

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

ЛАБОРАТОРНАЯ РАБОТА
по дисциплине «Объектно-ориентированное программирование»
Тема: создание игры

Студент гр. 4341

Четвертная В.Л.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы

Разработка объектно-ориентированной пошаговой игровой системы с применением принципов ООП, управлением ресурсами через семантику перемещения и реализацией сложных игровых механик взаимодействия между объектами.

Задачи

1. Реализация класса Player

- Создать класс для хранения характеристик игрока (здоровье, урон, очки)
 - Реализовать механизм перемещения по игровому полю
 - Добавить систему смерти игрока при окончании жизней
 - Реализовать переключение между ближним и дальним боем

2. Разработка класса Enemy

- Создать класс с параметрами здоровья и урона
- Реализовать механизмы автоматического управления врагами
- Добавить механизм нанесения урона при столкновении с игроком
 - Организовать преследование игрока

3. Создание системы игрового поля (GameField)

- Разработать класс для прямоугольного/квадратного поля
- Создать отдельный класс для клеток поля
- Реализовать конструкторы копирования и перемещения
- Добавить операторы присваивания с глубоким копированием
- Реализация непроходимых клеток
- Добавить тип непроходимых клеток
- Организовать проверку при перемещении объектов

- Заполнить поле непроходимыми клетками при создании

4. Система типов боя для игрока

- Реализовать перечисление для типов атаки (ближний/дальний)
- Добавить механизм смены типа боя
- Учесть затрату хода на переключение

5. Класс вражеских зданий (EnemyBuilding)

- Создать класс для генерации врагов
- Реализовать периодическое создание врагов
- Настроить систему отсчета ходов до создания врага

6. Замедляющие клетки

- Добавить тип замедляющих клеток
- Реализовать механизм пропуска хода для игрока
- Организовать проверку при перемещении

Архитектурные требования

- Применить правильные модификаторы доступа
- Использовать enum для ограниченных значений
- Исключить глобальные переменные
- Оптимизировать перемещение объектов

Задание

1. Создать класс игрока, который должен хранить информацию об игроке (его жизни, урон, очки, и т.д.). Объект класса игрока должен перемещаться по карте. Если у игрока кончаются жизни, то происходит конец игры.

2. 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.

3. 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 проходить глубокая копия).

4. 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.

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

6. 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.

7. Реализовать замедляющие клетки на поле. Если игрок переходит на такую клетку, то он не может двигаться на следующий ход.

Примечания:

- Не забывайте для полей и методов определять модификаторы доступа
- Для обозначения переменной, которая принимает небольшое ограниченное количество значений, используйте enum
- Не используйте глобальные переменные
- При реализации перемещения, не должно быть лишнего копирования
- У поля не должно быть методов возвращающих указатель на поле в явном виде, так как это небезопасно
- Информация о координатах игрока не должна дублироваться у игрока и у поля

Выполнение работы

UML-диаграмма классов

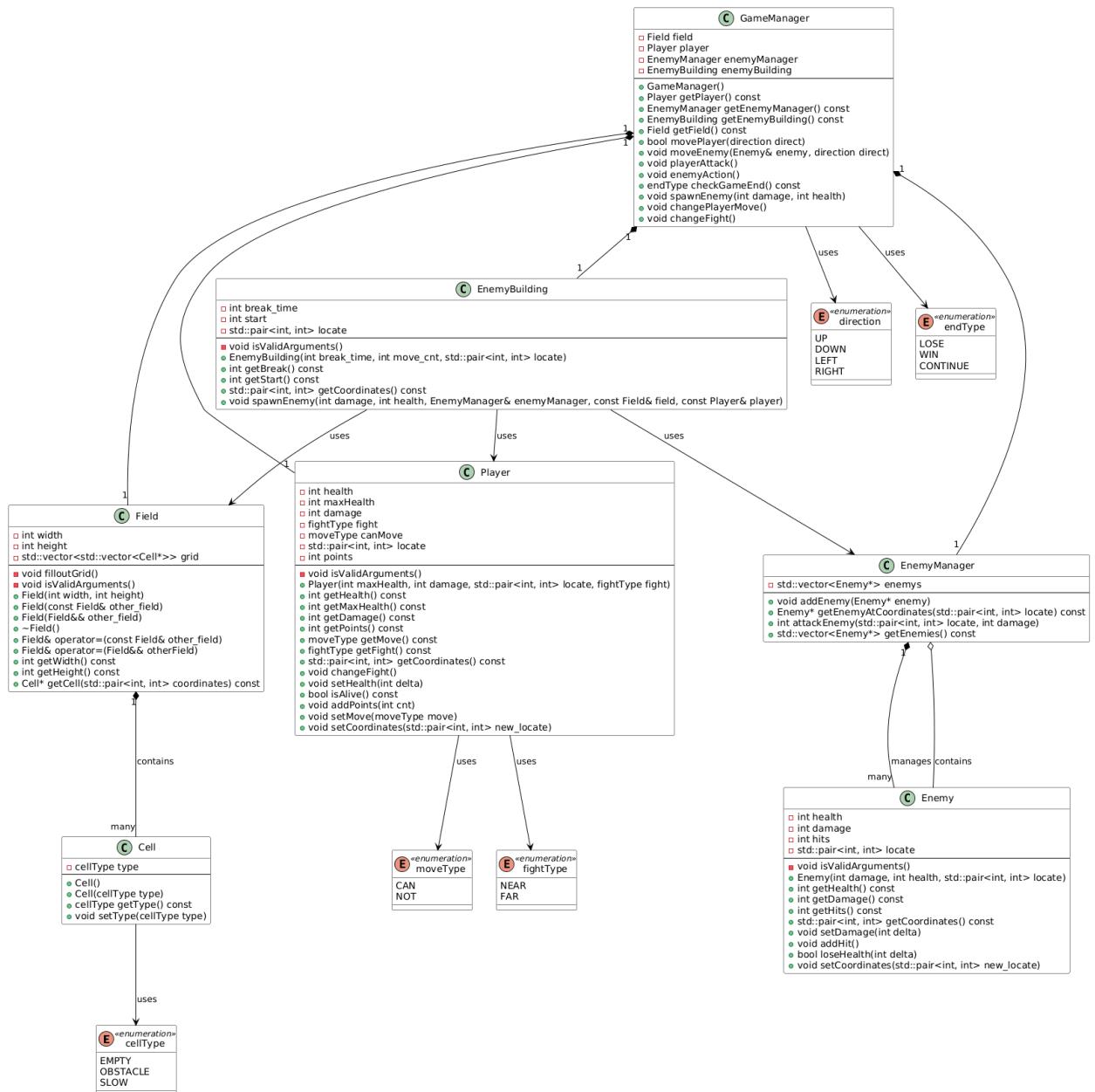


Рисунок 1. UML-диаграмма классов

Описание структуры кода

class Player — класс игрока.

Поля класса:

- *int health* - поле для хранения текущего значения здоровья;
- *int maxHealth* - поле для хранения максимального значения здоровья;
- *int damage* - поле для хранения урона;
- *fightType fight* - поле для хранения текущего типа атаки;
- *moveType canMove* - поле для хранения возможности перемещения;
- *std::pair <int, int> locate* - поле для хранения координат;
- *int points* - поле для хранения очков.

Методы класса:

void isValidArguments() — приватная функция проверки корректности аргументов при создании поля. В случае некорректности выбрасывает исключение.

Player(int maxHealth, int damage, std::pair <int, int> locate, fightType fight) — конструктор. Создаётся объект класса, максимальное и текущее значения здоровья устанавливаются равными *maxHealth*, урон задаётся равным *damage*, координаты задаются *std::pair <int, int> locate*, тип боя ближний *fight*, способность передвигаться задаётся возможной. В случае подачи отрицательного значения для здоровья или урона выбрасывается исключение. Вызывает функцию проверки корректности аргументов.

Принимаемые аргументы:

- *maxHealth: int* - максимальное значение здоровья;
- *damage: int* - урон.
- *std::pair <int, int> locate* - координаты игрока;
- *fight: fightType* - тип боя.

int getHealth() const — константная функция для получения текущего значения здоровья.

Возвращаемое значение:

- *health: int* - текущее значение здоровья.

int getMaxHealth() const — константная функция для получения максимального значения здоровья.

Возвращаемое значение:

- *maxHealth: int* - максимальное значение здоровья.

int getDamage() *const* — константная функция для получения значения урона.

Возвращаемое значение:

- *damage: int* - урон.

int getPoints() *const* — константная функция для получения количества очков.

Возвращаемое значение:

- *points: int* - очки.

moveType getMove() *const* — константная функция для получения информации о возможности перемещаться.

Возвращаемое значение:

- *canMove: moveType* - возможность перемещаться.

fightType getFight() *const* — константная функция для получения информации о типе боя.

Возвращаемое значение:

- *fight: fightType* - текущий тип боя.

std::pair <int, int> getCoordinates() *const* — константная функция для получения координат.

Возвращаемое значение:

- *locate: std::pair <int, int>* - координаты игрока.

void changeFight() — функция для изменения типа боя. Если текущий бой ближний, устанавливается дальний, и наоборот.

void setHealth(int delta) — функция для изменения здоровья. Изменяет здоровье на нужное число, максимальное значение здоровья - *maxHealth*.

Принимаемые аргументы:

- *delta: int* - на сколько необходимо изменить здоровье (для уменьшения здоровья подаётся отрицательное значение).

bool isAlive() const — функция для проверки жизней игрока.

Возвращаемое значение:

- *(health > 0): bool* - жив ли игрок.

void addPoints(int cnt) — функция для добавления очков.

Принимаемые аргументы:

- *cnt: int* - на сколько необходимо изменить очки (для уменьшения подаётся отрицательное значение).

void setMove(moveType move) — функция для изменения возможности перемещения.

Принимаемые аргументы:

- *move: moveType* - значение возможность перемещения.

void setCoordinates(std::pair <int, int> new_locate) — функция для изменения координат.

Принимаемые аргументы:

- *new_locate: std::pair <int, int>* - новые координаты.

class Enemey — класс врага.

Поля класса:

- *int health* - поле для хранения значения здоровья;
- *int damage* - поле для хранения урона;
- *int hits* - поле для хранения количества полученных ударов;
- *std::pair <int, int> locate* - поле для хранения координат.

Методы класса:

void isValidArguments() — приватная функция проверки корректности аргументов при создании врага. В случае некорректности выбрасывает исключение.

Enemy(int damage, int health, std::pair<int, int> locate) — конструктор. Создаётся объект класса врага с заданным уроном *damage*, здоровьем *health* и координатами *locate*, количество полученных ударов равно нулю. Вызывает функцию проверки корректности аргументов.

Принимаемые аргументы:

- *damage: int* - наносимый урон;
- *health: int* - здоровье врага;
- *locate: std::pair<int, int>* - координаты врага.

int getHealth() const — константная функция для получения текущего значения здоровья.

Возвращаемое значение:

- *health: int* - текущее значение здоровья.

int getDamage() const — константная функция для получения значения урона.

Возвращаемое значение:

- *damage: int* - урон.

int getHits() const — константная функция для получения количества полученных ударов.

Возвращаемое значение:

- *hits*: *int* - очки.

std::pair <int, int> getCoordinates() const — константная функция для получения координат.

Возвращаемое значение:

- *locate*: *std::pair <int, int>* - координаты врага.

void setDamage(int delta) — функция для изменения урона.

Принимаемые аргументы:

• *delta*: *int* - на сколько необходимо изменить урон (для уменьшения подаётся отрицательное значение).

void addHit() — функция для увеличения числа ударов. Увеличивает счётчик полученных ударов на 1.

bool loseHealth(int delta) — функция для изменения здоровья.

Уменьшает здоровье на нужное число.

Принимаемые аргументы:

- *delta*: *int* - на сколько необходимо уменьшить здоровье.

Возвращаемое значение:

- (*health > 0*): *bool* - жив ли враг.

void setCoordinates(std::pair <int, int> new_locate) — функция для изменения координат.

Принимаемые аргументы:

- *new_locate*: *std::pair <int, int>* - новые координаты.

class EnemyManager — класс для управления врагами.

Поля класса:

- *std::vector <Enemy*> enemys* - вектор для хранения врагов.

Методы класса:

void addEnemy(Enemy enemy)* — функция добавления врага.

Принимаемые аргументы:

- *enemy*: *Enemy** - запоминаемый враг;

void addEnemy(Enemy enemy)* — функция добавления врага.

Принимаемые аргументы:

- *enemy*: *Enemy** - запоминаемый враг;

Enemy getEnemyAtCoordinates(std::pair <int, int> locate) const* — функция поиска врага по заданным координатам. В случае отсутствия врага на клете возвращается *nullptr*.

Принимаемые аргументы:

- *locate*: *std::pair <int, int>* - координаты.

Возвращаемое значение:

- *enemys[i]*: *Enemy** - координаты.

int attackEnemy(std::pair <int, int> locate, int damage) — функция атаки врага. Возвращает количество полученных за атаку очков. Если враг на заданной координате есть, его здоровье уменьшается на урон и увеличивается на 1 количество ударов. После удара прибавляются 5 очков, если враг умер, добавляются дополнительные очки в количестве 20/(кол-во полученных врагом ударов). Умерший враг удаляется.

Принимаемые аргументы:

- *locate*: *std::pair <int, int>* - координаты;
- *damage*: *int* - получаемый урон.

Возвращаемое значение:

- *point*: *int* - очки.

class EnemyBuilding — класс вражеского здания.

Поля класса:

- *int break_time* - количество ходов между созданием новых врагов;
- *int start* - ход, на котором создан объект;
- *std::pair <int, int> locate* - координаты здания.

Методы класса:

void isValidArguments() — приватная функция проверки корректности аргументов при создании вражеского здания. В случае некорректности выбрасывает исключение.

EnemyBuilding(int break_time, int move_cnt, std::pair <int, int> locate) — конструктор. Создаётся объект класса вражеского здания с заданным временем перерыва *break_time*, стартовым ходом *move_cnt* и координатами *locate*. Вызывает функцию проверки корректности аргументов.

Принимаемые аргументы:

- *break_time: int* - время перерыва;
- *move_cnt: int* - ход создания;
- *locate: std::pair<int, int>* - координаты.

int getBreak() const — константная функция для получения количества ходов между созданием врагов.

Возвращаемое значение:

- *break_time: int* - время перерыва.

int getStart() const — константная функция для получения хода создания здания.

Возвращаемое значение:

- *start: int* - ход создания.

std::pair <int, int> getCoordinates() const — константная функция для получения координат.

Возвращаемое значение:

- *locate: std::pair <int, int>* - координаты.

void spawnEnemy(int damage, int health, EnemyManager& enemyManager, const Field& field, const Player& player) — функция создания нового врага. Находится свободная соседняя со зданием клетка, на ней порождается враг с заданными значениями урона и здоровья.

Принимаемые аргументы:

- *damage: int* - урон;
- *health: int* - здоровье;
- *enemyManager: EnemyManager&* - менеджер врагов;
- *field: const Field&* - поле (для проверки корректности координат);
- *player: const Player&* - игрок (для проверки занятости клетки);

class Cell — класс клетки

Поля класса:

- *cellType type* - поле для хранения типа клетки.

Методы класса:

Cell() — конструктор. Создаётся объект класса, тип задаётся *cellType::EMPTY*.

Cell(cellType type) — конструктор. Создаётся объект класса, тип задаётся *type*.

Принимаемые аргументы:

- *type: cellType* - тип клетки.

int getHealth() const — константная функция для получения текущего значения здоровья.

Возвращаемое значение:

- *health: int* - текущее значение здоровья.

cellType getType() const — константная функция для получения типа клетки.

Возвращаемое значение:

- *type: cellType* - тип клетки.

void setType(cellType type) — функция для изменения типа клетки.

Принимаемые аргументы:

- *type: cellType* - необходимый тип клетки.

class Field — класс поля.

Поля класса:

- *int width* - поле для хранения ширины поля;
- *int height* - поле для хранения высоты поля;
- *std::vector <std::vector <Cell*>> grid* - вектор для хранения клеток.

Методы класса:

void filloutGrid() — приватная функция для заполнения поля клетками различных типов. Заполнение происходит следующим образом: тип клеток, координаты которых соответствуют условию $((i+j+i*j)\%5 == 0)$ выбирается случайным образом из 3 возможных вариантов (пустая, замедляющая или непроходимая), остальные клетки становятся пустыми.

void isValidArguments() — приватная функция проверки корректности аргументов при создании поля. В случае некорректности выбрасывает исключение.

Field(int width, int height) — конструктор. Создаётся объект класса, ширина и высота задаются равными соответственно *width* и *height*, вызывается функция проверки корректности аргументов, затем вызывается функция *filloutGrid()* для заполнения поля.

Принимаемые аргументы:

- *width: int* - ширина;
- *height: int* - высота.

Field(const Field& other_field) — конструктор копирования. Через глубокое копирование создаётся новый объект класса идентичный переданному.

Принимаемые аргументы:

- *other_field: const Field&* - копируемый объект класса.

Field(Field&& other_field) — конструктор перемещения. Создаётся новый объект класса со значениями полей равными переданному, происходит перемещение вектора клеток.

Принимаемые аргументы:

- *other_field: Field&&* - перемещаемый объект класса.

~Field() — деструктор. Очищает память, выделенную для хранения клеток.

Field& operator=(const Field& other_field) — оператор копирования. Через оператор копирования создаётся временный объект, после чего происходит обмен полей временного объекта с полями текущего объекта.

Принимаемые аргументы:

- *other_field: const Field&* - копируемый объект класса.

Возвращаемое значение:

- **this: Field&* - ссылка на текущий объект.

Field(Field&& other_field) — оператор перемещения. Через move-семантику происходит перемещение полей заданного объекта.

Принимаемые аргументы:

- *other_field: Field&&* - перемещаемый объект класса.

Возвращаемое значение:

- **this: Field&* - ссылка на текущий объект.

int getWidth() const — константная функция получения ширины.

Возвращаемое значение:

- *width: int* - ширина поля.

int getHeight() const — константная функция получения ширины.

Возвращаемое значение:

- *height: int* - высота поля.

int getWidth() const — константная функция получения ширины.

Возвращаемое значение:

- *width: int* - ширина поля.

Cell getCell(std::pair <int, int> coordinates) const* — константная функция получения клетки по заданным координатам. В случае подачи некорректных координат выбрасывается исключение.

Принимаемые аргументы:

- *coordinates: std::pair <int, int>* - координаты клетки.

Возвращаемое значение:

- *grid[coordinates.second][coordinates.first]: Cell** - искомая клетка.

class GameManager — класс для осуществление взаимодействия между объектами игры из разных классов.

Поля класса:

- *Field field* - поле;
- *Player player* - игрок;
- *EnemyManager enemyManager* - менеджер врагов;
- *EnemyBuilding enemyBuilding* - вражеское здание.

Методы класса:

GameManager() — конструктор. Создаются поле размера 25*20, игрок со здоровьем 100, уроном 10 и ближним боем на клетке (0,0), вражеское здание на противоположной от игрока клетке, создающее врагов через каждые 5 ходов, и враг в центре поля.

Player getPlayer() const — константная функция для получения копии игрока.

EnemyManager getEnemyManager() const — константная функция для получения копии менеджера врагов.

EnemyBuilding getEnemyBuilding() const — константная функция для получения копии вражеского здания.

Field getField() const — константная функция для получения копии поля.

bool movePlayer(direction direct) — функция перемещения игрока. Выполняется перемещение игрока на одну клетку в заданном направлении, если игрок может перемещаться и необходимая клетка свободна. Если на клетке, в которую должен переместиться игрок, стоит враг, происходит атака врага, игрок не перемещается.

Принимаемые аргументы:

- *direct: direction* - направление перемещения.

Возвращаемое значение:

- *true/false: bool* - было ли совершено действие (перемещение или атака).

void moveEnemy(Enemy& enemy, direction direct) — функция перемещения заданного врага. Выполняется перемещение врага на одну клетку в заданном направлении, если необходимая клетка

свободна. Если на клетке, в которую должен переместиться враг, стоит игрок, происходит атака игрока, враг не перемещается.

Принимаемые аргументы:

- *enemy*: *Enemy&* - перемещаемый враг;
- *direct*: *direction* - направление перемещения.

void playerAttack() — функция атаки игрока. В зависимости от текущего типа боя атакуются враги находящиеся в радиусе $\text{std}::\min(\text{field.getHeight}(), \text{field.getWidth}())/10$ с нанесением полного урона для ближнего боя, или в радиусе $\text{std}::\min(\text{field.getHeight}(), \text{field.getWidth}())/5$ с нанесением половинного урона для дальнего боя.

void enemyAction() — функция движения врагов. Все враги сдвигаются на одну клетку в сторону игрока, либо атакуют игрока.

endType checkGameEnd() const — константная функция проверки условий завершения игры.

Возвращаемое значение:

- *endType::LOSE/endType::WIN/endType::CONTINUE*: *endType* — текущее условие окончания игры: проигрыш, если игрок мёртв, выигрыш — если очки ≥ 100 , либо продолжение игры.

Вывод

В ходе написания лабораторной работы разработана объектно-ориентированной пошаговой игровой системы с применением принципов ООП, управлением ресурсами через семантику перемещения и реализацией игровых механик взаимодействия между объектами. Реализованы классы игрока, врага, вражеского здания и игрового поля с необходимыми методами.

Игрок и враг могут перемещаться по полю, атаковать друг друга, нанося урон. Вражеское здание раз в заданное число ходов создаёт около себя нового врага. Для игрового поля реализованы конструкторы и операторы копирования и перемещения.

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ

Для проверки работоспособности классов и методов написано unit-тестирование с использованием gtests.

1. Проверка класса *Cell*

```
TEST(CellTest, BasicFunctionality) {
    Cell cell1;
    EXPECT_EQ(cell1.getType(), cellType::EMPTY);

    Cell cell2(cellType::OBSTACLE);
    EXPECT_EQ(cell2.getType(), cellType::OBSTACLE);

    cell1.setType(cellType::SLOW);
    EXPECT_EQ(cell1.getType(), cellType::SLOW);
}
```

2. Проверка класса *Enemy*

```
TEST(EnemyTest, ConstructorAndBasicMethods) {
    Enemy enemy(5, 30, make_pair(10, 10));

    EXPECT_EQ(enemy.getDamage(), 5);
    EXPECT_EQ(enemy.getHealth(), 30);
    EXPECT_EQ(enemy.getCoordinates(), make_pair(10, 10));

    enemy.setDamage(3);
    EXPECT_EQ(enemy.getDamage(), 8);

    enemy.setCoordinates(make_pair(5, 5));
    EXPECT_EQ(enemy.getCoordinates(), make_pair(5, 5));

    bool alive = enemy.loseHealth(10);
    EXPECT_TRUE(alive);
    EXPECT_EQ(enemy.getHealth(), 20);
}

TEST(EnemyTest, InvalidArguments) {
    EXPECT_THROW(Enemy(0, 30, make_pair(0, 0)), invalid_argument);
    EXPECT_THROW(Enemy(5, 0, make_pair(0, 0)), invalid_argument);
}
```

3. Проверка класса *Field*

```
TEST(FieldTest, CopyConstructor) {
    Field original(15, 15);
    original.getCell(make_pair(5, 5))->setType(cellType::OBSTACLE);
    original.getCell(make_pair(10, 10))->setType(cellType::SLOW);
```

```

    Field copy(original);

    EXPECT_EQ(copy.getWidth(), original.getWidth());
    EXPECT_EQ(copy.getHeight(), original.getHeight());

    Cell* origCell = original.getCell(make_pair(5, 5));
    Cell* copyCell = copy.getCell(make_pair(5, 5));
    EXPECT_NE(origCell, copyCell);
    EXPECT_EQ(copyCell->getType(), cellType::OBSTACLE);

    copyCell->setType(cellType::EMPTY);
    EXPECT_EQ(origCell->getType(), cellType::OBSTACLE);
}

TEST(FieldTest, MoveConstructor) {
    Field original(15, 15);
    original.getCell(make_pair(3, 3))->setType(cellType::SLOW);

    int originalWidth = original.getWidth();
    int originalHeight = original.getHeight();
    cellType savedType = original.getCell(make_pair(3, 3))->getType();

    Field moved(move(original));

    EXPECT_EQ(moved.getWidth(), originalWidth);
    EXPECT_EQ(moved.getHeight(), originalHeight);
    EXPECT_EQ(moved.getCell(make_pair(3, 3))->getType(), savedType);

    EXPECT_EQ(original.getWidth(), 0);
    EXPECT_EQ(original.getHeight(), 0);
    EXPECT_THROW(original.getCell(make_pair(0, 0)), invalid_argument);
}

TEST(FieldTest, CopyAssignment) {
    Field source(15, 15);
    Field destination(10, 10);

    source.getCell(make_pair(7, 7))->setType(cellType::OBSTACLE);

    destination = source;

    EXPECT_EQ(destination.getWidth(), 15);
    EXPECT_EQ(destination.getHeight(), 15);
    EXPECT_EQ(destination.getCell(make_pair(7, 7))->getType(),
cellType::OBSTACLE);

    Cell* sourceCell = source.getCell(make_pair(7, 7));
    Cell* destCell = destination.getCell(make_pair(7, 7));
    EXPECT_NE(sourceCell, destCell);

    destCell->setType(cellType::EMPTY);
}

```

```

        EXPECT_EQ(sourceCell->getType(), cellType::OBSTACLE);
    }

TEST(FieldTest, MoveAssignment) {
    Field source(15, 15);
    Field destination(20, 20);

    source.getCell(make_pair(4, 4))->setType(cellType::SLOW);
    int sourceWidth = source.getWidth();
    cellType savedType = source.getCell(make_pair(4, 4))->getType();

    destination = move(source);

    EXPECT_EQ(destination.getWidth(), sourceWidth);
    EXPECT_EQ(destination.getCell(make_pair(4, 4))->getType(),
    savedType);

    EXPECT_EQ(source.getWidth(), 0);
    EXPECT_THROW(source.getCell(make_pair(4, 4)), invalid_argument);
}

TEST(FieldTest, CellAccess) {
    Field field(15, 15);

    Cell* cell = field.getCell(make_pair(0, 0));
    EXPECT_NE(cell, nullptr);

    EXPECT_THROW(field.getCell(make_pair(-1, 0)), invalid_argument);
    EXPECT_THROW(field.getCell(make_pair(15, 0)), invalid_argument);
}

```

4. Проверка класса *EnemyManager*

```

TEST(EnemyManagerTest, BasicOperations) {
    EnemyManager manager;

    Enemy* enemy = new Enemy(5, 30, make_pair(5, 5));
    manager.addEnemy(enemy);

    EXPECT_FALSE(manager.getEnemies().empty());

    Enemy* found = manager.getEnemyAtCoordinates(make_pair(5, 5));
    EXPECT_NE(found, nullptr);
    EXPECT_EQ(found->getHealth(), 30);

    int points = manager.attackEnemy(make_pair(5, 5), 10);
    EXPECT_GT(points, 0);
}

```

5. Проверка класса *EnemyBuilding*

```

TEST(EnemyBuildingTest, BasicFunctionality) {

```

```

    EnemyBuilding building(5, 0, make_pair(10, 10));

    EXPECT_EQ(building.getBreak(), 5);
    EXPECT_EQ(building.getCoordinates(), make_pair(10, 10));
}

```

6. Проверка класса *GameManager*

```

TEST(GameManagerTest, PlayerMovement) {
    GameManager gm;

    bool moved = gm.movePlayer(direction::RIGHT);
    EXPECT_TRUE(moved);

    Player player = gm.getPlayer();
    EXPECT_EQ(player.getCoordinates(), make_pair(1, 0));
}

```

7. Проверка взаимодействия классов

```

TEST(IntegrationTest, PlayerEnemyCombat) {
    Field field(15, 15);
    Player player(100, 15, make_pair(5, 5), fightType::NEAR);
    EnemyManager enemyManager;

    Enemy* enemy = new Enemy(5, 30, make_pair(5, 5));
    enemyManager.addEnemy(enemy);

    int points = enemyManager.attackEnemy(make_pair(5, 5),
player.getDamage());
    EXPECT_GT(points, 0);
    EXPECT_EQ(enemy->getHealth(), 15);
    EXPECT_EQ(enemy->getHits(), 1);
}

TEST(IntegrationTest, EnemyBuildingSpawning) {
    Field field(15, 15);
    Player player(100, 10, make_pair(0, 0), fightType::NEAR);
    EnemyManager enemyManager;
    EnemyBuilding building(5, 0, make_pair(14, 14));

    building.spawnEnemy(5, 30, enemyManager, field, player);

    EXPECT_FALSE(enemyManager.getEnemies().empty());

    Enemy* spawned = enemyManager.getEnemyAtCoordinates(make_pair(13,
14));
    if (!spawned) spawned =
enemyManager.getEnemyAtCoordinates(make_pair(14, 13));
    if (!spawned) spawned =
enemyManager.getEnemyAtCoordinates(make_pair(13, 13));

    EXPECT_NE(spawned, nullptr);
}

```

}