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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Объектно-ориентированное программирование»

Студентка гр. 4384 _____ Мулюкина Е.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Изучить принципы объектно-ориентированного программирования. Написать программу на языке C++, которая будет прототипом пошаговой игры с перемещением персонажа и сражением с врагами.

Задание.

На 6/3/1 баллов:

Создать интерфейс карточки заклинания. Заклинание должно применяться игроком. На использование заклинания игрок тратит один ход.

Создать класс “руки” игрока, которая содержит все карточки заклинаний, которые игрок может применить в свой ход. Изначально рука игрока содержит только одно случайное заклинание. Реализовать возможность получать новые заклинание игроком, например, тратить очки на покупку или после уничтожения определенного кол-ва врагов. Размер “руки” должен быть ограничен и задается через конструктор.

Реализовать интерфейс заклинанием прямого урона. Это заклинание при использовании должно наносить урон врагу или вражескому зданию, если они находятся в достижимом радиусе. Если в качестве цели не выбран враг или вражеское здание, то заклинание не используется.

Реализовать интерфейс заклинания урона по площади. Это заклинание при использовании в допустимом радиусе наносит урон по области 2 на 2 клетки. Заклинание используется, даже если там нет никого.

Выполнение работы.

Была реализована программа, содержащая все указанные в условие лабораторной работы классы и их поля и методы, а именно:

- Интерфейс карточки заклинания.
- Класс руки игрока, который хранит в себе заклинания, которые можно использовать. Максимум 3.

- Родительский класс для двух заклинаний.
- Класс магазина заклинаний для того, чтобы игрок мог приобрести заклинания только за заработанные очки.
- Интерфейс заклинания прямого урона наносит урон врагу в радиусе 3 клеток.
- Интерфейс заклинания урона по площади наносит урон врагу по области 2*2 с центром-позицией игрока.
- Структура SpellOffer - структура предложения в магазине (заклинание, название, описание, цена)

Также были изменены некоторые предыдущие классы для взаимодействия с новыми классами.

Архитектура программы.

В программе реализована иерархия классов, соответствующая принципам ООП без «божественных» классов. Реализованы интерфейсы – классы чистых виртуальных функции, где все методы публичны и нет полей.

Дополнительные enum классы:

- CellType – класс обозначения типа клетки («p» – player, «e» – enemy, «.» - empty)
- MoveDirection – класс передвижений игрока

Пространства имен:

- GameConstants – пространство всех констант нужных для программы

Описание классов.

Основные классы:

- Класс GameObject

  Класс содержит общие характеристики и методы всех персонажей. Создан для избегания дублирования кода в классах Player и Enemy.

  Поля класса:

  - healthPoints – текущее здоровье объекта

- damagePoints – наносимый урон объектом
- currentPosition – текущая позиция объекта

Методы класса:

- getHealth(), getDamage(), getPosition() – базовые геттеры (получение количества здоровья, количества урона, позиции персонажа)
- setPosition(), takeDamage() – базовые сеттеры (изменение позиции на поле персонажа, изменение количества урона)
- isAlive() - проверка жизнеспособности
- validateHealth() - гарантия корректности здоровья

- Класс Player

Класс содержит характеристики и методы персонажа, за которого можно играть. Методы и поля наследуются от класса GameObject.

Поля класса:

- Score- количество очков игрока
- Hand - инвентарь для хранения заклинаний (максимум 3)

Методы класса:

- move() – перемещение игрока по полю на указанное смещение
- getScore(), addScore(), setScore() - управление очками игрока
- learnSpell() - изучение нового заклинания
- canLearnSpell() - проверка возможности изучения заклинания
- getHand() - получение доступа к инвентарю заклинаний
- castSpell() - применение заклинания по индексу
- canAffordSpell() - проверка возможности покупки заклинания
- buySpell() - покупка заклинания за очки
- removeSpell() - удаление заклинания из инвентаря

- Класс Enemy

Класс содержит характеристики и методы вражеских персонажей.

Методы и поля наследуются от класса GameObject.

Методы класса:

- canAttackPlayer() – проверяет, может ли враг атаковать врага (атака возможна только при совпадении клеток)

- Класс Cell

  Класс содержит характеристики и методы клеток поля.

  Поля класса:

  - cellType – тип содержимого клетки

  Методы класса:

  - getType() – геттер
  - setType() – сеттер
  - isEmpty() – проверка пуста ли клетка

- Класс GameField

  Класс содержит характеристики и методы игрового поля, а также методы взаимодействия различных классов с игровым полем.

  Поля класса:

  - field_width– ширина поля
  - field_height – высота поля
  - grid – двумерная сетка клеток

  Методы класса:

  - getWidth() – возвращает ширину поля
  - getHeight() – возвращает длину поля
  - isValidPosition() – проверка находится ли позиция внутри поля
  - isPositionEmpty() – проверка пуста ли данная клетка
  - setCellType() – устанавливает новое значение клетки
  - clearCell() - устанавливает тип клетки empty (после убийства врага)

- initializeGrid() – приватный метод инициализирует сетку клеток
- copyFrom() – приватный метод копирует данные из другого объекта
- moveFrom() – приватный метод перемещает данные из другого объекта

- Класс Position

  Класс содержит поля и методы взамодействия с позициеей объекта игры на поле.

Поля класса:

- xCoordinate – координата x
- yCoordinate – координата y

Методы класса:

- getX(), getY() – геттеры
- setX(), setY(), setPosition() – сеттеры
- operator==() – проверка равенства двух позиций
- operator!=() - проверка неравенства двух позиций (использует уже реализованный оператор сравнения выше)

- Класс SpellCard

  Поля класса:

- used - флаг использования заклинания

Методы класса:

- cast() - применение заклинания (виртуальный метод)
- getName() - получение названия заклинания (виртуальный метод)
- getDescription() - получение описания заклинания (виртуальный метод)

- canCast() - проверка возможности применения заклинания (виртуальный метод)
- isUsed() - проверка использовано ли заклинание
- markAsUsed() - пометка заклинания как использованного

- Класс SpellShop

  Поля класса:

  - availableSpells - вектор доступных для покупки заклинаний
  - gameController - указатель на игровой контроллер

  Методы класса:

  - initializeSpells() - заполняет список доступных заклинаний (Fire Bolt и Fireball)
  - getAvailableSpells() - возвращает список доступных заклинаний
  - buySpell() - покупает заклинание для игрока по указанному индексу, проверяет валидность индекса и возможность покупки

- Класс PlayerHand

  Поля класса:

  - spells - вектор заклинаний в руке игрока
  - maxHandSize - максимальный размер руки

  Методы класса:

  - addSpell() - добавляет заклинание в руку если есть место
  - removeSpell() - удаляет заклинание по индексу
  - canAddSpell() - проверяет можно ли добавить заклинание
  - isEmpty() - проверяет пуста ли рука
  - isFull() - проверяет заполнена ли рука
  - getSize() - возвращает текущее количество заклинаний
  - getMaxSize() - возвращает максимальный размер руки
  - getSpells() - возвращает вектор заклинаний

- getSpell() - возвращает заклинание по индекс
- clear() - очищает руку от всех заклинаний

- Класс DamageSpell

  Родительский класс для двух вариантов заклинаний.

  Поля класса:
  - spellName - название заклинания
  - spellDescription - описание заклинания
  - spellDamage - наносимый урон
  - spellRange - дальность применения
  - targetPosition - позиция цели
  - gameController - указатель на игровой контроллер

  Методы класса:
  - cast() - применяет заклинание к цели, отмечает как использованное
  - canCast() - проверяет можно ли применить заклинание (не использовано и есть valid targets)
  - getName() - возвращает название заклинания
  - getDescription() - возвращает описание заклинания
  - setTarget() - устанавливает позицию цели
  - isValidTarget() - проверяет валидность цели (живой враг в радиусе действия)
  - getValidTargets() - возвращает список валидных целей в радиусе действия
  - applyEffect() - чисто виртуальный метод применения эффекта
  - getDamage() - возвращает урон заклинания
  - getRange() - возвращает дальность заклинания

- Класс DirectDamageSpell

  Поля все наследует от родительского класса.

Методы класса:

- applyEffect() - наносит урон выбранной цели

- Класс AreaDamageSpell

  Поля класса:

  - areaSize - размер области поражения

  Методы класса:

  - applyEffect() - наносит урон всем врагам в области поражения
  - getAreaCells() - вычисляет все клетки в области поражения вокруг центра

- Класс GameController

  Класс содержит основную логику игру (перемещение игрока, аттака врага и игрока, завершение и начало игры).

Поля класса:

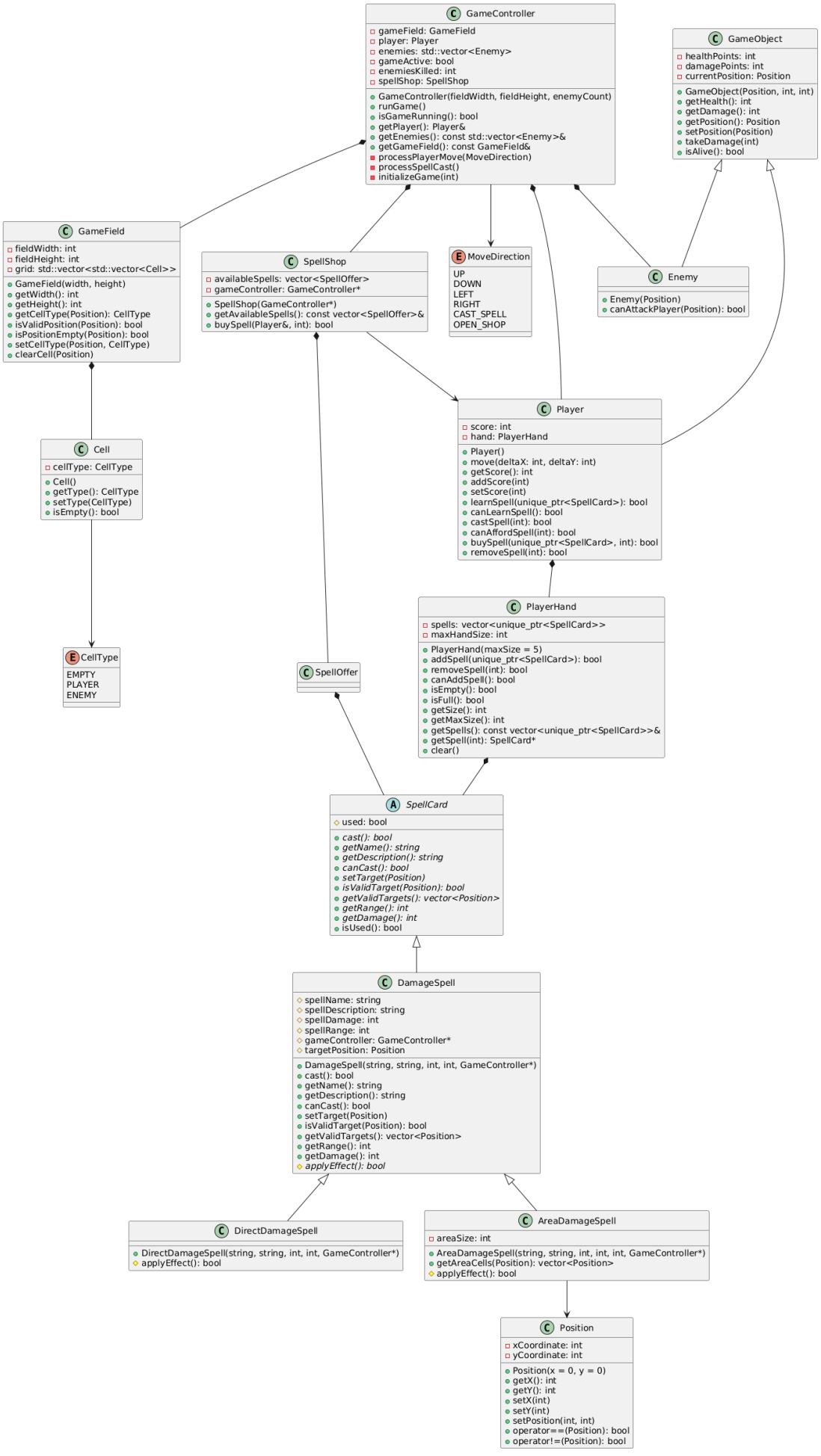
- gameField – игровое поле
- player – объект игрока
- enemies – вектор врагов
- gameActive – флаг активна ли игра
- spellShop - магазин заклинаний
- enemiesKilled - счетчик убитых врагов

Методы класса:

- isGameRunning() – возвращает активна ли игра
- runGame() – запускает игру
  - getPlayerInput() – запрашивает у пользователя команду передвижения (в случае неверного ввода очищает буфер и запрашивает заново)
  - displayGameState() – отображает текущее состояние игры (показывает статистику игрока и врагов, отрисовывает поле в консоли)

- processPlayerMove() – обрабатывает ход игрока (перемещает игрока)
- initializeGame() – инициализирует игру (создает поле)
- placePlayerAtStart() – размещает игрока на стартовой позиции
- tryMovePlayer() – пытается переместить игрока на новую позицию (проверяет возможно ли перемещение, обрабатывает взаимодействие с клеткой)
- placeEnemies() – создает и размещает врагов на поле
- checkGameConditions() – проверяет условия окончания игры (жив ли игрок, остались ли враги)
- applyDamageFromNearbyEnemies() – проверяет есть ли живые враги на соседней клетке с игроком. Если есть, атакует их.
- calculateNewPosition() – вычисляет новую позицию игрока после перемещения
- clearInputBuffer() – очищает буфер входного потока
- processSpellCast() - обработка применения заклинаний
- processShop() - обработка магазина заклинаний
- processDirectDamageSpell() - обработка заклинания прямого урона
- processAreaDamageSpell() - обработка заклинания площадного урона
- displaySpells() - отображение списка заклинаний
- displayShop() - отображение магазина
- displayTargetingHelp() - помощь по прицеливанию для прямого урона
- displayAreaTargetingHelp() - помощь по прицеливанию для площадного урона
- getTargetPosition() - получение позиции цели

UML-диаграммы классов.


GameController
- gameField: GameField
- player: Player
- enemies: std::vector<Enemy>
- gameActive: bool
- enemiesKilled: int
- spellShop: SpellShop
+ GameController(fieldWidth, fieldHeight, enemyCount)
+ runGame()
+ isGameRunning(): bool
+ getPlayer(): Player&
+ getEnemies(): const std::vector<Enemy>&
+ getGameField(): const GameField&
# processPlayerMove(MoveDirection)
# processSpellCast()
# initializeGame(int)
GameObject
- healthPoints: int
- damagePoints: int
- currentPosition: Position
+ GameObject(Position, int, int)
+ getHealth(): int
+ getDamage(): int
+ getPosition(): Position
+ setPosition(Position)
+ takeDamage(int)
+ isAlive(): bool
GameField
- fieldWidth: int
- fieldHeight: int
- grid: std::vector<std::vector<Cell>>
+ GameField(width, height)
+ getWidth(): int
+ getHeight(): int
+ getCellType(Position): CellType
+ isValidPosition(Position): bool
+ isPositionEmpty(Position): bool
+ setCellType(Position, CellType)
+ clearCell(Position)
SpellShop
- availableSpells: vector<SpellOffer>
- gameController: GameController*
+ SpellShop(GameController*)
+ getAvailableSpells(): const vector<SpellOffer>&
+ buySpell(Player&, int): bool
MoveDirection
UP
DOWN
LEFT
RIGHT
CAST_SPELL
OPEN_SHOP
Enemy
+ Enemy(Position)
+ canAttackPlayer(Position): bool
Player
- score: int
- hand: PlayerHand
+ Player()
+ move(deltaX: int, deltaY: int)
+ getScore(): int
+ addScore(int)
+ setScore(int)
+ learnSpell(unique_ptr<SpellCard>): bool
+ canLearnSpell(): bool
+ castSpell(int): bool
+ canAffordSpell(int): bool
+ buySpell(unique_ptr<SpellCard>, int): bool
+ removeSpell(int): bool
Cell
- cellType: CellType
+ Cell()
+ getType(): CellType
+ setType(CellType)
+ isEmpty(): bool
CellType
EMPTY
PLAYER
ENEMY
PlayerHand
- spells: vector<unique_ptr<SpellCard>>
- maxHandSize: int
+ PlayerHand(maxSize = 5)
+ addSpell(unique_ptr<SpellCard>): bool
+ removeSpell(int): bool
+ canAddSpell(): bool
+ isEmpty(): bool
+ isFull(): bool
+ getSize(): int
+ getMaxSize(): int
+ getSpells(): const vector<unique_ptr<SpellCard>>&
+ getSpell(int): SpellCard*
+ clear()
SpellOffer
SpellCard
used: bool
+ cast(): bool
+ getName(): string
+ getDescription(): string
+ canCast(): bool
+ setTarget(Position)
+ isValidTarget(Position): bool
+ getValidTargets(): vector<Position>
+ getRange(): int
+ getDamage(): int
+ isUsed(): bool
DamageSpell
spellName: string
spellDescription: string
spellDamage: int
spellRange: int
gameController: GameController*
targetPosition: Position
+ DamageSpell(string, string, int, int, GameController*)
+ cast(): bool
+ getName(): string
+ getDescription(): string
+ canCast(): bool
+ setTarget(Position)
+ isValidTarget(Position): bool
+ getValidTargets(): vector<Position>
+ getRange(): int
+ getDamage(): int
# applyEffect(): bool
DirectDamageSpell
+ DirectDamageSpell(string, string, int, int, GameController*)
# applyEffect(): bool
AreaDamageSpell
- areaSize: int
+ AreaDamageSpell(string, string, int, int, int, GameController*)
+ getAreaCells(Position): vector<Position>
# applyEffect(): bool
Position
- xCoordinate: int
- yCoordinate: int
+ Position(x = 0, y = 0)
+ getX(): int
+ getY(): int
+ setX(int)
+ setY(int)
+ setPosition(int, int)
+ operator==(Position): bool
+ operator!=(Position): bool

Выводы.

В процессе выполнения второй лабораторной работы я взяла за основу код из первой лабораторной работы и добавила туда выполнение нужных заданий. Были созданы новые классы как интерфейсы, так и обычные. Реализован родительский класс для заклинаний для того, чтобы избежать дублирования кода в двух дочерних классах. Два заклинания отличаются друг от друга областью нанесения урона. Также для получения заклинаний был реализован класс «магазина», где можно купить заклинания за полученные игроком очки в течение игры. Следуя условию задания, был реализован класс руки игрока, где хранятся заклинания, которые можно использовать. При попытке использовать заклинание нужно указать в консоли координаты врага. Для упрощения восприятия области действия заклинания, враги, попадаемые под действие выбранного заклинания, отображаются как «*» или «#». Если игрок ошибся с координатами, и на этой позиции нет врага или он не находится в области действия заклинания, программа не падает, а дает сделать ход ещё раз.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: game_object.h

```
#ifndef GAME_OBJECT_H
#define GAME_OBJECT_H

#include "position.h"

class GameObject {
public:
    GameObject(const Position& position, int health, int damage);
    virtual ~GameObject() = default;

    int getHealth() const;
    int getDamage() const;
    const Position& getPosition() const;
    void setPosition(const Position& position);
    void takeDamage(int damage);
    bool isAlive() const;

protected:
    int healthPoints;
    int damagePoints;
    Position currentPosition;

private:
    void validateHealth();
};

#endif
```

Название файла: game_object.cpp

```
#include "game_object.h"
#include <stdexcept>
```

```
GameObject::GameObject(const Position& position, int health, int
damage)
    : healthPoints(health),
    damagePoints(damage),
    currentPosition(position) {
}

int GameObject::getHealth() const {
    return healthPoints;
}

int GameObject::getDamage() const {
    return damagePoints;
}

const Position& GameObject::getPosition() const {
    return currentPosition;
}

void GameObject::setPosition(const Position& position) {
    currentPosition = position;
}

void GameObject::takeDamage(int damage) {
    if (damage < 0) {
        throw std::invalid_argument("Damage cannot be negative");
    }
    healthPoints -= damage;
    validateHealth();
}

bool GameObject::isAlive() const {
    return healthPoints > 0;
}

void GameObject::validateHealth() {
    if (healthPoints < 0) {
        healthPoints = 0;
    }
```

```
}
```

Название файла: player.h

```
#ifndef PLAYER_H
#define PLAYER_H

#include "game_object.h"
#include "game_constants.h"

class Player : public GameObject {
public:
    Player();
    void move(int deltaX, int deltaY);
};

#endif
```

Название файла: player.cpp

```
#include "player.h"

Player::Player()
    : GameObject(Position(0, 0),
        GameConstants::INITIAL_PLAYER_HEALTH,
        GameConstants::INITIAL_PLAYER_DAMAGE) {
}

void Player::move(int deltaX, int deltaY) {
    currentPosition.setX(currentPosition.getX() + deltaX);
    currentPosition.setY(currentPosition.getY() + deltaY);
}
```

Название файла: enemy.h

```
#ifndef ENEMY_H
#define ENEMY_H

#include "game_object.h"
#include "game_constants.h"

class Enemy : public GameObject {
public:
    Enemy(const Position& position);
```

```
    bool canAttackPlayer(const Position& playerPosition) const;

private:
    static constexpr int ATTACK_RANGE = 1;
};

#endif
```

Название файла: enemy.cpp

```
#include "enemy.h"
#include <cmath>

Enemy::Enemy(const Position& position)
    : GameObject(position,
        GameConstants::INITIAL_ENEMY_HEALTH,
        GameConstants::INITIAL_ENEMY_DAMAGE) {
}

bool Enemy::canAttackPlayer(const Position& playerPosition) const {
    return currentPosition == playerPosition;
}
```

Название файла: position.h

```
#ifndef POSITION_H
#define POSITION_H

class Position {
public:
    Position(int x = 0, int y = 0);

    int getX() const;
    int getY() const;
    void setX(int x);
    void setY(int y);
    void setPosition(int x, int y);

    bool operator==(const Position& other) const;
    bool operator!=(const Position& other) const;

private:
```

```
    int xCoordinate;
    int yCoordinate;
};

#endif
```

Название файла: position.cpp

```
#include "position.h"

Position::Position(int x, int y) : xCoordinate(x), yCoordinate(y) {}

int Position::getX() const {
    return xCoordinate;
}

int Position::getY() const {
    return yCoordinate;
}

void Position::setX(int x) {
    xCoordinate = x;
}

void Position::setY(int y) {
    yCoordinate = y;
}

void Position::setPosition(int x, int y) {
    xCoordinate = x;
    yCoordinate = y;
}

bool Position::operator==(const Position& other) const {
    return xCoordinate == other.xCoordinate && yCoordinate ==
other.yCoordinate;
}

bool Position::operator!=(const Position& other) const {
    return !(*this == other);
```

```
}
```

Название файла: cell.h

```
#ifndef CELL_H
#define CELL_H

enum class CellType {
    EMPTY,
    PLAYER,
    ENEMY
};

class Cell {
public:
    Cell();
    CellType getType() const;
    void setType(CellType type);
    bool isEmpty() const;

private:
    CellType cellType;
};

#endif
```

Название файла: cell.cpp

```
#include "cell.h"

Cell::Cell() : cellType(CellType::EMPTY) {}

CellType Cell::getType() const {
    return cellType;
}

void Cell::setType(CellType type) {
    cellType = type;
}

bool Cell::isEmpty() const {
    return cellType == CellType::EMPTY;
```

```
}
```

Название файла: main.cpp

```
#include "game_controller.h"
#include <iostream>

int main() {
    try {
        GameController gameController(15, 15, 3);
        gameController.runGame();

    }
    catch (const std::exception& exception) {
        std::cerr << "Error: " << exception.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Название файла: game_field.cpp

```
#include "game_field.h"
#include <stdexcept>

GameField::GameField(int width, int height)
    : fieldWidth(width), fieldHeight(height) {

    if (width < GameConstants::MIN_FIELD_SIZE ||
        width > GameConstants::MAX_FIELD_SIZE ||
        height < GameConstants::MIN_FIELD_SIZE ||
        height > GameConstants::MAX_FIELD_SIZE) {
        throw std::invalid_argument("Invalid field dimensions");
    }

    initializeGrid();
}

GameField::GameField(const GameField& other) //конструктор
копирования
    : fieldWidth(other.fieldWidth),
```

```
        fieldHeight(other.fieldHeight) {
            copyFrom(other);
        }

        GameField::GameField(GameField&& other) noexcept { //конструктор
перемещения
            moveFrom(std::move(other));
        }

        GameField& GameField::operator=(const GameField& other)
{ //оператор присваивания с копированием
            if (this != &other) {
                fieldWidth = other.fieldWidth;
                fieldHeight = other.fieldHeight;
                copyFrom(other);
            }
            return *this;
        }

        GameField& GameField::operator=(GameField&& other) noexcept
{ //оператор присваивания с перемещением
            if (this != &other) {
                moveFrom(std::move(other));
            }
            return *this;
        }

        int GameField::getWidth() const {
            return fieldWidth;
        }

        int GameField::getHeight() const {
            return fieldHeight;
        }

        CellType GameField::getCellType(const Position& position) const {
            if (!isValidPosition(position)) {
                throw std::out_of_range("Position is out of field bounds");
```

```
        }
        return grid[position.getY()][position.getX()].getType();
    }

    bool GameField::isValidPosition(const Position& position) const {
      return position.getX() >= 0 && position.getX() < fieldWidth &&
            position.getY() >= 0 && position.getY() < fieldHeight;
    }

    bool GameField::isPositionEmpty(const Position& position) const {
        return isValidPosition(position) &&
            grid[position.getY()][position.getX()].isEmpty();
    }

    void GameField::setCellType(const Position& position, CellType type)
{
        if (!isValidPosition(position)) {
            throw std::out_of_range("Position is out of field bounds");
        }
        grid[position.getY()][position.getX()].setType(type);
    }

    void GameField::clearCell(const Position& position) {
        setCellType(position, CellType::EMPTY);
    }

    void GameField::initializeGrid() {
        grid.resize(fieldHeight);
        for (int y = 0; y < fieldHeight; y++) {
            grid[y].resize(fieldWidth);
            for (int x = 0; x < fieldWidth; x++) {
                grid[y][x] = Cell();
            }
        }
    }

    void GameField::copyFrom(const GameField& other) {
        grid = other.grid;//копирование и указателей и элементов
    }
```

```
void GameField::moveFrom(GameField&& other) noexcept {
    fieldWidth = other.fieldWidth;
    fieldHeight = other.fieldHeight;
    grid = std::move(other.grid);

    other.fieldWidth = 0;
    other.fieldHeight = 0;
}
```

Название файла: game_field.h

```
#ifndef GAME_FIELD_H
#define GAME_FIELD_H

#include "cell.h"
#include "position.h"
#include "game_constants.h"
#include <vector>
#include <memory>

class GameField {
public:
    GameField(int width = GameConstants::DEFAULT_FIELD_SIZE,
        int height = GameConstants::DEFAULT_FIELD_SIZE);

    GameField(const GameField& other);
    GameField(GameField&& other) noexcept;

    GameField& operator=(const GameField& other);
    GameField& operator=(GameField&& other) noexcept;

    int getWidth() const;
    int getHeight() const;
    CellType getCellType(const Position& position) const;

    bool isValidPosition(const Position& position) const;
    bool isPositionEmpty(const Position& position) const;

    void setCellType(const Position& position, CellType type);
```

```
    void clearCell(const Position& position);

private:
    int fieldWidth;
    int fieldHeight;
    std::vector<std::vector<Cell>> grid;

    void initializeGrid();
    void copyFrom(const GameField& other);
    void moveFrom(GameField&& other) noexcept;
};

#endif
```