

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Объектно-ориентированное программирование»
Тема: Создание игрового поля

Студентка гр. 9381

Москаленко Е.М.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2020

Цель работы.

Изучить основные принципы объектно-ориентированного программирования, реализовать классы клетки и игрового поля на языке программирования C++ и так же реализовать Graphical user interface с помощью фреймворка Qt.

Задание.

Написать класс игрового поля, которое представляет из себя прямоугольник (двумерный массив). Для каждого элемента поля должен быть создан класс клетки. Клетка должна отображать, является ли она проходимой, а также информацию о том, что на ней находится. Также, на поле должны быть две особые клетки: вход и выход.

При реализации поля запрещено использовать контейнеры из stl

Обязательные требования:

- Реализован класс поля
- Реализован класс клетки
- Для класса поля написаны конструкторы копирования и перемещения, а также операторы присваивания и перемещения
- Поле сохраняет инвариант - из любой клетки можно провести путь до любой другой
- Гарантированно отсутствует утечки памяти

Дополнительные требования:

- Поле создается с использованием паттерна **Синглтон**
- Для обхода по полю используется паттерн **Итератор**. Итератор должен быть совместим со стандартной библиотекой.

Ход работы.

Написание работы происходило в среде разработки QtCreator с использованием фреймворка Qt. Написаны следующие классы:

1) **Square** – класс единичной клетки игрового поля. Имеет приватные поля **Type type** – тип клетки, где **Type** = {CLOSE, OPEN, ENTRY, EXIT} – перечисление, отвечающее за то, закрыта/открыта ли клетка, и **bool free** – булевая переменная, отвечающая за то, занята ли клетка в данный момент каким-либо объектом.

Помимо конструктора в классе реализованы следующие функции:

void setType(Type) – присваивает клетке тот или иной тип.

bool isFree() – возвращает true или false в зависимости от того, занята ли клетка.

Type getType() – возвращает тип клетки **Type**.

void setDestroyed() – делает клетку занятой/разрушенной, присваивая полю **free**

2) **Field** – класс поля. Создается с помощью паттерна Синглтон (одиночка). Реализован конструктор по умолчанию, а так же конструкторы копирования и перемещения, операторы присваивания и перемещения. Содержит приватные поля, отвечающие за количество клеток по вертикали и горизонтали, и двумерный массив типа класса **Square**. Также реализована функция **makeField()** для создания поля с клетками различного типа.

3) **SingletonDestroyer** - класс, предназначенный для автоматического разрушения уникального объекта **Field**. Класс **Field** имеет статический член **SingletonDestroyer**, который инициализируется при первом вызове **Field::getInstance()** создаваемым объектом **Field**. При завершении

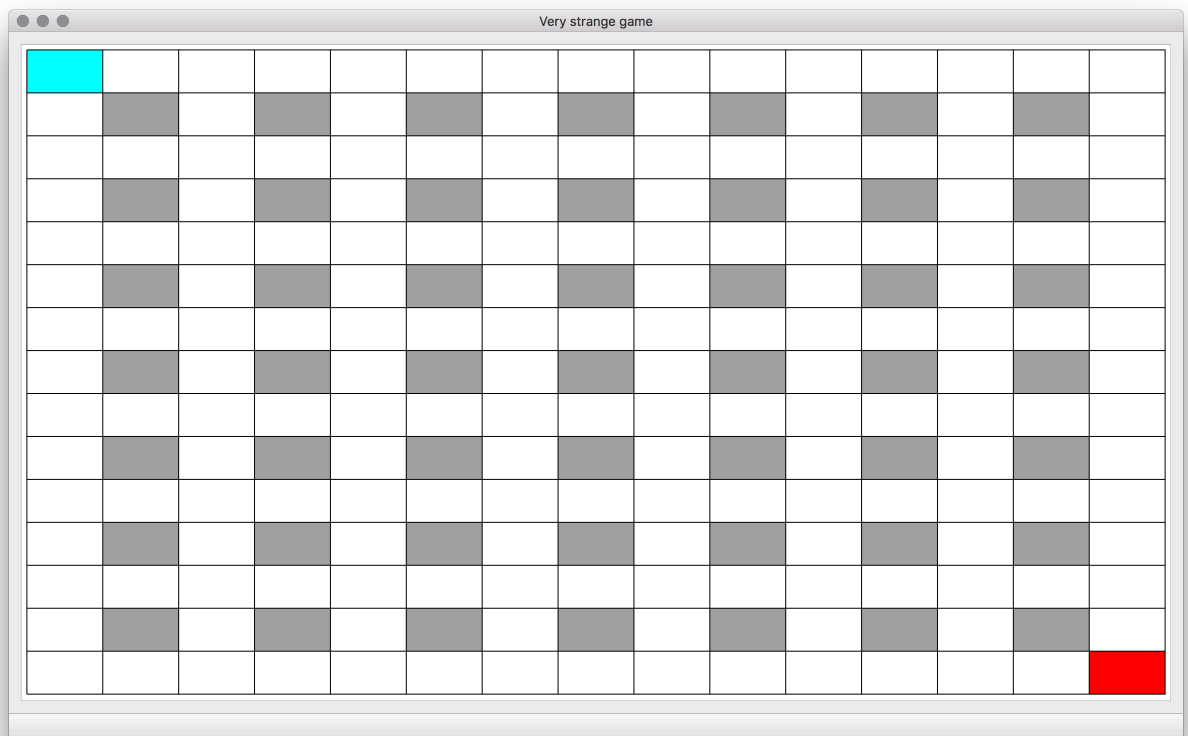
программы этот объект будет автоматически разрушен деструктором SingletonDestroyer (для этого SingletonDestroyer объявлен другом класса Field).

4) MainWindow – класс для графического отображения. Объект отображается в функции main(). Содержит приватные поля: указатель на поле типа field, указатель на графическую сцену QGraphicsScene* scene и массив указателей на прямоугольные объекты QGraphicsRectItem*** rects, которые отображаются на сцене.

Поле имеет две особенные точки – вход и выход. Вход расположен на позиции (0,0), выход – соответственно по диагонали, в правом нижнем углу поля. «Закрытые» клетки закрашены серым.

Тестирование.

Игровое поле успешно отображается.

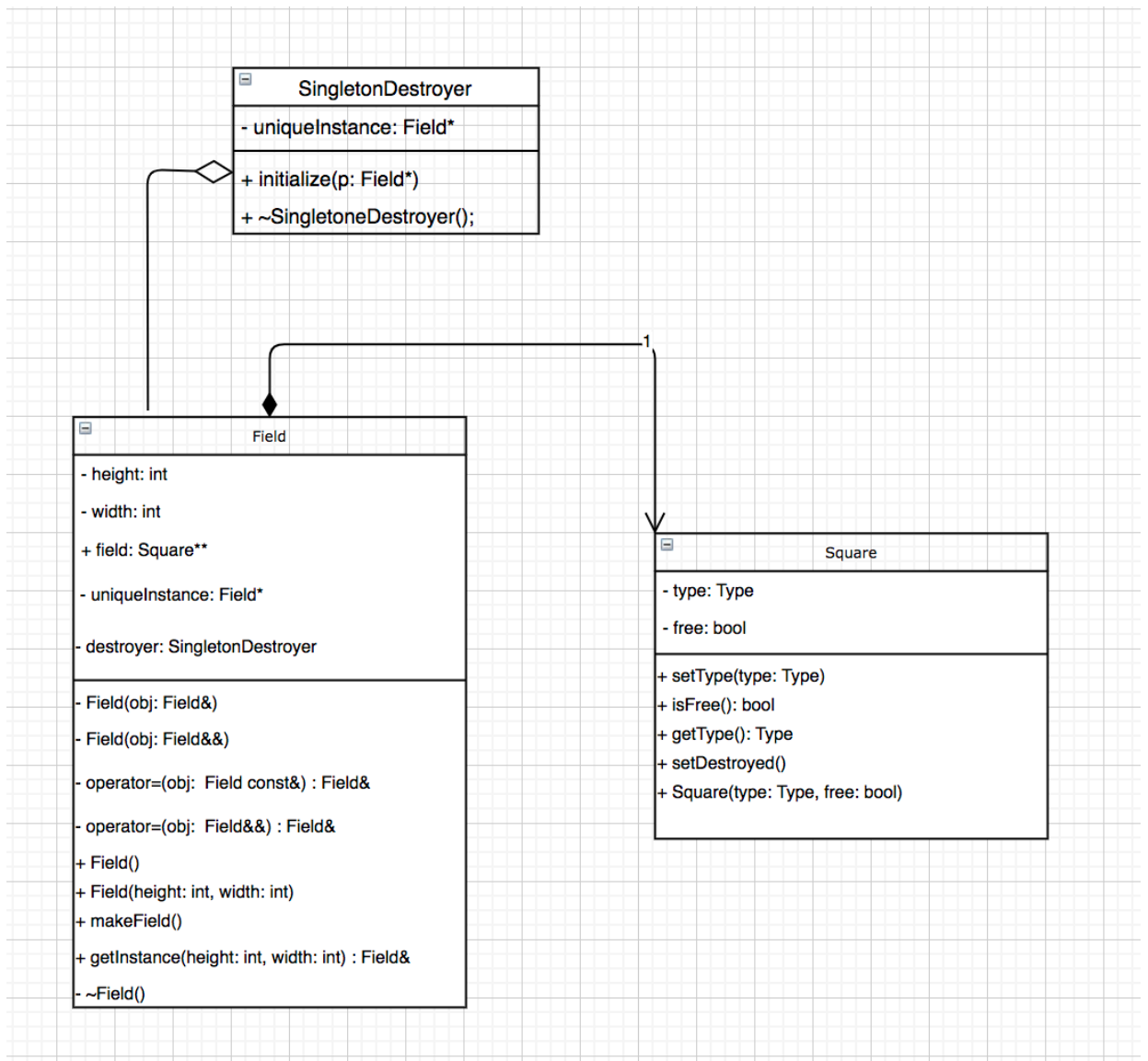


Паттерн Синглтон работает верно. Создается лишь один объект.

```
untitled12 [~/untitled12] - .../main.cpp
main.cpp
1  #include <iostream>
2  #include "Field.h"
3  #include "Square.h"
4
5  int main() {
6      std::cout << "Checking Singleton!" << std::endl;
7      Field* field = &Field::getInstance(HEIGHT,WIDTH);
8      std::cout << field << "\n";
9      Field* field1 = &Field::getInstance(HEIGHT,WIDTH);
10     std::cout << field1;
11     return 0;
12 }
```

```
/Users/elizaveta/untitled12/cmake-build-debug/untitled12
Checking Singleton!
0x7fb8d4c02800
0x7fb8d4c02800
Process finished with exit code 0
```

UML-диаграмма



Выводы.

Были изучены основные принципы объектно-ориентированного программирования, реализованы классы клетки и игрового поля с необходимыми конструкторами и операторами. Для отображения игрового поля реализован Graphical user interface.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

Файл square.h

```
#ifndef SQUARE_H
#define SQUARE_H
#include <QImage>
#include <QGraphicsRectItem>

enum Type{
    CLOSE, //0
    OPEN,  //1
    ENTRY, //2
    EXIT   //3
};

class Square
{
    // Q_OBJECT
public:
    Square(Type type = CLOSE, bool free = false):type(type),free(free){};
    void setType(Type);
    bool isFree();
    Type getType();
    void setDestroyed();
private:
    Type type;
    bool free;
};

#endif // SQUARE_H
```

Файл square.cpp

```
#include "square.h"

bool Square::isFree(){
    return free;
}

Type Square::getType(){
    return type;
}

void Square::setType(Type type){
    this->type = type;
    if (this->type == CLOSE)
        free = false;
    else
        free = true;
}

void Square::setDestroyed(){
    free = false;
    type = CLOSE;
}
```

Файл field.h

```
#ifndef FIELD_H
#define FIELD_H
#include "square.h"
#define HEIGHT 15
#define WIDTH 15
class Field;

class SingletonDestroyer
{
private:
    Field* uniqueInstance;
public:
    ~SingletonDestroyer();
    void initialize(Field* p);
};

class Field
{
public:
    static Field& getInstance(int height, int width);
    Field();
    Field(int height, int width);
    void makeField();
    Square** field;

private:
    Field(const Field& obj);
    Field(Field&& obj);
    Field& operator=(Field const& obj);
    Field& operator=(Field&& obj);
    ~Field();
    int height;
    int width;
    friend class SingletonDestroyer;
    static Field* uniqueInstance;
    static SingletonDestroyer destroyer;
};

#endif // FIELD_H
```

Файл field.cpp

```
#include "field.h"

Field *Field::uniqueInstance = nullptr;
SingletonDestroyer Field::destroyer;

Field::Field() {
    field = nullptr;
}

SingletonDestroyer::~SingletonDestroyer() {
    delete uniqueInstance;
}

void SingletonDestroyer::initialize(Field* p ) {
    uniqueInstance = p;
}

Field& Field::getInstance(int height,int width) {
```



```

        if(!uniqueInstance)    {
            uniqueInstance = new Field(height,width);
            destroyer.initialize(uniqueInstance);
        }
        return *uniqueInstance;
    }

Field::Field(int height, int width)
{
    this->height = height;
    this->width = width;
    field = new Square*[HEIGHT];
    for (auto i = 0; i < HEIGHT; i++)
        field[i] = new Square[WIDTH];
}

Field::~Field(){
    for (auto i = 0; i < HEIGHT; i++)
        delete[] field[i];
    delete [] field;
}

void Field::makeField(){
    for(auto i = 0; i < height; i++){
        for(auto j = 0; j < width; j++){
            if((i % 2 != 0 && j % 2 != 0) && //если ячейка нечетная по x
и y,
                (i < height-1 && j < width-1)) //и при этом находится в
пределах стен лабиринта
                field[i][j].setType(CLOSE); //то это СТЕНА
            else field[i][j].setType(OPEN); //в остальных
случаях это КЛЕТКА.
        }
    }
    field[0][0].setType(ENTRY);
    field[HEIGHT-1][WIDTH-1].setType(EXIT);
}

Field::Field(const Field& obj){ //конструктор копирования
    height = obj.height;
    width = obj.width;
    field = new Square*[height];
    for (auto i = 0; i < height; i++){
        field[i] = new Square[width];
        for (int j = 0; j < width; j++)
            field[i][j] = obj.field[i][j];
    }
}

Field::Field(Field&& obj): height(obj.height), width(obj.width){
    field = obj.field; //конструктор перемещения
    obj.height = 0;
    obj.width = 0;
    obj.field = nullptr;
}

Field &Field::operator=(Field const& obj){ //оператор присваивания
копированием
    if (this != &obj)
        for (auto i = 0; i < height; i++)

```

```

        delete [] field[i];
    delete [] field;
    height = obj.height;
    width = obj.width;
    field = new Square*[height];
    for (auto i = 0; i < height; i++){
        field[i] = new Square[width];
        for (auto j = 0; j < width; j++)
            field[i][j] = obj.field[i][j];
    }
    return *this;
}

```

```

Field& Field::operator=(Field&& obj){ //оператор присваивания перемещением
    if (&obj == this)
        return *this;
    for (auto i = 0; i < height; i++)
        delete[] field[i];
    delete [] field;
    field = obj.field;
    height = obj.height;
    width = obj.width;
    obj.field = nullptr;
    return *this;
}

```

Файл mainwindow.h

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QGraphicsScene>
#include <QMainWindow>
#include "square.h"
#include "field.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
private:
    Ui::MainWindow *ui;
    QGraphicsScene* scene;
    Field* field;
    QGraphicsRectItem*** rects;
};
#endif // MAINWINDOW_H

```

Файл mainwindow.cpp

```

#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QGraphicsRectItem>
#include <QDesktopWidget>

```

```

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
, ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    scene = new QGraphicsScene();
    field = &Field::getInstance(HEIGHT,WIDTH);
    field->makeField();

    QSize screenSize = QDesktopWidget().availableGeometry(this).size();
    ui->graphicsView->setFixedSize(screenSize*0.8);
    int rectHeight = screenSize.height()*0.8/HEIGHT;
    int rectWidth = screenSize.width()*0.8/WIDTH;

    QGraphicsRectItem*** rects = new QGraphicsRectItem**[HEIGHT];
    for (auto i = 0; i < HEIGHT; i++)
        rects[i] = new QGraphicsRectItem*[WIDTH];

    for (auto i = 0; i < WIDTH; i++){
        for(auto j = 0; j < HEIGHT; j++){
            rects[i][j] = scene->addRect(QRectF(0 + i*rectWidth, 0 + j*rectHeight,
rectWidth,rectHeight));
            Type cell = field->field[i][j].getType();
            switch(cell){
                case ENTRY:
                    rects[i][j]->setBrush(QBrush(Qt::cyan));
                    break;
                case CLOSE:
                    rects[i][j]->setBrush(QBrush(Qt::gray));
                    break;
                case EXIT:
                    rects[i][j]->setBrush(QBrush(Qt::red));
                    break;
                case OPEN:
                    rects[i][j]->setBrush(QBrush(Qt::white));
                    break;
            }
        }
    }
    ui->graphicsView->setScene(scene);
}

MainWindow::~MainWindow()
{
    delete ui;
}

```