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.

Задание

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

2. 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.

3. 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.

4. 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.

5. Реализовать интерфейс заклинания ловушки. Заклинание размещает на поле ловушку, если враг наступает на клетку с ловушкой, то ему наносится урон, и ловушка пропадает.

6. Создать класс вражеской башни. Вражеская башня размещается на поле, и если в радиусе ее атаки появляется игрок, то применяет ослабленную версию заклинания прямого урона. Не может применять заклинание несколько ходов подряд.

7. Реализовать интерфейс заклинания призыва. Заклинание создает союзника рядом с игроком, который перемещается самостоятельно.

8. Реализовать интерфейс заклинание улучшения. Заклинание улучшает следующее используемое заклинание:

- Заклинание прямого урона - увеличивает радиус применения
- Заклинание урона по площади - увеличивает площадь
- Заклинание ловушки - увеличивает урон
- Заклинание призыва - призывает больше союзников
- Заклинание улучшение - накапливает усиление, то есть при применении следующего заклинания отличного от улучшения, все улучшения применяются сразу

  Примечания:

- Интерфейс заклинания должен быть унифицирован, чтобы их можно было единообразно использовать через интерфейс. Не должно быть методов в интерфейсе, которые не используются каким-то классом наследником.
- Избегайте явных проверок на тип данных.

Выполнение работы

UML-диграмма классов

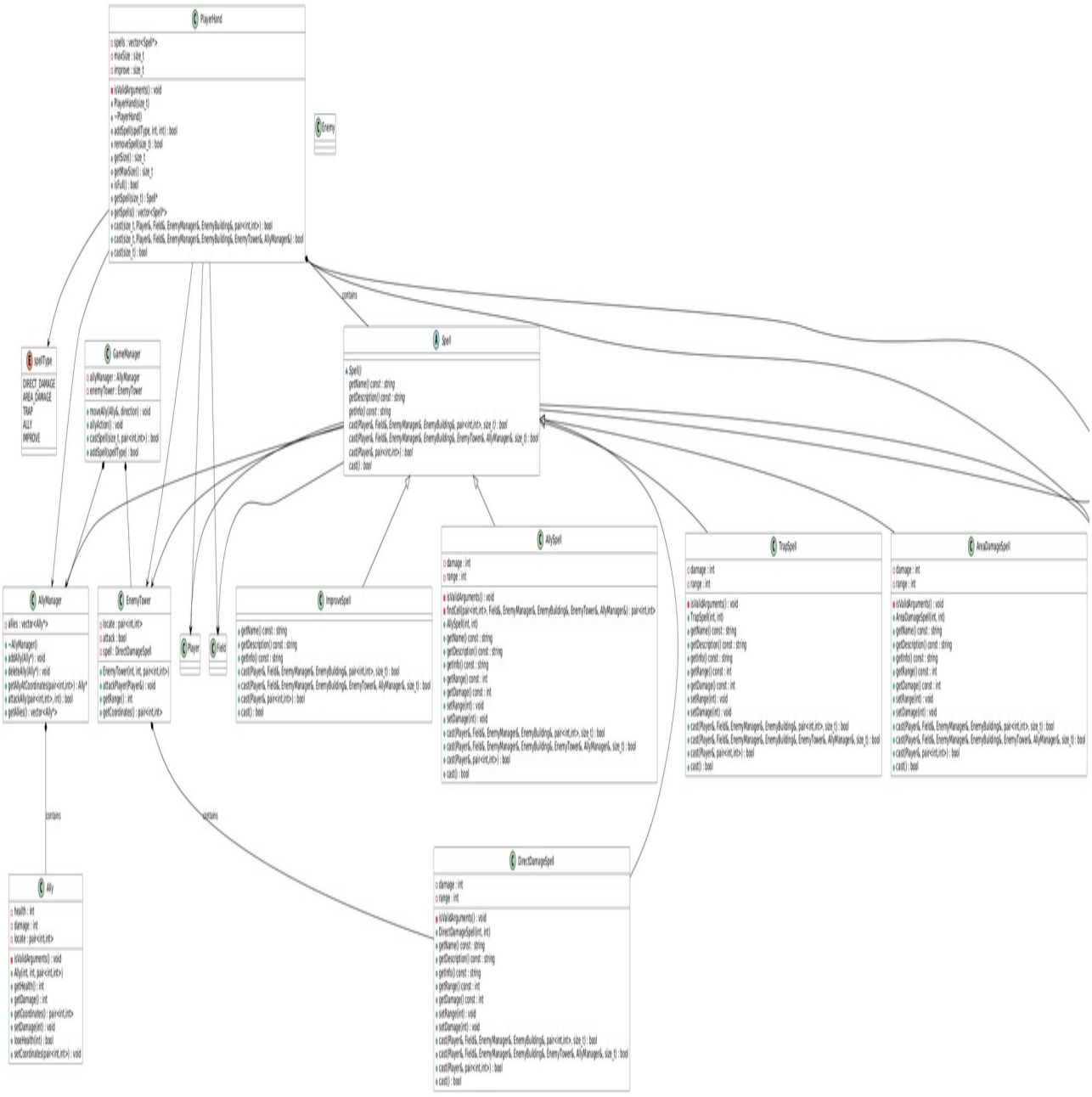

Рисунок 1. UML-диаграмма классов

Описание структуры кода

class Spell — интерфейс заклинания.

Методы класса:

virtual ~Spell() = default — виртуальный деструктор по умолчанию.

virtual std::string getName() const = 0 — виртуальный метод получения названия заклинания, обязательно должен быть переопределён.

virtual std::string getDescription() const = 0 — виртуальный метод получения описания заклинания, обязательно должен быть переопределён.

virtual bool cast(Player& player, Field& field, EnemyManager& enemyManager EnemyBuilding& enemyBuilding, std::pair <int, int> target, size_t improve) = 0 — виртуальный метод применения заклинания, возвращает успешность применения, обязательно должен быть переопределён.

bool cast(Player& player, Field& field, EnemyManager& enemyManager EnemyBuilding& enemyBuilding, EnemyTower& enemyTower, AllyManager& allyManager, size_t improve) = 0 — виртуальный метод применения заклинания, возвращает успешность применения, обязательно должен быть переопределён.

virtual bool cast(Player& target, std::pair <int, int> locate) = 0 — виртуальный метод применения заклинания, возвращает успешность применения, обязательно должен быть переопределён.

virtual bool cast() = 0 — виртуальный метод применения заклинания, возвращает успешность применения, обязательно должен быть переопределён.

class AreaDamageSpell — класс заклинания урона по площади. Наследуется от интерфейса заклинания.

Поля класса:

- *int damage* - поле для хранения значения урона;
- *int range* - поле для хранения значения дальности применения.

Методы класса:

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

AreaDamageSpell(int damage, int range) — конструктор. Создаётся объект класса поля *damage* и *range* задаются равными соответствующим аргументам. Вызывается функция проверки корректности аргументов.

Принимаемые аргументы:

- *damage: int* - значение урона;
- *range: int* - значение дальности применения.

std::string getName() const override — определение метода получения имени заклинания.

Возвращаемое значение:

- *std::string* - название заклинания.

std::string getDescription() const override — определение метода получения описания заклинания.

Возвращаемое значение:

- *std::string* - описание заклинания.

bool cast(Player& player, Field& field, EnemyManager& enemyManager, EnemyBuilding& enemyBuilding, std::pair <int, int> target) override — определение метода применения заклинания. Если выбранная цель находится вне дальности применения заклинания, возвращается *false*, иначе выполняется атака по полю 2*2 (выбранная клетка становится верхним левым углом), все находящиеся в поле сущности получают урон.

Принимаемые аргументы:

- *player*: *Player&* - игрок;
- *field*: *Field&* - поле;
- *enemyManager*: *EnemyManager&* - менеджер врагов;
- *enemyBuilding*: *EnemyBuilding&* - вражеское здание;
- *target*: *std::pair <int, int>* - выбранная координата применения.

Возвращаемое значение:

- *true/false*: *bool* - было ли применено заклинание.

bool cast(Player& target, std::pair <int, int> locate) override — определение метода применения заклинания. Для данного заклинания применение этого вызова невозможно.

Принимаемые аргументы:

- *target*: *Player&* - игрок;
- *locate*: *std::pair <int, int>* - координаты применяющей сущности.

Возвращаемое значение:

- *false: bool* - данное заклинание в такими аргументами применять нельзя.

class DirectDamageSpell — класс заклинания прямого урона. Наследуется от интерфейса заклинания.

Поля класса:

- *int damage* - поле для хранения значения урона;
- *int range* - поле для хранения значения дальности применения.

Методы класса:

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

DirectDamageSpell(int damage, int range) — конструктор. Создаётся объект класса поля *damage* и *range* задаются равными соответствующим аргументам. Вызывается функция проверки корректности аргументов.

Принимаемые аргументы:

- *damage: int* - значение урона;
- *range: int* - значение дальности применения.

std::string getName() const override — определение метода получения имени заклинания.

Возвращаемое значение:

- *std::string* - название заклинания.

std::string getDescription() const override — определение метода получения описания заклинания.

Возвращаемое значение:

- *std::string* - описание заклинания.

bool cast(Player& player, Field& field, EnemyManager& enemyManager, EnemyBuilding& enemyBuilding, std::pair <int, int> target) override — определение метода применения заклинания. Если на выбранная координата находится в области применения и на ней находится сущность, наносится урон. Иначе возвращается *false* - заклинание не применяется.

Принимаемые аргументы:

- *player*: *Player&* - игрок;
- *field*: *Field&* - поле;
- *enemyManager*: *EnemyManager&* - менеджер врагов;
- *enemyBuilding*: *EnemyBuilding&* - вражеское здание;
- *target*: *std::pair <int, int>* - выбранная координата применения.

Возвращаемое значение:

- *true/false*: *bool* - было ли применено заклинание.

bool cast(Player& target, std::pair <int, int> locate) override — определение метода применения заклинания. Если игрок находится в области применения заклинания, ему наносится урон. Иначе заклинание на выполняется.

Принимаемые аргументы:

- *target*: *Player&* - игрок;
- *locate*: *std::pair <int, int>* - координаты применяющей сущности.

Возвращаемое значение:

- *true/false*: *bool* - было ли применено заклинание.

class TrapSpell — класс заклинания ловушки. Наследуется от интерфейса заклинания.

Поля класса:

- *int damage* - поле для хранения значения урона;
- *int range* - поле для хранения значения дальности применения.

Методы класса:

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

TrapSpell(int damage, int range) — конструктор. Создаётся объект класса поля *damage* и *range* задаются равными соответствующим аргументам. Вызывается функция проверки корректности аргументов.

Принимаемые аргументы:

- *damage: int* - значение урона;
- *range: int* - значение дальности применения.

std::string getName() const override — определение метода получения имени заклинания.

Возвращаемое значение:

- *std::string* - название заклинания.

std::string getDescription() const override — определение метода получения описания заклинания.

Возвращаемое значение:

- *std::string* - описание заклинания.

bool cast(Player& player, Field& field, EnemyManager& enemyManager, EnemyBuilding& enemyBuilding, std::pair <int, int> target) override — определение метода применения заклинания. Если выбранная цель находится в области применения заклинания и выбранная клетке пуста, на клетку ставится ловушка. При попадании на ловушку враг получает урон, после чего ловушка пропадает.

Принимаемые аргументы:

- *player*: *Player&* - игрок;
- *field*: *Field&* - поле;
- *enemyManager*: *EnemyManager&* - менеджер врагов;
- *enemyBuilding*: *EnemyBuilding&* - вражеское здание;
- *target*: *std::pair <int, int>* - выбранная координата применения.

Возвращаемое значение:

- *true/false*: *bool* - было ли применено заклинание.

bool cast(Player& target, std::pair <int, int> locate) override — определение метода применения заклинания. Для данного заклинания применение этого вызова невозможно.

Принимаемые аргументы:

- *target*: *Player&* - игрок;
- *locate*: *std::pair <int, int>* - координаты применяющей сущности.

Возвращаемое значение:

- *false: bool* - данное заклинание в такими аргументами применять нельзя.

class PlayerHand — класс руки с заклинаниями.

Поля класса:

- *std::vector <Spell*> spells* - вектор для хранения заклинаний;
- *size_t maxSize* - поле для хранения максимального размера руки;
- *size_t improve* - поле для хранения количества накопленных улучшений.

Методы класса:

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

PlayerHand(size_t maxSize) — конструктор. Создаётся объект класса поле *maxSize* задаётся равным аргументу. Вызывается функция проверки корректности аргументов. При создании в руку добавляется одно случайное заклинание, количество улучшений устанавливается равным нулю.

Принимаемые аргументы:

- *maxSize: size_t* - максимальный размер руки.

bool addSpell(spellType spell, int damage, int range) — добавление заклинания. Если есть место создаётся и добавляется в руку новое заклинание нужного типа с заданными характеристиками.

Принимаемые аргументы:

- *spell*: *spellType* - тип заклинания;

- *damage: int* - значение урона;
- *range: int* - значение дальности применения заклинания.

Возвращаемое значение:

- *true/false*: *bool* - добавлено ли новое заклинание в руку.

bool removeSpell(size_t index) — удаление заклинания по индексу.

Принимаемые аргументы:

- *index*: *size_t* - индекс удаляемого заклинания.

Возвращаемое значение:

- *true/false*: *bool* - удалено ли заклинание.

void clear() — очистка руки.

size_t getSize() const — получение текущего размера руки.

Возвращаемое значение:

- *size_t* - количество заклинаний в руке на данным момент.

size_t getMaxSize() const — получение максимального размера руки.

Возвращаемое значение:

- *size_t* - максимальное количество заклинаний в руке.

Spell getSpell(size_t index) const* — получение заклинания по индексу.

Возвращаемое значение:

- *Spell** - искомое заклинание (при несуществующем индексе *nullptr*).

bool cast(size_t spell_id, Player& player, Field& field, EnemyManager& enemyManager, EnemyBuilding& enemyBuilding, std::pair <int, int> target) — применение заклинания. Если подан существующий индекс, вызывается соответствующее заклинание с передачей нужных аргументов. В случае выполнения заклинание количество улучшений сбрасывается в ноль.

Принимаемые аргументы:

- *spell_id*: *size_t* - индекс заклинания;

- *player: Player&* - игрок;
- *field: Field&* - поле;
- *enemyManager: EnemyManager&* - менеджер врагов;
- *enemyBuilding: EnemyBuilding&* - вражеское здание;
- *target: std::pair <int, int>* - координаты цели.

Возвращаемое значение:

- *true/false*: *bool* - выполнено ли заклинание.

bool cast(size_t spell_id) — применение заклинания. Если подан существующий индекс, вызывается соответствующее заклинание с передачей нужных аргументов. В случае выполнения заклинание количество улучшений увеличивается на 1, так как выполнение в данными аргументами используется в заклинании улучшения.

Принимаемые аргументы:

- *spell_id*: *size_t* - индекс заклинания;
- *player: Player&* - игрок.

Возвращаемое значение:

- *true/false*: *bool* - выполнено ли заклинание.

class Ally — класс союзника.

Поля класса:

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

Методы класса:

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

Ally(int damage, int health, std::pair<int, int> locate) — конструктор. Создаётся объект класса союзника с заданным уроном *damage*, здоровьем *health* и координатами *locate*. Вызывает функцию проверки корректности аргументов.

Принимаемые аргументы:

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

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.

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

Возвращаемое значение:

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

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

Принимаемые аргументы:

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

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 союзник.

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

Принимаемые аргументы:

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

class AllyManager — класс для управления союзниками.

Поля класса:

- *std::vector <Ally*> allies* - вектор для хранения союзников.

Методы класса:

void addAlly(Ally ally)* — функция добавления союзника.

Принимаемые аргументы:

- *ally*: *Ally** - добавляемый союзник.

void deleteAlly(Ally ally)* — функция удаления союзника.

Принимаемые аргументы:

- *ally*: *Ally** - удаляемый союзник.

Ally getAllyAtCoordinates(std::pair <int, int> locate) const* — функция поиска союзника по заданным координатам. В случае отсутствия союзника на клетке возвращается *nullptr*.

Принимаемые аргументы:

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

Возвращаемое значение:

- *allies[i]*: *Ally** - найденный союзник.

bool attackAlly(std::pair <int, int> locate, int damage) — функция атаки союзника. Возвращает был ли атакован союзник. Если союзник на заданной координате есть, его здоровье уменьшается на урон. Умерший союзник удаляется.

Принимаемые аргументы:

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

Возвращаемое значение:

- *true/false: bool* - был ли атакован союзник.

class AllySpell — класс заклинания призыва союзника. Наследуется от интерфейса заклинания.

Поля класса:

- *int damage* - поле для хранения значения урона;
- *int range* - поле для хранения значения дальности применения.

Методы класса:

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

AllySpell(int damage, int range) — конструктор. Создаётся объект класса поля *damage* и *range* задаются равными соответствующим аргументам. Вызывается функция проверки корректности аргументов.

Принимаемые аргументы:

- *damage: int* - значение урона;
- *range: int* - значение дальности применения.

std::string getName() const override — определение метода получения имени заклинания.

Возвращаемое значение:

- *std::string* - название заклинания.

std::string getDescription() const override — определение метода получения описания заклинания.

Возвращаемое значение:

- *std::string* - описание заклинания.

bool cast(Player& player, Field& field, EnemyManager& enemyManager, EnemyBuilding& enemyBuilding, std::pair <int, int> target, size_t improve) override — определение метода применения заклинания. Для данного заклинания применение этого вызова невозможно.

Принимаемые аргументы:

- *player*: *Player&* - игрок;
- *field*: *Field&* - поле;
- *enemyManager*: *EnemyManager&* - менеджер врагов;
- *enemyBuilding*: *EnemyBuilding&* - вражеское здание;
- *target*: *std::pair <int, int>* - выбранная координата применения.

Возвращаемое значение:

- *false*: *bool* - данное заклинание с такими аргументами применять нельзя.

bool cast(Player& player, Field& field, EnemyManager& enemyManager, EnemyBuilding& enemyBuilding, EnemyTower& enemyTower, AllyManager& allyManager, size_t improve) override — определение метода применения заклинания. Выполняется поиск пустой клетки рядом с игроком, в случае отсутствия места заклинание не выполняется, иначе на найденной клетке создаётся союзник. Процесс создания союзника повторяется improve+1 раз.

Принимаемые аргументы:

- *player*: *Player&* - игрок;
- *field*: *Field&* - поле;
- *enemyManager*: *EnemyManager&* - менеджер врагов;
- *enemyBuilding*: *EnemyBuilding&* - вражеское здание;
- *enemyTower*: *EnemyTower&* - вражеская башня;
- *allyManager*: *AllyManager&* - менеджер союзников;
- *improve*: *size_t* - количество улучшений.

Возвращаемое значение:

- *true/false*: *bool* - выполнено ли заклинание.

bool cast(Player& target, std::pair <int, int> locate) override — определение метода применения заклинания. Для данного заклинания применение этого вызова невозможно.

Принимаемые аргументы:

- *target*: *Player&* - игрок;
- *locate*: *std::pair <int, int>* - координаты применяющей сущности.

Возвращаемое значение:

- *false: bool* - данное заклинание с такими аргументами применять нельзя.

bool cast() override — определение метода применения заклинания. Для данного заклинания применение этого вызова невозможно.

Возвращаемое значение:

- *false: bool* - данное заклинание с такими аргументами применять нельзя.

class ImproveSpell — класс заклинания улучшения. Наследуется от интерфейса заклинания.

Методы класса:

std::string getName() const override — определение метода получения имени заклинания.

Возвращаемое значение:

- *std::string* - название заклинания.

std::string getDescription() const override — определение метода получения описания заклинания.

Возвращаемое значение:

- *std::string* - описание заклинания.

bool cast(Player& player, Field& field, EnemyManager& enemyManager, EnemyBuilding& enemyBuilding, std::pair <int, int> target, size_t improve) override — определение метода применения заклинания. Для данного заклинания применение этого вызова невозможно.

Принимаемые аргументы:

- *player*: *Player&* - игрок;
- *field*: *Field&* - поле;
- *enemyManager*: *EnemyManager&* - менеджер врагов;
- *enemyBuilding*: *EnemyBuilding&* - вражеское здание;
- *target*: *std::pair <int, int>* - выбранная координата применения.

Возвращаемое значение:

- *false*: *bool* - данное заклинание с такими аргументами применять нельзя.

bool cast(Player& player, Field& field, EnemyManager& enemyManager, EnemyBuilding& enemyBuilding, EnemyTower& enemyTower, AllyManager& allyManager, size_t improve) override — определение метода применения заклинания. Для данного заклинания применение этого вызова невозможно.

Принимаемые аргументы:

- *player*: *Player&* - игрок;
- *field*: *Field&* - поле;
- *enemyManager*: *EnemyManager&* - менеджер врагов;
- *enemyBuilding*: *EnemyBuilding&* - вражеское здание;
- *enemyTower*: *EnemyTower&* - вражеская башня;
- *allyManager*: *AllyManager&* - менеджер союзников;
- *improve*: *size_t* - количество улучшений.

Возвращаемое значение:

- *false*: *bool* - данное заклинание с такими аргументами применять нельзя.

bool cast(Player& target, std::pair <int, int> locate) override — определение метода применения заклинания. Для данного заклинания применение этого вызова невозможно.

Принимаемые аргументы:

- *target*: *Player&* - игрок;
- *locate*: *std::pair <int, int>* - координаты применяющей сущности.

Возвращаемое значение:

- *false: bool* - данное заклинание с такими аргументами применять нельзя.

bool cast() override — определение метода применения заклинания.

Возвращаемое значение:

- *true: bool* - выполнение заклинания.

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. Реализован интерфейс заклинания, от которого наследуются различные виды заклинаний. Все заклинания унифицированы. Создан класс руки игрока, для хранения карточек заклинаний. Класс вражеской башни, наносящей урон по игроку, если он попадает в радиус атаки. Добавлены возможности покупки и применения заклинаний.

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ

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

```
TEST(PlayerHandTest, ConstructorAndInitialSpell) {
    PlayerHand hand(3);

    EXPECT_EQ(hand.getMaxSize(), 3);
    EXPECT_EQ(hand.getSize(), 1);
    EXPECT_FALSE(hand.isFull());
}

TEST(PlayerHandTest, AddAndRemoveSpells) {
    PlayerHand hand(2);

    EXPECT_TRUE(hand.addSpell(spellType::DIRECT_DAMAGE, 15, 3));
    EXPECT_EQ(hand.getSize(), 2);

    EXPECT_FALSE(hand.addSpell(spellType::AREA_DAMAGE, 10, 4));

    EXPECT_TRUE(hand.removeSpell(0));
    EXPECT_EQ(hand.getSize(), 1);
}

TEST(PlayerHandTest, CastSpellWithTarget) {
    PlayerHand hand(2);
    Field field(15, 15);
    Player player(100, 10, make_pair(0, 0), 3, fightType::NEAR);
    EnemyManager enemyManager;
    EnemyBuilding building(5, 0, 10, make_pair(10, 10));

    hand.addSpell(spellType::DIRECT_DAMAGE, 20, 5);

    Enemy* enemy = new Enemy(5, 30, make_pair(2, 2));
    enemyManager.addEnemy(enemy);

    bool result = hand.cast(1, player, field, enemyManager, building,
make_pair(2, 2));
    EXPECT_TRUE(result);
    EXPECT_LT(enemy->getHealth(), 30);
}

TEST(PlayerHandTest, CastAreaDamageSpell) {
    PlayerHand hand(2);
    Field field(15, 15);
    Player player(100, 10, make_pair(0, 0), 3, fightType::NEAR);
    EnemyManager enemyManager;
```

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

    hand.addSpell(spellType::AREA_DAMAGE, 15, 4);

    Enemy* enemy1 = new Enemy(5, 30, make_pair(3, 3));
    Enemy* enemy2 = new Enemy(5, 30, make_pair(4, 3));
    enemyManager.addEnemy(enemy1);
    enemyManager.addEnemy(enemy2);

    bool result = hand.cast(1, player, field, enemyManager, building,
make_pair(3, 3));
    EXPECT_TRUE(result);
}

TEST(PlayerHandTest, CastTrapSpell) {
    PlayerHand hand(2);
    Field field(15, 15);
    Player player(100, 10, make_pair(0, 0), 3, fightType::NEAR);
    EnemyManager enemyManager;
    EnemyBuilding building(5, 0, 10, make_pair(10, 10));

    hand.addSpell(spellType::TRAP, 25, 3);

    bool result = hand.cast(1, player, field, enemyManager, building,
make_pair(2, 2));
    EXPECT_TRUE(result);

    EXPECT_EQ(field.getCell(make_pair(2, 2))->getType(),
cellType::TRAP);
}

TEST(PlayerHandTest, CastImproveSpell) {
    PlayerHand hand(2);

    hand.addSpell(spellType::IMPROVE, 0, 0);

    bool result = hand.cast(1);
    EXPECT_TRUE(result);
}

TEST(PlayerHandTest, CastAllySpell) {
    PlayerHand hand(2);
    Field field(15, 15);
    Player player(100, 10, make_pair(5, 5), 3, fightType::NEAR);
    EnemyManager enemyManager;
    EnemyBuilding building(5, 0, 10, make_pair(10, 10));
    EnemyTower tower(10, 3, make_pair(8, 8));
    AllyManager allyManager;

    hand.addSpell(spellType::ALLY, 10, 20);
```

```
    bool result = hand.cast(1, player, field, enemyManager, building,
tower, allyManager);
    EXPECT_TRUE(result);
    EXPECT_GT(allyManager.getAllies().size(), 0);
}

TEST(PlayerHandTest, InvalidCastOperations) {
    PlayerHand hand(2);
    Field field(15, 15);
    Player player(100, 10, make_pair(0, 0), 3, fightType::NEAR);
    EnemyManager enemyManager;
    EnemyBuilding building(5, 0, 10, make_pair(10, 10));

    EXPECT_FALSE(hand.cast(5, player, field, enemyManager, building,
make_pair(2, 2)));
    EXPECT_FALSE(hand.cast(5));
}

TEST(EnemyTowerTest, AttackPlayer) {
    EnemyTower tower(15, 4, make_pair(5, 5));
    Player player(100, 10, make_pair(3, 3), 3, fightType::NEAR);

    int initialHealth = player.getHealth();
    tower.attackPlayer(player);

    EXPECT_LT(player.getHealth(), initialHealth);
}

TEST(EnemyTowerTest, GetRangeAndCoordinates) {
    EnemyTower tower(10, 5, make_pair(3, 3));

    EXPECT_EQ(tower.getRange(), 5);
    EXPECT_EQ(tower.getCoordinates(), make_pair(3, 3));
}
```