**Пермский Национальный Исследовательский Политехнический Университет**

Творческая работа:

«Автоматизированное рабочее место наставника »

*Выполнил:*

*Ившин Максим Сергеевич РИС-23- 2б*

*Коротаев Александр Дмитриевич РИС-23-3Б*

*Проверила:*

*доцент кафедры ИТАС О.А. Полякова*

2024

**Целью данной работы** является разработка автоматизированной тестирующей системы консольных приложений на C++, с использованием компилятора g++.

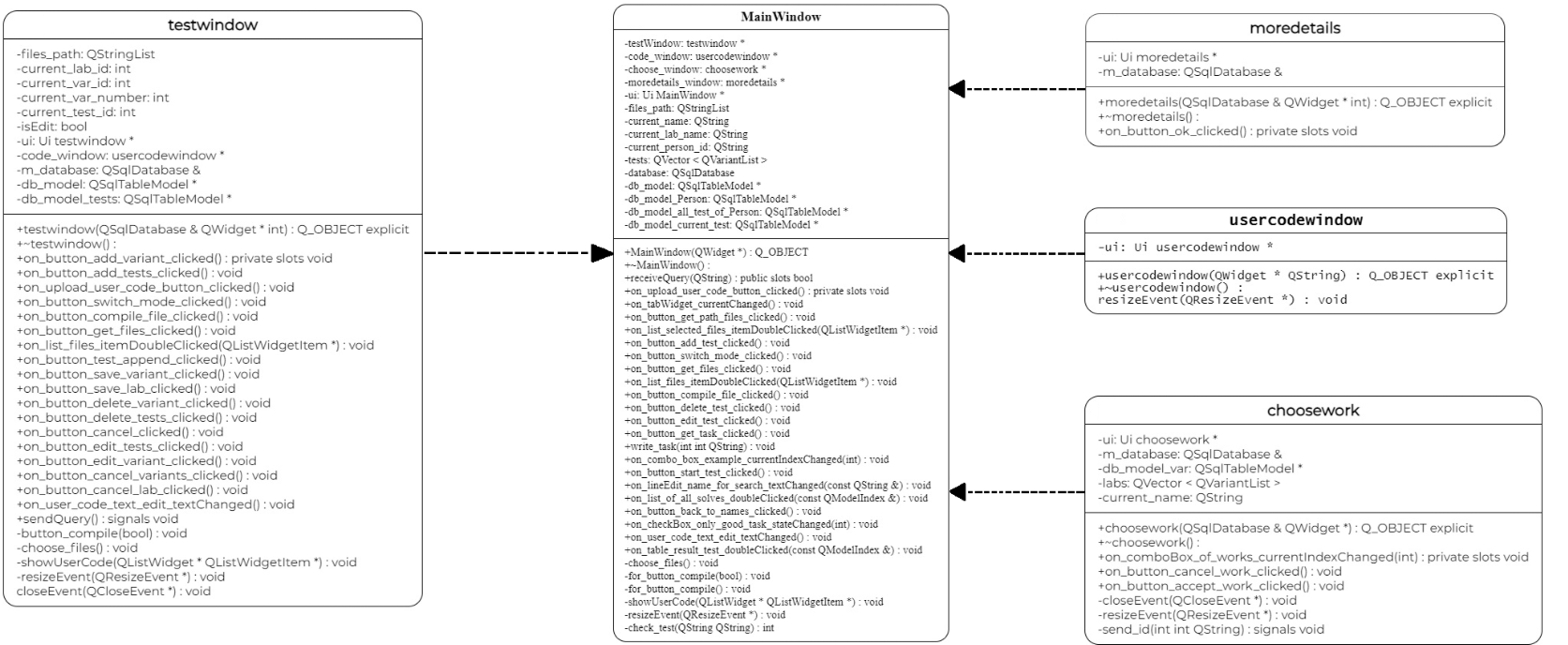
**Постановка задачи:**

Разработать приложение, которое позволит компилировать пользовательский код, создавать и редактировать тесты, для проверок правильности написанного кода на C++.

Анализ задачи:

1. Приложение представляет собой реализацию графического интерфейса для работы с базой данной, включая добавление и удаление лабораторных работ, создание и удаление тестов.
2. Код написан на C++ с использованием фреймворка Qt для создания графического интерфейса и работы с графическими элементами.
3. Класс MainWindow - основное окно, в котором реализованы функции для компиляции кода и проверки его на корректность в рамках выбранной лабораторной работы. Есть возможность пользователю проверить свой код (как в текстовом формате, так и уже готовые файлы с кодом) на своих тестах, а также в этом же окне, можно посмотреть информацию о всех проверках.
4. Класс testwindow - позволяет добавлять, удалять и редактировать лабораторные работы, варианты и тесты. Добавление тестов реализовано как через ручной ввод входных и выходных данных, так и через компиляцию уже готового пользовательского кода. Все добавленные элементы хранятся в соответствующих таблицах базы данных.
5. Класс usercodewindow представляет собой окно, которое позволяет пользователю получить информацию о выбранных им файлах с кодом, т.е. показывает пользователю содержимое выбранных файлов в диалоговом окне.
6. Класс moredetails представляет собой окно, которое дает больше информации о проверках лабораторных работ, например, на каком тесте произошел сбой и почему.
7. Класс choosework представляет собой окно, которое нужно, чтобы пользователь смог выбрать лабораторную работу, которую хочет проверить, при выборе работы из базы данных выбираются варианты для выбранной работы, а для этих вариантов выбираются соответствующие тесты для проверки пользовательского кода на корректность.
8. Вспомогательные файлы utils.h и database\_operations.h содержат полезные вспомогательные функции. В database\_operations.h реализованы функции для работы с базой данных, такие как вставка, удаление и выборка данных. В utils.h содержатся полезные функции для работы с файлами, например открытие файла, чтение и запись данных в файл.
9. Результат проекта – технология автоматизации проверки знаний и умений начинающих C++ разработчиков в сфере алгоритмизации различных проектов.

**UML Диаграммы:**



**Код**

mainwindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QListWidgetItem>

#include "textwindow.h"

#include "testwindow.h"

#include "choosework.h"

#include "moredetails.h"

#include <QSqlDatabase>

#include <QSqlError>

#include <QFileDialog>

#include <QMessageBox>

#include <QSqlTableModel>

#include <QDebug>

#include <QResizeEvent>

#include <memory>

#include "realization/database\_operations.h"

QT\_BEGIN\_NAMESPACE

*namespace* **Ui** { *class* **MainWindow**; }

QT\_END\_NAMESPACE

*class* **MainWindow** : *public* QMainWindow

{

Q\_OBJECT

*public*:

**MainWindow**(QWidget \*parent = *nullptr*);

~***MainWindow***();

*public* slots:

bool **receiveQuery**(QString query);

*private* slots:

void **on\_upload\_user\_code\_button\_clicked**();

void **on\_tabWidget\_currentChanged**();

void **on\_button\_get\_path\_files\_clicked**();

void **on\_list\_selected\_files\_itemDoubleClicked**(QListWidgetItem \*item);

void **on\_button\_add\_test\_clicked**();

void **on\_button\_switch\_mode\_clicked**();

void **on\_button\_get\_files\_clicked**();

void **on\_list\_files\_itemDoubleClicked**(QListWidgetItem \*item);

void **on\_button\_compile\_file\_clicked**();

void **on\_button\_delete\_test\_clicked**();

void **on\_button\_edit\_test\_clicked**();

void **on\_button\_get\_task\_clicked**();

void **write\_task**(int id\_lab, int id\_var, QString name);

void **on\_combo\_box\_example\_currentIndexChanged**(int index);

void **on\_button\_start\_test\_clicked**();

void **on\_lineEdit\_name\_for\_search\_textChanged**(*const* QString &arg1);

void **on\_list\_of\_all\_solves\_doubleClicked**(*const* QModelIndex &index);

void **on\_button\_back\_to\_names\_clicked**();

void **on\_checkBox\_only\_good\_task\_stateChanged**(int arg1);

void **on\_user\_code\_text\_edit\_textChanged**();

void **on\_table\_result\_test\_doubleClicked**(*const* QModelIndex &index);

*private*:

void **choose\_files**();

void **for\_button\_compile**(bool isWorkWithFile);

void **for\_button\_compile**();

void **showUserCode**(QListWidget\*, QListWidgetItem\*);

void ***resizeEvent***(QResizeEvent \*event);

int **check\_test**(QString input\_data, QString waiting\_output\_data);

testwindow \*testWindow;

usercodewindow \*code\_window;

choosework \*choose\_window;

moredetails \*moredetails\_window;

Ui::MainWindow \*ui;

*//* *пути* *до* *выбранных* *файлов*

QStringList files\_path;

*//* *имя* *в* *текущей* *проверке*

QString current\_name;

QString current\_lab\_name;

QString current\_person\_id;

*//* *список* *всех* *тестов* *для* *текущей* *работы*

QVector<QVariantList> tests;

*//* *для* *работы* *с* *базой* *данных*

QSqlDatabase database;

QSqlTableModel \*db\_model;

QSqlTableModel \*db\_model\_Person;

QSqlTableModel \*db\_model\_all\_test\_of\_Person;

QSqlTableModel \*db\_model\_current\_test;

};

#endif *//* *MAINWINDOW\_H*

Mainwindow.cpp

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include "realization//tab\_only\_code.cpp"

MainWindow::**MainWindow**(QWidget \*parent) : QMainWindow(*parent*), ui(*new* Ui::MainWindow)

{

ui->setupUi(*this*);

ui->statusbar->showMessage("Обратите внимание, что путь к файлам и их название не должны содержать кириллицу!");

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

*if* (tryToOpenDB(database, "Lab\_works"))

{

db\_model = *new* QSqlTableModel(*this*, database);

db\_model\_Person = *new* QSqlTableModel(*this*, database);

db\_model\_current\_test = *new* QSqlTableModel(*this*, database);

db\_model\_all\_test\_of\_Person = *new* QSqlTableModel(*this*, database);

db\_model->*setTable*("LabWork");

db\_model\_Person->*setTable*("Person");

db\_model\_current\_test->*setTable*("CheckedWorks");

db\_model\_all\_test\_of\_Person->*setTable*("CheckedWorks");

ui->list\_tests->resizeColumnsToContents();

ui->list\_of\_all\_solves->resizeColumnsToContents();

ui->table\_result\_test->resizeColumnsToContents();

db\_model->*select*();

db\_model\_Person->*setSort*(1, Qt::*DescendingOrder*); *//* *сортировка* *по* *дате*

db\_model\_Person->*select*();

db\_model->*setHeaderData*(1, Qt::*Horizontal*, tr("Название работы"));

db\_model\_Person->*setHeaderData*(1, Qt::*Horizontal*, tr("Дата последней проверки"));

db\_model\_Person->*setHeaderData*(2, Qt::*Horizontal*, tr("Имя"));

db\_model\_current\_test->*setHeaderData*(5, Qt::*Horizontal*, tr("Статус проверки"));

db\_model\_all\_test\_of\_Person->*setHeaderData*(2, Qt::*Horizontal*, tr("Дата проверки"));

db\_model\_all\_test\_of\_Person->*setHeaderData*(3, Qt::*Horizontal*, tr("Название работы"));

db\_model\_all\_test\_of\_Person->*setHeaderData*(5, Qt::*Horizontal*, tr("Статус проверки"));

ui->list\_tests->*setModel*(*db\_model*);

ui->list\_tests->setEditTriggers(QAbstractItemView::*NoEditTriggers*);

ui->list\_of\_all\_solves->*setModel*(*db\_model\_Person*);

ui->list\_of\_all\_solves->setEditTriggers(QAbstractItemView::*NoEditTriggers*);

ui->table\_result\_test->*setModel*(*db\_model\_current\_test*);

ui->table\_result\_test->setEditTriggers(QAbstractItemView::*NoEditTriggers*);

ui->list\_tests->setColumnHidden(0, *true*);

ui->list\_tests->setColumnHidden(2, *true*);

ui->list\_of\_all\_solves->setColumnHidden(0, *true*);

ui->table\_result\_test->setColumnHidden(0, *true*);

ui->table\_result\_test->setColumnHidden(1, *true*);

ui->table\_result\_test->setColumnHidden(2, *true*);

ui->table\_result\_test->setColumnHidden(3, *true*);

ui->table\_result\_test->setColumnHidden(4, *true*);

ui->table\_result\_test->setColumnHidden(6, *true*);

ui->table\_result\_test->setColumnHidden(7, *true*);

ui->table\_result\_test->setColumnHidden(8, *true*);

}

ui->text\_input\_example->setPlainText("Выберите работу");

ui->text\_output\_example->setPlainText("Выберите работу");

ui->lineEdit\_name\_for\_search->setPlaceholderText("Введите имя для поиска");

QPalette pal = ui->lineEdit\_name\_for\_search->palette();

pal.setColor(QPalette::*Text*, QColor(140, 140, 140));

ui->lineEdit\_name\_for\_search->setPalette(pal);

ui->checkBox\_only\_good\_task->hide();

ui->button\_back\_to\_names->hide();

ui->label\_name->hide();

ui->label\_name\_change->hide();

}

MainWindow::~***MainWindow***()

{

*delete* ui;

}

*//* *очистка* *интерфейса* *при* *смене* *вкладки*

void MainWindow::**on\_tabWidget\_currentChanged**()

{

*if* (ui->tabWidget->currentIndex() == 0)

{

db\_model\_current\_test->*setFilter*("id = -1");

db\_model\_current\_test->*select*();

ui->combo\_box\_example->setCurrentIndex(0);

ui->text\_info\_task->clear();

tests.clear();

*if* (ui->tabWidget->currentIndex() == 0)

ui->statusbar->showMessage("Обратите внимание, что путь к файлам и их название не должны содержать кириллицу!");

}

*else* *if* (ui->tabWidget->currentIndex() == 1)

{

ui->list\_tests->setColumnWidth(1, ui->list\_tests->width() + 138);

}

*else* *if* (ui->tabWidget->currentIndex() == 2)

{

ui->to\_user\_output\_data->setText("");

ui->user\_input\_data->clear();

*if* (ui->stacked\_widget\_modes->currentIndex())

ui->statusbar->showMessage("Обратите внимание, что путь к файлам и их название не должны содержать кириллицу!");

}

*else* *if* (ui->tabWidget->currentIndex() == 3)

{

on\_button\_back\_to\_names\_clicked();

}

ui->statusbar->showMessage("");

}

*//* *добавление* *работы* *по* *нажатию* *кнопки*

void MainWindow::**on\_button\_add\_test\_clicked**()

{

testWindow = *new* testwindow(*this->database*, *this*);

testWindow->setWindowTitle("Добавление");

connect(testWindow, &testwindow::sendQuery, *this*, &MainWindow::receiveQuery);

testWindow->setModal(*true*);

testWindow->*exec*();

db\_model->*select*();

}

void MainWindow::**on\_button\_switch\_mode\_clicked**()

{

*//* *очистка* *статус* *бара* *\* *виджетов* *ввода/вывода* *программы*

ui->statusbar->showMessage("");

ui->to\_user\_output\_data->clear();

ui->user\_input\_data->clear();

*if* (ui->stacked\_widget\_modes->currentIndex())

{

ui->stacked\_widget\_modes->setCurrentWidget(*ui->page\_code\_mode*);

ui->button\_switch\_mode->setText("Работа с файлами");

}

*else*

{

ui->stacked\_widget\_modes->setCurrentWidget(*ui->page\_file\_mode*);

ui->button\_switch\_mode->setText("Работа с кодом");

ui->statusbar->showMessage("Обратите внимание, что путь к файлам и их название не должны содержать кириллицу!");

}

}

bool MainWindow::**receiveQuery**(QString text\_query)

{

database.exec(text\_query);

*if* (database.lastError().type() != QSqlError::*NoError*)

{

qDebug() << "Ошибка при выполнении запроса: " << database.lastError().text();

*return* *false*;

}

*return* *true*;

}

*//* *удаление* *записи* *из* *списка* *с* *тестами* *и* *базы* *данных*

void MainWindow::**on\_button\_delete\_test\_clicked**()

{

int current\_row = ui->list\_tests->currentIndex().row();

int labwork\_id = db\_model->*index*(current\_row, 0).data().toInt();

*if* (current\_row != -1)

{

QMessageBox messageBox(QMessageBox::*Question*,

tr("Вы уверены?"),

tr("Вы уверены, что хотите удалить\nнавсегда и безвозвратно?"),

QMessageBox::*Yes* | QMessageBox::*No*,

*this*);

messageBox.setButtonText(QMessageBox::*Yes*, tr("Да"));

messageBox.setButtonText(QMessageBox::*No*, tr("Нет"));

*if* (messageBox.*exec*() == QMessageBox::*Yes*)

{

*//* *Удаление* *зависимостей* *из* *таблицы* *Tests*

QSqlQuery query;

query.prepare("DELETE FROM Tests WHERE variant\_id IN (SELECT id FROM Variants WHERE labwork\_id = :labwork\_id)");

query.bindValue(":labwork\_id", labwork\_id);

query.exec();

*//* *Удаление* *зависимостей* *из* *таблицы* *Variants*

query.prepare("DELETE FROM Variants WHERE labwork\_id = :labwork\_id");

query.bindValue(":labwork\_id", labwork\_id);

query.exec();

*//* *Удаление* *записи* *из* *таблицы* *LabWork*

db\_model->removeRow(current\_row);

db\_model->*select*();

}

}

*else*

{

QMessageBox::warning(*this*, "Ошибка", "Не выбран ни 1 элемент.");

*return*;

}

}

void MainWindow::***resizeEvent***(QResizeEvent \*event)

{

ui->list\_tests->setColumnWidth(1, ui->list\_tests->width() + 138);

*if* (ui->button\_back\_to\_names->isHidden())

{

ui->list\_of\_all\_solves->setColumnWidth(1, (*this*->width()-48) \* 0.3);

ui->list\_of\_all\_solves->setColumnWidth(2, (*this*->width()-48) \* 0.7 - 25);

}

*else*

{

ui->list\_of\_all\_solves->setColumnWidth(2, (*this*->width()-48) \* 0.3);

ui->list\_of\_all\_solves->setColumnWidth(3, (*this*->width()-48) \* 0.5 - 25);

ui->list\_of\_all\_solves->setColumnWidth(5, (*this*->width()-48) \* 0.2);

}

ui->table\_result\_test->setColumnWidth(5, ui->table\_result\_test->width() + 134);

event->accept();

}

void MainWindow::**on\_button\_edit\_test\_clicked**()

{

int current\_row = ui->list\_tests->currentIndex().row();

QModelIndex columnIndex = ui->list\_tests->model()->*index*(current\_row, 0);

int current\_lab\_id = ui->list\_tests->model()->*data*(columnIndex).toInt();

*if* (current\_lab\_id != 0)

{

testWindow = *new* testwindow(*this->database*, *this*, current\_lab\_id);

testWindow->setWindowTitle("Редактирование");

connect(testWindow, &testwindow::sendQuery, *this*, &MainWindow::receiveQuery);

testWindow->setModal(*true*);

testWindow->*exec*();

db\_model->*select*();

}

*else*

{

QMessageBox::warning(*this*, "Ошибка", "Не выбран ни 1 элемент.");

*return*;

}

}

void MainWindow::**on\_button\_get\_task\_clicked**()

{

db\_model\_current\_test->*setFilter*("id = -1");

db\_model\_current\_test->*select*();

ui->text\_input\_example->setPlainText("Выберите пример");

ui->text\_output\_example->setPlainText("Выберите пример");

ui->text\_info\_task->clear();

tests.clear();

ui->combo\_box\_example->setCurrentIndex(0);

choose\_window = *new* choosework(*this->database*);

choose\_window->setWindowTitle("Выбор работы");

choose\_window->setModal(*true*);

connect(choose\_window, &choosework::send\_id, *this*, &MainWindow::write\_task);

choose\_window->*exec*();

}

void MainWindow::**write\_task**(int id\_lab, int id\_var, QString name)

{

current\_name = name;

current\_lab\_name = getResult(getDBDataQuery("LabWork", "name", "id", QString::number(id\_lab))).toString();

QString lab\_task, var\_task;

lab\_task = getResult(getDBDataQuery("LabWork", "name", "id", QString::number(id\_lab))).toString() + "\n";

lab\_task += getResult(getDBDataQuery("LabWork", "description", "id", QString::number(id\_lab))).toString() + "\n\n\n";

var\_task = "Вариант №" + getResult(getDBDataQuery("Variants", "number\_var", "id", QString::number(id\_var))).toString() + "\n";

var\_task += getResult(getDBDataQuery("Variants", "conditions", "id", QString::number(id\_var))).toString();

QString task = lab\_task + var\_task;

ui->text\_info\_task->setPlainText(task);

tests = getAllInTableWhere(*this*->database, "Tests", "variant\_id", QString::number(id\_var));

}

void MainWindow::**on\_combo\_box\_example\_currentIndexChanged**(int index)

{

QString input\_text = "Входные данные:\n", output\_text = "Выходные данные:\n";

*if* (index == 0)

{

input\_text = "Выберите пример";

output\_text = "Выберите пример";

}

*else*

{

*if* (index - 1 < tests.size())

{

input\_text += tests[index-1][2].toString();

output\_text += tests[index-1][3].toString();

}

*else*

{

input\_text = "Примера пока нет";

output\_text = "Примера пока нет";

}

}

ui->text\_input\_example->setPlainText(input\_text);

ui->text\_output\_example->setPlainText(output\_text);

}

void MainWindow::**on\_lineEdit\_name\_for\_search\_textChanged**(*const* QString &arg1)

{

QPalette pal = ui->lineEdit\_name\_for\_search->palette();

*if* (arg1 != "")

{

pal.setColor(QPalette::*Text*, QColor(0, 0, 0));

ui->lineEdit\_name\_for\_search->setPalette(pal);

db\_model\_Person->*setFilter*(QString("person\_name LIKE '%%1%'").arg(arg1));

db\_model\_Person->*setSort*(1, Qt::*DescendingOrder*); *//* *сортировка* *по* *дате*

db\_model\_Person->*select*();

}

*else*

{

pal.setColor(QPalette::*Text*, QColor(140, 140, 140));

ui->lineEdit\_name\_for\_search->setPalette(pal);

db\_model\_Person->*setFilter*("");

db\_model\_Person->*setSort*(1, Qt::*DescendingOrder*); *//* *сортировка* *по* *дате*

db\_model\_Person->*select*();

}

}

void MainWindow::**on\_list\_of\_all\_solves\_doubleClicked**(*const* QModelIndex &index)

{

QModelIndex columnIndex = ui->list\_of\_all\_solves->model()->*index*(index.row(), 0);

QVariant data = ui->list\_of\_all\_solves->model()->*data*(columnIndex);

*if* (data.isValid())

{

*if* (!ui->lineEdit\_name\_for\_search->isHidden())

{

ui->lineEdit\_name\_for\_search->hide();

ui->button\_back\_to\_names->show();

ui->checkBox\_only\_good\_task->show();

ui->label\_name->show();

ui->label\_name\_change->show();

current\_person\_id = data.toString();

QString person\_name = getResult(getDBDataQuery("Person", "person\_name", "id", current\_person\_id)).toString();

ui->label\_name\_change->setText(person\_name);

ui->list\_of\_all\_solves->*setModel*(*db\_model\_all\_test\_of\_Person*);

ui->list\_of\_all\_solves->setEditTriggers(QAbstractItemView::*NoEditTriggers*);

ui->list\_of\_all\_solves->setColumnHidden(0, *true*);

ui->list\_of\_all\_solves->setColumnHidden(1, *true*);

ui->list\_of\_all\_solves->setColumnHidden(4, *true*);

ui->list\_of\_all\_solves->setColumnHidden(6, *true*);

ui->list\_of\_all\_solves->setColumnHidden(7, *true*);

ui->list\_of\_all\_solves->setColumnHidden(8, *true*);

ui->list\_of\_all\_solves->setColumnWidth(2, (*this*->width()-48) \* 0.3);

ui->list\_of\_all\_solves->setColumnWidth(3, (*this*->width()-48) \* 0.5 - 25);

ui->list\_of\_all\_solves->setColumnWidth(5, (*this*->width()-48) \* 0.2);

db\_model\_all\_test\_of\_Person->*setFilter*(QString("person\_id = %1").arg(current\_person\_id));

db\_model\_all\_test\_of\_Person->*setSort*(2, Qt::*DescendingOrder*); *//* *сортировка* *по* *дате*

db\_model\_all\_test\_of\_Person->*select*();

}

*else*

{

moredetails\_window = *new* moredetails(*database*, *this*, data.toInt());

moredetails\_window->setWindowTitle("Подробнее об отчете");

moredetails\_window->setModal(*true*);

moredetails\_window->*exec*();

}

}

}

void MainWindow::**on\_button\_back\_to\_names\_clicked**()

{

ui->lineEdit\_name\_for\_search->clear();

ui->label\_name\_change->clear();

ui->list\_of\_all\_solves->*setModel*(*db\_model\_Person*);

ui->list\_of\_all\_solves->setEditTriggers(QAbstractItemView::*NoEditTriggers*);

ui->list\_of\_all\_solves->showColumn(1);

ui->list\_of\_all\_solves->setColumnWidth(1, (*this*->width()-48) \* 0.3);

ui->list\_of\_all\_solves->setColumnWidth(2, (*this*->width()-48) \* 0.7 - 25);

db\_model\_Person->*setFilter*("");

db\_model\_Person->*setSort*(1, Qt::*DescendingOrder*); *//* *сортировка* *по* *дате*

db\_model\_Person->*select*();

ui->checkBox\_only\_good\_task->setCheckState(Qt::*Unchecked*);

ui->button\_back\_to\_names->hide();

ui->checkBox\_only\_good\_task->hide();

ui->label\_name->hide();

ui->label\_name\_change->hide();

ui->lineEdit\_name\_for\_search->show();

}

void MainWindow::**on\_checkBox\_only\_good\_task\_stateChanged**(int arg1)

{

*if* (arg1 == 2)

{

db\_model\_all\_test\_of\_Person->*setFilter*(QString("person\_id = '%1' AND status = '%2'").arg(current\_person\_id, QString("Зачтено")));

db\_model\_all\_test\_of\_Person->*setSort*(2, Qt::*DescendingOrder*); *//* *сортировка* *по* *дате*

db\_model\_all\_test\_of\_Person->*select*();

}

*else*

{

db\_model\_all\_test\_of\_Person->*setFilter*(QString("person\_id = %1").arg(current\_person\_id));

db\_model\_all\_test\_of\_Person->*setSort*(2, Qt::*DescendingOrder*); *//* *сортировка* *по* *дате*

db\_model\_all\_test\_of\_Person->*select*();

}

}

void MainWindow::**on\_user\_code\_text\_edit\_textChanged**()

{

QString text = ui->user\_code\_text\_edit->toPlainText();

QTextCursor cursor = ui->user\_code\_text\_edit->textCursor();

int cursorPosition = cursor.position();

*//* *если* *введен* *tab*

*if* (cursorPosition > 0 && text.at(cursorPosition - 1) == '\t')

{

*//* *удалить* *Tab*

cursor.deletePreviousChar();

*//* *вставить* *четыре* *пробела*

cursor.insertText(" ");

*//* *обновить* *курсор*

ui->user\_code\_text\_edit->setTextCursor(cursor);

}

}

void MainWindow::**on\_table\_result\_test\_doubleClicked**(*const* QModelIndex &index)

{

QModelIndex columnIndex = ui->table\_result\_test->model()->*index*(index.row(), 0);

QVariant data = ui->table\_result\_test->model()->*data*(columnIndex);

*if* (data.isValid())

{

moredetails\_window = *new* moredetails(*database*, *this*, data.toInt());

moredetails\_window->setWindowTitle("Подробнее об отчете");

moredetails\_window->setModal(*true*);

moredetails\_window->*exec*();

}

}

Testwindow.h

#ifndef TESTWINDOW\_H

#define TESTWINDOW\_H

#include <QDialog>

#include <QCloseEvent>

#include <QListWidgetItem>

#include <QSqlDatabase>

#include <QSqlTableModel>

#include <QResizeEvent>

#include "textwindow.h"

*namespace* **Ui** {*class* **testwindow**;}

*class* **testwindow** : *public* QDialog

{

Q\_OBJECT

*public*:

*explicit* **testwindow**(QSqlDatabase& database, QWidget \*parent = *nullptr*, int current\_lab = 0);

~***testwindow***();

*private* slots:

void **on\_button\_add\_variant\_clicked**();

void **on\_button\_add\_tests\_clicked**();

void **on\_upload\_user\_code\_button\_clicked**();

void **on\_button\_switch\_mode\_clicked**();

void **on\_button\_compile\_file\_clicked**();

void **on\_button\_get\_files\_clicked**();

void **on\_list\_files\_itemDoubleClicked**(QListWidgetItem \*item);

void **on\_button\_test\_append\_clicked**();

void **on\_button\_save\_variant\_clicked**();

void **on\_button\_save\_lab\_clicked**();

void **on\_button\_delete\_variant\_clicked**();

void **on\_button\_delete\_tests\_clicked**();

void **on\_button\_cancel\_clicked**();

void **on\_button\_edit\_tests\_clicked**();

void **on\_button\_edit\_variant\_clicked**();

void **on\_button\_cancel\_variants\_clicked**();

void **on\_button\_cancel\_lab\_clicked**();

void **on\_user\_code\_text\_edit\_textChanged**();

signals:

void **sendQuery**(QString);

*private*:

void **button\_compile**(bool isWorkWithFile);

void **choose\_files**();

void **showUserCode**(QListWidget\* widget, QListWidgetItem \*item);

void ***resizeEvent***(QResizeEvent \*event);

QStringList files\_path;

int current\_lab\_id = 0;

int current\_var\_id = 0;

int current\_var\_number = 1;

int current\_test\_id = 0;

bool isEdit = *false*;

Ui::testwindow \*ui;

usercodewindow \*code\_window;

QSqlDatabase& m\_database;

QSqlTableModel\* db\_model;

QSqlTableModel\* db\_model\_tests;

*protected*:

void ***closeEvent***(QCloseEvent \*evnt);

};

#endif *//* *TESTWINDOW\_H*

Testwindow.cpp

#include "testwindow.h"

#include "ui\_testwindow.h"

#include "mainwindow.h"

#include <QMessageBox>

#include <QDebug>

#include "realization/utils.h"

#include "realization/database\_operations.h"

testwindow::testwindow(QSqlDatabase& database, QWidget \*parent, int current\_lab)

: QDialog(parent), ui(*new* Ui::testwindow), m\_database(database)

{

ui->setupUi(*this*);

db\_model\_tests = *new* QSqlTableModel(*this*, m\_database);

db\_model\_tests->setTable("Tests");

ui->list\_of\_tests->resizeColumnsToContents();

db\_model\_tests->setHeaderData(2, Qt::Horizontal, tr("Входные данные"));

db\_model\_tests->setHeaderData(3, Qt::Horizontal, tr("Выходные данные"));

ui->list\_of\_tests->setModel(db\_model\_tests);

ui->list\_of\_tests->setEditTriggers(QAbstractItemView::NoEditTriggers);

ui->list\_of\_tests->setColumnHidden(0, *true*);

ui->list\_of\_tests->setColumnHidden(1, *true*);

db\_model = *new* QSqlTableModel(*this*, m\_database);

db\_model->setTable("Variants");

ui->list\_variants->resizeColumnsToContents();

db\_model->setHeaderData(3, Qt::Horizontal, tr("Номер варианта"));

ui->list\_variants->setModel(db\_model);

ui->list\_variants->setEditTriggers(QAbstractItemView::NoEditTriggers);

ui->list\_variants->setColumnHidden(0, *true*);

ui->list\_variants->setColumnHidden(1, *true*);

ui->list\_variants->setColumnHidden(2, *true*);

*if* (current\_lab)

{

isEdit = *true*;

current\_lab\_id = current\_lab;

QString query\_name = getResult(getDBDataQuery("LabWork", "name", "id", QString::number(current\_lab\_id))).toString();

QString query\_desc = getResult(getDBDataQuery("LabWork", "description", "id", QString::number(current\_lab\_id))).toString();

ui->lineEdit\_lab\_name->setText(query\_name);

ui->TextEdit\_lab\_desc->setPlainText(query\_desc);

db\_model->setFilter(QString("labwork\_id = %1").arg(current\_lab\_id));

db\_model->setSort(3, Qt::SortOrder::AscendingOrder);

db\_model->select();

}

}

testwindow::~testwindow()

{

*delete* ui;

}

void testwindow::closeEvent(QCloseEvent \*evnt)

{

*if* (evnt->spontaneous())

{

QMessageBox messageBox(QMessageBox::Question,

tr("Вы уверены?"),

tr("Несохраненные изменения будут утеряны."),

QMessageBox::Yes | QMessageBox::No,

*this*);

messageBox.setButtonText(QMessageBox::Yes, tr("Да"));

messageBox.setButtonText(QMessageBox::No, tr("Нет"));

*if* (messageBox.exec() == QMessageBox::Yes)

{

evnt->accept(); *//* *Закрыть* *окно*

}

*else*

{

evnt->ignore(); *//* *Отменить* *закрытие* *окна*

}

}

}

void testwindow::on\_button\_add\_variant\_clicked()

{

*if* (ui->lineEdit\_lab\_name->text() != "" && ui->TextEdit\_lab\_desc->toPlainText() != "")

{

isEdit = *false*;

QString query;

ui->stackedWidget->setCurrentIndex(1);

db\_model\_tests->setFilter(QString("variant\_id = %1").arg(current\_var\_id));

db\_model\_tests->select();

current\_var\_number = getNextFreeNumberVar(*this*->m\_database, current\_lab\_id);

ui->label\_number\_variant->setText(QString::number(current\_var\_number));

ui->TextEdit\_variant->clear();

QStringList tables = {"name", "description"};

QStringList data;

data << ui->lineEdit\_lab\_name->text() << ui->TextEdit\_lab\_desc->toPlainText();

*if* (current\_lab\_id)

{

QString query\_name = getResult(getDBDataQuery("LabWork", "name", "id", QString::number(current\_lab\_id))).toString();

QString query\_desc = getResult(getDBDataQuery("LabWork", "description", "id", QString::number(current\_lab\_id))).toString();

*if* (query\_name != data[0] || query\_desc != data[1])

{

query = updateData("LabWork", tables, data, "id = " + QString::number(current\_lab\_id));

*emit* sendQuery(query);

}

}

*else*

{

query = makeInsertQuery("LabWork", tables, data, *this*->m\_database);

*emit* sendQuery(query);

query = getDBDataQuery("LabWork", "id", "name", data[0]);

current\_lab\_id = getResult(query).toInt();

}

}

*else*

{

QMessageBox::warning(*this*, "Куда торопимся?", "Не все поля заполнены.");

}

}

void testwindow::on\_button\_add\_tests\_clicked()

{

QString variant = ui->TextEdit\_variant->toPlainText();

*if* (variant.size())

{

isEdit = *false*;

ui->stackedWidget->setCurrentIndex(2);

ui->user\_input\_data->clear();

ui->to\_user\_output\_data->clear();

QStringList tables = {"labwork\_id", "conditions" , "number\_var"};

QStringList data;

data << QString::number(current\_lab\_id) << variant << QString::number(current\_var\_number);

QString query;

*if* (current\_var\_id)

{

QString conditions = getResult(getDBDataQuery("Variants", "conditions", "id", QString::number(current\_var\_id))).toString();

*if* (conditions != data[1])

{

query = updateData("Variants", tables, data, "id = " + QString::number(current\_var\_id));

*emit* sendQuery(query);

}

}

*else*

{

query = makeInsertQuery("Variants", tables, data, *this*->m\_database);

*emit* sendQuery(query);

query = getDBDataQuery("Variants", "id", "conditions", data[1]);

current\_var\_id = getResult(query).toInt();

}

}

*else*

{

QMessageBox::warning(*this*, "Куда торопимся?", "Не все поля заполнены.");

}

}

*//* *вывод* *текста* *выбранного* *файла* *в* *новом* *окне*

*//* *item\_index* *-* *индекс* *элемента* *списка* *(и* *пути* *этого* *элемента* *в* *files\_path),* *на* *который* *нажал* *пользователь*

void testwindow::showUserCode(QListWidget\* widget, QListWidgetItem \*item)

{

QString text\_code;

int item\_index = widget->currentRow();

readFromFile(files\_path[item\_index], text\_code);

code\_window = *new* usercodewindow(*this*, text\_code);

code\_window->setWindowTitle(item->text());

code\_window->show();

code\_window->setSizePolicy(QSizePolicy::MinimumExpanding, QSizePolicy::MinimumExpanding);

}

void testwindow::choose\_files()

{

QStringList temp\_paths; *//* *нужен,* *если* *пользователь* *случайно* *нажал* *(не* *хотел* *выбирать* *файлы),* *чтобы* *список* *не* *очищался*

QString current\_path; *//* *для* *запоминания* *последней* *директории*

*if* (files\_path.isEmpty())

current\_path = QDir::currentPath();

*//* *выбор* *файла*

temp\_paths = QFileDialog::getOpenFileNames(*this*, "Выберите файл", current\_path, "Cpp and Header Files (\*.\*)");

*//* *проверка* *на* *пустую* *директорию*

*if* (temp\_paths.isEmpty())

{

*return*;

}

files\_path = temp\_paths;

current\_path = files\_path[0];

*//* *очистка* *listWidget*

ui->list\_files->clear();

*//* *заполнение* *списка*

fillSelecteFilesTable(files\_path, ui->list\_files);

}

*//* *запускает* *компиляцию* *и* *выводит* *на* *пользовательские* *окна*

void testwindow::button\_compile(bool isWorkWithFile)

{

QString file\_bat\_path; *//* *путь* *до* *файла* *.bat*

QString file\_cpp\_path; *//* *путь* *до* *файла* *.cpp*

QString file\_input\_path; *//* *путь* *до* *файла* *.txt*

QString current\_path; *//* *путь* *до* *рабочей* *папки*

QTextStream writeStream; *//* *поток* *для* *записи* *в* *файл*

QStringList names;

QStringList list\_del\_names = { "user\_input.txt", "user\_output.txt" }; *//* *список* *для* *удаления* *файлов*

*if* (isWorkWithFile)

{

*if* (files\_path.size() == 0)

{

QMessageBox::warning(*this*, "Ошибка", "Файлы не выбраны!");

*return*;

}

names = get\_name\_from\_path(files\_path);

QString path = files\_path[0];

int index;

*for* (index = path.size()-1; index > 0 && path[index] != "/"; index--);

current\_path = path.remove(index+1, path.size()-index);

}

*else*

{

*if* (!ui->user\_code\_text\_edit->toPlainText().size())

{

QMessageBox::warning(*this*, "Ошибка", "Нельзя скомпилировать пустой код.");

*return*;

}

names.append("user\_main\_code.cpp");

list\_del\_names.append("user\_main\_code.cpp");

current\_path = QDir::currentPath().remove(QDir::currentPath().size()-3, 3);

file\_cpp\_path = current\_path + "user\_main\_code.cpp"; *//* *путь* *до* *файла* *user\_main\_code.cpp*

QString user\_code = ui->user\_code\_text\_edit->toPlainText(); *//* *код* *из* *тектового* *поля* *ввода*

*//* *открытие* *user\_main\_code.cpp* *и* *запись* *в* *него* *код* *из* *пользовательского* *окна*

writeToFile(file\_cpp\_path, user\_code);

}

QString user\_input\_data = ui->user\_input\_data->toPlainText(); *//* *входные* *данные* *из* *поля* *ввода*

file\_bat\_path = current\_path + "start\_compile.bat"; *//* *путь* *до* *файла* *start\_compile.bat*

file\_input\_path = current\_path + "user\_input.txt"; *//* *путь* *до* *файла* *user\_input.txt*

*//* *очистка* *вывода* *пользователю*

ui->to\_user\_output\_data->setPlainText("");

*//* *запись* *в* *user\_input.txt* *из* *пользовательского* *окна* *ввода*

writeToFile(file\_input\_path, user\_input\_data);

QProcess process;

compile\_code(current\_path, names);

*//* *запуск* *start\_compile.bat*

process.start(current\_path + "start\_compile.bat");

*//* *проверка* *на* *ошибки* *и* *вывод*

process.waitForReadyRead();

QByteArray errors = process.readAllStandardError();

*if* (errors.size() != 0)

{

delete\_file(current\_path, list\_del\_names);

QMessageBox::information(*this*, "Ошибка компиляции", errors);

*return*;

}

*if* (!process.waitForFinished(15000))

{

WinExec("taskkill /im user\_main\_code.exe /f", SW\_HIDE);

delete\_file(current\_path, list\_del\_names);

QMessageBox::critical(*this*, "Runtime Error", "Превышено время ожидания выполнения программы.");

*return*;

}

QString output\_data = QString::fromLocal8Bit(process.readAll());

ui->to\_user\_output\_data->setPlainText(output\_data);

delete\_file(current\_path, list\_del\_names);

}

void testwindow::on\_upload\_user\_code\_button\_clicked()

{

button\_compile(*false*);

}

void testwindow::on\_button\_switch\_mode\_clicked()

{

ui->to\_user\_output\_data->setPlainText("");

ui->user\_input\_data->setPlainText("");

*//* *очистка* *виджетов* *ввода/вывода* *программы*

ui->to\_user\_output\_data->setPlainText("");

ui->user\_input\_data->setPlainText("");

*if* (ui->stacked\_widget\_modes->currentIndex())

{

ui->stacked\_widget\_modes->setCurrentWidget(ui->page\_code\_mode);

ui->button\_switch\_mode->setText("Работа с файлами");

}

*else*

{

ui->stacked\_widget\_modes->setCurrentWidget(ui->page\_file\_mode);

ui->button\_switch\_mode->setText("Работа с кодом");

}

}

void testwindow::on\_button\_compile\_file\_clicked()

{

button\_compile(*true*);

}

void testwindow::on\_button\_get\_files\_clicked()

{

choose\_files();

}

void testwindow::on\_list\_files\_itemDoubleClicked(QListWidgetItem \*item)

{

showUserCode(ui->list\_files, item);

}

void testwindow::on\_button\_test\_append\_clicked()

{

QString input\_data, output\_data;

input\_data = ui->user\_input\_data->toPlainText();

output\_data = ui->to\_user\_output\_data->toPlainText();

*if* (input\_data.size() || output\_data.size())

{

ui->user\_input\_data->clear();

ui->to\_user\_output\_data->clear();

QStringList tables = {"variant\_id", "input\_data", "output\_data"};

QStringList data;

data << QString::number(current\_var\_id) << input\_data << output\_data;

QString query;

*if* (current\_test\_id)

{

query = updateData("Tests", tables, data, "id = " + QString::number(current\_test\_id));

}

*else*

{

query = makeInsertQuery("Tests", tables, data, *this*->m\_database);

}

*emit* sendQuery(query);

current\_test\_id = 0;

*if* (!isEdit)

{

QMessageBox::information(*this*, tr("Успешно!"), tr("Тест успешно добавлен."));

db\_model\_tests->setFilter(QString("variant\_id = %1").arg(current\_var\_id));

db\_model\_tests->select();

}

*else*

{

QMessageBox::information(*this*, tr("Успешно!"), tr("Тест успешно изменен."));

ui->stackedWidget->setCurrentIndex(1);

db\_model\_tests->setFilter(QString("variant\_id = %1").arg(current\_var\_id));

db\_model\_tests->select();

}

}

*else*

{

QMessageBox::warning(*this*, tr("Ошибка"), tr("Хотя бы одно поле должно быть заполнено"));

}

}

void testwindow::on\_button\_save\_variant\_clicked()

{

QString variant\_data;

variant\_data = ui->TextEdit\_variant->toPlainText();

*if* (variant\_data.size())

{

QStringList tables = {"labwork\_id", "conditions", "number\_var"};

QStringList data;

data << QString::number(current\_lab\_id) << variant\_data << QString::number(current\_var\_number);

QString query;

*if* (current\_var\_id)

{

QString conditions = getResult(getDBDataQuery("Variants", "conditions", "id", QString::number(current\_var\_id))).toString();

*if* (conditions != data[1])

{

query = updateData("Variants", tables, data, "id = " + QString::number(current\_var\_id));

*emit* sendQuery(query);

}

}

*else*

{

query = makeInsertQuery("Variants", tables, data, *this*->m\_database);

*emit* sendQuery(query);

query = getDBDataQuery("Variants", "id", "conditions", data[1]);

current\_var\_id = getResult(query).toInt();

}

db\_model->setFilter(QString("labwork\_id = %1").arg(current\_lab\_id));

db\_model->setSort(3, Qt::SortOrder::AscendingOrder);

db\_model->select();

current\_var\_id = 0;

ui->stackedWidget->setCurrentIndex(0);

}

*else*

{

QMessageBox::warning(*this*, "Куда торопимся?", "Не все поля заполнены.");

}

}

void testwindow::on\_button\_save\_lab\_clicked()

{

*if* (ui->lineEdit\_lab\_name->text() != "" && ui->TextEdit\_lab\_desc->toPlainText() != "")

{

QStringList tables = {"name", "description"};

QStringList data;

QString query = "";

data << ui->lineEdit\_lab\_name->text() << ui->TextEdit\_lab\_desc->toPlainText();

*if* (current\_lab\_id)

{

QString query\_name = getResult(getDBDataQuery("LabWork", "name", "id", QString::number(current\_lab\_id))).toString();

QString query\_desc = getResult(getDBDataQuery("LabWork", "description", "id", QString::number(current\_lab\_id))).toString();

*if* (query\_name != data[0] || query\_desc != data[1])

{

query = updateData("LabWork", tables, data, "id = " + QString::number(current\_lab\_id));

}

}

*else*

{

query = makeInsertQuery("LabWork", tables, data, *this*->m\_database);

}

*if* (query.size())

*emit* sendQuery(query);

current\_lab\_id = 0;

current\_var\_number = 1;

close();

}

*else*

{

QMessageBox::warning(*this*, "Куда торопимся?", "Не все поля заполнены.");

}

}

void testwindow::on\_button\_delete\_variant\_clicked()

{

int current\_row = ui->list\_variants->currentIndex().row();

int variant\_id = db\_model->index(current\_row, 0).data().toInt();

*if* (current\_row != -1)

{

QMessageBox messageBox(QMessageBox::Question,

tr("Вы уверены?"),

tr("Вы уверены, что хотите удалить\nнавсегда и безвозвратно?"),

QMessageBox::Yes | QMessageBox::No,

*this*);

messageBox.setButtonText(QMessageBox::Yes, tr("Да"));

messageBox.setButtonText(QMessageBox::No, tr("Нет"));

*if* (messageBox.exec() == QMessageBox::Yes)

{

*//* *Удаление* *зависимостей* *из* *таблицы* *Tests*

QSqlQuery query;

query.prepare("DELETE FROM Tests WHERE variant\_id = :variant\_id");

query.bindValue(":variant\_id", variant\_id);

query.exec();

*//* *Удаление* *записи* *из* *таблицы* *Variants*

db\_model->removeRow(current\_row);

db\_model->setSort(3, Qt::SortOrder::AscendingOrder);

db\_model->select();

}

}

*else*

{

QMessageBox::warning(*this*, "Ошибка", "Не выбран ни 1 элемент.");

*return*;

}

}

void testwindow::on\_button\_delete\_tests\_clicked()

{

int current\_row = ui->list\_of\_tests->currentIndex().row();

*if* (current\_row != -1)

{

QMessageBox messageBox(QMessageBox::Question,

tr("Вы уверены?"),

tr("Вы уверены, что хотите удалить\nнавсегда и безвозвратно?"),

QMessageBox::Yes | QMessageBox::No,

*this*);

messageBox.setButtonText(QMessageBox::Yes, tr("Да"));

messageBox.setButtonText(QMessageBox::No, tr("Нет"));

*if* (messageBox.exec() == QMessageBox::Yes)

{

db\_model\_tests->removeRow(current\_row);

db\_model\_tests->select();

}

}

*else*

{

QMessageBox::warning(*this*, "Ошибка", "Не выбран ни 1 элемент.");

*return*;

}

}

void testwindow::on\_button\_cancel\_clicked()

{

QString input\_data, output\_data;

input\_data = ui->user\_input\_data->toPlainText();

output\_data = ui->to\_user\_output\_data->toPlainText();

QString input\_data\_db = "", output\_data\_db = "";

*if* (isEdit)

{

input\_data\_db = getResult(getDBDataQuery("Tests", "input\_data", "id", QString::number(current\_test\_id))).toString();

output\_data\_db = getResult(getDBDataQuery("Tests", "output\_data", "id", QString::number(current\_test\_id))).toString();

}

*if* (input\_data != input\_data\_db || output\_data != output\_data\_db)

{

QMessageBox messageBox(QMessageBox::Question,

tr("Вы уверены?"),

tr("Совершенные изменения не сохранятся."),

QMessageBox::Yes | QMessageBox::No,

*this*);

messageBox.setButtonText(QMessageBox::Yes, tr("Да"));

messageBox.setButtonText(QMessageBox::No, tr("Нет"));

*if* (messageBox.exec() == QMessageBox::Yes)

{

ui->stackedWidget->setCurrentIndex(1);

}

}

*else*

{

ui->stackedWidget->setCurrentIndex(1);

}

current\_test\_id = 0;

}

void testwindow::on\_button\_edit\_tests\_clicked()

{

isEdit = *true*;

int current\_row = ui->list\_of\_tests->currentIndex().row();

*if* (current\_row != -1)

{

QModelIndex index = ui->list\_of\_tests->model()->index(current\_row, 0);

current\_test\_id = index.data().toInt();

QString input\_data, output\_data;

input\_data = getResult(getDBDataQuery("Tests", "input\_data", "id", QString::number(current\_test\_id))).toString();

output\_data = getResult(getDBDataQuery("Tests", "output\_data", "id", QString::number(current\_test\_id))).toString();

ui->user\_input\_data->setPlainText(input\_data);

ui->to\_user\_output\_data->setPlainText(output\_data);

ui->stackedWidget->setCurrentIndex(2);

}

*else*

{

QMessageBox::warning(*this*, "Ошибка", "Не выбран ни 1 элемент.");

*return*;

}

}

void testwindow::on\_button\_edit\_variant\_clicked()

{

isEdit = *true*;

int var\_number = ui->list\_variants->currentIndex().row() + 1;

*if* (var\_number != 0)

{

current\_var\_number = var\_number;

QString query = getDBDataQuery("Variants", "id", {"labwork\_id", "number\_var"},

{QString::number(current\_lab\_id), QString::number(current\_var\_number)});

current\_var\_id = getResult(query).toInt();

query = getDBDataQuery("Variants", "conditions", "id", QString::number(current\_var\_id));

ui->TextEdit\_variant->setPlainText(getResult(query).toString());

ui->label\_number\_variant->setText(QString::number(current\_var\_number));

ui->stackedWidget->setCurrentIndex(1);

db\_model\_tests->setFilter(QString("variant\_id = %1").arg(current\_var\_id));

db\_model\_tests->select();

}

*else*

{

QMessageBox::warning(*this*, "Ошибка", "Не выбран ни 1 элемент.");

*return*;

}

}

void testwindow::resizeEvent(QResizeEvent \*event)

{

ui->list\_variants->setColumnWidth(3, ui->list\_variants->width());

ui->list\_of\_tests->setColumnWidth(2, (ui->list\_of\_tests->width())/2);

ui->list\_of\_tests->setColumnWidth(3, (ui->list\_of\_tests->width())/2);

event->accept();

}

void testwindow::on\_button\_cancel\_variants\_clicked()

{

QString var\_data;

var\_data = ui->TextEdit\_variant->toPlainText();

QString var\_data\_db = "";

*if* (isEdit)

{

var\_data\_db = getResult(getDBDataQuery("Variants", "conditions", "id", QString::number(current\_var\_id))).toString();

}

*if* (var\_data != var\_data\_db)

{

QMessageBox messageBox(QMessageBox::Question,

tr("Вы уверены?"),

tr("Совершенные изменения не сохранятся."),

QMessageBox::Yes | QMessageBox::No,

*this*);

messageBox.setButtonText(QMessageBox::Yes, tr("Да"));

messageBox.setButtonText(QMessageBox::No, tr("Нет"));

*if* (messageBox.exec() == QMessageBox::Yes)

{

ui->stackedWidget->setCurrentIndex(0);

}

}

*else*

{

ui->stackedWidget->setCurrentIndex(0);

}

current\_var\_id = 0;

}

void testwindow::on\_button\_cancel\_lab\_clicked()

{

QString name\_data, desc\_data;

name\_data = ui->lineEdit\_lab\_name->text();

desc\_data = ui->TextEdit\_lab\_desc->toPlainText();

QString name\_data\_db = "", desc\_data\_db = "";

*if* (isEdit)

{

name\_data\_db = getResult(getDBDataQuery("LabWork", "name", "id", QString::number(current\_lab\_id))).toString();

desc\_data\_db = getResult(getDBDataQuery("LabWork", "description", "id", QString::number(current\_lab\_id))).toString();

}

*if* (name\_data != name\_data\_db || desc\_data != desc\_data\_db)

{

QMessageBox messageBox(QMessageBox::Question,

tr("Вы уверены?"),

tr("Совершенные изменения не сохранятся."),

QMessageBox::Yes | QMessageBox::No,

*this*);

messageBox.setButtonText(QMessageBox::Yes, tr("Да"));

messageBox.setButtonText(QMessageBox::No, tr("Нет"));

*if* (messageBox.exec() == QMessageBox::Yes)

{

close();

}

}

*else*

{

close();

}

}

void testwindow::on\_user\_code\_text\_edit\_textChanged()

{

QString text = ui->user\_code\_text\_edit->toPlainText();

QTextCursor cursor = ui->user\_code\_text\_edit->textCursor();

int cursorPosition = cursor.position();

*//* *если* *введен* *tab*

*if* (cursorPosition > 0 && text.at(cursorPosition - 1) == '\t')

{

*//* *удалить* *Tab*

cursor.deletePreviousChar();

*//* *вставить* *четыре* *пробела*

cursor.insertText(" ");

*//* *обновить* *курсор*

ui->user\_code\_text\_edit->setTextCursor(cursor);

}

}

Textwindow.h

#ifndef TEXTWINDOW\_H

#define TEXTWINDOW\_H

#include <QDialog>

#include <QResizeEvent>

*namespace* **Ui** {*class* **usercodewindow**;}

*class* **usercodewindow** : *public* QDialog

{

Q\_OBJECT

*public*:

*explicit* **usercodewindow**(QWidget \*parent = *nullptr*, QString text\_code = "");

~***usercodewindow***();

*private*:

Ui::usercodewindow \*ui;

*protected*:

void ***resizeEvent***(QResizeEvent \*event);

};

#endif *//* *TEXTWINDOW\_H*

Textwindow.cpp

#include "textwindow.h"

#include "ui\_usercodewindow.h"

usercodewindow::**usercodewindow**(QWidget \*parent, QString text\_code) : QDialog(*parent*), ui(*new* Ui::usercodewindow)

{

ui->setupUi(*this*);

ui->text\_user\_code->setText(text\_code);

*this*->setMinimumSize(300,300);

}

usercodewindow::~***usercodewindow***()

{

*delete* ui;

}

void usercodewindow::***resizeEvent***(QResizeEvent \*event)

{

ui->text\_user\_code->setGeometry(0, 0, event->size().width(), event->size().height());

}

Moredetails.h

#ifndef MOREDETAILS\_H

#define MOREDETAILS\_H

#include <QDialog>

#include <QCloseEvent>

#include <QListWidgetItem>

#include <QSqlDatabase>

#include <QSqlTableModel>

#include <QResizeEvent>

#include <QDebug>

#include <QVector>

#include <QCloseEvent>

*namespace* **Ui** {

*class* **moredetails**;

}

*class* **moredetails** : *public* QDialog

{

Q\_OBJECT

*public*:

*explicit* **moredetails**(QSqlDatabase& database, QWidget \*parent = *nullptr*, int test\_id = 0);

~***moredetails***();

*private* slots:

void **on\_button\_ok\_clicked**();

*private*:

Ui::moredetails \*ui;

QSqlDatabase& m\_database;

};

#endif *//* *MOREDETAILS\_H*

Moredetails.cpp

#include "moredetails.h"

#include "ui\_moredetails.h"

#include "realization/database\_operations.h"

#include "realization/utils.h"

moredetails::**moredetails**(QSqlDatabase& database, QWidget \*parent, int test\_id) :

QDialog(*parent*),

ui(*new* Ui::moredetails), m\_database(database)

{

ui->setupUi(*this*);

ui->textBrowser\_task->setPlainText(getResult(getDBDataQuery("CheckedWorks", "task", "id", QString::number(test\_id))).toString());

ui->label\_status->setText(getResult(getDBDataQuery("CheckedWorks", "status", "id", QString::number(test\_id))).toString());

QString input\_text, output\_text, expected\_text;

input\_text = getResult(getDBDataQuery("CheckedWorks", "input\_data", "id", QString::number(test\_id))).toString();

expected\_text = getResult(getDBDataQuery("CheckedWorks", "expected\_output", "id", QString::number(test\_id))).toString();

output\_text = getResult(getDBDataQuery("CheckedWorks", "output\_data", "id", QString::number(test\_id))).toString();

*if* (input\_text.size() && expected\_text.size() && output\_text.size())

{

ui->textBrowser\_input\_data->setText("Входные данные:\n" + input\_text);

ui->textBrowser\_expected\_output->setText("Ожидаемые выходные данные:\n" + expected\_text);

ui->textBrowser\_output\_data->setText("Выходные данные:\n" + output\_text);

}

*else*

{

QString text = "Все тесты пройдены!";

ui->textBrowser\_input\_data->setText(text);

ui->textBrowser\_expected\_output->setText(text);

ui->textBrowser\_output\_data->setText(text);

}

}

moredetails::~***moredetails***()

{

*delete* ui;

}

void moredetails::**on\_button\_ok\_clicked**()

{

close();

}

Choosework.h

#ifndef CHOOSEWORK\_H

#define CHOOSEWORK\_H

#include <QDialog>

#include <QCloseEvent>

#include <QListWidgetItem>

#include <QSqlDatabase>

#include <QSqlTableModel>

#include <QResizeEvent>

#include <QDebug>

#include <QVector>

#include <QCloseEvent>

*namespace* **Ui** {

*class* **choosework**;

}

*class* **choosework** : *public* QDialog

{

Q\_OBJECT

*public*:

*explicit* **choosework**(QSqlDatabase& database, QWidget \*parent = *nullptr*);

~***choosework***();

*private* slots:

void **on\_comboBox\_of\_works\_currentIndexChanged**(int index);

void **on\_button\_cancel\_work\_clicked**();

void **on\_button\_accept\_work\_clicked**();

*private*:

Ui::choosework \*ui;

QSqlDatabase& m\_database;

QSqlTableModel\* db\_model\_var;

void ***closeEvent***(QCloseEvent \*evnt);

void ***resizeEvent***(QResizeEvent \*event);

QVector<QVariantList> labs;

QString current\_name;

signals:

void **send\_id**(int id\_lab, int id\_var, QString current\_name);

};

#endif *//* *CHOOSEWORK\_H*

Choosework.cpp

#include "choosework.h"

#include "ui\_choosework.h"

#include "realization/database\_operations.h"

#include "realization/utils.h"

choosework::choosework(QSqlDatabase& database, QWidget \*parent) :

QDialog(parent),

ui(*new* Ui::choosework), m\_database(database)

{

ui->setupUi(*this*);

db\_model\_var = *new* QSqlTableModel(*this*, m\_database);

db\_model\_var->setTable("Variants");

ui->list\_of\_variants->resizeColumnsToContents();

db\_model\_var->setHeaderData(3, Qt::Horizontal, tr("Номер варианта"));

ui->list\_of\_variants->setModel(db\_model\_var);

ui->list\_of\_variants->setColumnHidden(0, *true*);

ui->list\_of\_variants->setColumnHidden(1, *true*);

ui->list\_of\_variants->setColumnHidden(2, *true*);

ui->list\_of\_variants->setColumnWidth(3, *this*->width()-31);

labs = getAllInTableWhere(database, "LabWork", "id", QString::number(0), *true*);

*for* (int i = 0; i < labs.size(); i++)

{

ui->comboBox\_of\_works->addItem(labs[i][1].toString());

}

}

choosework::~choosework()

{

*delete* ui;

}

void choosework::on\_comboBox\_of\_works\_currentIndexChanged(int index)

{

*if* (index != 0)

{

int current\_lab\_id = labs[index-1][0].toInt();

db\_model\_var->setFilter(QString("labwork\_id = %1").arg(current\_lab\_id));

db\_model\_var->setSort(3, Qt::SortOrder::AscendingOrder);

db\_model\_var->select();

}

*else*

{

db\_model\_var->setFilter(QString("labwork\_id = %1").arg(-1));

db\_model\_var->select();

}

}

void choosework::closeEvent(QCloseEvent \*evnt)

{

*if* (ui->comboBox\_of\_works->currentIndex() != 0 || ui->lineEdit\_name->text() != "")

{

QMessageBox messageBox(QMessageBox::Question,

tr("Вы уверены?"),

tr("Несохраненные изменения будут утеряны."),

QMessageBox::Yes | QMessageBox::No,

*this*);

messageBox.setButtonText(QMessageBox::Yes, tr("Да"));

messageBox.setButtonText(QMessageBox::No, tr("Нет"));

*if* (messageBox.exec() == QMessageBox::Yes)

{

evnt->accept(); *//* *Закрыть* *окно*

}

*else*

{

evnt->ignore(); *//* *Отменить* *закрытие* *окна*

}

}

}

void choosework::on\_button\_cancel\_work\_clicked()

{

close();

}

void choosework::on\_button\_accept\_work\_clicked()

{

int current\_index\_lab = ui->comboBox\_of\_works->currentIndex();

int current\_index\_var = ui->list\_of\_variants->currentIndex().row();

*if* (current\_index\_lab != -1 && current\_index\_var != -1)

{

int id\_lab, id\_var;

id\_lab = labs[current\_index\_lab-1][0].toInt();

QModelIndex index = ui->list\_of\_variants->model()->index(current\_index\_var, 0);

id\_var = index.data().toInt();

current\_name = ui->lineEdit\_name->text();

*emit* send\_id(id\_lab, id\_var, current\_name);

ui->comboBox\_of\_works->setCurrentIndex(0);

ui->lineEdit\_name->clear();

close();

}

*else*

{

QMessageBox::warning(*this*, "Ошибка", "Не выбрано ни 1 работы.");

*return*;

}

}

void choosework::resizeEvent(QResizeEvent \*event)

{

ui->list\_of\_variants->setColumnWidth(3, *this*->width()-31);

event->accept();

}

Tab\_files.cpp

#include "realization/database\_operations.h"

#include "realization/utils.h"

void MainWindow::choose\_files()

{

QStringList temp\_paths; *//* *нужен,* *если* *пользователь* *случайно* *нажал* *(не* *хотел* *выбирать* *файлы),* *чтобы* *список* *не* *очищался*

QString current\_path; *//* *для* *запоминания* *последней* *директории*

*if* (files\_path.isEmpty())

current\_path = QDir::currentPath();

*//* *выбор* *файла*

temp\_paths = QFileDialog::getOpenFileNames(*this*, "Выберите файл", current\_path, "Cpp and Header Files (\*.\*)");

*//* *проверка* *на* *пустую* *директорию*

*if* (temp\_paths.isEmpty())

{

*return*;

}

files\_path = temp\_paths;

current\_path = files\_path[0];

*//* *очистка* *listWidget*

ui->list\_selected\_files->clear();

ui->list\_files->clear();

*//* *заполнение* *списка*

fillSelecteFilesTable(files\_path, ui->list\_files);

fillSelecteFilesTable(files\_path, ui->list\_selected\_files);

}

*//* *вывод* *текста* *выбранного* *файла* *в* *новом* *окне*

*//* *item\_index* *-* *индекс* *элемента* *списка* *(и* *пути* *этого* *элемента* *в* *files\_path),* *на* *который* *нажал* *пользователь*

void MainWindow::showUserCode(QListWidget\* widget, QListWidgetItem \*item)

{

QString text\_code;

int item\_index = widget->currentRow();

readFromFile(files\_path[item\_index], text\_code);

code\_window = *new* usercodewindow(*this*, text\_code);

code\_window->setWindowTitle(item->text());

code\_window->show();

code\_window->setSizePolicy(QSizePolicy::MinimumExpanding, QSizePolicy::MinimumExpanding);

}

*//* *вывод* *текста* *выбранного* *файла* *в* *новом* *окне,* *вызванное* *из* *вкладки* *""*

void MainWindow::on\_list\_selected\_files\_itemDoubleClicked(QListWidgetItem \*item)

{

showUserCode(ui->list\_selected\_files,item);

}

*//* *вывод* *текста* *выбранного* *файла* *в* *новом* *окне*

void MainWindow::on\_list\_files\_itemDoubleClicked(QListWidgetItem \*item)

{

showUserCode(ui->list\_files,item);

}

void MainWindow::on\_button\_get\_path\_files\_clicked()

{

choose\_files();

}

void MainWindow::on\_button\_get\_files\_clicked()

{

choose\_files();

}

Tab\_only\_code.cpp

#include "realization/database\_operations.h"

#include "realization/utils.h"

*//* *запускает* *компиляцию* *и* *выводит* *на* *пользовательские* *окна*

void MainWindow::for\_button\_compile(bool isWorkWithFile)

{

QString file\_bat\_path; *//* *путь* *до* *файла* *.bat*

QString file\_cpp\_path; *//* *путь* *до* *файла* *.cpp*

QString file\_input\_path; *//* *путь* *до* *файла* *.txt*

QString current\_path; *//* *путь* *до* *рабочей* *папки*

QTextStream writeStream; *//* *поток* *для* *записи* *в* *файл*

QStringList names;

QStringList list\_del\_names = { "user\_input.txt", "user\_output.txt" }; *//* *список* *для* *удаления* *файлов*

*if* (isWorkWithFile)

{

*if* (files\_path.size() == 0)

{

ui->statusbar->showMessage("Файлы не выбраны!");

*return*;

}

names = get\_name\_from\_path(files\_path);

QString path = files\_path[0];

int index;

*for* (index = path.size()-1; index > 0 && path[index] != "/"; index--);

current\_path = path.remove(index+1, path.size()-index);

}

*else*

{

*if* (!ui->user\_code\_text\_edit->toPlainText().size())

{

QMessageBox::warning(*this*, "Ошибка", "Нельзя скомпилировать пустой код.");

*return*;

}

names.append("user\_main\_code.cpp");

list\_del\_names.append("user\_main\_code.cpp");

current\_path = QDir::currentPath().remove(QDir::currentPath().size()-3, 3);

file\_cpp\_path = current\_path + "user\_main\_code.cpp"; *//* *путь* *до* *файла* *user\_main\_code.cpp*

QString user\_code = ui->user\_code\_text\_edit->toPlainText(); *//* *код* *из* *тектового* *поля* *ввода*

*//* *открытие* *user\_main\_code.cpp* *и* *запись* *в* *него* *код* *из* *пользовательского* *окна*

writeToFile(file\_cpp\_path, user\_code, *false*);

}

QString user\_input\_data = ui->user\_input\_data->toPlainText(); *//* *входные* *данные* *из* *поля* *ввода*

file\_bat\_path = current\_path + "start\_compile.bat"; *//* *путь* *до* *файла* *start\_compile.bat*

file\_input\_path = current\_path + "user\_input.txt"; *//* *путь* *до* *файла* *user\_input.txt*

*//* *очистка* *статус-бара* *|* *вывода* *пользователю*

ui->statusbar->showMessage("");

ui->to\_user\_output\_data->setText("");

*//* *запись* *в* *user\_input.txt* *из* *пользовательского* *окна* *ввода*

writeToFile(file\_input\_path, user\_input\_data);

QProcess process;

compile\_code(current\_path, names);

*//* *запуск* *start\_compile.bat*

process.start(current\_path + "start\_compile.bat");

*//* *проверка* *на* *ошибки* *и* *вывод*

process.waitForReadyRead();

QByteArray errors = process.readAllStandardError();

*if* (errors.size() != 0)

{

ui->statusbar->showMessage("Ошибка компиляции");

delete\_file(current\_path, list\_del\_names);

QMessageBox::information(*this*, "Ошибка компиляции", errors);

*return*;

}

*if* (!process.waitForFinished(15000))

{

WinExec("taskkill /im user\_main\_code.exe /f", SW\_HIDE);

ui->statusbar->showMessage("Runtime Error");

delete\_file(current\_path, list\_del\_names);

QMessageBox::critical(*this*, "Runtime Error", "Превышено время ожидания выполнения программы.");

*return*;

}

QString output\_data = QString::fromLocal8Bit(process.readAll());

ui->to\_user\_output\_data->setText(output\_data);

ui->statusbar->showMessage("Успешное выполнение кода!");

delete\_file(current\_path, list\_del\_names);

}

*//* *запускает* *компиляцию* *и* *выводит* *на* *пользовательские* *окна*

void MainWindow::for\_button\_compile()

{

QString file\_bat\_path; *//* *путь* *до* *файла* *.bat*

QString current\_path; *//* *путь* *до* *рабочей* *папки*

QStringList names;

names = get\_name\_from\_path(files\_path);

QString path = files\_path[0];

int index;

*for* (index = path.size()-1; index > 0 && path[index] != "/"; index--);

current\_path = path.remove(index+1, path.size()-index);

file\_bat\_path = current\_path + "start\_compile.bat"; *//* *путь* *до* *файла* *start\_compile.bat*

*//* *очистка* *статус-бара* *|* *вывода* *пользователю*

ui->statusbar->showMessage("");

QProcess process;

compile\_code(current\_path, names, *true*);

*//* *запуск* *start\_compile.bat*

process.start(current\_path + "start\_compile.bat");

*//* *проверка* *на* *ошибки* *и* *вывод*

process.waitForReadyRead();

QByteArray errors = process.readAllStandardError();

*if* (errors.size() != 0)

{

qDebug() << errors;

ui->statusbar->showMessage("Ошибка компиляции");

QMessageBox::information(*this*, "Ошибка компиляции", errors);

*return*;

}

process.waitForFinished();

}

int MainWindow::check\_test(QString input\_data, QString waiting\_output\_data)

{

QString file\_input\_path; *//* *путь* *до* *файла* *.txt*

QString current\_path; *//* *путь* *до* *рабочей* *папки*

QTextStream writeStream; *//* *поток* *для* *записи* *в* *файл*

QString path = files\_path[0];

int index;

*for* (index = path.size()-1; index > 0 && path[index] != "/"; index--);

current\_path = path.remove(index+1, path.size()-index);

file\_input\_path = current\_path + "user\_input.txt"; *//* *путь* *до* *файла* *user\_input.txt*

*//* *очистка* *статус-бара* *|* *вывода* *пользователю*

ui->statusbar->showMessage("");

*//* *запись* *в* *user\_input.txt* *входных* *данных* *теста*

writeToFile(file\_input\_path, input\_data, *true*);

QFile temp\_file(current\_path + "start\_check\_test.bat");

*if* (!temp\_file.exists())

bat\_for\_check\_test(current\_path);

std::unique\_ptr<QProcess> process(*new* QProcess(*this*));

*//* *запуск* *start\_check\_test.bat*

process->start(current\_path + "start\_check\_test.bat");

*//* *проверка* *на* *ошибки* *и* *вывод*

process->waitForReadyRead();

QByteArray errors = process->readAllStandardError();

*if* (errors.size() != 0)

{

*return* -1;

}

*if* (!process->waitForFinished(4000))

{

WinExec("taskkill /im user\_main\_code.exe /f", SW\_HIDE);

*return* 0;

}

QString output\_data = QString::fromLocal8Bit(process->readAll());

writeToFile(current\_path + "user\_output.txt", output\_data);

output\_data.replace("\r\n", "\n");

*if* (output\_data == waiting\_output\_data)

{

*return* 1;

}

*return* 2;

}

*//* *запуск* *компиляции* *c* *кодом* *в* *соответствующем* *поле*

void MainWindow::on\_upload\_user\_code\_button\_clicked()

{

for\_button\_compile(*false*);

}

*//* *запуск* *компиляции* *c* *кодом* *из* *файлов*

void MainWindow::on\_button\_compile\_file\_clicked()

{

*if* (isCyrillic(files\_path))

{

ui->statusbar->showMessage("Ошибка! В имени файла или в пути к директории файла присутствует кириллица");

*return*;

}

for\_button\_compile(*true*);

}

void MainWindow::on\_button\_start\_test\_clicked()

{

*if* (files\_path.size() == 0)

{

ui->statusbar->showMessage("Файлы не выбраны!");

*return*;

}

for\_button\_compile();

QString current\_path = files\_path[0];

int index;

*for* (index = current\_path.size()-1; index > 0 && current\_path[index] != "/"; index--);

current\_path.remove(index+1, current\_path.size()-index);

QStringList list\_del\_names = { "user\_input.txt", "user\_output.txt", "user\_main\_code.exe" };

QString test\_result, input\_data = "", output\_data = "", expected\_output = "";

int test\_index;

bool isLatestTestAccepted = *true*;

*for* (test\_index = 0; test\_index < tests.size() && isLatestTestAccepted; test\_index++)

{

int result = check\_test(tests[test\_index][2].toString(), tests[test\_index][3].toString());

*switch* (result)

{

*case* -1:

test\_result = "Ошибка при выполнении кода (Тест " + QString::number(test\_index+1) + ")";

output\_data = "Ошибка в коде";

isLatestTestAccepted = *false*;

*break*;

*case* 0:

test\_result = "Превышено время выполнения кода (Тест " + QString::number(test\_index+1) + ")";

output\_data = "Runtime Error";

isLatestTestAccepted = *false*;

*break*;

*case* 2:

test\_result = "Неверный ответ (Тест " + QString::number(test\_index+1) + ")";

readFromFile(current\_path + "user\_output.txt", output\_data);

isLatestTestAccepted = *false*;

*break*;

*case* 1:

*break*;

*default*:

*break*;

}

}

delete\_file(current\_path, {"start\_check\_test.bat"});

delete\_file(current\_path, list\_del\_names);

QString date = getCurrentDateTime();

QString query;

QSqlQuery q(database);

QString person\_id;

*if* (current\_name == "")

current\_name = "Неизвестный";

*if* (existsRecord("Person", "person\_name", current\_name))

query = updateData("Person", {"date"}, {date}, "person\_name = '" + current\_name + "'");

*else*

query = makeInsertQuery("Person", {"date", "person\_name"}, {date, current\_name}, database);

q.prepare(query);

q.exec();

q.clear();

db\_model\_Person->select();

person\_id = getResult(getDBDataQuery("Person", "id", "person\_name", current\_name)).toString();

*if* (isLatestTestAccepted)

{

test\_result = "Зачтено";

}

*else*

{

input\_data = tests[test\_index-1][2].toString();

expected\_output = tests[test\_index-1][3].toString();

}

query = makeInsertQuery("CheckedWorks", {"person\_id", "date", "work\_name", "task", "status", "input\_data", "expected\_output", "output\_data"},

{person\_id, date, current\_lab\_name, ui->text\_info\_task->toPlainText(), test\_result, input\_data, expected\_output, output\_data}, database);

q.prepare(query);

q.exec();

int test\_id = getResult(getDBDataQuery("CheckedWorks", "id", {"person\_id", "date", "work\_name", "task", "status", "input\_data", "expected\_output", "output\_data"},

{person\_id, date, current\_lab\_name, ui->text\_info\_task->toPlainText(), test\_result, input\_data, expected\_output, output\_data})).toInt();

db\_model\_current\_test->setFilter(QString("id = %1").arg(test\_id));

db\_model\_current\_test->select();

}

Utils.h

#ifndef UTILS\_H

#define UTILS\_H

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <QFileDialog>

#include <QProcess>

#include <QFile>

#include <QByteArray>

#include <QString>

#include <QDebug>

#include <QMessageBox>

#include <QDateTime>

#include <QStandardItemModel>

#include <Windows.h>

*//* *сравнение* *строковых* *списков* *на* *равенство*

bool areListsEqual(*const* QStringList& list1, *const* QStringList& list2);

*//* *попытка* *отрыть* *файл,* *если* *такого* *нет,* *то* *он* *создается* *по* *пути* *file\_path*

QFile\* tryToOpenFile(QString file\_path, bool isTrunc = *false*);

*//* *считывание* *данных* *из* *файла,* *находящегося* *по* *пути* *path\_to\_file,* *и* *запись* *их* *в* *where\_to\_read*

void readFromFile(QString current\_path, QString& text\_code);

*//* *Получение* *имени* *файла* *с* *расширением*

QString getFIleName(QString path);

*//* *удаление* *всех* *файлов* *с* *именами* *file\_names,* *которые* *лежат* *по* *пути* *current\_path*

void delete\_file(QString current\_path, QStringList file\_names);\

*//* *заполнение* *окна* *с* *выбранными* *файлами*

void fillSelecteFilesTable(QStringList& paths, QListWidget\* list);

*//* *проверка* *расширения* *файла:* *1* *-* *.cpp* *\* *2* *-* *.h* *\* *0* *-* *другое*

int isCppOrHeader(QString fileName);

*//* *проверка* *наличие* *кириллицы* *в* *путях*

bool isCyrillic (QStringList files\_path);

*//* *запись* *в* *файл,* *находящегося* *по* *пути* *path\_to\_file,* *данных* *data\_to\_write*

void writeToFile(QString path\_to\_file, QString data\_to\_write, bool isTrunc = *false*);

*//* *получает* *список* *путей* *до* *файлов,* *и* *возвращает* *список* *имен* *файлов* *.cpp* *по* *этим* *путям*

QStringList get\_name\_from\_path(QStringList paths);

*//* *заключение* *в* *одинарные* *кавычки* *всех* *элементов* *списка*

void setToQuote(QStringList& data);

*//* *Компилирует* *код* *|* *directory\_path* *передается* *с* *"/"* *|*

void compile\_code(QString directory\_path, QStringList file\_names, bool isTest = *false*);

*//* *Создает* *батник* *для* *проверки* *тестов*

void bat\_for\_check\_test(QString directory\_path);

*//* *Получение* *текущей* *даты* *и* *времени*

QString getCurrentDateTime();

#endif *//* *UTILS\_H*

Utils.cpp

#include "utils.h"

***////*** ***проверка*** ***расширения*** ***файла:*** ***1*** ***-*** ***.cpp*** ***\*** ***2*** ***-*** ***.h*** ***\*** ***0*** ***-*** ***другое***

int isCppOrHeader(QString fileName)

{

int pos = fileName.size() - 1;

*while* (pos && fileName[pos] != ".")

{

pos--;

}

*if* (fileName.remove(0, ++pos) == "cpp")

*return* 1;

*else* *if* (fileName == "h")

*return* 2;

*else*

*return* 0;

}

*//* *попытка* *отрыть* *файл,* *если* *такого* *нет,* *то* *он* *создается* *по* *пути* *file\_path*

QFile\* tryToOpenFile(QString file\_path, bool isTranc)

{

QFile \*file = *new* QFile;

file->setFileName(file\_path);

*if* (isTranc)

{

*if* (!file->open(QIODevice::WriteOnly | QIODevice::Truncate))

{

*delete* file;

QMessageBox::critical(*nullptr*, "Ошибка", "Ошибка при открытии файла по пути:\n" + file\_path);

*return* *nullptr*;

}

}

*else* *if* (!file->open(QIODevice::ReadWrite))

{

*delete* file;

QMessageBox::critical(*nullptr*, "Ошибка", "Ошибка при открытии файла по пути:\n" + file\_path);

*return* *nullptr*;

}

*return* file;

}

*//* *запись* *в* *файл,* *находящегося* *по* *пути* *path\_to\_file,* *данных* *data\_to\_write*

void writeToFile(QString path\_to\_file, QString data\_to\_write, bool isTrunc)

{

QTextStream writeStream;

QFile \*file;

file = tryToOpenFile(path\_to\_file, isTrunc);

*if* (file == *nullptr*)

*return*;

writeStream.setDevice(file);

writeStream << data\_to\_write;

file->close();

*delete* file;

}

*//* *считывание* *данных* *из* *файла,* *находящегося* *по* *пути* *path\_to\_file,* *и* *запись* *их* *в* *where\_to\_read*

void readFromFile(QString path\_to\_file, QString& text\_code)

{

QFile \*file;

file = tryToOpenFile(path\_to\_file);

*if* (file == *nullptr*)

*return*;

QByteArray data = file->readAll();

text\_code = QString::fromUtf8(data); *//* *utf-8* *и* *utf-8* *со* *спецификацией*

file->close();

}

*//* *Получение* *имени* *файла* *с* *расширением*

QString getFIleName(QString path)

{

int pos = path.size() - 1;

*while* (pos && path[pos] != "/")

{

pos--;

}

*return* path.remove(0, ++pos);

}

*//* *заполнение* *окна* *с* *выбранными* *файлами*

void fillSelecteFilesTable(QStringList& paths, QListWidget\* list)

{

QStringList temp\_list\_path;

*for* (*const* QString& file\_path : qAsConst(paths))

{

QString file\_name = getFIleName(file\_path);

int extension = isCppOrHeader(file\_name);

*if* (extension)

{

temp\_list\_path.append(file\_path);

QString image\_path;

*if* (extension == 1)

{

image\_path = ":/img/image/cpp\_file.png";

}

*else* *if* (extension == 2)

{

image\_path = ":/img/image/h\_file.png";

}

QListWidgetItem \*item = *new* QListWidgetItem(QIcon(image\_path), file\_name);

list->addItem(item);

}

}

paths = temp\_list\_path; *//* *чтобы* *не* *было* *лишних* *путей*

}

*//* *проверка* *наличие* *кириллицы* *в* *путях*

bool isCyrillic (QStringList files\_path)

{

QString alphabets = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдеёжзийклмнпорстуфхцчшщъыьэюя";

*for* (*const* QString& file\_path : qAsConst(files\_path))

{

*for* (int index\_alph = file\_path.size()-1; index\_alph > 0; index\_alph--)

{

*for* (*const* QChar& alphs : qAsConst(alphabets))

{

*if* (file\_path[index\_alph] == alphs)

*return* 1;

}

}

}

*return* 0;

}

*//* *удаление* *всех* *файлов* *с* *именами* *file\_names,* *которые* *лежат* *по* *пути* *current\_path*

void delete\_file(QString current\_path, QStringList file\_names)

{

QString del\_command = "@echo off\n"

"chcp 1251 > null\n"

"cd /D " + current\_path + "\n";

*for* (*const* QString& file\_name : qAsConst(file\_names))

{

del\_command.append("del " + file\_name + "\n");

}

del\_command.append("\n del start\_compile.bat\ndel null\ndel delete\_bat.bat\nexit");

writeToFile(current\_path + "delete\_bat.bat", del\_command);

QProcess process\_del;

process\_del.start(current\_path + "delete\_bat.bat");

process\_del.waitForFinished();

}

*//* *получает* *список* *путей* *до* *файлов,* *и* *возвращает* *список* *имен* *файлов* *.cpp* *по* *этим* *путям*

QStringList get\_name\_from\_path(QStringList paths)

{

QStringList temp\_name;

*for* (QString path : paths)

{

*if* (path[path.size()-1] == "p")

{

int index;

*for* (index = path.size()-1; index > 0 && path[index] != "/"; index--);

path.remove(0, index+1);

temp\_name.append(path);

}

}

*return* temp\_name;

}

bool areListsEqual(*const* QStringList& list1, *const* QStringList& list2)

{

*return* list1.size() == list2.size() && std::includes(list1.begin(), list1.end(), list2.begin(), list2.end());

}

*//* *заключение* *в* *одинарные* *кавычки* *всех* *элементов* *списка*

void setToQuote(QStringList& data)

{

*for* (int i = 0; i < data.size(); i++)

{

data[i] = "'" + data[i] + "'";

}

}

*//* *Компилирует* *код* *|* *directory\_path* *передается* *с* *"/"* *|*

void compile\_code(QString directory\_path, QStringList file\_names, bool isTest)

{

QString compiled\_files = "";

*for* (*const* QString& file : qAsConst(file\_names))

{

compiled\_files.append('"' + file + '"');

compiled\_files.append(' ');

}

compiled\_files.chop(1);

QString str = "";

*if* (!isTest)

str = "user\_main\_code.exe < user\_input.txt\n"

"del user\_main\_code.exe\n";

*//* *открытие* *и* *запись* *в* *start\_compile.bat*

QString command = "@echo off\n"

"chcp 65001 > null\n"

"cd /D " + directory\_path + "\n"

"g++ -Wall " + compiled\_files + " -o user\_main\_code.exe\n"

+ str +

"exit";

writeToFile(directory\_path + "start\_compile.bat", command);

}

*//* *Создает* *батник* *для* *проверки* *тестов*

void bat\_for\_check\_test(QString directory\_path)

{

*//* *открытие* *и* *запись* *в* *start\_check\_test.bat*

QString command = "@echo off\n"

"chcp 65001 > null\n"

"cd /D " + directory\_path + "\n"

"user\_main\_code.exe < user\_input.txt\n"

"exit";

writeToFile(directory\_path + "start\_check\_test.bat", command);

}

*//* *Получение* *текущей* *даты* *и* *времени*

QString getCurrentDateTime()

{

QDateTime currentDateTime = QDateTime::currentDateTime();

QString formattedDateTime = currentDateTime.toString("HH:mm dd.MM.yyyy");

*return* formattedDateTime;

}

Database\_operations.h

#ifndef DATABASE\_OPERATIONS\_H

#define DATABASE\_OPERATIONS\_H

#include <QDir>

#include <QDebug>

#include <QSqlQuery>

#include <QSqlRecord>

#include <QSqlDatabase>

#include <QVariant>

#include <QVector>

*//* *открытие* *базы* *данных*

bool **tryToOpenDB**(QSqlDatabase, QString);

*//* *получение* *рузультата* *запроса*

QVariant **getResult**(QString query);

*//* *нахождение* *первого* *своббодного* *варианта*

int **getNextFreeNumberVar**(QSqlDatabase& db, int labworkId);

*//* *Нахождение* *первого* *свободного* *id* *в* *таблице*

int **getNextAvailableId**(QSqlDatabase& db, QString table, QString field);

*//* *создание* *готового* *запроса* *для* *БД*

QString **makeInsertQuery**(QString table, QStringList columns, QStringList data, QSqlDatabase& db);

*//* *Обновление* *данных* *в* *базе* *данных*

QString **updateData**(QString table, QStringList columns, QStringList data, QString condition);

*//* *Функция* *для* *определения* *существует* *ли* *запись* *с* *таким* *именем* *в* *базе* *данных*

bool **existsRecord**(QString table, QString column, QString value);

*//* *Функция* *для* *получения* *информации* *из* *поля*

QString **getDBDataQuery**(QString table, QString data, QString column, QString column\_condition);

*//* *Функция* *для* *получения* *информации* *из* *поля* *(несколько* *условий)*

QString **getDBDataQuery**(QString table, QString data, QStringList column, QStringList column\_condition);

*//* *получение* *всех* *данных,* *с* *нужным* *полем*

QVector<QVariantList> **getAllInTableWhere**(QSqlDatabase db, QString table, QString field, QString condition, bool NotEqual = *false*);

*//* *получение* *всех* *данных,* *с* *полемм,* *содержащем* *подстроку*

QVector<QVariantList> **getAllLIKE**(QSqlDatabase db, QString table, QString field, QString substr);

*//* *заключение* *в* *одинарные* *кавычки* *всех* *элементов* *списка*

void **setToQuote**(QStringList&);

#endif *//* *DATABASE\_OPERATIONS\_H*

Database\_operations.cpp

#include "realization/database\_operations.h"

#include "realization/utils.h"

#include "mainwindow.h"

#include "testwindow.h"

*//* *открытие* *базы* *данных*

bool tryToOpenDB(QSqlDatabase database, QString db\_name)

{

database.setDatabaseName(QDir::currentPath() + "/" + db\_name + ".db");

*if* (!database.open())

{

qDebug() << "Ошибка открытия базы данных:\n" + database.lastError().text();

*return* *false*;

}

*//* *проверка* *на* *наличие* *нужных* *таблиц*

*if* (db\_name == "Lab\_works")

{

*if* (database.tables().size() != 6)

{

database.exec("CREATE TABLE LabWork (id INTEGER, name TEXT UNIQUE, description TEXT,PRIMARY KEY(id AUTOINCREMENT))");

database.exec("CREATE TABLE Variants (id INTEGER PRIMARY KEY AUTOINCREMENT, labwork\_id INTEGER, conditions TEXT, number\_var INTEGER, FOREIGN KEY (labwork\_id) REFERENCES LabWork(id));");

database.exec("CREATE TABLE Tests (id INTEGER PRIMARY KEY AUTOINCREMENT, variant\_id INTEGER, input\_data TEXT, output\_data TEXT, FOREIGN KEY (variant\_id) REFERENCES Variants(id));");

database.exec("CREATE TABLE Person (id INTEGER,date TEXT, person\_name TEXT, PRIMARY KEY(id AUTOINCREMENT))");

database.exec("CREATE TABLE CheckedWorks (id INTEGER, person\_id INTEGER, date TEXT, work\_name TEXT, task TEXT, status TEXT, input\_data TEXT, expected\_output TEXT, output\_data TEXT,PRIMARY KEY( id AUTOINCREMENT))");

}

}

*return* *true*;

}

*//* *получение* *рузультата* *запроса*

QVariant getResult(QString query)

{

QSqlQuery q(query);

*if* (q.next())

{

*return* q.value(0);

}

*return* -1;

}

*//* *нахождение* *первого* *свободного* *варианта*

int getNextFreeNumberVar(QSqlDatabase& db, int labworkId)

{

QSqlQuery query(db);

*//* *Найти* *максимальный* *number\_var* *для* *заданного* *labwork\_id*

query.prepare("SELECT COALESCE(MAX(number\_var), 0) FROM Variants WHERE labwork\_id = :labworkId");

query.bindValue(":labworkId", labworkId);

*if* (!query.exec() || !query.next())

{

qWarning() << "Ошибка при выполнении запроса:" << query.lastError().text();

*return* -1;

}

int lastVar = query.value(0).toInt();

*//* *Проверить* *каждый* *number\_var* *от* *1* *до* *lastVar*

*for* (int i = 1; i <= lastVar + 1; ++i)

{

query.prepare("SELECT COUNT(\*) FROM Variants WHERE labwork\_id = :labworkId AND number\_var = :numberVar");

query.bindValue(":labworkId", labworkId);

query.bindValue(":numberVar", i);

*if* (!query.exec() || !query.next())

{

qWarning() << "Ошибка при выполнении запроса:" << query.lastError().text();

*return* -1;

}

int count = query.value(0).toInt();

*if* (count == 0)

{

*return* i; *//* *возвращается* *первый* *свободный* *number\_var*

}

}

*//* *Если* *все* *number\_var* *от* *1* *до* *lastVar* *заняты,* *возвращается* *следующий* *number\_var*

*return* lastVar + 1;

}

*//* *Нахождение* *первого* *свободного* *id* *в* *таблице*

int getNextAvailableId(QSqlDatabase& db, QString table, QString field)

{

QSqlQuery query(db);

query.prepare("SELECT " + field + " FROM " + table + " ORDER BY " + field + " DESC LIMIT 1");

*if* (!query.exec())

{

qDebug() << "Ошибка при выполнении запроса: " << query.lastError().text();

*return* -1; *//* *Возвращаем* *-1* *или* *другое* *значение,* *указывающее* *на* *ошибку*

}

*if* (query.next())

{

int lastId = query.value(0).toInt();

*for* (int i = 1; i <= lastId; ++i)

{

query.prepare("SELECT " + field + " FROM " + table + " WHERE " + field + " = :id");

query.bindValue(":id", i);

*if* (!query.exec() || !query.next())

{

*return* i; *//* *Возвращаем* *первый* *свободный* *id*

}

}

*return* lastId + 1; *//* *Если* *все* *id* *от* *1* *до* *lastId* *заняты,* *возвращаем* *следующий* *id*

}

*else*

{

*//* *Если* *нет* *записей,* *возвращаем* *следующий* *id,* *который* *был* *бы* *автоматически* *сгенерирован*

*return* 1;

}

}

*//* *создание* *готового* *запроса* *для* *БД*

QString makeInsertQuery(QString table, QStringList columns, QStringList data, QSqlDatabase& db)

{

setToQuote(data);

int id = getNextAvailableId(db, table, "id");

QString query = "INSERT INTO " + table + "(id, " + columns.join(", ") + ") "

"VALUES ('" + QString::number(id) + "', " + data.join(", ") + ");";

*return* query;

}

*//* *Обновление* *данных* *в* *базе* *данных*

QString updateData(QString table, QStringList columns, QStringList data, QString condition)

{

setToQuote(data);

QString query = "UPDATE " + table + " SET ";

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

{

query += columns[i] + " = " + data[i];

*if* (i < columns.size() - 1)

query += ", ";

}

query += " WHERE " + condition + ";";

*return* query;

}

*//* *Функция* *для* *определения* *существует* *ли* *запись* *с* *таким* *именем* *в* *базе* *данных*

bool existsRecord(QString table, QString column, QString value)

{

QString q = "SELECT COUNT(\*) FROM " + table + " WHERE " + column + " = '" + value + "';";

QSqlQuery query(q);

*if* (query.lastError().type() != QSqlError::NoError)

{

qDebug() << "Ошибка при выполнении запроса: " << query.lastError().text();

*return* *false*;

}

*//* *Проверка* *результата* *запроса*

*if* (query.next())

{

int count = query.value(0).toInt();

*return* count > 0;

}

*return* *false*;

}

*//* *Функция* *для* *получения* *информации* *из* *поля*

QString getDBDataQuery(QString table, QString data, QString column, QString column\_condition)

{

*return* getDBDataQuery(table, data, QStringList (column), QStringList(column\_condition));

}

*//* *Функция* *для* *получения* *информации* *из* *поля* *(несколько* *условий)*

QString getDBDataQuery(QString table, QString data, QStringList column, QStringList column\_condition)

{

QString condition;

*for* (int i = 0; i < column.size(); i++)

{

condition += " AND " + column[i] + " = '" + column\_condition[i] + "'";

}

condition.remove(0, 5);

QString query = "SELECT " + data + " FROM " + table + " WHERE ("

+ condition + ");";

*return* query;

}

*//* *получение* *всех* *данных,* *с* *нужным* *полем*

QVector<QVariantList> getAllInTableWhere(QSqlDatabase db, QString table, QString field, QString condition, bool NotEqual)

{

QSqlQuery query(db);

QVector<QVariantList> result;

*//* *Формирование* *SQL* *запроса*

*if* (!NotEqual)

query.prepare("SELECT \* FROM " + table + " WHERE " + field + " = :cond");

*else*

query.prepare("SELECT \* FROM " + table + " WHERE " + field + " != :cond");

query.bindValue(":cond", condition);

*//* *Выполнение* *запроса*

*if* (query.exec())

{

*//* *Считка* *результата*

*while* (query.next())

{

QVariantList row;

*for* (int i = 0; i < query.record().count(); ++i)

{

row.append(query.value(i));

}

result.append(row);

}

}

*else*

{

*//* *Обработка* *ошибки* *выполнения* *запроса*

qDebug() << "Ошибка при выполнении запроса:" << query.lastError().text();

}

*return* result;

}

*//* *получение* *всех* *данных,* *с* *полемм,* *содержащем* *подстроку*

QVector<QVariantList> getAllLIKE(QSqlDatabase db, QString table, QString field, QString substr)

{

QSqlQuery query(db);

QVector<QVariantList> result;

*//* *Формирование* *SQL* *запроса*

query.prepare("SELECT \* FROM " + table + " WHERE " + field + " LIKE :substr");

query.bindValue(":substr", substr);

*//* *Выполнение* *запроса*

*if* (query.exec())

{

*//* *Считка* *результата*

*while* (query.next())

{

QVariantList row;

*for* (int i = 0; i < query.record().count(); ++i)

{

row.append(query.value(i));

}

result.append(row);

}

}

*else*

{

*//* *Обработка* *ошибки* *выполнения* *запроса*

qDebug() << "Ошибка при выполнении запроса:" << query.lastError().text();

}

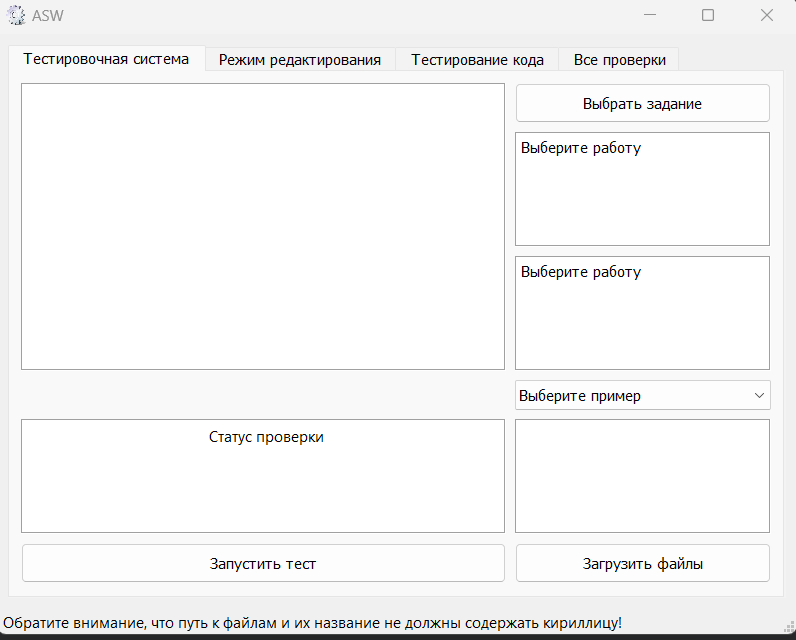
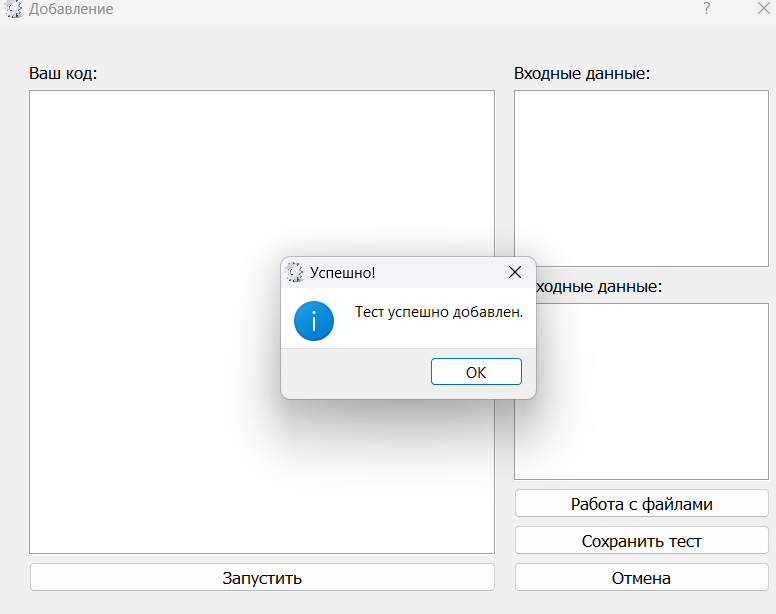
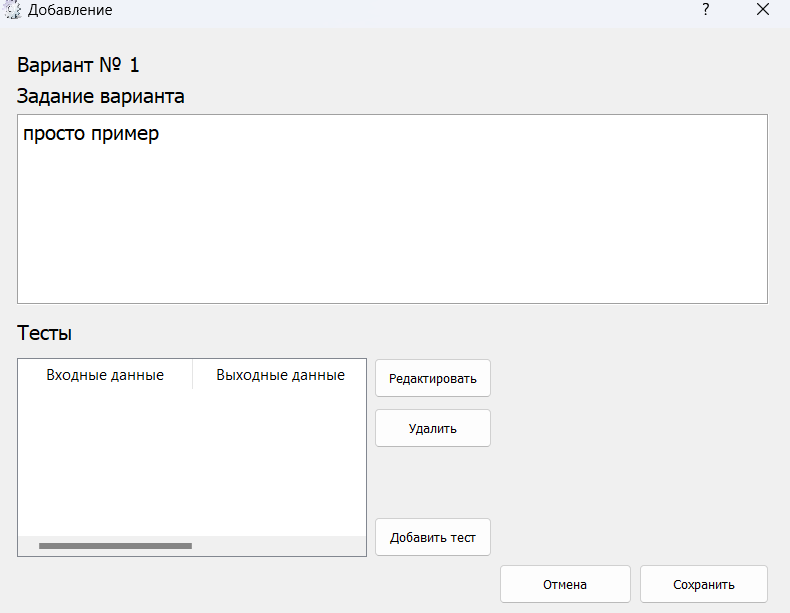
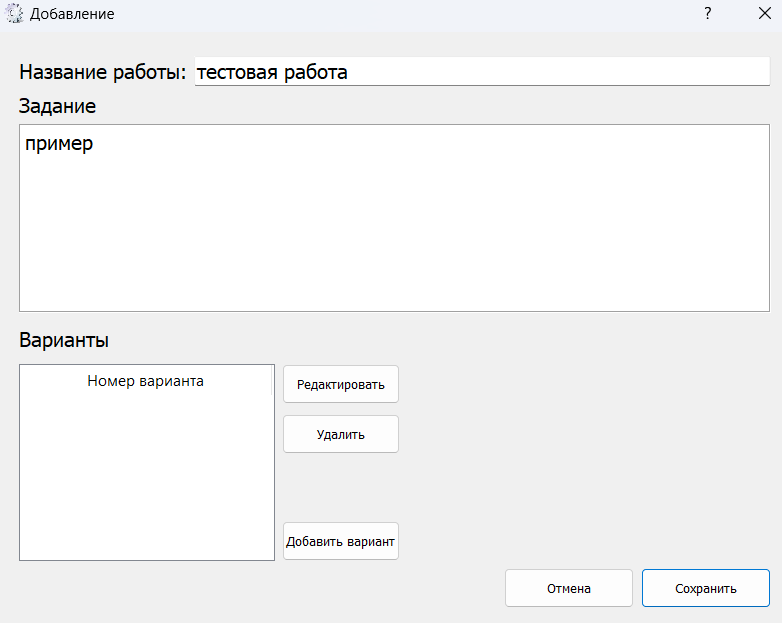
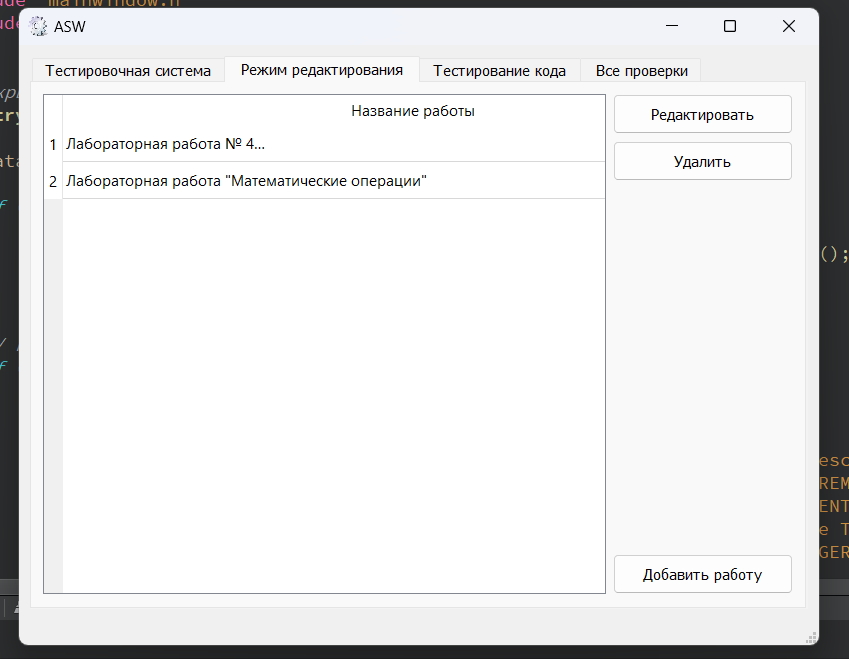
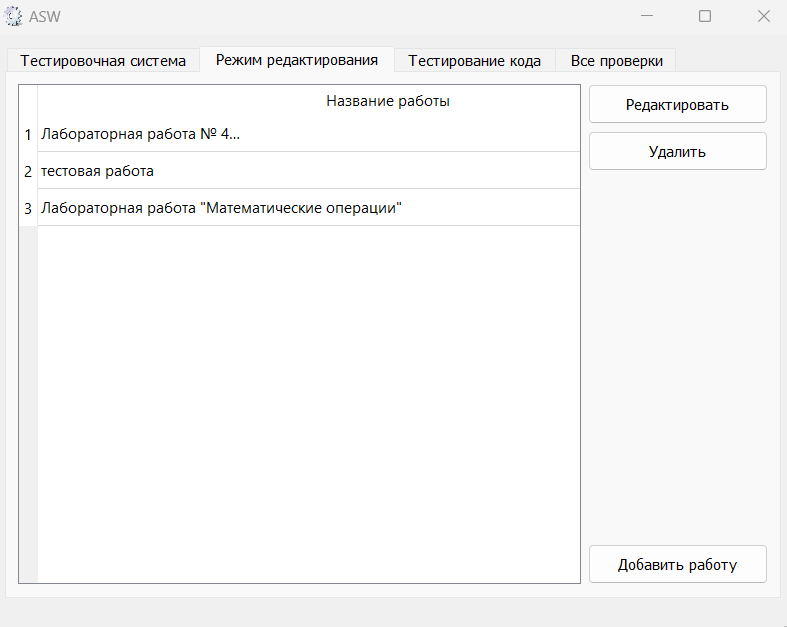
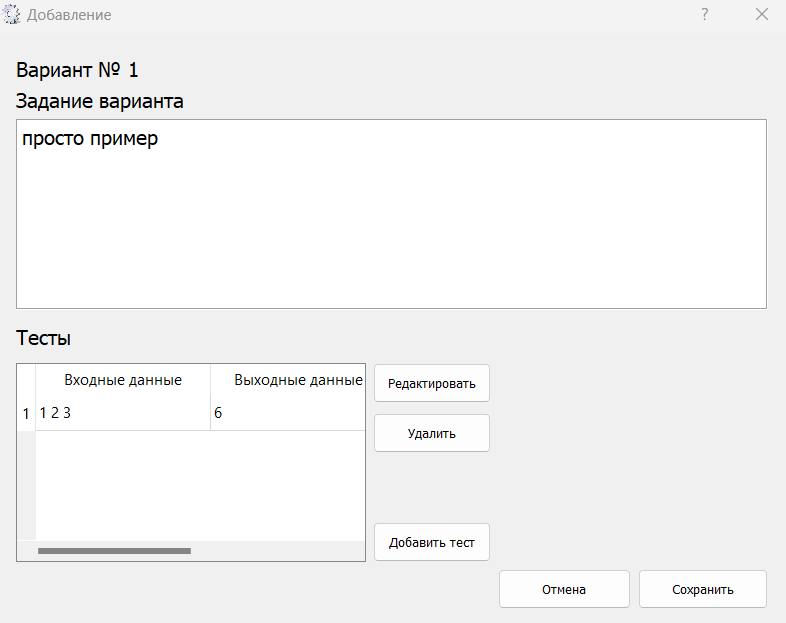
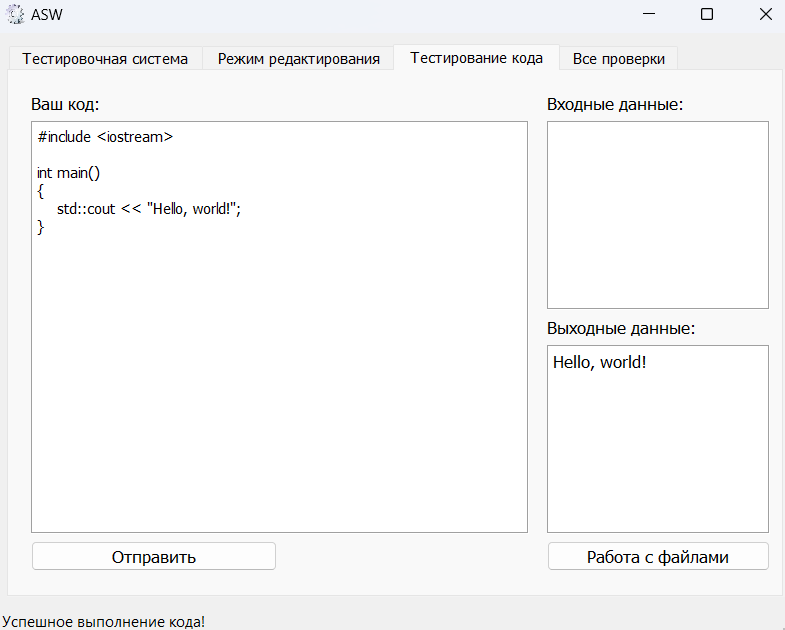
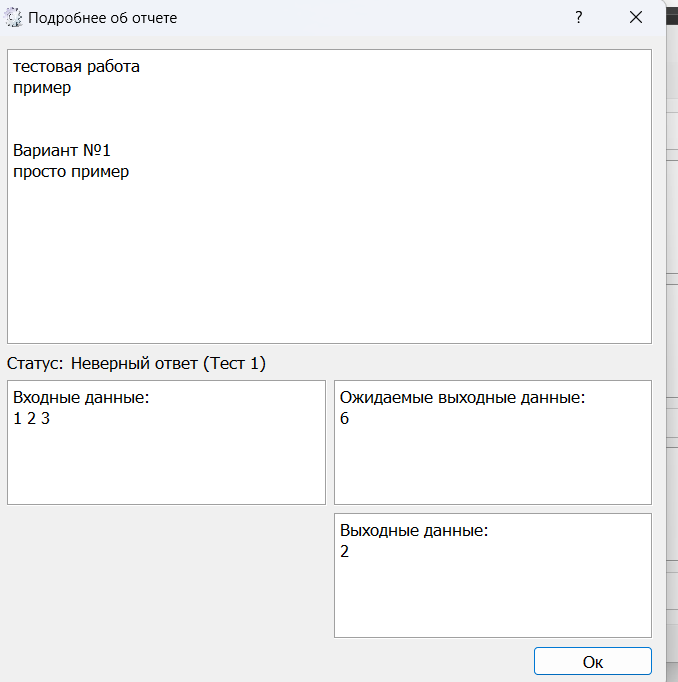
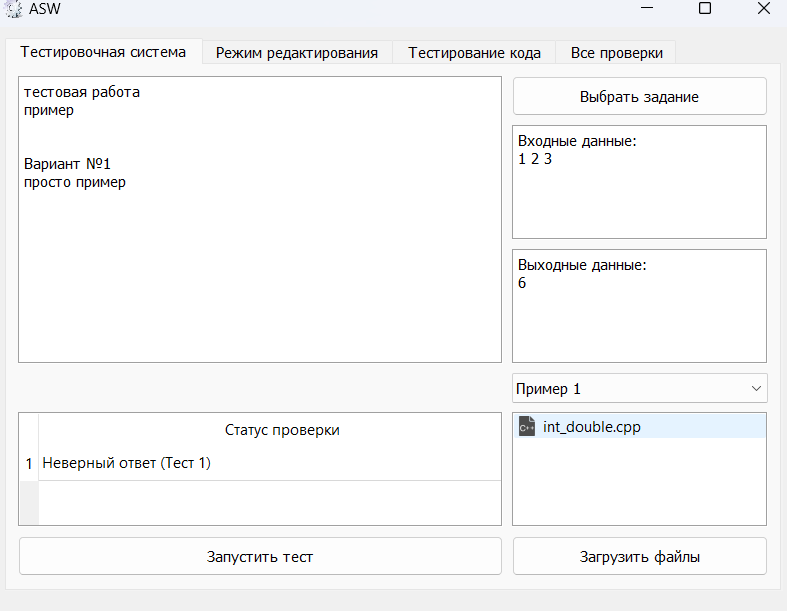
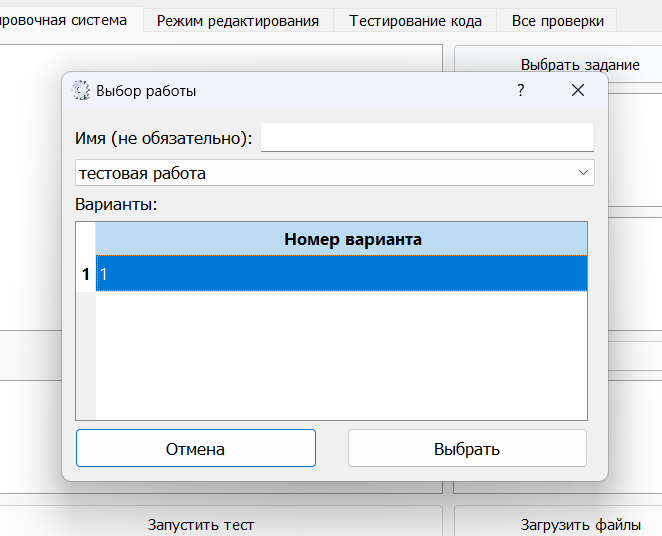
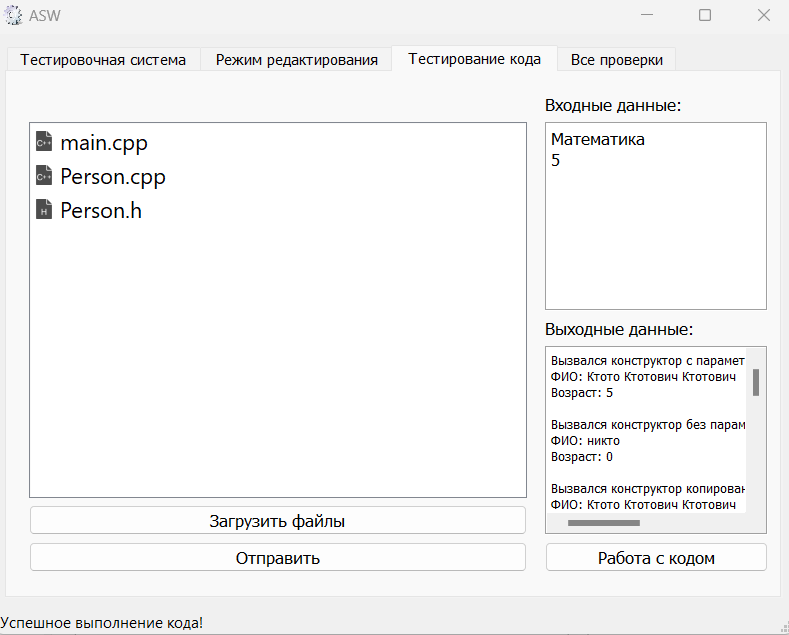
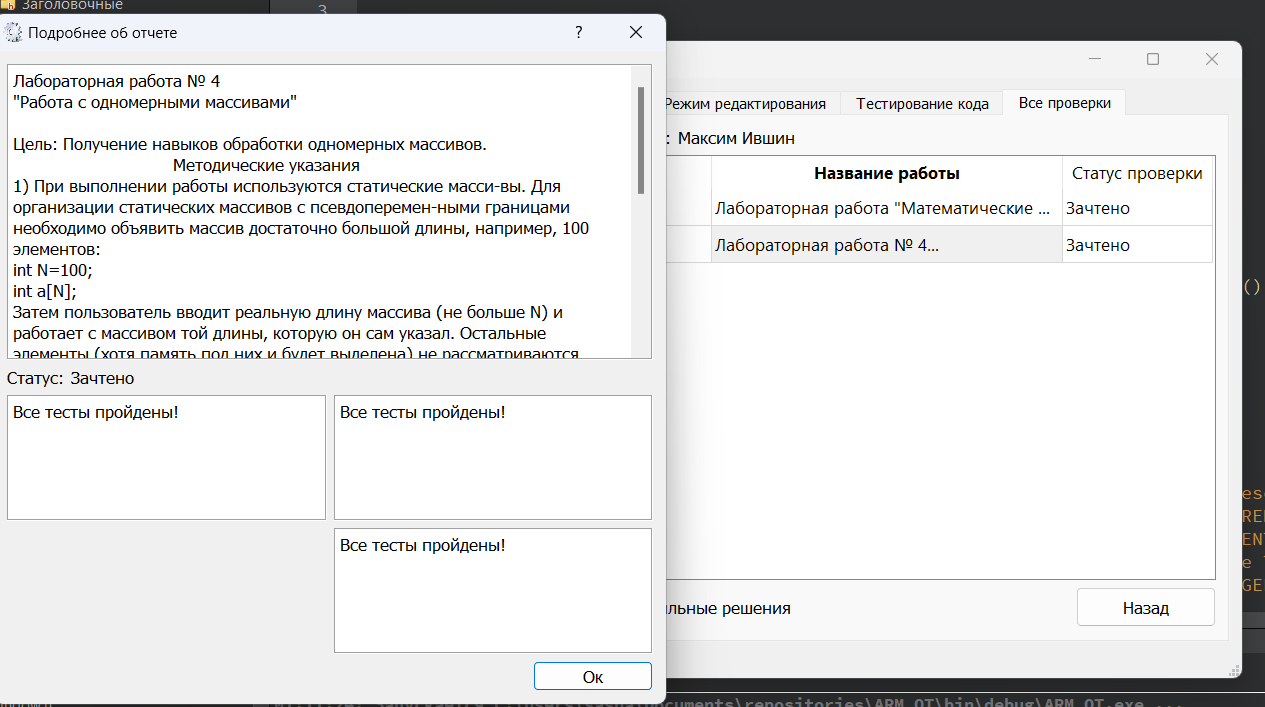
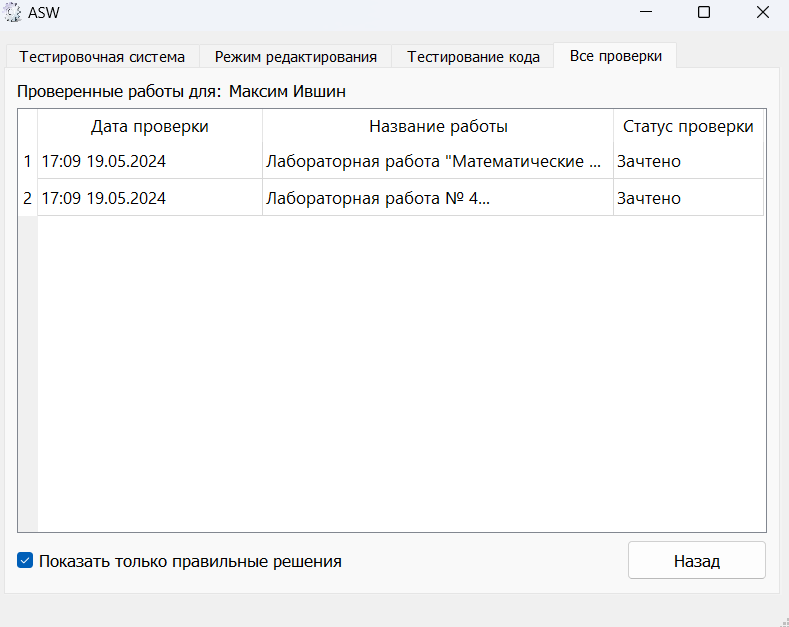
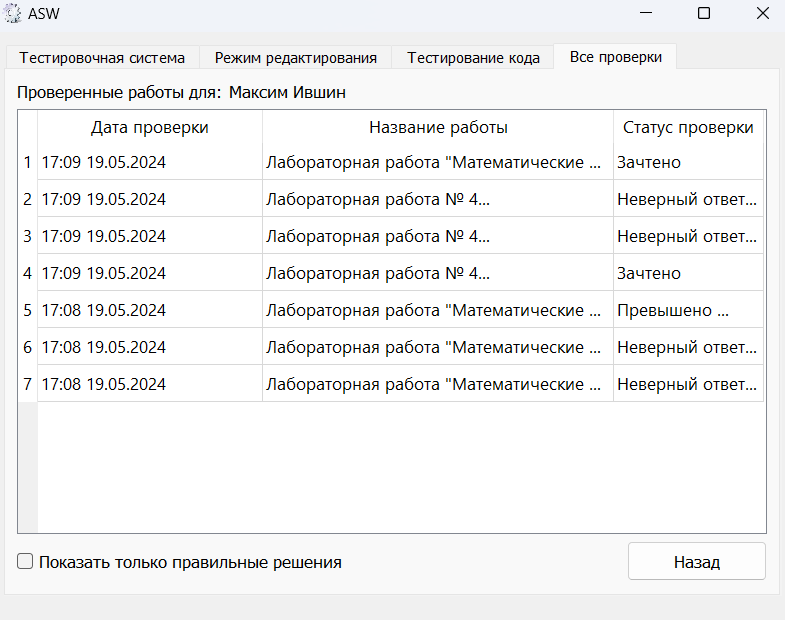
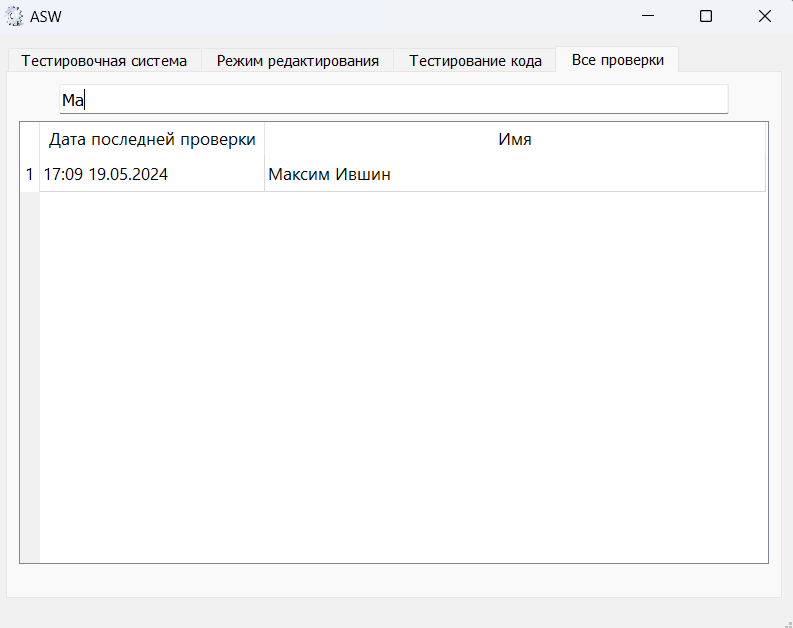
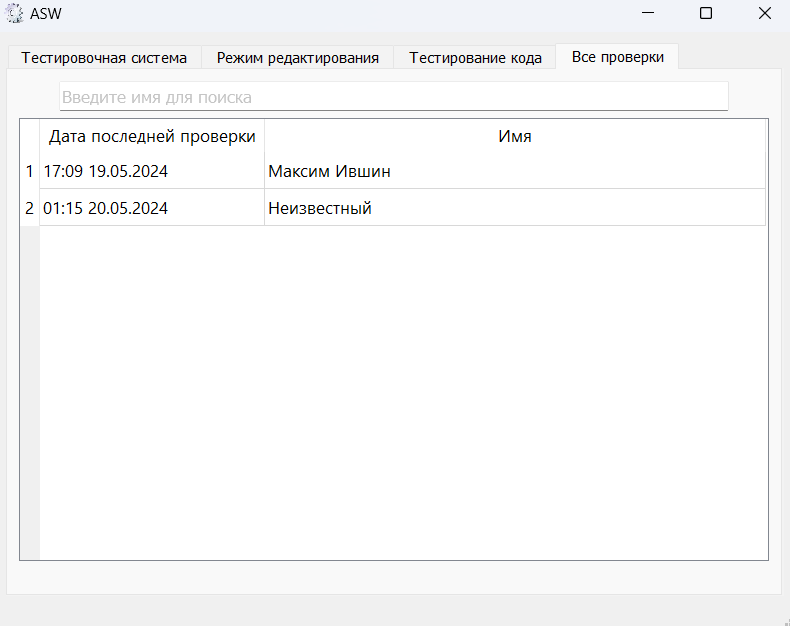
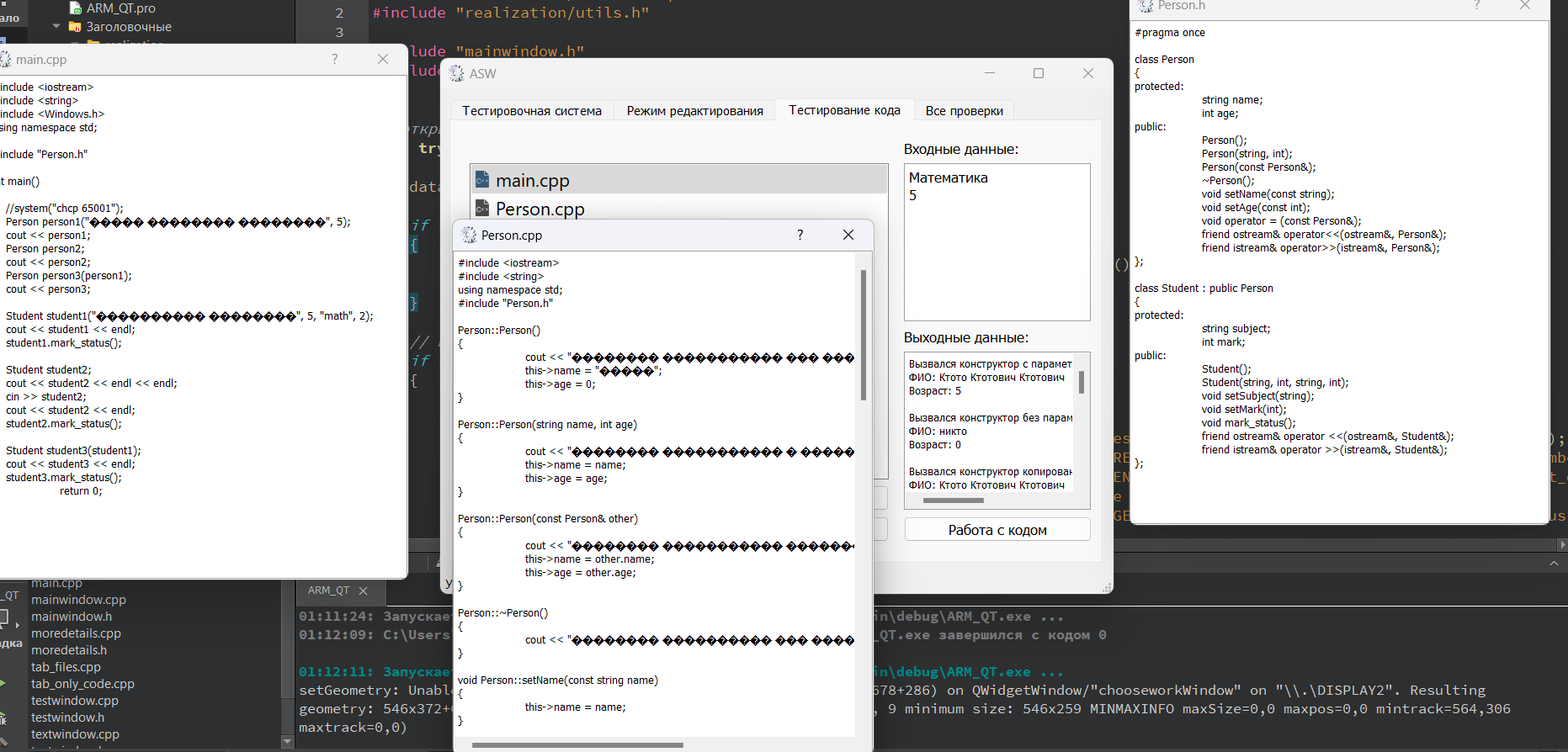
*return* result;

}

**Вывод**

Было создано приложение, позволяющее создавать лабораторные работы и тесты к ним, также их редактирование. Было реализовано тестирование пользовательского кода, в соответствии с выбранными им тестами. Во время выполнения работы авторы ознакомились с фреймворком Qt, получили опыт создания оконных приложений, отточили навыки программирования на C++.

**Результаты работы**

************

**GitHub**

[**ARM\_QT**](https://github.com/geroineee/ARM_QT)