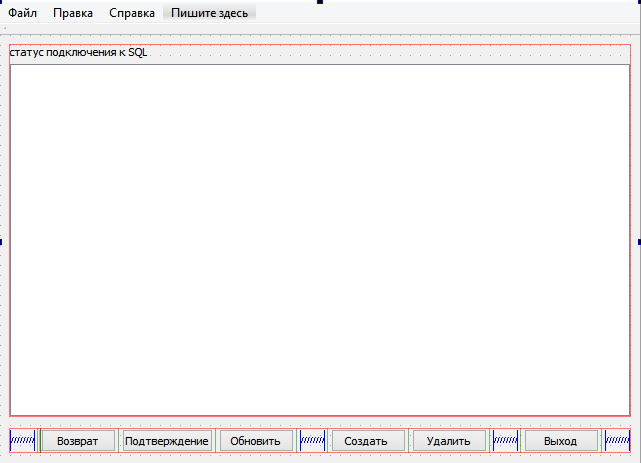
model->*setEditStrategy*(QSqlTableModel::OnManualSubmit);

//изменение в таблице только после подтверждения

model->*setTable*("NAME");

model->*select*();

Созданная в редакторе форма главного окна программы



Использованные элементы:

Label – “статус подключения к SQL”, будет отслеживать подключение к файлу базы данных

TableView – центральный элемент окна, на него будет выводиться информация из таблицы

Кнопки

Возврат – отменяет последние НЕВНЕСЕННЫЕ данные в таблицу

Подтверждение – подтверждает внесение данных в таблицу

Обновить – обновить данные на экране из файла SQL

Создать – создает новую строку в базе данных

Удалить – удаляет выделенную строку

Выход – закрытие программы

Между кнопками находятся элементы Horizontal spacer для рационального распределения кнопок по окну при изменении размера рабочего экрана.

Коды кнопок:

Кнопка «Возврат»

void MainWindow::on\_pushButton\_clicked

{

model->revertAll();

}

Кнопка «Подтверждение»

void MainWindow::on\_OkButton\_clicked

{

model->submitAll();

}

Кнопка «Обновить»

void MainWindow::on\_pushButton\_2\_clicked() {

model->*select*();

}

Кнопка «Создать»

void MainWindow::on\_CreateButton\_clicked

{

qDebug() << "inserting row" << model->insertRow(model->*rowCount*());

}

Кнопка «Удалить»

void MainWindow::on\_DeleteButton\_clicked()

{

int selectedRow = ui->tableView->

currentIndex().row();

if (selectedRow >= 0)

{

qDebug() << "deleting row:" << model->

removeRow(selectedRow);

}

else

{

qDebug() << "NO ROW SELECTED";

}

}

Кнопка «Выход»

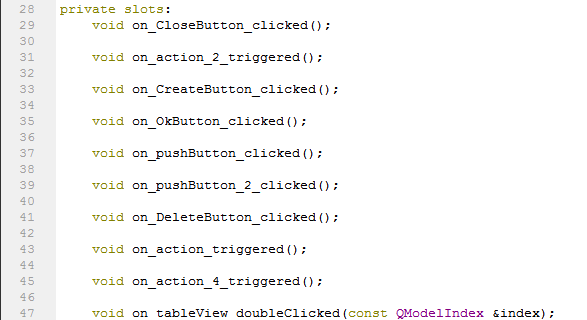
void MainWindow::on\_CloseButton\_clicked

{

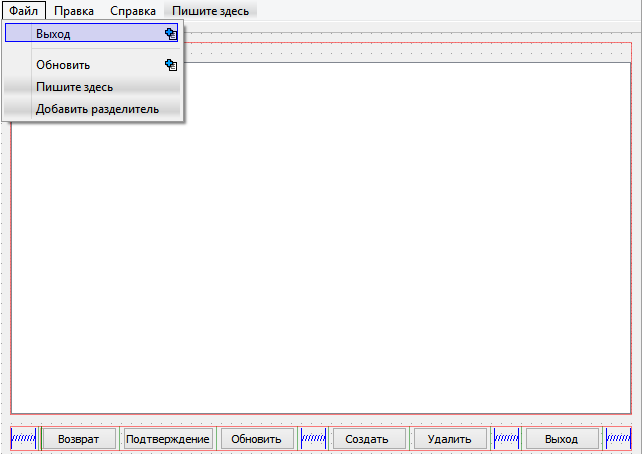
close();

}

Удобство QT в том, что при выборе элемента, можно выбрать на какое действие он будет отзываться, после чего класс создается автоматически и прописывается в header – файле:

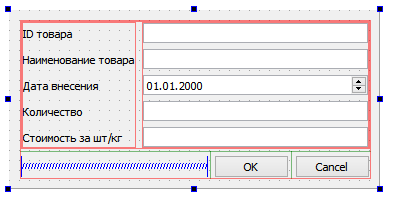


Также можно организовать верхнюю строку меню.



Дополнительное окно для редактирования

При работе с базами данных, нужно визуально разделять рабочие зоны, для лучшего понимания информации. Потому, для редактирования информации в таблице используем дополнительное окно, вызываемое двойным щелчком по таблице:



Использованные элементы:

Label’s

ID товара

Наименование товара

Дата внесения

Количество

Стоимость за шт/кг

Эти строки привязываются к другим элементам и служет для визуального восприятия информации

lineEdit\_ID – строка для вывода и изменения ID товара

lineEdit\_NAME – строка для вывода и изменения наименования товара

dateEdit – строка для вывода и изменения даты внесения товара

lineEdit\_NUM – строка для вывода и изменения количества внесенного товара

lineEdit\_COST – строка для вывода и изменения цены на товар

Также в интерфейсе есть кнопки:

OK – подтверждение внесенных данных и перенос их в основное окно

Close – отмена внесенных изменений и закрытие окна редактирования.

Для упрощения связи с главным окном, используется mapper-ы.

Это своего рода функция для установления связи между элементами вызываемого окна с элементами главной.

Подключение происходит с помощью

#include <QDataWidgetMapper>

Объявляется класс public

QDataWidgetMapper \*mapper;

Прописываем класс «соединений»

void Dialogedit::setModel(QAbstractItemModel \*model)

{

mapper->setModel(model);

mapper->addMapping(ui->lineEdit\_ID, 0);

mapper->addMapping(ui->lineEdit\_NAME, 1);

mapper->addMapping(ui->dateEdit, 2);

mapper->addMapping(ui->lineEdit\_NUM, 3);

mapper->addMapping(ui->lineEdit\_COST, 4);

}

Устанавливаем передачу изменений вручную

mapper->setSubmitPolicy(QDataWidgetMapper::ManualSubmit);

И прописываем в слот TableView по двойному нажатию вызов окна и передачу строки в нажатом окне

void MainWindow::on\_tableView\_doubleClicked(const QModelIndex &index)

{

editForm->mapper->setCurrentModelIndex(index);

editForm->show();

}

Кнопка «OK»

void Dialogedit::on\_OKbtn\_clicked()

{

mapper->submit();

close();

}

Кнопка «Cancel»

void Dialogedit::on\_Cancelbtn\_clicked()

{

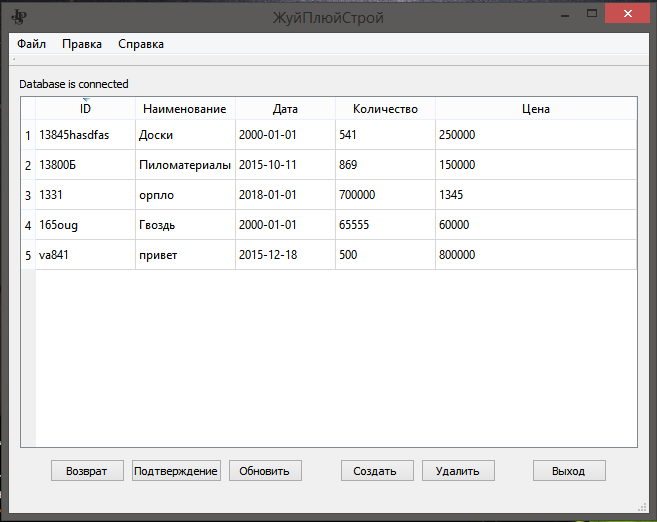
close();

}

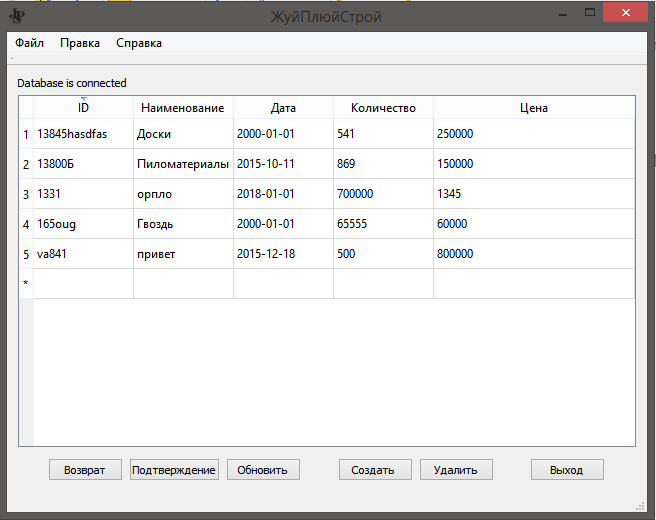
Для удаления возможности изменить таблицу с главного окна, выделяем TanleView, находим пункт в окне настроек edittriggers и выставляем галочку NoEditTriggers, в результате все остальные галочки в подпункте должны убраться.

СКРИНШОТЫ РАБОТЫ ПРОГРАММЫ

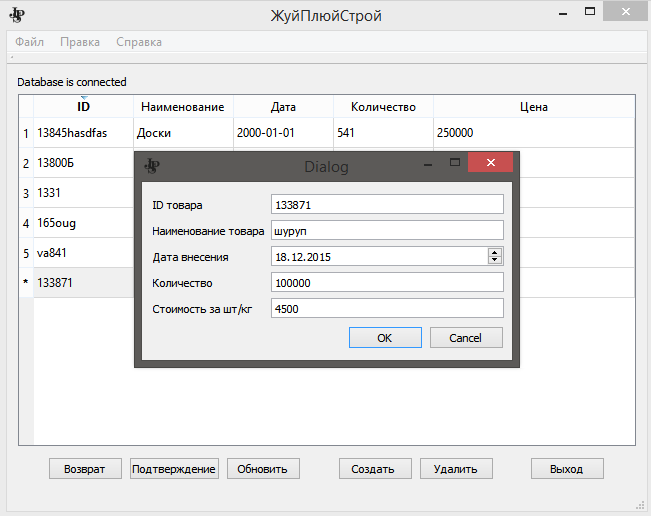
Главное окно, с базой данных соединилось



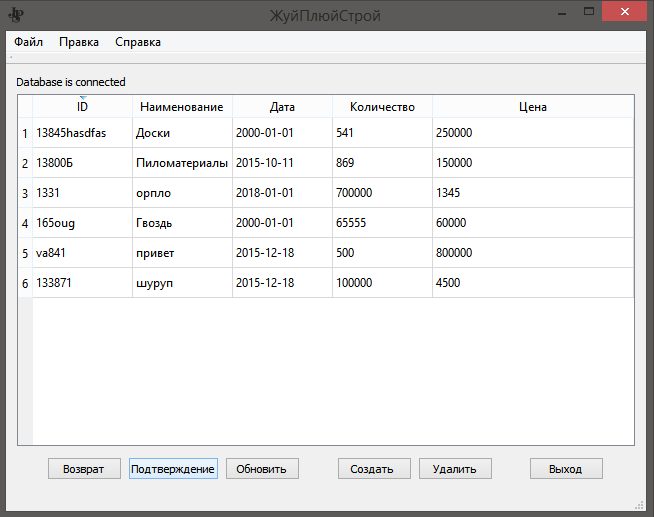
Создание новой строки



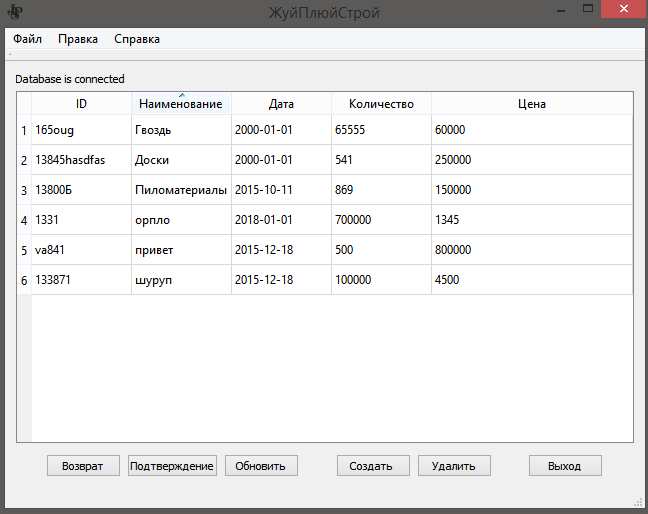
Редактирование информации



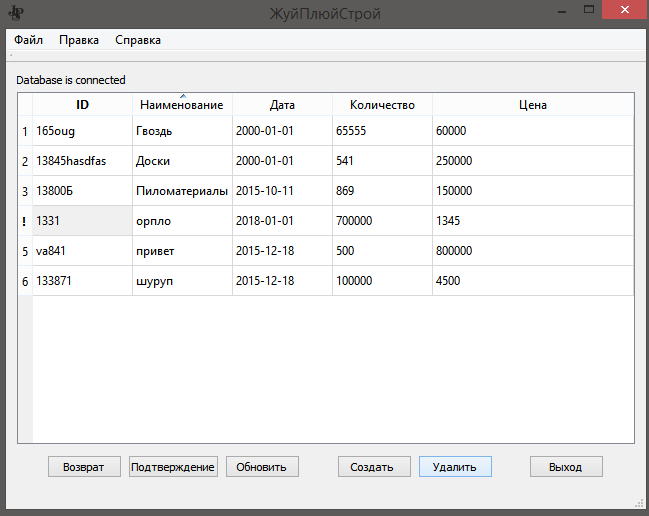
Подтверждение внесенных изменений



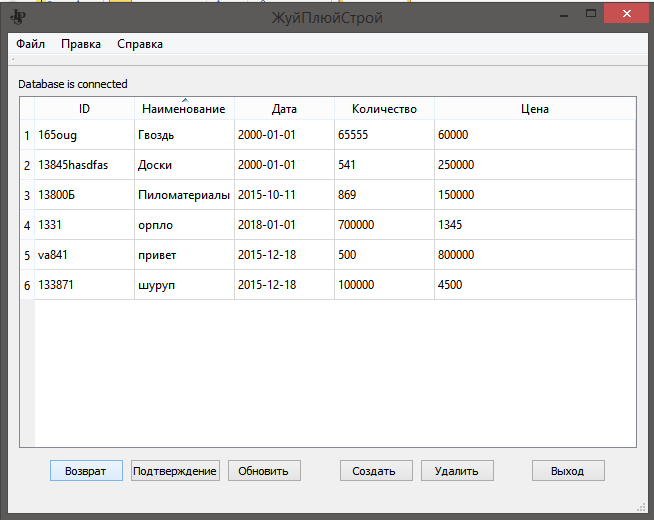
Сортировка по столбцу массива

****

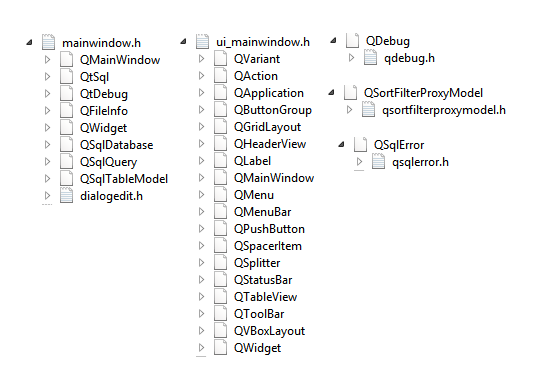
**Удаление элемента таблицы**

****

**Откат последних изменений**

****

**ИТОГОВОЕ ДЕРЕВО КЛАССОВ**

****

**ЗАКЛЮЧЕНИЕ**

В рамках данной работы был разработан компьютерный вариант рассчетной программы учета строительных материалов. Программа отлично подходит для хранения таких структурированных данных, обладает абсолютной автономностью, не требует наличия MySQL сервера.

Программа подойдет любой компании с наличием склада, для прихода расхода материалов. В будущих версиях можно перейти на расширенную базу данных, чтоб хранить больше ячеек. Также можно переназначить колонки по желанию клиента. Встает вопрос удобства интерфейсных кнопок

**СПИСОК ЛИТЕРАТУРЫ**

1.<https://www.youtube.com/watch?v=WZV5hUAVVjU&index=1&list=PLbmlzoDQrXVHDN2nBogk7qy-4DzgyNhNq> – видеоуроки по QT

2.<https://www.youtube.com/watch?v=IWOs5FqJGlM&list=PLfTV0gRh8sNK6km5thab8dMNvuog2HSGp&index=35> – видеоуроки по QT, создание и объединение qt с базами данных SQL

3.Макконел С. «Совершенный код. Мастер-класс». СПб.: Питер, 2005 – 896 стр.; ил. ISBN 5-7502-0064-7

4.Дейтел. «Как программировать на С++» г.Москва, издательство «Бином»

5.<http://goodprogrammer.ru/real-programming-2/lessons/11-databases-sqlite-install> - видеоурок, понятие и программирование баз данных SQLite

**ПРИЛОЖЕНИЕ А. ЛИСТИНГ ПРОГРАММЫ**

Mainwindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QtSql>

#include <QtDebug>

#include <QFileInfo>

#include <QWidget>

#include <QSqlDatabase>

#include <QSqlQuery>

#include <QSqlTableModel>

#include "dialogedit.h"

namespace Ui {

class MainWindow;

}

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

explicit MainWindow(QWidget \*parent = 0);

~*MainWindow*();

private slots:

void on\_CloseButton\_clicked();

void on\_action\_2\_triggered();

void on\_CreateButton\_clicked();

void on\_OkButton\_clicked();

void on\_pushButton\_clicked();

void on\_pushButton\_2\_clicked();

void on\_DeleteButton\_clicked();

void on\_action\_triggered();

void on\_action\_4\_triggered();

void on\_tableView\_doubleClicked(const QModelIndex &index);

private:

Ui::MainWindow \*ui;

QSqlDatabase mydb;

QSqlTableModel\* model;

Dialogedit\* editForm;

};

#endif // MAINWINDOW\_H

**Mainwindow.cpp**

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <QSqlError>

#include <QDebug>

#include <QSortFilterProxyModel>

MainWindow::MainWindow(QWidget \*parent) :

QMainWindow(parent),

ui(new Ui::MainWindow)

{

ui->setupUi(this);

mydb=QSqlDatabase::addDatabase("QSQLITE");

mydb.setDatabaseName("C:/JPS/mydatabase.sqlite"); //путь к таблице БД

if (!mydb.open()) //вывод в статусбар подключилось ли к БД

ui->label->setText("Failed to open the database");

else

ui->label->setText("Database is connected");

model = new QSqlTableModel(this, mydb);

model->*setEditStrategy*(QSqlTableModel::OnManualSubmit); //изменение в таблице только после подтверждения

model->*setTable*("NAME");

model->*select*();

QSortFilterProxyModel\* proxy = new QSortFilterProxyModel(this); //эти строки для сортировки без изменения ID

proxy->*setSourceModel*(model);

ui->tableView->*setModel*(model);

editForm = new Dialogedit();

editForm->setParent(this,Qt::Window);

editForm->setModel(model);

}

MainWindow::~*MainWindow*() //деструктор

{

delete ui;

}

void MainWindow::on\_CloseButton\_clicked()

//закрытие основного окна

{

close();

}

void MainWindow::on\_action\_2\_triggered()

//закрытие основного окна

{

close();

}

void MainWindow::on\_CreateButton\_clicked()

//создание новой строки БД

{

qDebug() << "inserting row" << model->insertRow(model->*rowCount*());

}

void MainWindow::on\_OkButton\_clicked()

//подтверждение изменений таблицы

{

model->submitAll();

}

void MainWindow::on\_pushButton\_clicked()

//откат изменений

{

model->revertAll();

}

void MainWindow::on\_pushButton\_2\_clicked() //обновление таблицы

{

model->*select*();

}

void MainWindow::on\_DeleteButton\_clicked()

//удаление выделенной строки

{

int selectedRow = ui->tableView->currentIndex().row();

if (selectedRow >= 0)

{

qDebug() << "deleting row:" << model->removeRow(selectedRow);

}

else

{

qDebug() << "NO ROW SELECTED";

}

}

void MainWindow::on\_action\_triggered()

{

qDebug() << "inserting row" << model->insertRow(model->*rowCount*());

}

void MainWindow::on\_action\_4\_triggered()

//удаление выбранной строки из верхнего меню

{

int selectedRow = ui->tableView->currentIndex().row();

if (selectedRow >= 0)

{

qDebug() << "deleting row:" << model->removeRow(selectedRow);

}

else

{

qDebug() << "NO ROW SELECTED";

}

}

void MainWindow::on\_tableView\_doubleClicked(const QModelIndex &index)

{

editForm->mapper->setCurrentModelIndex(index);

editForm->show();

}

**Dialogedit.h**

#ifndef DIALOGEDIT\_H

#define DIALOGEDIT\_H

#include <QDialog>

#include <QDataWidgetMapper>

namespace Ui {

class Dialogedit;

}

class Dialogedit : public QDialog

{

Q\_OBJECT

public:

explicit Dialogedit(QWidget \*parent = 0);

~*Dialogedit*();

void setModel(QAbstractItemModel\* model);

QDataWidgetMapper \*mapper;

private slots:

void on\_OKbtn\_clicked();

void on\_Cancelbtn\_clicked();

private:

Ui::Dialogedit \*ui;

};

#endif // DIALOGEDIT\_H

**Dialogedit.cpp**

#include "dialogedit.h"

#include "ui\_dialogedit.h"

Dialogedit::Dialogedit(QWidget \*parent) :

QDialog(parent),

ui(new Ui::Dialogedit)

{

ui->setupUi(this);

mapper = new QDataWidgetMapper(this);

mapper->setSubmitPolicy(QDataWidgetMapper::ManualSubmit);

}

Dialogedit::~*Dialogedit*()

{

delete ui;

}

void Dialogedit::setModel(QAbstractItemModel \*model)

{

mapper->setModel(model);

mapper->addMapping(ui->lineEdit\_ID, 0);

mapper->addMapping(ui->lineEdit\_NAME, 1);

mapper->addMapping(ui->dateEdit, 2);

mapper->addMapping(ui->lineEdit\_NUM, 3);

mapper->addMapping(ui->lineEdit\_COST, 4);

}

void Dialogedit::on\_OKbtn\_clicked()

{

mapper->submit();

close();

}

void Dialogedit::on\_Cancelbtn\_clicked()

{

close();

}

**Main.cpp**

#include "mainwindow.h"

#include <QApplication>

int main(int argc, char \*argv[])

{

QApplication a(argc, argv);

MainWindow w;

w.show();

return a.exec();

}