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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Объектно-ориентированное программирование»

Тема: Полиморфизм

Студент гр. 2300 _____ Жохов К.С.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2023

Цель работы.

Изучить понятие такого принципа объектно-ориентированного программирования, как полиморфизм. Также познакомиться с виртуальными методами и абстрактными классами. Продумать и реализовать интерфейс игрового события, разработать класс, генерирующий игровое поле.

Задание.

а) Создать интерфейс игрового события. Интерфейс должен обеспечивать срабатывание события, когда игрок наступает на клетку.

б) Реализовать интерфейс игрового события тремя конкретными событиями. Одно событие должно положительно влиять на характеристики игрока, второе должно негативно влиять на характеристики игрока, третье изменять координаты игрока на поле. При желании можно реализовать больше событий и/или события меняющие само поле (например, делать из непроходимой клетки проходимую).

в) В классе управления игроком добавить проверку на наличие события на клетке, если событие присутствует, то оно должно сработать. Срабатывание должно происходить через интерфейс события, и не должно быть никаких проверок на тип события (реализация через динамический полиморфизм)

г) Создать класс создающий поле. Предусмотреть возможность создания 2 разных уровней. По желанию можно сделать случайную генерацию уровней. Должно гарантироваться, что игрок может дойти от входа до выхода.

Выполнение работы.

1. Класс `IEvent`. Представляет собой интерфейс игрового события. Содержит 3 чисто виртуальных `public` метода, которые будут перегружены в классах-наследниках.

2. Класс `DamageEvent`. Унаследован от класса `IEvent`. Представляет собой реализацию интерфейса игрового события, наносящего урон игроку. Содержит одно `private` поле, хранящее величину урона, которое получит игрок в случае срабатывания данного игрового события.

- Конструктор `DamageEvent(int damage = DEFAULT_DAMAGE)`. Модификатор доступа – `public`. Принимает на вход необязательный параметр: величину урона.
- Метод `void activationEvent(Controller& controller)`. Модификатор доступа – `public`. Принимает на вход ссылку на объект класса, управляющего игроком. Активирует игровое событие – игрок получает урон. После срабатывания, игровое событие удаляется из клетки.
- Метод `DamageEvent* clone()`. Модификатор доступа – `public`. Не принимает аргументов, возвращает указатель на динамическую память, выделенную под текущий объект класса. Требуется для осуществления копирования класса клетки.

3. Класс `HealEvent`. Унаследован от класса `IEvent`. Представляет собой реализацию интерфейса игрового события, увеличивающего здоровье игрока. Содержит одно `private` поле, хранящее количество единиц здоровья, которое получит игрок в случае срабатывания данного игрового события.

- Конструктор `HealEvent(int heal = DEFAULT_HEAL)`. Модификатор доступа – `public`. Принимает на вход необязательный параметр: количество единиц здоровья.
- Метод `void activationEvent(Controller& controller)`. Модификатор доступа – `public`. Принимает на вход ссылку на объект класса, управляющего игроком. Активирует игровое событие – игрок получает дополнительное здоровье. После срабатывания, игровое событие удаляется из клетки.
- Метод `HealEvent* clone()`. Модификатор доступа – `public`. Не принимает аргументов, возвращает указатель на динамическую память, выделенную под

текущий объект класса. Требуется для осуществления копирования класса клетки.

4. Класс `ScoreEvent`. Унаследован от класса `IEvent`. Представляет собой реализацию интерфейса игрового события, которое добавляет некоторое количество очков игроку. Содержит одно `private` поле, хранящее количество очков, которое получит игрок в случае срабатывания данного игрового события.

- Конструктор `ScoreEvent(int bonus = DEFAULT_BONUS)`. Модификатор доступа – `public`. Принимает на вход необязательный параметр: количество очков.

- Метод `void activationEvent(Controller& controller)`. Модификатор доступа – `public`. Принимает на вход ссылку на объект класса, управляющего игроком. Активирует игровое событие – игрок получает дополнительные очки. После срабатывания, игровое событие удаляется из клетки.

- Метод `ScoreEvent* clone()`. Модификатор доступа – `public`. Не принимает аргументов, возвращает указатель на динамическую память, выделенную под текущий объект класса. Требуется для осуществления копирования класса клетки.

5. Класс `TeleportEvent`. Унаследован от класса `IEvent`. Представляет собой реализацию интерфейса игрового события, которое перемещает игрока в какую-то из точек поля. Содержит два `private` поля, хранящих координаты точки игрового поля, в которую произойдёт перемещение игрока, и флаг, сигнализирующий о том, что игровое событие уже было обыграно ранее.

- Конструктор `TeleportEvent(int x, int y)`. Модификатор доступа – `public`. Принимает на вход два параметра: координаты точки назначения.

- Метод `void activationEvent(Controller& controller)`. Модификатор доступа – `public`. Принимает на вход ссылку на объект класса, управляющего игроком. Активирует игровое событие – перемещает игрока

в одну из доступных точек игрового поля. После срабатывания, игровое событие не может быть обыграно ещё раз.

- Метод `TeleportEvent* clone()`. Модификатор доступа – `public`. Не принимает аргументов, возвращает указатель на динамическую память, выделенную под текущий объект класса. Требуется для осуществления копирования класса клетки.

6. Класс `FieldCreator`. Предназначен для генерирования игрового поля.

- Метод `Field createLevel_1()`. Модификатор доступа – `public`. Не принимает аргументов, создаёт игровое поле первого уровня (наглядную модель игрового поля первого уровня можно увидеть на рисунке 1). Возвращает объект класса `Field` (сгенерированное игровое поле).

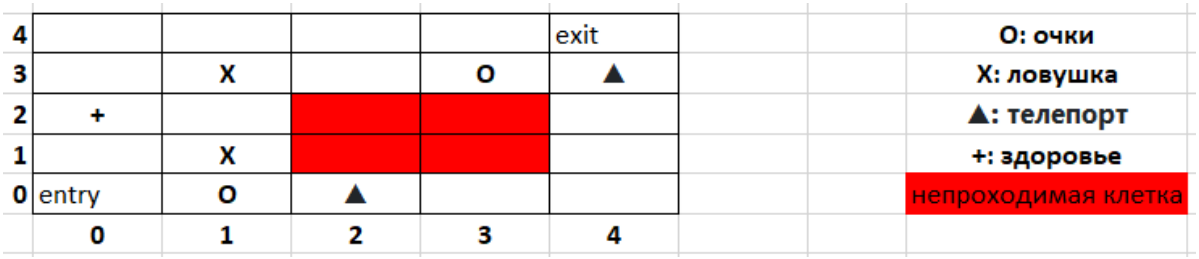


Рисунок 1 – Игровое поле первого уровня

- Метод `Field createLevel_2()`. Модификатор доступа – `public`. Не принимает аргументов, создаёт игровое поле второго уровня (наглядную модель игрового поля первого уровня можно увидеть на рисунке 2). Возвращает объект класса `Field` (сгенерированное игровое поле).

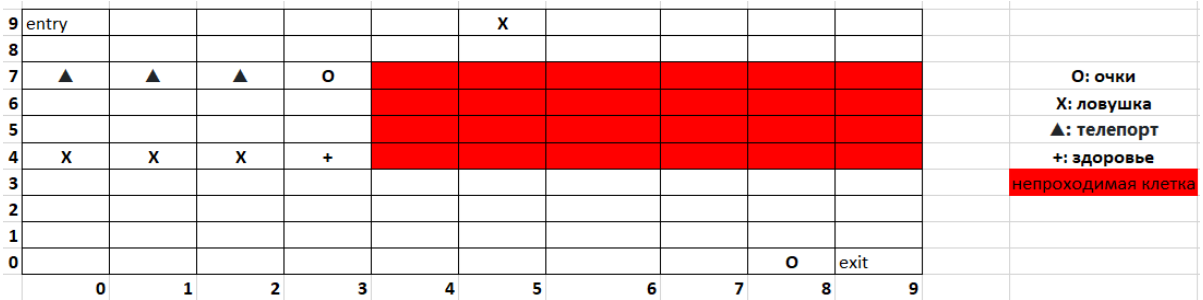


Рисунок 2 – Игровое поле второго уровня

Разработанные UML-диаграммы классов см. в приложении А.

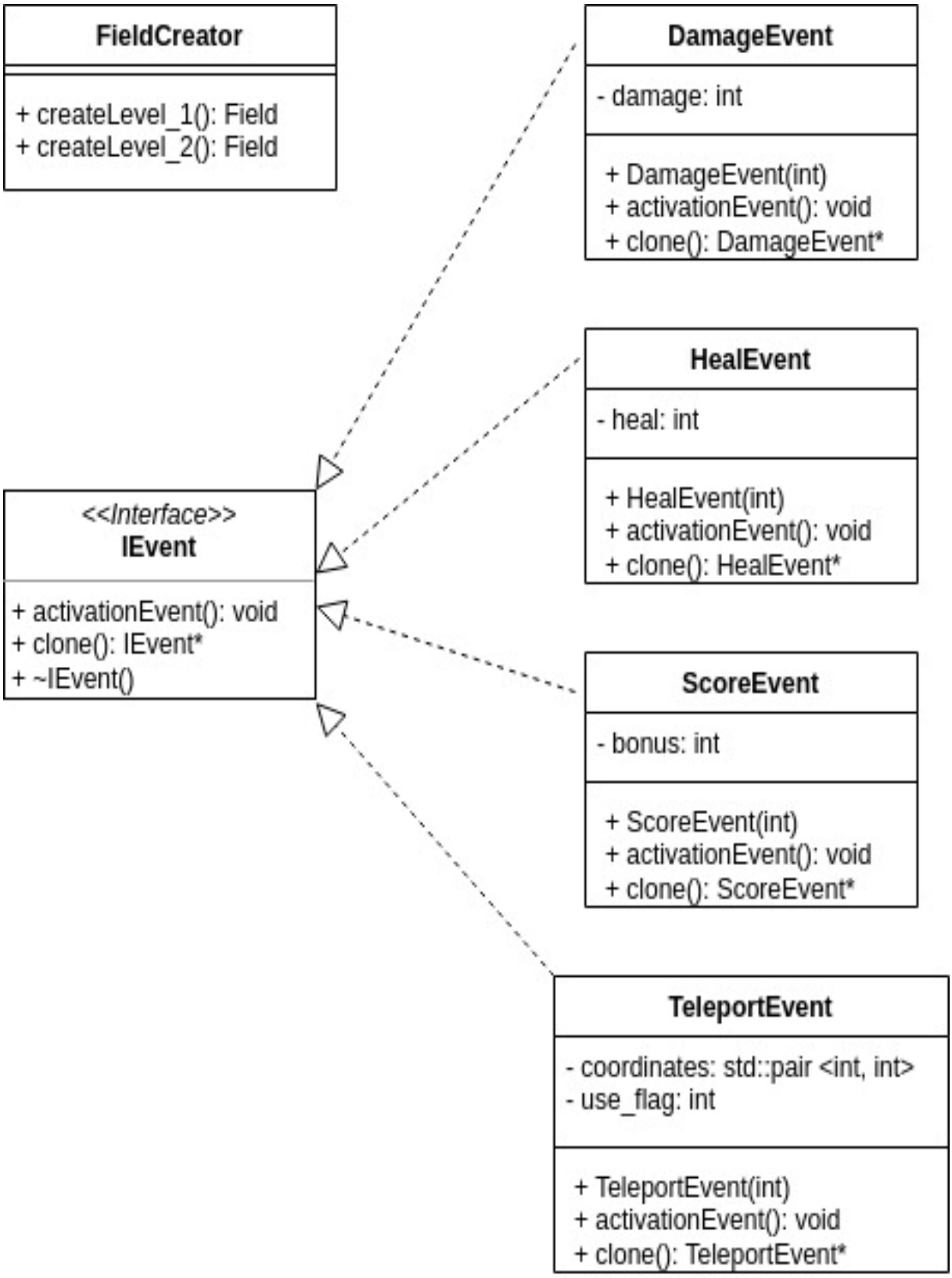
Результаты тестирования см. в приложении Б.

Выводы.

В ходе лабораторной работы были освоены навыки работы с виртуальными методами и абстрактными классам. Также был реализован интерфейс различных игровых событий и разработан класс, генерирующий игровое поле.

ПРИЛОЖЕНИЕ А

UML-ДИАГРАММЫ КЛАССОВ



ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные	Комментарии

1.	//check DamageEvent
	Player player;
	Field field(5, 5);
	field.getCell(1, 0) = Cell(true, new DamageEvent(50));
	Controller controller(player, field);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Health: " << player.getHealth() << "\n\n";
	controller.move(Direction::right);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Health: " << player.getHealth() << '\n';
	controller.move(Direction::right);
	std::cout << controller.getCoordinates().first << '\t' <<
		0 0
		Health: 100
		
		1 0
		Health: 50
		2 0
		1 0
		Health: 50
			Проверка работоспособности игрового события, наносящего урон игроку

	controller.getCoordinates().second << '\n'; controller.move(Direction::left); std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n'; std::cout << "Health: " << player.getHealth() << '\n';		

2.	//checkHealEvent
	Player player;
	Field field(5, 5);
	field.getCell(1, 0) = Cell(true, new DamageEvent(50));
	field.getCell(2, 0) = Cell(true, new HealEvent(30));
	Controller controller(player, field);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Health: " << player.getHealth() << "\n\n";
	controller.move(Direction::right);
	std::cout << controller.getCoordinates().first << '\t' <<
		0 0
		Health: 100
		
		1 0
		Health: 50
		
		2 0
		Health: 80
		
		1 0
		Health: 80
		
		2 0
		Health: 80
			Проверка работоспособности игрового события, увеличивающего очки здоровья игрока

	controller.getCoordinates().second << '\n';
	std::cout << "Health: " << player.getHealth() << "\n\n";
	controller.move(Direction::right);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Health: " << player.getHealth() << "\n\n";
	controller.move(Direction::left);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Health: " << player.getHealth() << "\n\n";
	controller.move(Direction::right);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Health: " << player.getHealth() << '\n';		

3.	//checkScoreEvent Player player; Field field(5, 5);	0 0 Score: 0	Проверка работоспособности игрового события,

	field.getCell(0, 1) = Cell(true, new ScoreEvent(25));
	Controller controller(player, field);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Score: " << player.getScore() << "\n\n";
	controller.move(Direction::up);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Score: " << player.getScore() << "\n\n";
	controller.move(Direction::down);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
	std::cout << "Score: " << player.getScore() << "\n\n";
	controller.move(Direction::up);
	std::cout << controller.getCoordinates().f
		0 1
		Score: 25
		
		0 0
		Score: 25
		
		0 1
		Score: 25
			увеличивающего очки игрока

	irst << '\t' << controller.getCoordinates().second << '\n'; std::cout << "Score: " << player.getScore() << "\n\n";		

4.	//checkTeleportEvent
	Player player;
	Field field(5, 5);
	field.getCell(0, 1) = Cell(true, new TeleportEvent(3, 3));
	field.getCell(3, 3) = Cell(true, new TeleportEvent(0, 3));
	field.getCell(0, 3) = Cell(true, new TeleportEvent(4, 4));
	Controller controller(player, field);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << "\n\n";
	controller.move(Direction::up);
	std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << "\n\n";
		0 0
		
		4 4
			Проверка работоспособности игрового события, перемещающего игрока в какую-то клетку поля

5.	// Level_1 Player player; FieldCreator field_creator;	Coordinates before: 0 0 Score before: 0 Coordinates after: 0 1	Проверка работоспособности игровых событий в рамках

	Field field = field_creator.createLevel_1(); //field.FieldView(); Controller controller(player, field); std::pair <int, int> coordinates_before = controller.getCoordinates(); std::cout << "Coordinates before: " << coordinates_before.first << '\t' <<coordinates_before.second << '\n'; std::cout << "Score before: " << player.getScore() << '\n'; controller.move(Direction::right); controller.move(Direction::right); std::pair <int, int> coordinates_after_teleport1 = controller.getCoordinates(); std::cout << "Coordinates after: " << coordinates_after_teleport1.first << '\t' <<coordinates_after_teleport1.second << '\n';	Score after first bonus: 50 Health before damage: 100 Health after damage: 50 Health after heal: 100 Health after damage: 50 Coordinates after: 0 3 Score after second bonus: 100 Health (no damage at the trap location): 50 Is the player dead? false	игрового поля первого уровня

```
std::cout << "Score after first bonus: " << player.getScore() << "\n\n";

std::cout << "Health before damage: " << player.getHealth() << '\n';
controller.move(Direction::right);
std::cout << "Health after damage: " << player.getHealth() << "\n\n";

//controller.move(Direction::right);
//std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';

controller.move(Direction::left);
controller.move(Direction::up);
std::cout << "Health after heal: " << player.getHealth() << '\n';

controller.move(Direction::up);
controller.move(Direction::right);
```

```
std::cout << "Health after damage: " << player.getHealth() << "\n\n";

controller.move(Direction::right);
controller.move(Direction::right);
controller.move(Direction::right);
std::pair <int, int> coordinates_after_teleport2 = controller.getCoordinates();
std::cout << "Coordinates after: " << coordinates_after_teleport2.first << '\t' <<coordinates_after_teleport2.second << '\n';
std::cout << "Score after second bonus: " << player.getScore() << "\n";

controller.move(Direction::right);
std::cout << "Health (no damage at the trap location): " << player.getHealth() << "\n\n";

for (size_t i = 0; i < 3; i++) {
controller.move(Direction::right);
```

	}
	controller.move(Direction::up);
	std::cout << std::boolalpha << "Is the player dead? " << player.isDead() << '\n';		

6.	//Level_2
	Player player;
	FieldCreator creator;
	Field field = creator.createLevel_2();
	//field.FieldView();
	Controller controller(player, field);
	std::pair <int, int> coordinates_before = controller.getCoordinates();
	std::cout << "Coordinates before: " << coordinates_before.first << '\t' <<coordinates_before.second << '\n';
	std::cout << "Score before: " << player.getScore() << '\n';
	std::cout << "Health before damage: " << player.getHealth() << "\n\n";
	
	controller.move(Direction::down);
		Coordinates before: 0 9
		Score before: 0
		Health before damage: 100
		
		Coordinates after: 5 9
		Health after damage: 1
		
		2 8
		Coordinates after: 5 9
		Health (no damage at the trap location) : 1
		
		3 0
		Health after heal: 100
		
		9 0
		Finish Score: 100
			Проверка работоспособности игровых событий в рамках игрового поля второго уровня

```
controller.move(Direction::down);
    std::pair <int, int> coordinates_after_teleport1 = controller.getCoordinates();
    std::cout << "Coordinates after: " << coordinates_after_teleport1.first << '\t' <<coordinates_after_teleport1.second << '\n';
    std::cout << "Health after damage: " << player.getHealth() << "\n\n";

    for (size_t x = 5; x > 2; x--) {

controller.move(Direction::left);
    }

controller.move(Direction::down);
    std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';

controller.move(Direction::down);
```

```
    std::pair <int, int> coordinates_after_teleport2 = controller.getCoordinates();
    std::cout << "Coordinates after: " << coordinates_after_teleport2.first << '\t' <<coordinates_after_teleport2.second << '\n';
    std::cout << "Health (no damage at the trap location) : " << player.getHealth() << "\n\n";


controller.move(Direction::left);

controller.move(Direction::left);
    for (int y = 9; y >= 0; y--)
{

controller.move(Direction::down);
    }
    std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
```

```
    std::cout << "Health after heal: " << player.getHealth() << "\n\n";
    for (int x = 3; x <= 9; x++)
{

controller.move(Direction::right);
    }
    std::cout << controller.getCoordinates().first << '\t' << controller.getCoordinates().second << '\n';
    std::cout << "Finish Score: " << player.getScore() << '\n';
```