МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное   
учреждение высшего образования

**"Южно-Уральский государственный университет**

**(национальный исследовательский университет)"**

**Высшая школа электроники и компьютерных наук**

**Кафедра системного программирования**

**ОТЧЕТ   
по учебной практике**

бакалавра направления 02.03.02 "Фундаментальная информатика   
и информационные технологии"

|  |  |
| --- | --- |
|  | Выполнил:\_\_\_\_\_\_\_\_\_\_\_\_ студент группы КЭ-201  Гордеев А.С.  Проверил:\_\_\_\_\_\_\_\_\_\_\_\_\_  Докт. физ.-мат. наук, доцент,  Проф. кафедры СП  Макаровских Т.А.  Дата:\_\_\_\_\_\_\_\_, Оценка:\_\_\_\_\_\_\_\_\_\_ |

Челябинск-2022

Министерство науки и высшего образования Российской Федерации

Южно-Уральский государственный университет

Кафедра системного программирования

УТВЕРЖДАЮ

Зав. кафедрой

системного программирования

\_\_\_\_\_\_\_\_\_\_\_\_\_ Л.Б. Соколинский

**ЗАДАНИЕ**

**по** **учебной практике**

1. **Цель работы**

Разработать GUI-приложение, работающее с входной информацией, вводимой пользователем с помощью управляемых элементов формы, либо из текстового файла.

1. **Исходные данные к работе**
2. База данных пословиц, поговорок, афоризмов, каламбуров и других словесных курьезов.
3. Хранение данных в файле в формате json.
4. Файл хранит такую информацию как: автор, тема и фраза.
5. **Перечень подлежащих разработке вопросов**
6. Определение структуры приложения (по модулям), структур данных, используемых для хранения основной пользовательской информации.
7. Дизайн оконного интерфейса, анализ структуры входных данных и их защита от некорректного ввода информации.
8. Разработка основного функционала приложения: основных форм и механизмов получения информации из их компонентов и их файлов; основного алгоритма функционирования приложения; тестирование приложения.
9. Подготовка руководства пользователя и документации для программиста.
10. **Сроки**

Дата выдачи задания: "27" июня 2022 г.

Срок сдачи законченной работы: "23" июля 2022 г.

**Руководитель:**

Докт. физ-мат наук, проф. кафедры СП \_\_\_\_\_\_\_\_\_\_\_\_\_ Макаровских Т.А.

должность, ученая степень подпись ФИО руководителя

**Задание принял к исполнению:**

\_\_\_\_\_\_\_\_\_\_\_\_\_ Гордеев А.С.

подпись ФИО студента

**ОГЛАВЛЕНИЕ**

[1. ПОСТАНОВКА ЗАДАЧИ 4](#_Toc107416469)

[2. ДИЗАЙН ОКОННОГО ИНТЕРФЕЙСА 7](#_Toc107416470)

[3. РАЗРАБОТКА ФУНКЦИОНАЛА ОСНОВНЫХ ФОРМ И МЕХАНИЗМОВ ПОЛУЧЕНИЯ ИНФОРМАЦИИ 9](#_Toc107416471)

[4. РАЗРАБОТКА ОСНОВНОГО МЕХАНИЗМА ФУНКЦИОНИРОВАНИЯ ПРИЛОЖЕНИЯ 11](#_Toc107416472)

[5. ТЕСТИРОВАНИЕ 12](#_Toc107416473)

[5.1. Автономное тестирование 12](#_Toc107416474)

[5.2. Комплексное тестирование 12](#_Toc107416475)

[6. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 14](#_Toc107416476)

[7. РУКОВОДСТВО РАЗРАБОТЧИКА 15](#_Toc107416477)

[ЗАКЛЮЧЕНИЕ 16](#_Toc107416478)

[ЛИТЕРАТУРА 17](#_Toc107416479)

# ПОСТАНОВКА ЗАДАЧИ

Требуется написать справочник пословиц, поговорок, афоризмов, каламбуров, других словесных курьезов. Программа выполняет классификацию по авторам и источникам, поиск по темам и автору. Программа должна обеспечивать поиск по базе по заданным критериям, позволять редактировать и дополнять базу. Записи базы данных будут иметь следующие типы (табл.1).

Таблица 1. Переменные, используемые для хранения записей базы данных

|  |  |  |
| --- | --- | --- |
| Переменная | Тип переменной | Содержательный смысл |
| id | int | Идентификатор записи |
| author | QString | Автор (источник) |
| theme | QString | Тема |
| phrase | QString | Фраза |

Реализовать решение данной задачи можно с помощью такой известной структуры как префиксное дерево, или же бор. Каждый его элемент — вершина дерева, включающая в себя символ поиска строки, номер вершины, хэш-таблицу с номерами следующих вершин в виде: ключ-символ, значение-номер; булеву переменную, является ли вершина листом и массив типа QVector для хранения идентификаторов записей. Само префиксное дерево будет состоять из символов (вершин) строки с названием источника (автора) и темы для их поиска по данным критериям, а каждый лист массив идентификаторов.

Листинг 1. Пример структуры вершины и класса, реализующего поиск в префиксном дереве

struct Node {

QMap<QChar, int> nextNodes;

bool isLeaf = false;

QVector<int> ids;

};

class PrefixSearcher {

public:

PrefixSearcher() : countNodes\_(1), countStrings\_(0) {}

int getCountStrings() { return countStrings\_; }

void insert(const QString& str, int input\_id);

void erase(const QString& str, int erase\_id);

QVector<int> find(const QString& prefix);

private:

QMap<int, Node> trie\_;

int countNodes\_;

int countStrings\_;

bool canGoNode(int vertice, QChar sym);

void createNode(int vertice, QChar sym);

int getNextVertice(int vertice, QChar sym);

bool isLeaf(int vretice);

void depthFirstSearch(QVector<int>& result, int vertice);

};

Разрабатываемое приложение состоит из пяти оконных форм:

1. Главное окно программы.
2. Диалоговое окно добавление записи в базу.
3. Окно редактирования записи.
4. Окно с информацией о приложении.
5. Окно с информацией об авторе.

Каждой из разработанных оконных форм соответствует три файла (\*.ui, \*.h, \*.cpp), приведенные в таблице 2. Там же приведена информация о файле prefixsearcher.h, в котором содержится информация о классе, реализующем работу префиксного дерева.

Таблица 2. Модули создаваемого проекта

|  |  |  |  |
| --- | --- | --- | --- |
| Имя файла | Описание информации, содержащейся в нем | Функциональное назначение | Файлы проекта, подключенные к текущему файлу посредством директивы #include и файл разметки интерфейса |
| prefixsearcher.cpp | Класс реализации поиска в дереве | Работа с префиксном деревом | prefixsearcher.h |
| tabledata.cpp | Класс, реализующий базу данных таблицы | Хранение и обработка данных таблицы | tabledata.h |
| mainwindow.cpp | Класс основного рабочего окна приложения | Работа с основным окном приложения. | mainwindow.h  mainwindow.ui |
| additemdialog.cpp | Класс окна добавления записи | Ввод данных пользователя и добавление в базу | additemdialog.h  additemdialog.ui |
| edititemdialog.cpp | Класс окна редактирования записи | Редактирование данных в базе | edititemdialog.h  edititemdialog.ui |
| helpwindow.cpp | Класс окна с информацией о приложении | Вывод информации о приложении | helpwindow.h  helpwindow.ui |
| aboutwindow.cpp | Класс окна с информацией об авторе | Вывод информаци об авторе | aboutwindow.h  aboutwindow.ui |

На рисунке 1 представлена схема взаимодействия классов и оконных форм в приложении.

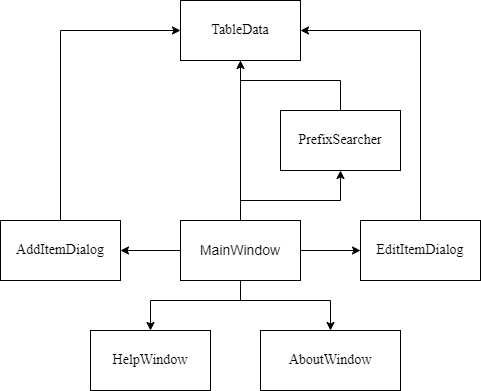


Рисунок 1. Блок-схема взаимодействия классов

# ДИЗАЙН ОКОННОГО ИНТЕРФЕЙСА

В данном разделе приводится описание всех оконных форм, используемых для функционирования приложения:

* Основная форма для работы с таблицей.
* Форма добавление записи в таблицу.
* Форма редактирования записи таблицы.
* Форма информации о приложении.
* Форма информации об авторе.
* Вспомогательные окна.

При описании интерфейса приводится изображение соответствующей формы и приводится перечень помещенных на нее компонентов для ввода/вывода данных, оформления, и пр.

Для реализации работы с данными можно воспользоваться таблицей, реализованной в классе QTableView. На рисунке 2 представлено оформление основной формы приложения MainWindow класса QMainWindow.

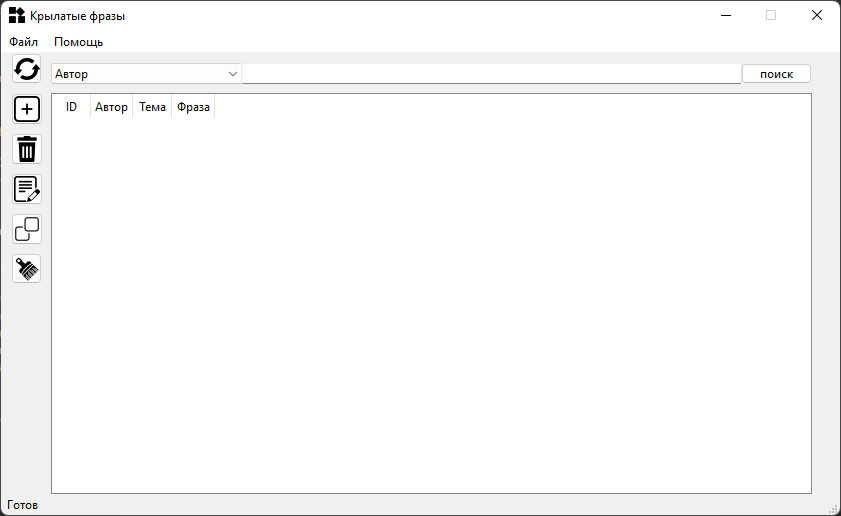


Рисунок 2. Оформление формы основного окна приложения MainWindow

Размещенные на форме компоненты и перечень методов и событий, которые необходимо реализовать приведен в таблице 3.

Таблица 3. Компоненты основного окна приложения

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя компоненты | Тип | Ограничения для ввода информации | Реализованные события | Функциональное назначение |
| mainTableView | QTableView | Таблица с данными | Нет | Отображение таблицы из класса TableData |
| criterionComboBox | QComboBox | Выпадающее меню с пунктами: Автор, Тема | onCriterionCombo-  BoxActivated | Выбор критерия для поиска в таблице |
| searchLineEdit | QLineEdit | Ввод данных в виде строки | Нет | Пользователь вводит строку для поиска значения из таблицы по заданному критерию |
| searchButton | QPushButton | Инициализация поиска с заданной строкой | onSearchButtonClicked | Пользователю выводится таблица с заданным критерием поиска |
| refreshButton | QPushButton | Обновление таблицы | onRefreshButtonClicked | Пользователь обновляет весь интерфейс таблицы |
| addButton | QPushButton | Вывод формы для добавления записи в таблицу | onAddButtonClicked | При нажатии таблицы открывается форма для заполнения |
| removeButton | QPushButton | Удаление записи с таблицы | onRemoveButtonClicked | При нажатии на кнопку осуществляется удаление выделенной записи |
| editButton | QPushButton | Вывод формы для редактирования записи | onEditButtonClicked | При нажатии на кнопку пользователь может редактировать выделенную запись таблицы |
| copyButton | QPushButton | Копирование записи таблицы | onCopyButtonClicked | При нажатии на кнопку выделенная запись копируется в буфер обмена |
| clearButton | QPushButton | Очистка таблицы | onClearButtonClicked | Пользователь может очистить таблицу от всех записей |
| menuBar | QMenuBar | Выбор пункта из выпадающего меню | Нет | Пользователь может обратиться к пунктам меню |
| actonSave | QAction | Формат файла json | onSaveFileClicked | Пользователь может сохранить свои изменения в файле |
| actionLoad | QAction | Формат файла json | onLoadFileClicked | Пользователь может загрузить необходимый ему файл с данными |
| actionHelp | QAction | Нет | onHelpActionClicked | Отображает окно с помощью для работы с программой |
| actionAbout | QAction | Нет | onAboutActionClicked | Отображает окно с информацией об авторе |

Добавление и редактирование записи из таблицы реализованы в похожих формах AddItemDialog и EditItemDialog соответственно и имеют класс QDialog. На рисунках 3 и 4 представлено оформление этих форм, а их компоненты в таблицах 4 и 5. Главные отличия в них, это название кнопок “Добавить” и “Изменить”, и в том, что данные в форме EditItemDialog берутся из таблицы для редактирования.

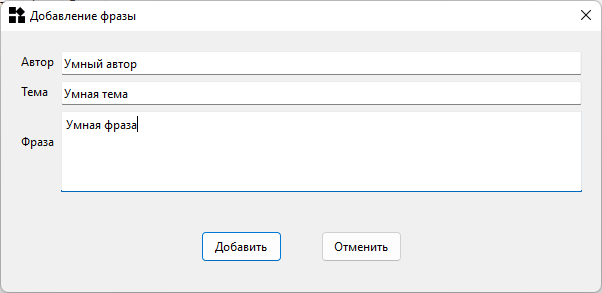


Рисунок 3. Оформление формы добавления записи AddItemDialog

Таблица 4. Компоненты диалогового окна AddItemDialog

|  |  |  |  |
| --- | --- | --- | --- |
| Имя компоненты | Тип | Ограничения для ввода информации | Функциональное назначение |
| authorLabel | QLabel | Текстовая информация | Информация о вводимом поле “Автор” |
| authorLineEdit | QLineEdit | Ввод строки | Пользователь вводит имя автора или источника |
| themeLabel | QLabel | Текстовая информация | Информация о вводимом поле ‘Тема” |
| themeLineEdit | QLineEdit | Ввод строки | Пользователь вводит название темы фразы |
| phraseLabel | QLabel | Текстовая информация | Информация о вводимом поле “Фраза” |
| phraseTextEdit | QTextEdit | Ввод многострочных данных | Пользователь вводит фразу |
| acceptButton | QPushButton | Соглашение о вводимых данных | При нажатии на кнопку пользователь добавляет запись таблицы |
| cancelButton | QPushButton | Отмена изменений | При нажатии на кнопку пользователь отказывается от вводимых или измененных данных |

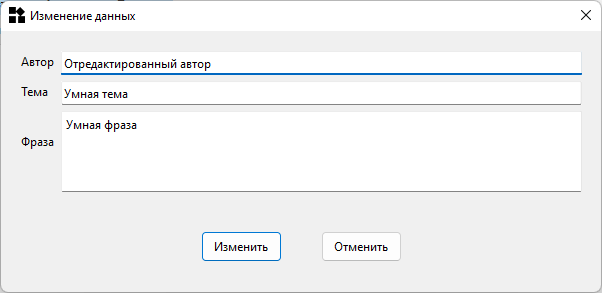


Рисунок 4. Оформление формы редактирования записи EditItemDialog

Таблица 5. Компоненты диалогового окна EditItemDialog

|  |  |  |  |
| --- | --- | --- | --- |
| Имя компоненты | Тип | Ограничения для ввода информации | Функциональное назначение |
| authorLabel | QLabel | Текстовая информация | Информация о редактируемом поле “Автор” |
| authorLineEdit | QLineEdit | Ввод строки | Пользователь редактирует имя автора или источника |
| themeLabel | QLabel | Текстовая информация | Информация о вводимом поле ‘Тема” |
| themeLineEdit | QLineEdit | Ввод строки | Пользователь редактирует название темы |
| phraseLabel | QLabel | Текстовая информация | Информация о вводимом поле “Фраза” |
| phraseTextEdit | QTextEdit | Ввод многострочных данных | Пользователь редактирует фразу |
| acceptButton | QPushButton | Соглашение о вводимых данных | При нажатии на кнопку пользователь изменяет запись таблицы |
| cancelButton | QPushButton | Отмена изменений | При нажатии на кнопку пользователь отказывается от вводимых или измененных данных |

Информационная форма содержит полезные подсказки для пользователя и представлена на рисунке 5, ее компоненты в таблице 6.

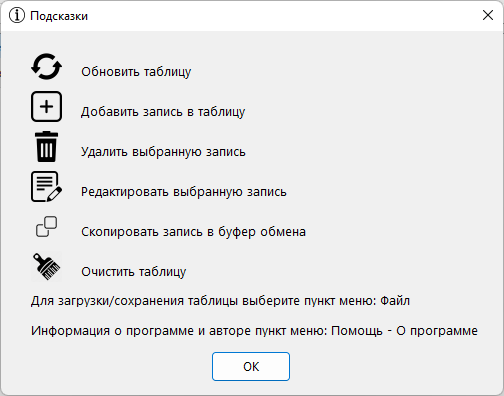


Рисунок 5. Оформление информационной формы HelpWindow

Таблица 6. Компоненты информационной формы HelpWindow

|  |  |  |
| --- | --- | --- |
| Имя компоненты | Тип | Функциональное назначение |
| updateIcon | QLabel | Иконка кнопки “Обновить” |
| updateLabel | QLabel | Подпись иконки “Обновить” |
| addIcon | QLabel | Иконка кнопки “Добавить” |
| addLabel | QLabel | Подпись иконки “Добавить” |
| removeIcon | QLabel | Иконка кнопки “Удалить” |
| removeLabel | QLabel | Подпись иконки “Удалить” |
| editIcon | QLabel | Иконка кнопки “Редактировать” |
| editLabel | QLabel | Подпись иконки “Редактировать” |
| copyIcon | QLabel | Иконка кнопки “Скопировать” |
| copyLabel | QLabel | Подпись иконки “Скопировать” |
| clearIcon | QLabel | Иконка кнопки “Очистить” |
| clearLabel | QLabel | Подпись иконки “Очистить” |
| fileLabel | QLabel | Информация о работе с файлами |
| aboutLabel | QLabel | Информация как вызвать окно “О программе” |
| okButton | QPushButton | Позволяет выйти из формы |

Форма об авторе содержит полезную информацию о программе и ее авторе и представлена на рисунке 6, компоненты в таблице 7.

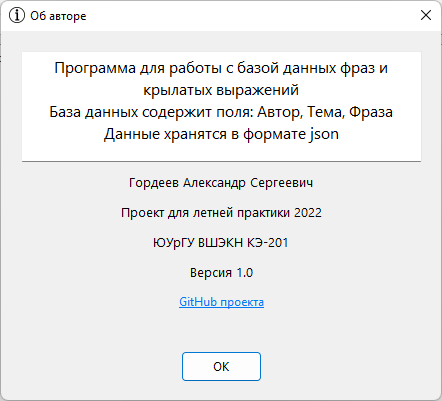


Рисунок 6. Оформление формы об авторе AboutWindow

Таблица 6. Компоненты формы об авторе AboutWindow

|  |  |  |
| --- | --- | --- |
| Имя компоненты | Тип | Функциональное назначение |
| aboutProgram | QTextEdit | Информация о программе |
| aboutLabel | QLabel | Информация об авторе |
| okButton | QPushButton | Позволяет выйти из формы |

Алгоритмы и программная реализация приведенных в этой главе событий и методов приведены в следующем разделе.

# РАЗРАБОТКА ФУНКЦИОНАЛА ОСНОВНЫХ ФОРМ И МЕХАНИЗМОВ ПОЛУЧЕНИЯ ИНФОРМАЦИИ

Главная форма MainWindow содержит множество кнопок, которые привязаны к событиям QAbstractButton::clicked, а также QAction к событиям QAction::triggered. Здесь приводится информация обо всех событиях и методах.

При нажатии на кнопку refreshButton вызывается метод onRefreshButtonClicked, который обновляет таблицу и сбрасывает состояние поиска. Исходный код представлен в листинге 2.

Листинг 2. Метод onRefreshButtonClicked

void MainWindow::onRefreshButtonClicked() {

QAbstractItemModel\* ui\_model = ui\_->mainTableView->model();

if (ui\_model != tableData\_->getModel() && ui\_model != nullptr) {

delete ui\_model;

}

ui\_->mainTableView->setModel(tableData\_->getModel());

ui\_->criterionComboBox->setCurrentIndex(0);

ui\_->searchLineEdit->clear();

statusBar()->showMessage("Таблица обновлена");

}

Кнопка addButton вызывает метод onAddButtonClicked, который в свою очередь открывает форму для добавления записи в таблицу AddItemDialog. Исходный код в листинге 3.

Листинг 3. Метод onAddButtonClicked

void MainWindow::onAddButtonClicked() {

AddItemDialog dialog(this);

dialog.exec();

if (dialog.result() == QDialog::Accepted) {

QVector<QString> data = {dialog.getAuthor(), dialog.getTheme(),

dialog.getPhrase()};

tableData\_->addRow(data);

refreshTable();

statusBar()->showMessage("Запись успешно добавлена в таблицу");

}

}

Кнопка removeButton удаляет выделенную строку в таблице. Удаление происходит по идентификатору (ID) записи. Исходный код представлен в листинге 4.

Листинг 4. Метод onRemoveButtonClicked

void MainWindow::onRemoveButtonClicked() {

QModelIndexList selected\_rows =

ui\_->mainTableView->selectionModel()->selectedRows();

if (selected\_rows.size() != 1) {

buttonMessageBox("Выберите строку для удаления");

return;

}

int id = selected\_rows.first().data().toInt();

tableData\_->removeRow(id);

refreshTable();

statusBar()->showMessage("Запись успешно удалена");

}

Кнопка editButton вызывает форму редактирования выделенной записи таблицы EditItemDialog. Нужная запись также выбирается по идентификатору. Исходный код в листинге 5.

Листинг 5. Метод onEditButtonClicked

void MainWindow::onEditButtonClicked() {

QModelIndexList selected\_rows =

ui\_->mainTableView->selectionModel()->selectedRows();

if (selected\_rows.size() != 1) {

buttonMessageBox("Выберите строку для редактирования");

return;

}

int id = selected\_rows.first().data().toInt();

QVector<QString> data = tableData\_->getRowData(id);

EditItemDialog dialog(data, this);

dialog.exec();

if (dialog.result() == QDialog::Accepted) {

QVector<QString> data = {dialog.getAuthor(), dialog.getTheme(),

dialog.getPhrase()};

tableData\_->editRow(data, id);

refreshTable();

statusBar()->showMessage("Запись в таблице успешно изменена");

}

}

Кнопка copyButton копирует выделенную запись таблицы в буфер обмена. Код в листинге 6.

Листинг 6. Метод onCopyButtonClicked

void MainWindow::onCopyButtonClicked() {

QModelIndexList selected\_rows =

ui\_->mainTableView->selectionModel()->selectedRows();

if (selected\_rows.size() != 1) {

buttonMessageBox("Выберите строку для копирования");

return;

}

int id = selected\_rows.first().data().toInt();

QVector<QString> data = tableData\_->getRowData(id);

QString result = data[1];

for (int i = 2; i < data.size(); ++i) {

result += '\t' + data[i];

}

QApplication::clipboard()->setText(result);

statusBar()->showMessage("Строка скопирована");

}

Кнопка clearButton полностью очищает таблицу от записей. Исходный код представлен в листинге 7.

Листинг 7. Метод onClearButtonClicked

void MainWindow::onClearButtonClicked() {

QAbstractItemModel\* model = tableData\_->clearModel();

if (model) {

QAbstractItemModel\* ui\_model = ui\_->mainTableView->model();

if (ui\_model != nullptr) {

delete ui\_model;

}

ui\_->mainTableView->setModel(model);

statusBar()->showMessage("Таблица успешно очищена");

} else {

statusBar()->showMessage("Ошибка очистки таблицы");

}

}

Выбор в criterionComboBox меняет критерий поиска, а также устанавливает столбец для QCompleter (о нем будет сказано позднее). Метод представлен в листинге 8.

Листинг 8. Метод onCriterionComboBoxActivated

void MainWindow::onCriterionComboBoxActivated(int index) {

compl\_->setCompletionColumn(index + 1);

}

Кнопка searchButton вызывает метод onSearchButtonClicked, который делегирует свою работу методу refreshTable. Исходный код этих двух методов представлен в листинге 9.

Листинг 9. Метод onSearchButtonClicked и refreshTable

void MainWindow::onSearchButtonClicked() { refreshTable(); }

void MainWindow::refreshTable() {

QAbstractItemModel\* ui\_model = ui\_->mainTableView->model();

if (ui\_model != tableData\_->getModel() && ui\_model != nullptr) {

delete ui\_model;

}

QString criterion = ui\_->criterionComboBox->currentText();

QString search\_string = ui\_->searchLineEdit->text();

if (search\_string.isNull() || search\_string.isEmpty()) {

QAbstractItemModel\* model = tableData\_->getModel();

ui\_->mainTableView->setModel(model);

compl\_->setModel(model);

return;

}

QAbstractItemModel\* model = tableData\_->search(criterion, search\_string);

ui\_->mainTableView->setModel(model);

compl\_->setModel(model);

}

Вызов меню actionSave позволяет выбрать место для сохранения данных таблицы в формате json. Код в листинге 10.

Листинг 10. Метод onSaveFileClicked

void MainWindow::onSaveFileClicked() {

QString path = QFileDialog::getSaveFileName(this, "Сохранить файл", "", "Фразы (\*.json)");

if (path.isNull() || path.isEmpty())

return;

if (tableData\_->saveToJson(path)) {

statusBar()->showMessage("Файл успешно сохранен");

} else {

statusBar()->showMessage("Ошибка сохранения файла");

}

}

Вызов меню actionLoad открывает окно с выбором файла для загрузки данных таблицы в формате json. Код представлен в листинге 11.

Листинг 11. Метод onLoadFileClicked

void MainWindow::onLoadFileClicked() {

QString path = QFileDialog::getOpenFileName(this, "Загрузить файл", "", "Фразы (\*.json)");

if (path.isNull() || path.isEmpty()) {

return;

}

QAbstractItemModel\* model = tableData\_->loadFromJson(path);

if (model) {

ui\_->mainTableView->setModel(model);

compl\_->setModel(model);

statusBar()->showMessage("Файл успешно загружен");

} else {

statusBar()->showMessage("Ошибка загрузки файла");

}

}

Вызов меню actionHelp открывает окно с подсказками для пользователя HelpWindow. Исходный код вызывающего метода в листинге 12.

Листинг 12. Метод onHelpActionClicked

void MainWindow::onHelpActionClicked() {

HelpWindow wnd(this);

wnd.exec();

}

Вызов меню actionAbout открывает окно AboutWindow с информацией об авторе и самой программе. Код метода в листинге 13.

Листинг 13. Метод onAboutActionClicked

void MainWindow::onAboutActionClicked() {

AboutWindow wnd(this);

wnd.exec();

}

Вспомогательный метод для вызова окон с сообщениями о различных ошибках типа QMessageBox. Исходный код метода в листинге 14.

Листинг 14. Метод buttonMessageBox

void MainWindow::buttonMessageBox(const QString& text) {

QMessageBox msg\_box;

msg\_box.setWindowTitle("Таблица");

msg\_box.setWindowIcon(windowIcon());

msg\_box.setIcon(QMessageBox::Information);

msg\_box.setText(text);

msg\_box.exec();

}

# РАЗРАБОТКА ОСНОВНОГО МЕХАНИЗМА ФУНКЦИОНИРОВАНИЯ ПРИЛОЖЕНИЯ

В данном разделе описываются методы классов, в которых происходит сохранение или загрузка файла и изменение данных приложения:

* Добавление элемента
* Удаление элемента
* Изменение элемента
* Поиск по заданному критерию
* Чтение файла
* Запись в файл

Добавление записи в таблицу начинается с вызова и заполнения формы AddItemDialog (рисунок 3). Данные отправляются в метод addRow класса TableData, который и взаимодействует с моделью таблицы QAbstractItemModel. Модель самостоятельно реагирует на изменения и отображается в главном окне. Также для возможности поиска по критериям заполняется бор, реализованный в классе PrefixSearcher, о котором будет рассказано далее. Реализованный метод представлен в листинге 15.

Листинг 15. Метод addRow

bool TableData::addRow(const QVector<QString>& columns) {

if (columns.size() > model\_->columnCount()) {

return false;

}

int row\_count = model\_->rowCount();

int last\_id = lastID();

model\_->insertRow(row\_count);

model\_->setData(model\_->index(row\_count, 0), last\_id);

for (int i = 0; i < columns.size(); ++i) {

QModelIndex index = model\_->index(row\_count, i + 1);

model\_->setData(index, columns[i]);

criterionSearch\_[headers\_[i + 1]]->insert(columns[i], last\_id);

}

return true;

}

Удаление выбранной записи осуществляется нажатием кнопки removeButton (событие в листинге 4). Вызывается метод removeRow с параметром идентификатора выбранной строки. Запись также удаляется из поиска. Метод реализован в листинге 16.

Листинг 16. Метод removeRow

bool TableData::removeRow(int id) {

int row = findRow(id);

for (int i = 1; i < headers\_.size(); ++i) {

QModelIndex index = model\_->index(row, i);

QString data = model\_->data(index).toString();

criterionSearch\_[headers\_[i]]->erase(data, id);

}

return model\_->removeRow(row);

}

Изменение записи происходит в форме EditItemDialog (рисунок 4), вызванный кнопкой editButton (событие в листинге 5). В метод editRow отправляются измененные данные и идентификатор строки. Изменение также касается поиска. Исходный код в листинге 17.

Листинг 17. Метод editRow

void TableData::editRow(const QVector<QString>& columns, int id) {

if (columns.size() > model\_->columnCount()) {

return;

}

int row = findRow(id);

for (int i = 0; i < columns.size(); ++i) {

QModelIndex index = model\_->index(row, i + 1);

QString data = model\_->data(index).toString();

criterionSearch\_[headers\_[i + 1]]->erase(data, id);

criterionSearch\_[headers\_[i + 1]]->insert(columns[i], id);

model\_->setData(index, columns[i]);

}

}

Поиск осуществляется путем выбора критерия для поиска из criterionComboBox, ввода данных для поиска в searchLineEdit и нажатием кнопки searchButton. Выпадающий список при вводе в searchLineEdit, или же QCompleter, самостоятельно адаптируется под выбор критерия и предлагает подходящие варианты поиска. В метод search отправляется выбранный критерий и строка для поиска, для этого используется класс PrefixSearcher. Данный метод возвращает новую модель (QAbstractModel) для отображения в таблице. Исходный код представлен в листинге 18.

Листинг 18. Метод search

QAbstractItemModel\* TableData::search(const QString& header, const QString& search\_string) {

QVector<int> ids = criterionSearch\_[header]->find(search\_string);

QAbstractItemModel\* searchModel\_ =

new QStandardItemModel(ids.size(), headers\_.size());

for (int i = 0; i < headers\_.size(); ++i) {

searchModel\_->setHeaderData(i, Qt::Horizontal, headers\_[i]);

}

for (int i = 0; i < ids.count(); ++i) {

for (int j = 0; j < model\_->rowCount(); ++j) {

QModelIndex index = model\_->index(j, 0);

int id = model\_->data(index).toInt();

if (id == ids[i]) {

for (int k = 0; k < model\_->columnCount(); ++k) {

index = model\_->index(j, k);

QString data = model\_->data(index).toString();

QModelIndex index\_search = searchModel\_->index(i, k);

searchModel\_->setData(index\_search, data);

}

break;

}

}

}

return searchModel\_;

}

Класс PrefixSearcher реализует такую структуру данных как «Префиксное дерево» или же по-другому «Бор». Бор — структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца. Структура класса была представлена в листинге 1, а сам класс хранится и объявлен в классе TableData как хэш таблица QHash<QString, PrefixSearcher\*>criterionSearch\_ для удобства обращения к нужным критериям с типом QString. В листингах 19 и 20 представлены методы вставки и удаления.

Листинг 19. Метод вставки в префиксное дерево

void PrefixSearcher::insert(const QString& str, int input\_id) {

int vertice = 0;

for (QChar sym : str) {

if (!canGoNode(vertice, sym)) {

createNode(vertice, sym);

}

vertice = getNextVertice(vertice, sym);

}

trie\_[vertice].isLeaf = true;

trie\_[vertice].ids.push\_back(input\_id);

}

Листинг 20. Метод удаления в префиксном дереве

void PrefixSearcher::erase(const QString& str, int erase\_id) {

int vertice = 0;

for (QChar sym : str) {

if (!canGoNode(vertice, sym)) {

return;

}

vertice = getNextVertice(vertice, sym);

}

Node& node = trie\_[vertice];

auto it = std::find(node.ids.begin(), node.ids.end(), erase\_id);

if (it != node.ids.end()) {

node.ids.erase(it);

}

node.isLeaf = node.ids.size();

}

Так как это префиксное дерево, в нем можно найти все строки, начинающиеся определенной подстрокой. Для данной задачи идеально подойдет такой алгоритм обхода графов (в нашем случае дерева) как поиск в глубину (или же DFS – Depth First Search). Метод поиска, который будет собирать результат и начнет вызов рекурсивного поиска в глубину представлен в листинге 21, а сам алгоритм DFS в листинге 22.

Листинг 21. Метод поиска всех подстрок в префиксном дереве

QVector<int> PrefixSearcher::find(const QString& prefix) {

int vertice = 0;

for (QChar sym : prefix) {

if (!canGoNode(vertice, sym)) {

return QVector<int>();

}

vertice = getNextVertice(vertice, sym);

}

QVector<int> result;

depthFirstSearch(result, vertice);

std::sort(result.begin(), result.end());

return result;

}

Листинг 22. Алгоритм поиска в глубину (DFS)

void PrefixSearcher::depthFirstSearch(QVector<int>& result, int vertice) {

if (isLeaf(vertice)) {

for (int id : trie\_[vertice].ids) {

result.push\_back(id);

}

}

QList vertices = trie\_[vertice].nextNodes.values();

for (int& next\_vertice : vertices) {

depthFirstSearch(result, next\_vertice);

}

}

Чтение файла начинается с выбора файла для чтения с помощью встроенного QFileDialog (листинг 10). Далее путь до файла отправляется в метод saveToJson, описанный в листинге 23. В этом методе проверяется, возможно ли открыть файл, и имеет ли он параметр TABLE\_DATA. Данный параметр указывает на то, что файл был сохранен именно этой программой и имеет правильную архитектуру формата json и поля для считывания данных. Данные заполняются в модель таблицы.

Листинг 23. Метод чтения и заполнения таблицы из файла json

QAbstractItemModel\* TableData::loadFromJson(const QString& path) {

QFile file(path);

if (!file.open(QIODevice::ReadOnly)) {

return nullptr;

}

QJsonDocument load\_doc(QJsonDocument::fromJson(file.readAll()));

if (!load\_doc.object().contains("TABLE\_DATA")) {

return nullptr;

}

QJsonArray array = load\_doc.object()["TABLE\_DATA"].toArray();

if (model\_ != nullptr) {

delete model\_;

}

model\_ = new QStandardItemModel(array.size(), headers\_.size());

initHeaders();

clearSearcher();

initSearcher();

for (int i = 0; i < array.size(); ++i) {

int id = array[i].toObject()[headers\_[0]].toString().toInt();

for (int j = 0; j < headers\_.size(); ++j) {

QModelIndex index = model\_->index(i, j);

QString data = array[i].toObject()[headers\_[j]].toString();

model\_->setData(index, data);

if (j > 0) {

criterionSearch\_[headers\_[j]]->insert(data, id);

}

}

}

return model\_;

}

Запись в файл происходит аналогичным образом. В файл json обязательно устанавливается опция TABLE\_DATA, описанная выше. Метод приведен в листинге 24.

Листинг 24. Метод сохранения данных таблицы в файл json

bool TableData::saveToJson(const QString& path) {

QFile file(path);

if (!file.open(QIODevice::WriteOnly)) {

return false;

}

QJsonArray array;

for (int i = 0; i < model\_->rowCount(); ++i) {

QJsonObject obj;

for (int j = 0; j < model\_->columnCount(); ++j) {

QModelIndex index = model\_->index(i, j);

QString data = model\_->data(index).toString();

obj[headers\_[j]] = data;

}

array.push\_back(obj);

}

QJsonObject save\_data;

save\_data.insert("TABLE\_DATA", array);

file.write(QJsonDocument(save\_data).toJson());

return true;

}

# ТЕСТИРОВАНИЕ

## Автономное тестирование

В данном разделе приводятся протоколы тестирования всех основных механизмов, приведенных в главе 4: чтение, редактирование, вставка (вставка в хвост), поиск по разным ключам, удаление элемента. Необходимо для каждого случая предусмотреть несколько принципиально различных наборов входных данных, не обязательно корректных. Количество наборов данных зависит от количества возвращаемых кодов ошибок и операторов возврата из функции.

Таблица 7. Тестирование операции … (пример заполнения таблицы)

|  |  |  |
| --- | --- | --- |
| **Входные данные** | **Выходные данные** | **Верно ли?** |
| Перечисляются все параметры, передаваемые в функцию (корректные) | Приводится возвращаемое значение | **Да** |
| Приводится набор данных, приводящих к аварийному завершению функции | Возвращаемый код ошибки | **Да** |
| Приводится набор данных, позволяющий добраться до каждого размещенного в функции оператора return. | Приводится возвращаемое значение | **Да** |

## Комплексное тестирование

В данном разделе приводятся протоколы тестирования приложения. В таблице необходимо отразить следующие ситуации. Для каждой из них возможно указать несколько тестов.

1. Программа работает корректно для введенных данных.
2. Пользователь ввел некорректные данные (либо попытался ввести некорректные данные)
3. Пользователь завершил работу системного диалога и отказался от выбора имени файла.
4. Имя входного/выходного файла задано некорректно. Входной/выходной файл не открывается.
5. Невозможно загрузить файл помощи.
6. Невозможно открыть заставку приложения.
7. Прочие критические и аварийные ситуации.

Полученный результат описывается текстом, но допускается приведение скриншотов оконных форм.

Табл.5. Тестирование головного модуля (примеры заполнения таблицы).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № п/п | Описание ситуации | Входные данные | Выходные данные | Тест пройден? |
| 1. | Программа работает корректно | Источник входных данных | Описание полученного результата | Да |
| 2. | Невозможно открыть заставку приложения | Путь к файлу с заставкой | Описание полученного результата, текст выводимого сообщения | Да |
| 3. | Невозможно загрузить файл помощи | Путь к файлу помощи | текст выводимого сообщения об ошибке | Да |

# РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Данный раздел следует разбить на следующие пункты:

1. Запуск приложения, приветственные экраны, какие ресурсы необходимы для выполнения программы.
2. Различные способы ввода и вывода информации.
3. Описание всех возможностей программы.
4. Перечень ошибок и способов их устранения.

Следует отметить, что пользователь может быть не знаком с языками программирования и объектно-ориентированным подходом. Пользователь – это человек, который запускает исполняемый файл и работает с ним. От пользователя следует скрывать все детали реализации, поскольку эта информация для него не актуальна. Чтобы было легче написать данный раздел представьте, что вы рассказываете о функционировании своей программы потенциальному покупателю и показываете ему все возможности вашей разработки.

# ЗАКЛЮЧЕНИЕ

Данная работа посвящена разработке оконного приложения <вставить название либо описать основной функционал> с использованием языка C++. Для хранения данных использованы объекты класса <вставить название>, являющегося наследником класса List, приведенного в задании. При организации функционирования класса была создана перегруженная функция <вставить название>, используемая для поиска элемента списка с ключами разных типов, а также с различными наборами ключевых полей.

В ходе выполнения работы были пройдены все основные этапы разработки программного обеспечения: анализ, написание спецификации, проектирование, разработка алгоритмов, кодирование, тестирование и сопровождение. Проведено автономное тестирование основных методов разработанного класса и комплексное тестирование программы в целом. В результате тестирования ошибок не обнаружено. В данных пунктах описана проделанная в каждом случае работа и ее результаты.

Привести направления дальнейшей разработки. Что в разработке может быть расширено, улучшено, оптимизировано?

# ЛИТЕРАТУРА

1. Панюкова Т.А., Панюков А.В. Языки и методы программирования: Путеводитель по языку С++. Учебное пособие. – М.: Книжный дом «Либроком», 2013. – 216 с.
2. Панюкова Т.А. Документирование программного обеспечения: В помощь техническому писателю: Учебное пособие. – М.: Книжный дом «Либроком», 2012. – 264 с.
3. Пахомов Б. И. C/C++ и MS Visual C++ 2012 для начинающих. — СПб.: БХВ-Петербург, 2013. – 512 с.