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 классов, конструкторов и методов классов

Студент гр. 0304 _____ Максименко Е.М.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2021

Цель работы.

Изучить работу с классами в языке программирования C++, научиться применять объектно-ориентированный подход на примере создания игрового поля и его элементов.

Задание

Игровое поле представляет из себя прямоугольную плоскость разбитую на клетки. На поле на клетках в дальнейшем будут располагаться игрок, враги, элементы взаимодействия. Клетка может быть проходимой или непроходимой, в случае непроходимой клетки, на ней ничего не может располагаться. На поле должны быть две особые клетки: вход и выход. В дальнейшем игрок будет появляться на клетке входа, а затем выполнив определенный набор задач дойти до выхода.

При реализации класса поля запрещено использовать контейнеры из stl

Требования:

- Реализовать класс поля, который хранит набор клеток в виде двумерного массива.
- Реализовать класс клетки, которая хранит информацию о ее состоянии, а также того, что на ней находится.
- Создать интерфейс элемента клетки.
- Обеспечить появление клеток входа и выхода на поле. Данные клетки не должны быть появляться рядом.
- Для класса поля реализовать конструкторы копирования и перемещения, а также соответствующие операторы.
- Гарантировать отсутствие утечки памяти.

Выполнение работы

Исходный код программы приведен в приложении А. Диаграмма классов приведена в приложении Б.

1. Создание класса *Cell*, отвечающего за работу с клеткой поля. Данный класс содержит в себе такие данные, как расположение ячейки на поле в виде полей *size_t x* и *size_t y*, наличие в данной клетки стены *bool wall*, вектор элементов, которые будут располагаться на клетке *std::vector<std::shared_ptr<Item>> items*. Конструктор *Cell(size_t x, size_t y, bool wall)* создает клетку согласно заданным координатам и параметру наличия стены в данной клетке. Класс содержит виртуальный метод *CellType getType()*, который возвращает тип клетки: он может быть одним из 3-х, объявленных в перечислении *CellType*. Также класс содержит перегруженные конструкторы копирования и перемещения, а так же соответствующие операторы. Класс содержит геттеры *size_t getX()*, *size_t getY()*, *bool getHasWall()*, возвращающие значение параметров клетки. Дополнительно в классе определены виртуальные методы *std::string getTextureAlias()* и *std::unique_ptr<Cell> createUniquePtr()*: первый отвечает за алиас используемой при отрисовке текстуры, второй — за создание умного указателя определенного типа, зависящего от того, какого типа *CellType* данная клетка.

2. Создание классов *StartCell* и *EndCell*, которые наследуются от класса *Cell*. В данных классах конструкторы *StartCell(size_t x, size_t y)* и *EndCell(size_t x, size_t y)* вызывают родительский конструктор с параметрами *x, y, false*, т. к. данные клетки не могут содержать стену. Также в классах переопределены методы *std::string getTextureAlias()* и *std::string getTextureAlias()*.

3. Создание класса *Field*, отвечающего за работу с игровым полем. Класс содержит в себе такие поля, как *size_t width* и *size_t height*, отвечающие за размеры игрового поля, а также двумерный массив умных указателей на клетки данного поля. Конструктор *Field(size_t width, size_t height)* создает неинициализированный двумерный массив клеток согласно заданным размерам, в последствии данный массив необходимо будет заполнить. Также были реализованы конструкторы копирования и перемещения для данного класса, были реализованы соответствующие операторы присваивания. При копировании объекта новый объект создается по размерам переданного, однако копирования клеток не происходит, в следствие чего нужно заново инициализировать массив клеток. При перемещении используется функция

std::move(). Были реализованы геттеры *size_t getWidth()*, *size_t getHeight()*, возвращающие соответствующие размеры поля, а также геттер *Cell& getCell(size_t x, size_t y)*, возвращающий константную ссылку на клетку.

4. Для генерации поля был разработан класс *FieldGenerator*. Полями данного класса являются *size_t width* и *size_t height*, отвечающие за размеры игрового поля, *size_t start_x*, *size_t start_y*, *size_t end_x*, *size_t end_y*, которые отвечают за позицию клеток старта и финиша, *double wall_chance*, отвечающая за процентный шанс генерации стены на клетке, а также вектор сгенерированных клеток *std::vector<std::shared_ptr<Cell>> cells*. Конструктор *FieldGenerator(size_t width, size_t height, double wall_chance)* задает параметры генерации, шанс на генерацию стены по умолчанию равен 10%. Также был реализован метод *std::vector<cell_ptr> generateCells()*, отвечающий непосредственно за генерацию клеток, в данной реализации генерация происходит полностью случайно. Реализован приватный метод *void generateStartEnd()*, отвечающий за генерацию координат клеток старта и финиша, в методе также проверяется, чтобы данные клетки не находились рядом, в связи с чем на игровое поле накладываются определенные ограничения: размеры должны быть больше 2x2.

5. Создание интерфейса элемента клетки, который имеет название *Item*. Данный интерфейс описывает такие методы, как *void attach()* для прикрепления объекта к определенной клетке, *void onAdd()*, *void onPickUp()*, *void destroy()*, описывающие поведение данного объекта.

6. Создание классов, отвечающих за отрисовку игры. Графика в игре сделана с помощью библиотеки SFML. Первым классом стал класс *Renderer*, который занимается непосредственно отрисовкой игры на экране. Конструктор *Renderer(size_t w_width, size_t w_height, std::string title)* создает новое окно, которое сохраняется в поле *sf::RenderWindow window*, а также сохраняет параметры окна в поля *size_t w_width*, *size_t w_height*, *std::string title*. Также данный класс имеет еще 2 поля: *std::vector<sf::Sprite> static_render_objects* и *std::vector<sf::Sprite> render_objects* — данные поля хранят в себе спрайты — легковесные объекты, служащие для хранения изображений; отличие данных полей лишь в том, что параметры спрайтов из *render_objects* вероятно будут

изменяться во время игры. Для добавления нового спрайта создан метод *void addObject(size_t x, size_t y, size_t object_w, size_t object_h, const std::string& texture_alias, bool is_static)*, который отвечает за добавление спрайта по заданным координатам, с заданными параметрами ширины и высоты, с заданной текстурой, которая определяется посредством алиаса, также в зависимости от значения *is_static* спрайт будет добавлен в список изменяемых либо не изменяемых в процессе игры спрайтов. Также для очистки полей *render_objects* и *static_render_objects* создан метод *void flushObjects(bool flush_static)*: в зависимости от значения *flush_static* будет очищен вектор *render_objects* либо одновременно оба вектора со спрайтами. Также созданы два метода работы с окном: геттер для получения экземпляра окна по ссылке и метод проверки окна на то, является ли оно открытым. Также создан метод отрисовки сцены *bool renderFrame()*, в котором в цикле выводятся спрайты, сохраненные в полях *render_objects* и *static_render_objects*. В деструкторе класса происходит закрытие окна.

7. Создание класса *TextureManager*, который отвечает за хранение текстур объектов. Данный класс реализован с помощью паттерна проектирования «*Singleton*», что означает, что данный объект может быть создан в единственном экземпляре на всю программу. Данный класс имеет поле *std::map<std::string, sf::Texture> textures*, в котором хранятся текстуры, доступ к которым происходит посредством алиасов. В классе реализован метод *bool addTexture(const std::string& alias, const std::string& path)*, который пытается загрузить текстуру из файла, указанного в переменной *path*: в случае успеха текстура помещается в словарь *textures* по ключу, соответствующему алиасу, и возвращается значение *true*, в случае провала — возвращается значение *false*. Также реализован метод *const sf::Texture& get(const std::string& alias, const std::string& path = "")*, который возвращает ссылку на нужную текстуру по алиасу, в случае, если такая текстура не найдена в словаре, вызывается метод *addTexture()* для попытки ее загрузки из файла.

8. Создание класса *Game*, отвечающего за связь между всеми компонентами игры. Данный класс содержит в себе поля *std::unique_ptr<Renderer> renderer*, *std::unique_ptr<Field> field*, *FieldGenerator generator*, отвечающие за

компоненты отрисовки игры, работы с полем, а также его генерации. В конструкторе класса *Game()* выполняется создание экземпляров отрисовщика *renderer*, поля *field* и генератора поля *generator* и вызов метода инициализации клеток поля *fillField()*. Метод *bool fillField()* вызывает генератор клеток, после чего, посредством полного прохода, добавляет клетки в двумерный массив клеток, находящихся на поле, также каждая клетка добавляется в качестве спрайта в отрисовщик. В классе также имеется геттер *Renderer& getRenderer()*, который возвращает ссылку на объект отрисовщика для упрощения последующего взаимодействия с ним. Также в классе реализован метод *bool run()*, который запускает игровой цикл, в котором, в данный момент, происходит лишь отрисовка кадров игры.

9. Создание функции *int main()*, в которой создается экземпляр объекта *Game* и вызывается метод *run()* у данного объекта.

Тестирование

Результаты тестирования программы представлены на рис. 1 и рис. 2

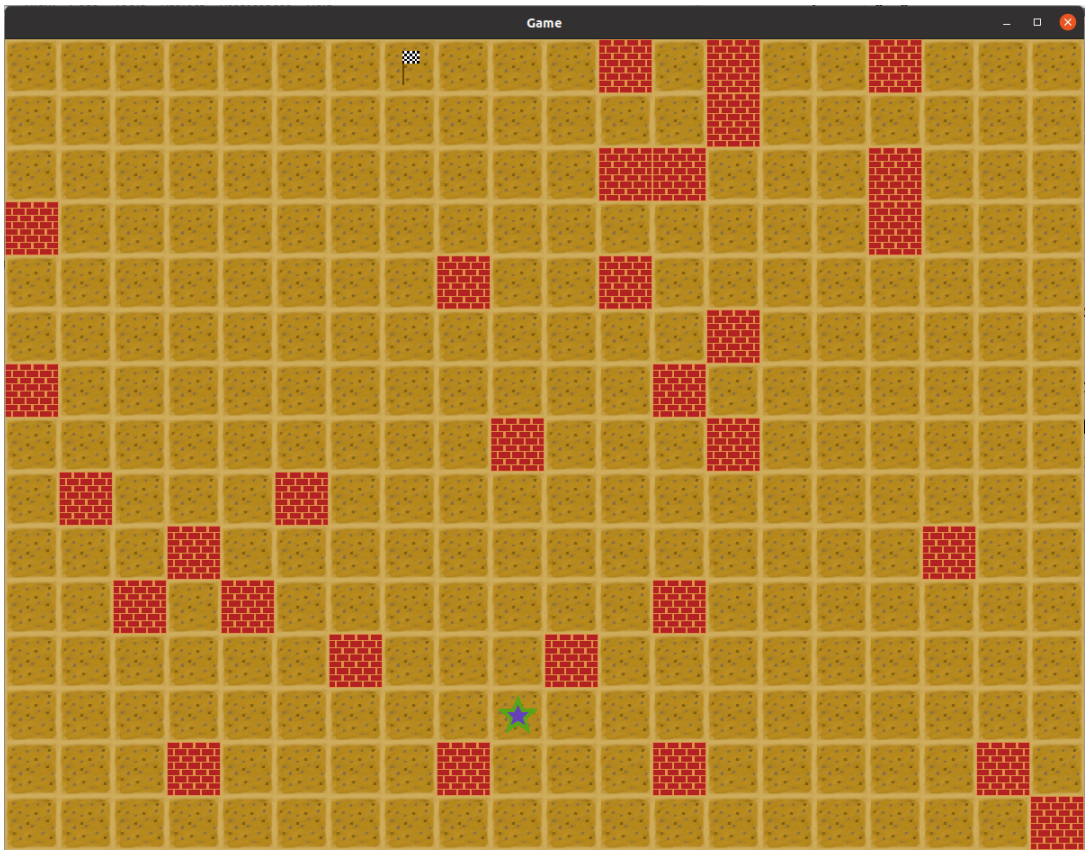

Рисунок 1

Рисунок 2

Выводы.

Был изучен процесс создания классов в языке C++, были изучены и применены конструкторы копирования и перемещения, было затронуто наследование классов, а также разработка с использованием паттернов проектирования.

Была разработана программа, создающее игровое поле со случайно сгенерированными клетками на данном поле. Клетки подразделяются на типы: клетка старта, клетка финиша и обычная клетка. Обычная клетка может содержать стену либо набор объектов. Разработан графический интерфейс для вывода результата на экран.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: *main.cpp*

```
#include "game.h"

int main()
{
	Game game;
	game.run();
	return 0;
}
```

Название файла: *game.h*

```
#ifndef GAME_H
#define GAME_H

#include <iostream>
#include <memory>
#include <vector>
#include <ctime>
#include <chrono>
#include <thread>

#include "utils/field_generator.h"
#include "field.h"

#include "graphics/renderer.h"

class Game
{
public:
	Game();
	~Game() = default;

	bool run();
	bool fillField();

	Renderer& getRenderer() const;
private:
	std::unique_ptr<Renderer> renderer;
	std::unique_ptr<Field> field;
	FieldGenerator generator;
};

#endif
```

Название файла: *game.cpp*

```
#include "game.h"

const int width = 20;
const int height = 15;
const int cell_size = 60;
```

```
Game::Game()
{
	this->field = std::make_unique<Field>(Field(width, height));
	this->renderer = std::make_unique<Renderer>(
		Renderer(cell_size * width, cell_size * height, "Game")
	);
	this->generator = FieldGenerator(width, height);
	this->fillField();
}

bool Game::run()
{
	using namespace std::this_thread;
    using namespace std::chrono_literals;
    using std::chrono::system_clock;

    bool running = true;

	std::cout << "Running" << std::endl;
	while (running)
	{
		std::chrono::system_clock::time_point renderStarts =
std::chrono::system_clock::now();
		running = this->getRenderer().renderFrame();
		sleep_until(renderStarts + .2s);
	}
	return true;
}

bool Game::fillField()
{
	std::vector<std::shared_ptr<Cell>> generated_cells = this-
>generator.generateCells();
	for (size_t y = 0; y < this->field->getHeight(); y++)
	{
		for (size_t x = 0; x < this->field->getWidth(); x++)
		{
			this->field->setCell(
				x,
				y,
				generated_cells.at(y * this->field->getWidth() +
x).get()
			);
			this->getRenderer().addObject(
				x, y,
				cell_size, cell_size,
				generated_cells.at(
					y * this->field->getWidth() + x
				).get()->getTextureAlias(),
				true
			);
		}
	}
	return true;
}

Renderer& Game::getRenderer() const
```

```
{
	return *this->renderer.get();
}
```

Название файла: *field.h*

```
#ifndef FIELD_H
#define FIELD_H

#include <memory>
#include "cells/cell.h"
#include "cells/start_cell.h"
#include "cells/end_cell.h"
#include <stdexcept>

class Field
{
public:
	Field(size_t width, size_t height);
	~Field() = default;

	Field(const Field& other);
	Field(Field&& other);

	void setCell(size_t x, size_t y, Cell *cell);

	Field& operator =(const Field& other);
	Field& operator =(Field&& other);

	size_t getWidth() const;
	size_t getHeight() const;
	const Cell&  getCell(size_t x, size_t y) const;
private:
	using cell_ptr = std::unique_ptr<Cell>;
	using cell_row = std::unique_ptr<cell_ptr[]>;

	size_t width;
	size_t height;
	std::unique_ptr<cell_row[]> cells;
};

#endif
```

Название файла: *field.cpp*

```
#include "field.h"

Field::Field(size_t width, size_t height)
{
	if (width <= 0 || height <= 0)
		throw std::runtime_error("Длина и ширина должны быть
положительными числами!");
	this->width = width;
	this->height = height;

	cells = std::make_unique<cell_row[]>(this->height);
	for (size_t i = 0; i < this->height; i++)
	{
```

```
            cells[i] = std::make_unique<cell_ptr[]>(this->width);
      }
}

Field::Field(const Field& other) : Field(other.width, other.height)
{}

Field::Field(Field&& other) : width{other.width}, height{other.height}
{
      cells = std::move(other.cells);
}

void Field::setCell(size_t x, size_t y, Cell *cell)
{
      if (x < 0 || y < 0 || x >= this->width || y >= this->height)
            throw std::runtime_error("Координаты не должны выходить за
границы поля!");
      cells[y][x] = cell->createUniquePtr();
}

const Cell& Field::getCell(size_t x, size_t y) const
{
      if (x < 0 || y < 0 || x >= this->width || y >= this->height)
            throw std::runtime_error("Координаты не должны выходить за
границы поля!");

      return *this->cells[y][x].get();
}

Field& Field::operator =(const Field& other)
{
      if (&other == this)
            return *this;

      this->width = other.width;
      this->height = other.height;

      cells = std::make_unique<cell_row[]>(this->height);
      for (size_t i = 0; i < this->height; i++)
      {
            cells[i] = std::make_unique<cell_ptr[]>(this->height);
      }

      return *this;
}

Field& Field::operator =(Field&& other)
{
      if (&other == this)
            return *this;

      std::swap(this->width, other.width);
      std::swap(this->height, other.height);
      std::swap(this->cells, other.cells);

      return *this;
}
```

```
size_t Field::getHeight() const
{
	return this->height;
}

size_t Field::getWidth() const
{
	return this->width;
}
```

Название файла: *cells/cell.h*

```
#ifndef CELL_H
#define CELL_H

#include <memory>
#include <vector>
#include <string>

#include "../interfaces/item.h"

enum class CellType {
	RegularCell,
	StartCell,
	EndCell
};

class Cell
{
public:
	Cell(size_t x, size_t y, bool wall = false);
	virtual ~Cell() = default;

	virtual CellType getType();

	Cell(const Cell& other);
	Cell(Cell&& other);
	Cell(Cell* other);

	Cell& operator =(const Cell& other);
	Cell& operator =(Cell&& other);

	size_t getX() const;
	size_t getY() const;
	bool   getHasWall() const;

	virtual std::string getTextureAlias() const;

	virtual std::unique_ptr<Cell> createUniquePtr();
protected:
	using item_ptr = std::shared_ptr<Item>;

	size_t x, y;
	bool wall;
	std::vector<item_ptr> items;
};
```

```
#endif
```

Название файла: *cells/cell.cpp*

```
#include "cell.h"

Cell::Cell(size_t x, size_t y, bool wall) : x{x}, y{y}, wall{wall}
{}

CellType Cell::getType()
{
	return CellType::RegularCell;
}

Cell::Cell(const Cell& other): x{other.x}, y{other.y}, wall{other.wall}
{
	for (size_t i = 0; i < other.items.size(); i++)
		this->items.push_back(other.items.at(i));
}

Cell::Cell(Cell&& other): x{other.x}, y{other.y}, wall{other.wall}
{
	std::swap(this->items, other.items);
}

Cell::Cell(Cell* other): x{other->x}, y{other->y}, wall{other->wall}
{
	for (size_t i = 0; i < other->items.size(); i++)
		this->items.push_back(other->items.at(i));
}

Cell& Cell::operator =(const Cell& other)
{
	if (&other == this)
		return *this;

	this->x    = other.x;
	this->y    = other.y;
	this->wall = other.wall;
	for (size_t i = 0; i < other.items.size(); i++)
		this->items.push_back(other.items.at(i));

	return *this;
}

Cell& Cell::operator =(Cell&& other)
{
	if (&other == this)
		return *this;

	std::swap(this->x, other.x);
	std::swap(this->y, other.y);
	std::swap(this->wall, other.wall);
	std::swap(this->items, other.items);

	return *this;
}
```

```
size_t Cell::getX() const
{
	return this->x;
}
size_t Cell::getY() const
{
	return this->y;
}
bool   Cell::getHasWall() const
{
	return this->wall;
}

std::unique_ptr<Cell> Cell::createUniquePtr()
{
	return std::make_unique<Cell>(this);
}

std::string Cell::getTextureAlias() const
{
	if (this->wall)
		return "cell_wall";
	return "cell_regular";
}
```

Название файла: *cells/start_cell.h*

```
#ifndef STARTCELL_H
#define STARTCELL_H

#include "cell.h"

class StartCell: public Cell
{
public:
	StartCell(size_t x, size_t y);
	~StartCell() = default;

	CellType getType();

	std::string getTextureAlias() const;

	std::unique_ptr<Cell> createUniquePtr();
};

#endif
```

Название файла: *cells/start_cell.cpp*

```
#include "start_cell.h"

StartCell::StartCell(size_t x, size_t y) : Cell{x, y, false}
{}

CellType StartCell::getType()
{
	return CellType::StartCell;
}
```

```
std::unique_ptr<Cell> StartCell::createUniquePtr()
{
	return std::make_unique<StartCell>(*this);
}

std::string StartCell::getTextureAlias() const
{
	return "cell_start";
}
```

Название файла: *cells/end_cell.h*

```
#ifndef END_H
#define END_H

#include "cell.h"

class EndCell: public Cell
{
public:
	EndCell(size_t x, size_t y);
	~EndCell() = default;

	CellType getType();

	std::string getTextureAlias() const;

	std::unique_ptr<Cell> createUniquePtr();
};

#endif
```

Название файла: *cells/end_cell.cpp*

```
#include "end_cell.h"

EndCell::EndCell(size_t x, size_t y) : Cell{x, y, false}
{}

CellType EndCell::getType()
{
	return CellType::EndCell;
}

std::unique_ptr<Cell> EndCell::createUniquePtr()
{
	return std::make_unique<EndCell>(*this);
}

std::string EndCell::getTextureAlias() const
{
	return "cell_end";
}
```

Название файла: *interfaces/item.h*

```
#ifndef ITEM_H
```

```
#define ITEM_H

class Cell;

class Item
{
public:
	virtual void attach(Cell& holder) = 0;
	virtual void onAdd() = 0;
	virtual void onPickUp() = 0;
	virtual void destroy() = 0;
};

#endif
```

Название файла: *utils/field_generator.h*

```
#ifndef FIELD_GENERATOR_H
#define FIELD_GENERATOR_H

#include "../cells/cell.h"
#include "../cells/end_cell.h"
#include "../cells/start_cell.h"

#include <vector>
#include <cstdlib>
#include <ctime>

using cell_ptr = std::shared_ptr<Cell>;

class FieldGenerator
{
public:
	FieldGenerator(size_t width = 0, size_t height = 0, double
wall_chance = 0.1);
	~FieldGenerator() = default;

	std::vector<cell_ptr> generateCells();
private:
	std::vector<cell_ptr> cells;

	size_t width;
	size_t height;

	size_t start_x, start_y;
	size_t end_x, end_y;

	double wall_chance;

	void generateStartEnd();
};

#endif
```

Название файла: *utils/field_generator.cpp*

```
#include "field_generator.h"
```

```
FieldGenerator::FieldGenerator(size_t width, size_t height, double
wall_chance) :
      width{width}, height{height}, wall_chance{wall_chance}
{}

std::vector<cell_ptr> FieldGenerator::generateCells()
{
      std::srand(std::time(NULL));
      bool has_wall;
      int wall_offset = (int) (1 / this->wall_chance);
      this->generateStartEnd();
      std::shared_ptr<Cell> _cell_ptr;

      for (size_t y = 0; y < this->height; y++)
      {
            for (size_t x = 0; x < this->width; x++)
            {
                  has_wall = false;
                  if (x == this->start_x && y == this->start_y)
                  {
                        _cell_ptr = std::make_shared<StartCell>(x, y);
                  } else if (x == this->end_x && y == this->end_y)
                  {
                        _cell_ptr = std::make_shared<EndCell>(x, y);
                  } else
                  {
                        has_wall = (std::rand() % wall_offset) == 0;
                        _cell_ptr = std::make_shared<Cell>(x, y, has_wall);
                  }
                  this->cells.push_back(_cell_ptr);
            }
      }

      return this->cells;
}

void FieldGenerator::generateStartEnd()
{
      std::srand(std::time(NULL));
      bool valid_start_end = false;

      if (this->width <=2 && this->height <= 2)
            return; // mb throw error

      while (!valid_start_end)
      {
            this->start_x = std::rand() % this->width;
            this->end_x = std::rand() % this->width;
            this->start_y = std::rand() % this->height;
            this->end_y = std::rand() % this->height;

            if (abs(this->start_x - this->end_x) > 1 && abs(this->start_y
- this->end_y) > 1)
                  valid_start_end = true;
      }
}
```

Название файла: *graphics/renderer.h*

```
#ifndef RENDERER_H
#define RENDERER_H

#include <SFML/Graphics.hpp>
#include <string>
#include <memory>

#include "texture_manager.h"

class Renderer
{
public:
	Renderer(size_t w_width = 0, size_t w_height = 0, std::string title
= "");
	~Renderer();

	Renderer(const Renderer& other);
	Renderer(Renderer&& other);

	Renderer& operator =(const Renderer& other);
	Renderer& operator =(Renderer&& other);

	sf::RenderWindow& getWindow();

	bool renderFrame();
	bool isWindowOpen();

	void addObject(
		size_t x,
		size_t y,
		size_t object_w,
		size_t object_h,
		const std::string& texture_alias,
		bool is_static = false
	);

	void flushObjects(bool flush_static = false);

	// void modifyObject()
private:
	sf::RenderWindow window;
	size_t w_width, w_height;
	std::string title;
	std::vector<sf::Sprite> static_render_objects;
	std::vector<sf::Sprite> render_objects;
};

#endif
```

Название файла: *graphics/renderer.cpp*

```
#include "renderer.h"

Renderer::Renderer(size_t w_width, size_t w_height, std::string title):
	w_width{w_width}, w_height{w_height}, title{title}
{
```

```
	this->window.create(
		sf::VideoMode(w_width, w_height),
		title
	);

	TextureManager::instance().addTexture("cell_regular",
"assets/textures/cell_regular.png");
	TextureManager::instance().addTexture("cell_start",
"assets/textures/cell_start.png");
	TextureManager::instance().addTexture("cell_end",
"assets/textures/cell_end.png");
	TextureManager::instance().addTexture("cell_wall",
"assets/textures/cell_wall.png");

	static_render_objects.resize(w_width * w_height);
}

Renderer::~Renderer()
{
	this->getWindow().close();
}

bool Renderer::renderFrame()
{
	if (!this->isWindowOpen())
		return false;

	this->getWindow().clear(sf::Color(255, 255, 255, 0));

	for (auto& object: this->static_render_objects)
	{
		this->getWindow().draw(object);
	}
	for (auto& object: this->render_objects)
	{
		this->getWindow().draw(object);
	}

	sf::Event event;
	while (this->getWindow().pollEvent(event))
	{
		if (event.type == sf::Event::Closed)
		{
			this->getWindow().close();
			return false;
		}
	}

	this->getWindow().display();

	return true;
}

bool Renderer::isWindowOpen()
{
	return this->getWindow().isOpen();
}
```

```
sf::RenderWindow& Renderer::getWindow()
{
	return this->window;
}

Renderer::Renderer(const Renderer& other):
	w_width{other.w_width}, w_height{other.w_height},
title{other.title}
{
	this->window.create(
		sf::VideoMode(w_width, w_height),
		title
	);
}

Renderer::Renderer(Renderer&& other):
	w_width{other.w_width}, w_height{other.w_height},
title{other.title}
{
	this->window.create(
		sf::VideoMode(other.w_width, other.w_height),
		other.title
	);
}

Renderer& Renderer::operator =(const Renderer& other)
{
	if (&other == this)
		return *this;

	this->w_width = other.w_width;
	this->w_height = other.w_height;
	this->title = other.title;
	this->window.create(
		sf::VideoMode(w_width, w_height),
		title
	);

	return *this;
}

Renderer& Renderer::operator =(Renderer&& other)
{
	if (&other == this)
		return *this;

	std::swap(this->w_height, other.w_height);
	std::swap(this->w_width, other.w_width);
	std::swap(this->title, other.title);
	this->window.create(
		sf::VideoMode(w_width, w_height),
		title
	);

	return *this;
}

void Renderer::addObject(
```

```
            size_t x,
            size_t y,
            size_t object_w,
            size_t object_h,
            const std::string& texture_name,
            bool is_static
      )
{
      sf::Texture texture = TextureManager::instance().get(texture_name);
      sf::Sprite object;
      object.setPosition(
            x * object_w,
            y * object_h
      );
      object.setTexture(TextureManager::instance().get(texture_name));
      object.setScale(
            object_w / static_cast<float>(texture.getSize().x),
            object_h / static_cast<float>(texture.getSize().y)
      );

      if (is_static)
            static_render_objects.push_back(object);
      else
            render_objects.push_back(object);
}

void Renderer::flushObjects(bool flush_static)
{
      render_objects.clear();
      if (flush_static)
            static_render_objects.clear();
}
```

Название файла: *graphics/texture_manager.h*

```
#ifndef TEXTURE_MANAGER_H
#define TEXTURE_MANAGER_H

#include <SFML/Graphics.hpp>
#include <string>
#include <map>
#include <stdexcept>

class TextureManager
{
public:
      static TextureManager& instance();

      TextureManager(const TextureManager&) = delete;
      TextureManager& operator =(const TextureManager&) = delete;

      const sf::Texture& get(const std::string& alias, const std::string&
path = "");
      bool addTexture(const std::string& alias, const std::string& path);
private:
      TextureManager() = default;
      std::map<std::string, sf::Texture> textures;
};
```

```
#endif
```

Название файла: *graphics/texture_manager.cpp*

```
#include "texture_manager.h"

TextureManager& TextureManager::instance()
{
	static TextureManager manager;
	return manager;
}

const sf::Texture& TextureManager::get(const std::string& alias, const
std::string& path)
{
	if (textures.find(alias) == textures.end())
		addTexture(alias, path);
	return textures.at(alias);
}

bool TextureManager::addTexture(const std::string& alias, const
std::string& path)
{
	if (textures[alias].loadFromFile(path))
		return false;
	return true;
}
```

Название файла: *Makefile*

```
BUILD_DIR = ./build
INTERFACE_PATH = ./interfaces
CELLS_PATH = ./cells
BUILD_PATH = ./build
UTILS_PATH = ./utils
GRAPHICS_PATH = ./graphics
FILES = main.cpp field.cpp $(CELLS_PATH)/cell.cpp
TARGETS = $(BUILD_DIR)/main.o $(BUILD_DIR)/field.o $(BUILD_DIR)/cell.o \
	$(BUILD_DIR)/end_cell.o $(BUILD_DIR)/start_cell.o
$(BUILD_DIR)/game.o \
	$(BUILD_DIR)/field_generator.o $(BUILD_DIR)/renderer.o \
	$(BUILD_DIR)/texture_manager.o
HEADERS = field.h $(CELLS_PATH)/cell.h $(CELLS_PATH)/end_cell.h \
	$(CELLS_PATH)/start_cell.h $(INTERFACE_PATH)/item.h game.h \
	$(UTILS_PATH)/field_generator.h $(GRAPHICS_PATH)/renderer.h \
	$(GRAPHICS_PATH)/texture_manager.h
COMP = g++ -std=c++2a
SFML_LINK = -lsfml-graphics -lsfml-window -lsfml-system

all: compile

compile: $(TARGETS)
	$(COMP) $(TARGETS) -o main $(SFML_LINK)

$(BUILD_DIR)/main.o: main.cpp $(HEADERS)
	$(COMP) main.cpp -c -o $(BUILD_DIR)/main.o
```

```
$(BUILD_DIR)/field.o: field.cpp $(HEADERS)
	$(COMP) field.cpp -c -o $(BUILD_DIR)/field.o

$(BUILD_DIR)/cell.o: $(CELLS_PATH)/cell.cpp $(CELLS_PATH)/cell.h $
(INTERFACE_PATH)/item.h
	$(COMP) $(CELLS_PATH)/cell.cpp -c -o $(BUILD_DIR)/cell.o

$(BUILD_DIR)/end_cell.o: $(CELLS_PATH)/end_cell.cpp $(CELLS_PATH)/cell.h
$(INTERFACE_PATH)/item.h $(CELLS_PATH)/end_cell.h
	$(COMP) $(CELLS_PATH)/end_cell.cpp -c -o $(BUILD_DIR)/end_cell.o

$(BUILD_DIR)/start_cell.o: $(CELLS_PATH)/start_cell.cpp
$(CELLS_PATH)/cell.h $(INTERFACE_PATH)/item.h $(CELLS_PATH)/start_cell.h
	$(COMP) $(CELLS_PATH)/start_cell.cpp -c -o
$(BUILD_DIR)/start_cell.o

$(BUILD_DIR)/game.o: game.cpp $(HEADERS)
	$(COMP) game.cpp -c -o $(BUILD_DIR)/game.o

$(BUILD_DIR)/field_generator.o: $(UTILS_PATH)/field_generator.cpp $
(HEADERS)
	$(COMP) $(UTILS_PATH)/field_generator.cpp -c -o
$(BUILD_DIR)/field_generator.o

$(BUILD_DIR)/renderer.o: $(GRAPHICS_PATH)/renderer.cpp
$(GRAPHICS_PATH)/renderer.h $(GRAPHICS_PATH)/texture_manager.h
	$(COMP) $(GRAPHICS_PATH)/renderer.cpp -c -o $(BUILD_DIR)/renderer.o

$(BUILD_DIR)/texture_manager.o: $(GRAPHICS_PATH)/texture_manager.cpp $
(GRAPHICS_PATH)/texture_manager.h
	$(COMP) $(GRAPHICS_PATH)/texture_manager.cpp -c -o
$(BUILD_DIR)/texture_manager.o
```

ПРИЛОЖЕНИЕ Б

ДИАГРАММА КЛАССОВ