

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: Реализация консольной игры с использованием ООП.

Студент гр. 4384

Мазеев В.А.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Целью работы является разработка консольной игры с использованием принципов объектно-ориентированного программирования, демонстрирующей взаимодействие классов, инкапсуляцию, наследование.

Задание.

На 6/3/1 баллов:

- Создать класс игрока, который должен хранить информацию об игроке (его жизни, урон, очки, и т.д. - студент сам определяет необходимые для работы характеристики). Объект класса игрока должен перемещаться по карте. Если у игрока кончаются жизни, то происходит конец игры.
- Создать класс врага, который хранит параметры жизней и урона. Объектами класса врага управляет компьютер. При перемещении, если враг пытается перейти на клетку с игроком, то перемещение не происходит, и игроку наносится урон.
- Создать класс квадратного/прямоугольного игрового поля, по которому перемещаются игрок и враги. Игровое поле не должно быть меньше 10 на 10 клеток, и не больше 25 на 25 клеток. Размеры поля задаются через конструктор. Рекомендуется для хранения информации об отдельных клетках поля создать отдельный класс.
- Реализовать конструкторы перемещения и копирования для поля, а также соответствующие операторы присваивания с копированием и перемещением (должна происходить глубокая копия).

На 8/4/1.5 баллов:

- Реализовать непроходимые клетки на поле. При попытке врагов или игрока перейти на такую клетку, перемещение не происходит. Заполнение поля непроходимыми клетками происходит в момент создания поля.

- Добавить возможность для игрока переключаться на ближний или дальний бой с изменением значения наносимого урона. Такое переключение требует один ход.

На 10/5/2 баллов:

- Добавить класс вражеского здания. Такое здание размещается на карте, и раз в несколько ходов создает нового врага возле себя. Количество ходов до создания нового врага задается в конструкторе.
- Реализовать замедляющие клетки на поле. Если игрок переходит на такую клетку, то он не может двигаться на следующий ход.

Выполнение работы.

Результатом работы стала программа на языке C++, реализующая простую консольную игру с элементами пошагового взаимодействия между игроком и врагами. Программа основана на применении объектно-ориентированного подхода и разделена на несколько классов, каждый из которых отвечает за отдельную часть логики:

Программа состоит из семи основных классов:

1. Класс *Cell*.

Назначение: представляет собой одну клетку игрового поля.

Основные методы:

setType(CellType t) – устанавливает тип клетки(пуста, игрок, враг, стена)

getType() – возвращает текущий тип клетки.

isEmpty(), *isWall()*, *isOccupied()* – проверяют состояние клетки.

2. Класс *Entity*

Назначение: базовый класс для всех существ на поле (игрок, враг).

Основные методы:

takeDamage(int damage) – уменьшает здоровье сущности.

isAlive() – проверяет, жива ли сущность.

getDamage() – возвращает урон (чисто виртуальный метод).

getDisplayChar() – возвращает символ для отображения (чисто виртуальный метод).

3. Класс *Player*.

Назначение: описывает игрока на поле. Управляется пользователем, может атаковать врагов и переключать режим боя.

Основные методы:

move(int dx, int dy, Field& field) – перемещает игрока по полю.

attack(int targetX, int targetY, Field& field) – атакует врага в указанной клетке.

switchCombatMode() – переключает режим боя (ближний/дальний).

getScore(), *addScore()* – управление очками игрока.

4. Класс *Enemy*.

Назначение: описывает врага. Враги перемещаются случайным образом и атакуют игрока при столкновении.

Основные методы:

move(int dx, int dy, Field& field) – перемещает врага по полю.

attack(int targetX, int targetY, Field& field) – атакует игрока.

canMove() – проверяет возможность перемещения.

5. Класс *Field*.

Назначение: управляет игровым полем, хранит сетку клеток и размещённые на ней сущности.

Основные методы:

setPlayer(std::unique_ptr<Player> p) – размещает игрока на поле.

addEnemy(std::unique_ptr<Enemy> enemy) – добавляет врага.

draw() – отрисовывает поле в консоли.

isValidPosition(), *isWall()*, *isEmpty()* – проверяют состояние клеток.

6. Класс *Game*.

Назначение: управляет игровым процессом, содержит основной цикл игры.

Основные методы:

initializeGame() – инициализирует игрока и врагов.

run() – основной игровой цикл.

processEnemiesTurn() – обрабатывает ходы врагов.

checkWin(), checkLose() – проверяют условия победы/поражения.

7. Функция *main()*.

Назначение: точка входа в программу.

Создает объект класса *Game*, вызывает метод *run()* для запуска игрового цикла, после завершения освобождает память.

Пример работы программы представлены в приложении А.

Исходный код программы представлен в приложении В.

Архитектура.

Выбранная архитектура программы основана на принципах объектно-ориентированного программирования, что обеспечивает четкое разделение ответственности между классами. Каждый класс решает конкретную задачу: *Cell* отвечает за состояние клетки поля, *Entity* является базовым классом для всех сущностей, *Player* и *Enemy* реализуют поведение персонажей, *Field* управляет игровым полем, а *Game* координирует игровой процесс.

Базовый класс *Entity* содержит общую логику для всех сущностей, а производные классы *Player* и *Enemy* переопределяют виртуальные методы, что позволяет обрабатывать разные типы сущностей через единый интерфейс. Это обеспечивает легкость расширения системы - для добавления новых типов персонажей достаточно наследоваться от *Entity*.

Инкапсуляция данных в классах защищает состояние объектов от некорректного изменения и обеспечивает контролируемое взаимодействие через четко определенные методы. Композиция и агрегация, реализованные через включение *Field* в *Game* и использование умных указателей,

обеспечивают безопасное управление памятью и автоматическое освобождение ресурсов.

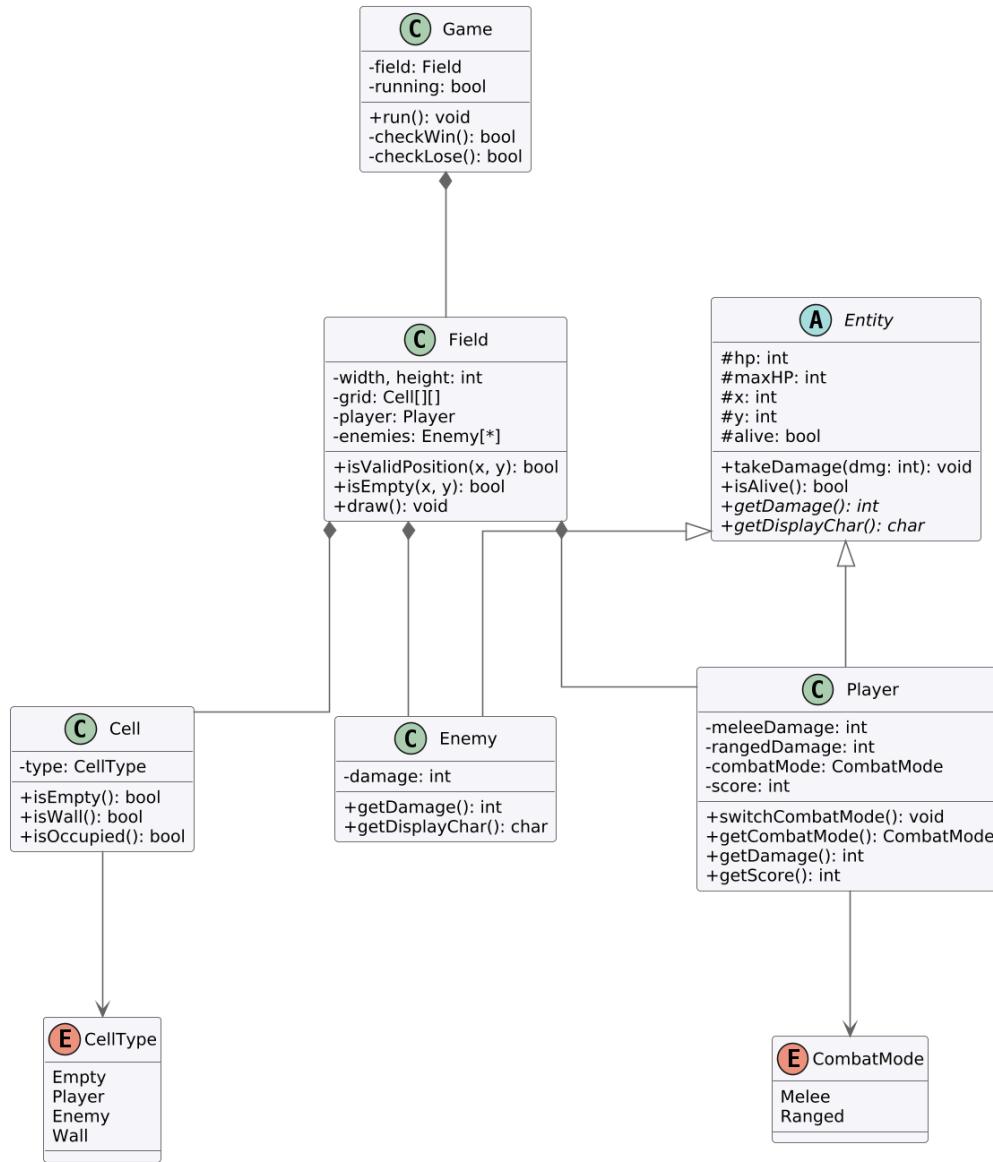


Рис. 1 UML-диаграмма классов.

Выводы.

В ходе лабораторной работы была разработана консольная игра на языке C++, реализованная с применением объектно-ориентированного программирования. В процессе выполнения были созданы и связаны между собой классы Game, Field, Entity, Player, Enemy и Cell, каждый из которых выполняет строго определенную роль в архитектуре программы.

Разработанная структура демонстрирует принципы инкапсуляции, наследования и слабой связанности компонентов, что облегчает поддержку и масштабирование кода. Игровой процесс: игрок перемещается по полю, враги совершают случайные ходы и атакуют при столкновении, поле обновляется в реальном времени.

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ

tests.cpp

```
#include <iostream>
#include <cassert>

// Include all game headers
#include "Cell.h"
#include "Entity.h"
#include "Player.h"
#include "Enemy.h"
#include "Field.h"

// ===== Testing Cell =====
void testCell() {
    std::cout << "Testing Cell..." << std::endl;

    Cell cell;

    // Check initial state
    assert(cell.isEmpty() == true);
    assert(cell.isWall() == false);

    // Check type setting
    cell.setType(CellType::Wall);
    assert(cell.isWall() == true);
    assert(cell.isEmpty() == false);

    cell.setType(CellType::Player);
    assert(cell.getType() == CellType::Player);
    assert(cell.isOccupied() == true);

    std::cout << "Cell tests passed!" << std::endl;
}

// ===== Testing Entity =====
void testEntity() {
    std::cout << "Testing Entity..." << std::endl;

    // Create test class for abstract Entity
    class TestEntity : public Entity {
    public:
        TestEntity(int x, int y, int hp) : Entity(x, y, hp) {}
        int getDamage() const override { return 10; }
        char getDisplayChar() const override { return 'T'; }
    };

    TestEntity entity(5, 5, 100);

    // Check position and health
    assert(entity.getX() == 5);
    assert(entity.getY() == 5);
    assert(entity.getHP() == 100);
    assert(entity.isAlive() == true);

    // Check damage taking
}
```

```

entity.takeDamage(30);
assert(entity.getHP() == 70);

// Check death
entity.takeDamage(100);
assert(entity.getHP() == 0);
assert(entity.isAlive() == false);

std::cout << "Entity tests passed!" << std::endl;
}

// ===== Testing Player =====
void testPlayer() {
    std::cout << "Testing Player..." << std::endl;

    Player player(2, 2, 100, 15, 5);

    // Check initial state
    assert(player.getHP() == 100);
    assert(player.getCombatMode() == CombatMode::Melee);
    assert(player.getDamage() == 15); // Melee mode

    // Check combat mode switching
    player.switchCombatMode();
    assert(player.getCombatMode() == CombatMode::Ranged);
    assert(player.getDamage() == 5); // Ranged mode

    // Check score system
    assert(player.getScore() == 0);
    player.addScore(10);
    assert(player.getScore() == 10);

    // Check display character
    assert(player.getDisplayChar() == 'P');

    std::cout << "Player tests passed!" << std::endl;
}

// ===== Testing Enemy =====
void testEnemy() {
    std::cout << "Testing Enemy..." << std::endl;

    Enemy enemy(3, 3, 50, 8);

    // Check initial state
    assert(enemy.getHP() == 50);
    assert(enemy.getDamage() == 8);
    assert(enemy.isAlive() == true);

    // Check damage taking
    enemy.takeDamage(20);
    assert(enemy.getHP() == 30);
    assert(enemy.isAlive() == true);

    // Check display character
    assert(enemy.getDisplayChar() == 'E');

    std::cout << "Enemy tests passed!" << std::endl;
}

```

```

}

// ===== Testing Field =====
void testField() {
    std::cout << "Testing Field..." << std::endl;

    // Create field with valid size (10x10 minimum)
    Field field(10, 10);

    // Check dimensions
    assert(field.getWidth() == 10);
    assert(field.getHeight() == 10);

    // Check position validation
    assert(field.isValidPosition(0, 0) == true);
    assert(field.isValidPosition(9, 9) == true);
    assert(field.isValidPosition(-1, 0) == false);
    assert(field.isValidPosition(10, 10) == false);

    // Add player
    auto player = std::make_unique<Player>(1, 1);
    field.setPlayer(std::move(player));

    // Check player was added
    assert(field.getPlayer() != nullptr);
    assert(field.hasPlayerAt(1, 1) == true);

    // Add enemy
    auto enemy = std::make_unique<Enemy>(2, 2);
    field.addEnemy(std::move(enemy));

    // Check enemy was added
    assert(field.getEnemies().size() == 1);
    assert(field.hasEnemyAt(2, 2) == true);

    std::cout << "Field tests passed!" << std::endl;
}

// ===== Testing Movement =====
void testMovement() {
    std::cout << "Testing Movement..." << std::endl;

    Field field(10, 10);
    auto player = std::make_unique<Player>(5, 5);
    field.setPlayer(std::move(player));

    // Just verify that player was placed correctly
    assert(field.getPlayer() != nullptr);
    assert(field.getPlayer()->getX() == 5);
    assert(field.getPlayer()->getY() == 5);

    std::cout << "Movement tests passed!" << std::endl;
}

// ===== Main testing function =====
int main() {
    std::cout << "==== Starting Simple Game Tests ===" << std::endl;
        std::cout << "Running basic functionality tests...\n" <<
std::endl;
}

```

```

try {
    testCell();
    testEntity();
    testPlayer();
    testEnemy();
    testField();
    testMovement();

    std::cout << "\n==== ALL TESTS PASSED! ===" << std::endl;
    std::cout << "Basic game functionality is working correctly!" << std::endl;
}

} catch (const std::exception& e) {
    std::cerr << "\nTest failed: " << e.what() << std::endl;
    return 1;
} catch (...) {
    std::cerr << "\nUnknown test failure" << std::endl;
    return 1;
}

return 0;
}

```

```

Running basic functionality tests...

Testing Cell...
Cell tests passed!
Testing Entity...
Entity tests passed!
Testing Player...
Player tests passed!
Ranged fight
Testing Enemy...
Enemy tests passed!
Testing Field...
Field tests passed!
Testing Movement...
Movement tests passed!

==== ALL TESTS PASSED! ===
Basic game functionality is working correctly!

```

Рис. 2 Результаты тестирования.

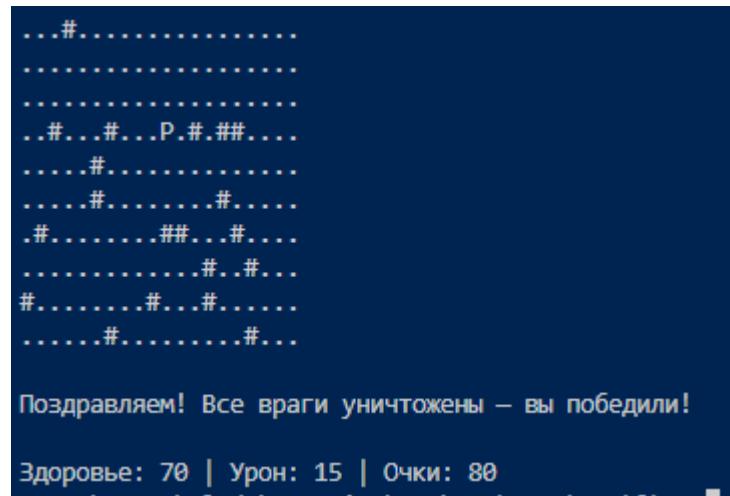


Рис. 3 Начало игры.

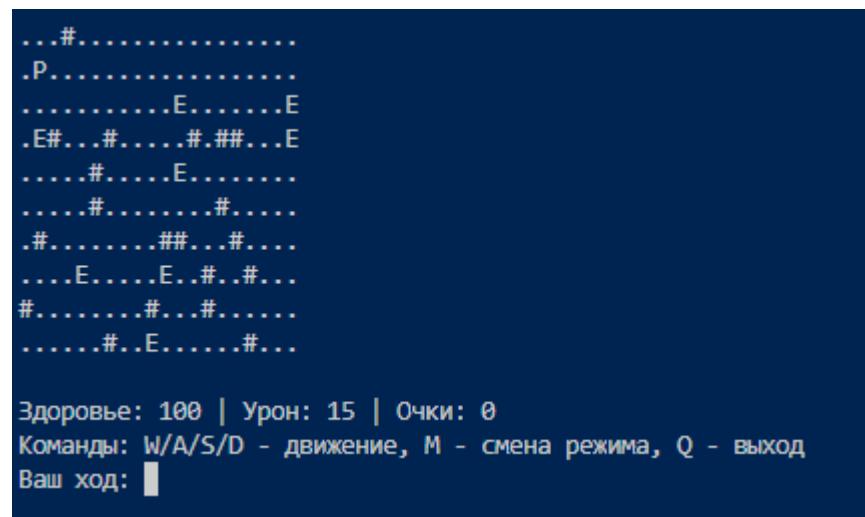


Рис. 4 Конец игры.

ПРИЛОЖЕНИЕ В ИСХОДНЫЙ КОД

main.cpp

```
#include "Game.h"

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

Entity.cpp

```
#include "Entity.h"

Entity::Entity(int startX, int startY, int health)
    : hp(health), maxHP(health), x(startX), y(startY), alive(true) {}

void Entity::takeDamage(int damage) {
    hp -= damage;
    if (hp <= 0) {
        hp = 0;
        alive = false;
    }
}

bool Entity::isAlive() const {
    return alive && hp > 0;
}

int Entity::getX() const {
    return x;
}

int Entity::getY() const {
    return y;
}

int Entity::getHP() const {
    return hp;
}

int Entity::getMaxHP() const {
    return maxHP;
}

void Entity::setPosition(int newX, int newY) {
    x = newX;
    y = newY;
}
```

Entity.h

```
#pragma once
#include <memory>

class Field;
```

```

enum class CombatMode {
    Melee,
    Ranged
};

class Entity {
protected:
    int hp;
    int maxHP;
    int x, y;
    bool alive;

public:
    Entity(int startX, int startY, int health);
    virtual ~Entity() = default;

    Entity(const Entity& other) = default;
    Entity& operator=(const Entity& other) = default;
    Entity(Entity&& other) noexcept = default;
    Entity& operator=(Entity&& other) noexcept = default;

    virtual void takeDamage(int damage);
    virtual bool isAlive() const;
    virtual int getDamage() const = 0;

    int getX() const;
    int getY() const;
    int getHP() const;
    int getMaxHP() const;

    void setPosition(int newX, int newY);
    virtual char getDisplayChar() const = 0;
};


```

Cell.cpp

```

#include "Cell.h"

Cell::Cell() : type(CellType::Empty) {}

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

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

bool Cell::isEmpty() const {
    return type == CellType::Empty;
}

bool Cell::isWall() const {
    return type == CellType::Wall;
}

bool Cell::isOccupied() const {

```

```
        return type == CellType::Player || type == CellType::Enemy;
    }
```

Cell.h

```
#pragma once

enum class CellType { Empty, Player, Enemy, Wall };

class Cell {
private:
    CellType type;

public:
    Cell();
    Cell(const Cell& other) = default;
    Cell& operator=(const Cell& other) = default;
    Cell(Cell&& other) noexcept = default;
    Cell& operator=(Cell&& other) noexcept = default;

    void setType(CellType t);
    CellType getType() const;
    bool isEmpty() const;
    bool isWall() const;
    bool isOccupied() const;
};
```

Player.cpp

```
#include "Player.h"
#include "Field.h"
#include "Enemy.h"
#include <iostream>

Player::Player(int startX, int startY, int h, int melee, int ranged)
    : Entity(startX, startY, h), meleeDamage(melee),
rangedDamage(ranged),
combatMode(CombatMode::Melee), score(0) {}

Player::Player(const Player& other)
    : Entity(other), meleeDamage(other.meleeDamage),
rangedDamage(other.rangedDamage),
combatMode(other.combatMode), score(other.score) {}

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

    Entity::operator=(other);
    meleeDamage = other.meleeDamage;
    rangedDamage = other.rangedDamage;
    combatMode = other.combatMode;
    score = other.score;

    return *this;
}
```

```

Player::Player(Player&& other) noexcept
    : Entity(std::move(other)), meleeDamage(other.meleeDamage),
      rangedDamage(other.rangedDamage), combatMode(other.combatMode),
      score(other.score) {}

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

    Entity::operator=(std::move(other));
    meleeDamage = other.meleeDamage;
    rangedDamage = other.rangedDamage;
    combatMode = other.combatMode;
    score = other.score;

    return *this;
}

bool Player::canMove(int dx, int dy, const Field& field) const {
    int newX = x + dx;
    int newY = y + dy;

    if (!field.isValidPosition(newX, newY))
        return false;

    if (field.isWall(newX, newY))
        return false;

    return true;
}

bool Player::move(int dx, int dy, Field& field) {
    if (!canMove(dx, dy, field))
        return false;

    int newX = x + dx;
    int newY = y + dy;

    if (field.hasEnemyAt(newX, newY)) {
        return attack(newX, newY, field);
    }

    field.clearCell(x, y);
    setPosition(newX, newY);
    field.placeEntity(this);

    return true;
}

bool Player::attack(int targetX, int targetY, Field& field) {
    auto enemy = field.getEnemyAt(targetX, targetY);
    if (!enemy) return false;

    enemy->takeDamage(getDamage());
    std::cout << "Вы атаковали врага! Урон: " << getDamage() <<
    std::endl;

    if (!enemy->isAlive()) {

```

```

        std::cout << "Враг убит! +10 очков\n";
        addScore(10);
        field.removeEnemy(enemy);
    }

    return false;
}

void Player::switchCombatMode() {
    combatMode = (combatMode == CombatMode::Melee) ?
CombatMode::Ranged : CombatMode::Melee;
    std::cout << "Игрок переключился в " <<
        (combatMode == CombatMode::Melee ? "Melee" : "Ranged") << "
fight\n";
}

CombatMode Player::getCombatMode() const {
    return combatMode;
}

int Player::getDamage() const {
    return (combatMode == CombatMode::Melee) ? meleeDamage :
rangedDamage;
}

int Player::getScore() const {
    return score;
}

void Player::addScore(int points) {
    score += points;
}

char Player::getDisplayChar() const {
    return 'P';
}

```

Player.h

```

#pragma once
#include "Entity.h"

// Forward declaration
class Field;

class Player : public Entity {
private:
    int meleeDamage;
    int rangedDamage;
    CombatMode combatMode;
    int score;

public:
    Player(int startX, int startY, int h = 100, int melee = 15, int
ranged = 5);

    Player(const Player& other);

```

```

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

bool canMove(int dx, int dy, const Field& field) const;
bool move(int dx, int dy, Field& field);
bool attack(int targetX, int targetY, Field& field);

void switchCombatMode();
CombatMode getCombatMode() const;
int getDamage() const override;
int getScore() const;
void addScore(int points);

char getDisplayChar() const override;
};

```

Enemy.cpp

```

#include "Enemy.h"
#include "Player.h"
#include "Field.h"
#include <iostream>

Enemy::Enemy(int startX, int startY, int h, int d)
    : Entity(startX, startY, h), damage(d) {}

Enemy::Enemy(const Enemy& other)
    : Entity(other), damage(other.damage) {}

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

    Entity::operator=(other);
    damage = other.damage;

    return *this;
}

Enemy::Enemy(Enemy&& other) noexcept
    : Entity(std::move(other)), damage(other.damage) {}

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

    Entity::operator=(std::move(other));
    damage = other.damage;

    return *this;
}

bool Enemy::canMove(int dx, int dy, const Field& field) const {
    int newX = x + dx;
    int newY = y + dy;

```

```

        if (!field.isValidPosition(newX, newY))
            return false;

        if (field.isWall(newX, newY))
            return false;

        return true;
    }

bool Enemy::move(int dx, int dy, Field& field) {
    if (!canMove(dx, dy, field))
        return false;

    int newX = x + dx;
    int newY = y + dy;

    // Check if there's a player to attack
    if (field.hasPlayerAt(newX, newY)) {
        return attack(newX, newY, field);
    }

    // Clear current position and move
    field.clearCell(x, y);
    setPosition(newX, newY);
    field.placeEntity(this);

    return true;
}

bool Enemy::attack(int targetX, int targetY, Field& field) {
    auto player = field.getPlayer();
    if (!player || player->getX() != targetX || player->getY() != targetY)
        return false;

    player->takeDamage(damage);
    std::cout << "Enemy attacked player! Damage: " << damage << std::endl;

    return false; // Attack doesn't move the enemy
}

int Enemy::getDamage() const {
    return damage;
}

char Enemy::getDisplayChar() const {
    return 'E';
}

```

Enemy.h

```
#pragma once
#include "Entity.h"
```

```

// Forward declarations
class Field;

class Enemy : public Entity {
private:
    int damage;

public:
    Enemy(int startX, int startY, int h = 30, int d = 5);

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

    bool canMove(int dx, int dy, const Field& field) const;
    bool move(int dx, int dy, Field& field);
    bool attack(int targetX, int targetY, Field& field);

    int getDamage() const override;
    char getDisplayChar() const override;
};


```

Field.cpp

```

#include "Field.h"
#include "Player.h"
#include "Enemy.h"
#include <stdexcept>
#include <cstdlib>
#include <algorithm>

Field::Field(int w, int h) : width(w), height(h) {
    if (w < 10 || h < 10 || w > 25 || h > 25)
        throw std::invalid_argument("Field size must be between 10x10 and 25x25");

    // Initialize grid
    grid = new Cell*[height];
    for (int i = 0; i < height; i++)
        grid[i] = new Cell[width];

    initializeWalls();
}

Field::Field(const Field& other) : width(other.width),
height(other.height) {
    // Copy grid
    grid = new Cell*[height];
    for (int i = 0; i < height; i++) {
        grid[i] = new Cell[width];
        for (int j = 0; j < width; j++)
            grid[i][j] = other.grid[i][j];
    }

    // Copy player
}


```

```

    if (other.player) {
        player = std::make_unique<Player>(*other.player);
    }

    // Copy enemies
    enemies.reserve(other.enemies.size());
    for (const auto& enemy : other.enemies) {
        enemies.push_back(std::make_unique<Enemy>(*enemy));
    }
}

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

    // Clean up current resources
    for (int i = 0; i < height; i++)
        delete[] grid[i];
    delete[] grid;

    player.reset();
    enemies.clear();

    // Copy new data
    width = other.width;
    height = other.height;

    grid = new Cell*[height];
    for (int i = 0; i < height; i++) {
        grid[i] = new Cell[width];
        for (int j = 0; j < width; j++)
            grid[i][j] = other.grid[i][j];
    }

    if (other.player) {
        player = std::make_unique<Player>(*other.player);
    }

    enemies.reserve(other.enemies.size());
    for (const auto& enemy : other.enemies) {
        enemies.push_back(std::make_unique<Enemy>(*enemy));
    }

    return *this;
}

Field::Field(Field&& other) noexcept
    : width(other.width), height(other.height), grid(other.grid),
      player(std::move(other.player)),
      enemies(std::move(other.enemies)) {
    other.grid = nullptr;
    other.width = 0;
    other.height = 0;
}

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

```

```

        // Clean up current resources
        for (int i = 0; i < height; i++)
            delete[] grid[i];
        delete[] grid;

        // Move new data
        width = other.width;
        height = other.height;
        grid = other.grid;
        player = std::move(other.player);
        enemies = std::move(other.enemies);

        // Reset other
        other.grid = nullptr;
        other.width = 0;
        other.height = 0;

        return *this;
    }

Field::~Field() {
    for (int i = 0; i < height; i++)
        delete[] grid[i];
    delete[] grid;
}

void Field::initializeWalls() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (std::rand() % 10 == 0)
                grid[i][j].setType(CellType::Wall);
        }
    }
}

void Field::setPlayer(std::unique_ptr<Player> p) {
    player = std::move(p);
    if (player) {

        grid[player->getY()][player->getX()].setType(CellType::Player);
    }
}

void Field::addEnemy(std::unique_ptr<Enemy> enemy) {
    if (enemy) {
        grid[enemy->getY()][enemy->getX()].setType(CellType::Enemy);
        enemies.push_back(std::move(enemy));
    }
}

void Field::removeEnemy(Enemy* enemy) {
    auto it = std::find_if(enemies.begin(), enemies.end(),
                           [enemy](const std::unique_ptr<Enemy>& e) { return e.get() == enemy; });

    if (it != enemies.end()) {
        grid[*it->getY()][(*it)->getX()].setType(CellType::Empty);
    }
}

```

```

        enemies.erase(it);
    }
}

bool Field::isValidPosition(int x, int y) const {
    return x >= 0 && x < width && y >= 0 && y < height;
}

bool Field::isWall(int x, int y) const {
    if (!isValidPosition(x, y)) return false;
    return grid[y][x].isWall();
}

bool Field::isEmpty(int x, int y) const {
    if (!isValidPosition(x, y)) return false;
    return grid[y][x].isEmpty();
}

bool Field::hasPlayerAt(int x, int y) const {
    return player && player->getX() == x && player->getY() == y;
}

bool Field::hasEnemyAt(int x, int y) const {
    return getEnemyAt(x, y) != nullptr;
}

Player* Field::getPlayer() const {
    return player.get();
}

Enemy* Field::getEnemyAt(int x, int y) const {
    for (const auto& enemy : enemies) {
        if (enemy->getX() == x && enemy->getY() == y) {
            return enemy.get();
        }
    }
    return nullptr;
}

const std::vector<std::unique_ptr<Enemy>>& Field::getEnemies() const
{
    return enemies;
}

void Field::clearCell(int x, int y) {
    if (isValidPosition(x, y)) {
        grid[y][x].setType(CellType::Empty);
    }
}

void Field::placeEntity(Entity* entity) {
    if (entity && isValidPosition(entity->getX(), entity->getY())) {
        CellType type = dynamic_cast<Player*>(entity) ?
CellType::Player : CellType::Enemy;
        grid[entity->getY()][entity->getX()].setType(type);
    }
}

```

```

int Field::getWidth() const {
    return width;
}

int Field::getHeight() const {
    return height;
}

void Field::draw() const {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            switch (grid[i][j].getType()) {
                case CellType::Empty: std::cout << "."; break;
                case CellType::Player: std::cout << "P"; break;
                case CellType::Enemy: std::cout << "E"; break;
                case CellType::Wall: std::cout << "#"; break;
            }
        }
        std::cout << "\n";
    }
}

```

Field.h

```

#pragma once
#include "Cell.h"
#include "Entity.h"
#include <memory>
#include <vector>
#include <iostream>

// Forward declarations
class Player;
class Enemy;

class Field {
private:
    int width, height;
    Cell** grid;
    std::unique_ptr<Player> player;
    std::vector<std::unique_ptr<Enemy>> enemies;

    void initializeWalls();

public:
    Field(int w, int h);

    // Copy constructor and assignment operator
    Field(const Field& other);
    Field& operator=(const Field& other);

    // Move constructor and assignment operator
    Field(Field&& other) noexcept;
    Field& operator=(Field&& other) noexcept;

    ~Field();
}

```

```

// Entity management
void setPlayer(std::unique_ptr<Player> p);
void addEnemy(std::unique_ptr<Enemy> enemy);
void removeEnemy(Enemy* enemy);

// Field queries
bool isValidPosition(int x, int y) const;
bool isWall(int x, int y) const;
bool isEmpty(int x, int y) const;
bool hasPlayerAt(int x, int y) const;
bool hasEnemyAt(int x, int y) const;

// Entity access
Player* getPlayer() const;
Enemy* getEnemyAt(int x, int y) const;
const std::vector<std::unique_ptr<Enemy>>& getEnemies() const;

// Field operations
void clearCell(int x, int y);
void placeEntity(Entity* entity);

// Getters
int getWidth() const;
int getHeight() const;

// Display
void draw() const;
};


```

Game.cpp

```

#include "Game.h"
#include <cstdlib>
#include <ctime>
#include <iostream>

Game::Game() : field(10, 10), running(true) {
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    std::srand(static_cast<unsigned int>(std::time(nullptr)));
}

    initializeGame();
}

void Game::initializeGame() {
    // Create and place player
    auto player = std::make_unique<Player>(1, 1);
    field.setPlayer(std::move(player));

    // Create and place enemies
    int enemyCount = 3;
    int width = field.getWidth();
    int height = field.getHeight();

    for (int i = 0; i < enemyCount; i++) {
        int x, y;

```

```

        do {
            x = std::rand() % width;
            y = std::rand() % height;
        } while (!field.isEmpty(x, y) ||
                  (x == field.getPlayer()->getX() && y ==
field.getPlayer()->getY()));

        field.addEnemy(std::make_unique<Enemy>(x, y));
    }
}

void Game::run() {
    char command;

        while (running && field.getPlayer() &&
field.getPlayer()->isAlive()) {
        system("cls");
        field.draw();
        showStats();

        std::cout << "Commands: W/A/S/D - movement, M - switch mode,
Q - quit\n";
        std::cout << "Your turn: ";
        std::cin >> command;

        int dx = 0, dy = 0;
        switch (std::tolower(command)) {
            case 'w': dy = -1; break;
            case 's': dy = 1; break;
            case 'a': dx = -1; break;
            case 'd': dx = 1; break;
            case 'm':
                if (field.getPlayer()) {
                    field.getPlayer()->switchCombatMode();
                }
                continue;
            case 'q': running = false; continue;
            default:
                std::cout << "Invalid command!\n";
                continue;
        }

        if ((dx != 0 || dy != 0) && field.getPlayer()) {
            field.getPlayer()->move(dx, dy, field);
        }

        if (checkWin()) break;

        processEnemiesTurn();

        if (checkLose()) break;
    }
}

void Game::processEnemiesTurn() {
    for (const auto& enemy : field.getEnemies()) {
        int dx = (std::rand() % 3) - 1; // -1, 0, 1
        int dy = (std::rand() % 3) - 1;
    }
}

```

```

        if (dx != 0 || dy != 0) {
            enemy->move(dx, dy, field);
        }

        if (!field.getPlayer() || !field.getPlayer()->isAlive()) {
            break;
        }
    }
}

bool Game::checkWin() const {
    if (field.getEnemies().empty()) {
        system("cls");
        field.draw();
        std::cout << "\nCongratulations! All enemies destroyed - you
won!\n";
        showStats();
        return true;
    }
    return false;
}

bool Game::checkLose() const {
    if (!field.getPlayer() || !field.getPlayer()->isAlive()) {
        system("cls");
        field.draw();
        std::cout << "\nGame over! You lost.\n";
        return true;
    }
    return false;
}

void Game::showStats() const {
    if (field.getPlayer()) {
        std::cout << "\nHealth: " << field.getPlayer()->getHP()
            << " | Damage: " << field.getPlayer()->getDamage()
            << " | Score: " << field.getPlayer()->getScore() <<
"\n";
    }
}

```

Game.h

```

#pragma once
#include <iostream>
#include <windows.h>
#include <cstdlib>
#include <ctime>
#include <memory>

#include "Field.h"
#include "Player.h"
#include "Enemy.h"

class Game {

```

```
private:
    Field field;
    bool running;

    void processPlayerTurn();
    void processEnemiesTurn();
    bool checkWin() const;
    bool checkLose() const;
    void showStats() const;
    void initializeGame();

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

    // Disable copy and move to prevent multiple game instances
    Game(const Game& other) = delete;
    Game& operator=(const Game& other) = delete;
    Game(Game&& other) = delete;
    Game& operator=(Game&& other) = delete;

    void run();
};
```