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. Создание классов, реализующих основной цикл игры. При прохождении уровня игрок выбирает, перейти на следующий уровень или завершить игру. Реализация системы сохранения и загрузки игры, обработка исключительных ситуаций при работе с файлами сохранения.

Задание

1. Создать класс(ы) игры, который реализует основной цикл игры, и которому передаются команды от пользователя. Игровой цикл состоит из следующих шагов:

a) Начало игры

b) Запуск уровня

c) Ход игрока. Ход, атака или применение заклинания.

d) Ход союзников - если имеются

e) Ход врагов

f) Ход вражеской базы и башни - если имеются

Условие прохождения уровня студент определяет самостоятельно. Если игрок проигрывает, то игроку должно предлагаться начать заново игру, либо выйти из программы.

Все взаимодействие должно происходить через классы игры.

2. Реализовать систему сохранения и загрузки игры. Пользователь должен иметь возможность сохранить игру в любой момент. Пользователь должен иметь возможность загружаться при запуске программы (или выбрать новую), либо во время игры. Сохранения должны оставаться в консистентном состоянии между запусками игры.

3. Добавить обработку исключительных ситуаций для загрузки/сохранения, например, невозможность записать в файл, нельзя загрузиться так как файл не существует или в нем некорректные данные.

4. Реализовать переход на следующий уровень, после прохождения уровня. При переходе на следующий уровень создается новое поле другого размера с более сильными врагами. При переходе на следующий уровень,

значение жизни игрока восстанавливается, и половина его карточек заклинаний случайным образом удаляется.

5. Реализовать прокачку игрока при переходе между уровнями. Пользователь может улучшить характеристики игрока или улучшить заклинание (что и как улучшать определяет студент). Для этого нужно расширить игровой цикл.

Примечания:

- Класс игры может знать о игровых сущностях, но не наоборот
- При работе с файлом используйте идиому RAII.
- Исключения должны обязательно обрабатываться, и программа не должна завершаться
- Исключения должны быть информативными (содержать информацию о том, что и где произошло), на разные виды исключительных ситуаций должны быть свои исключения

Выполнение работы

UML-диграмма классов

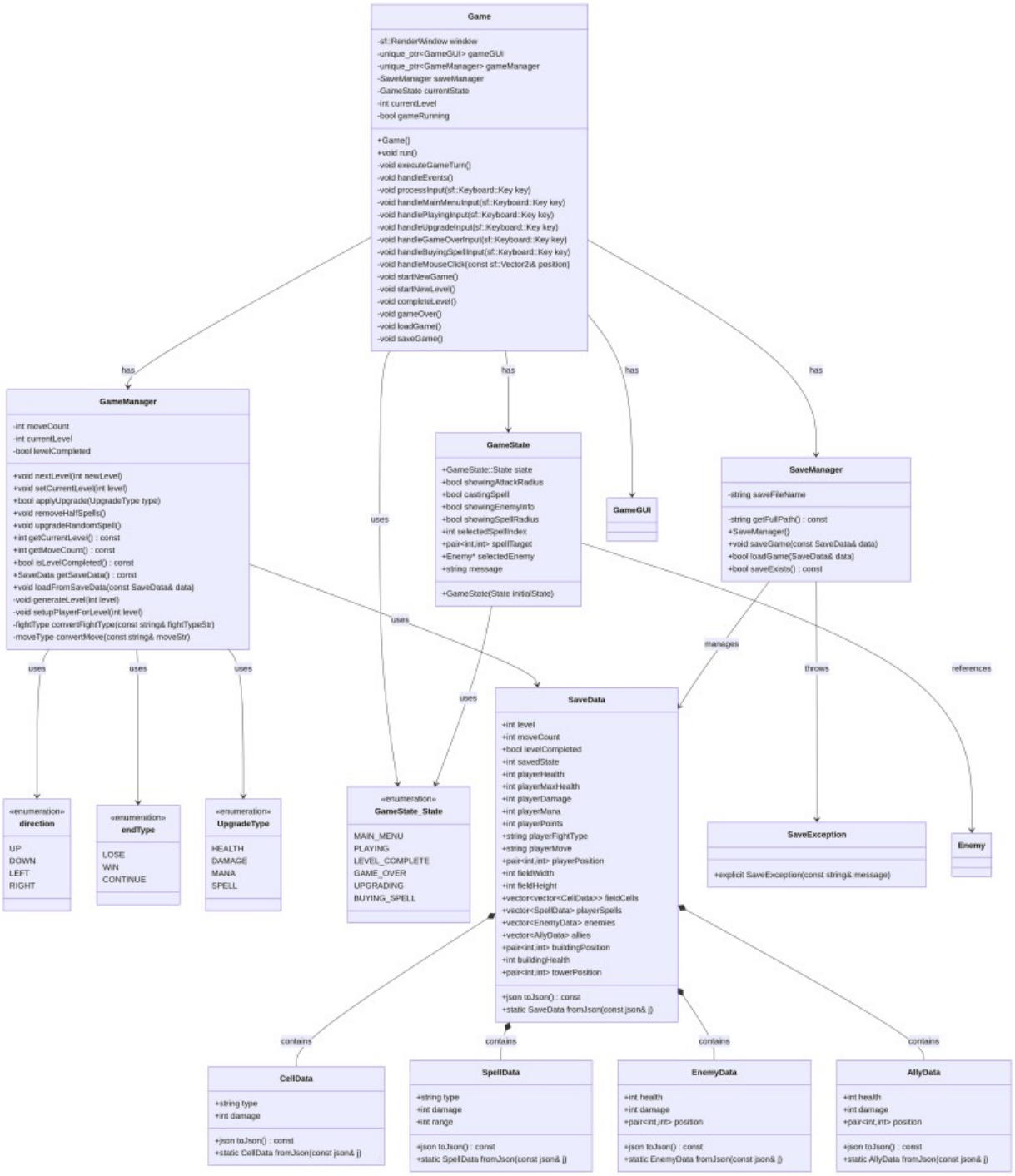

Рисунок 1. UML-диаграмма классов

Описание структуры кода

class GameManager — класс для осуществление взаимодействия между объектами разных классов. Данный класс реализован в первой лабораторной работе, в данной работе добавлены отдельные поля и методы для перехода на следующие уровни, сохранения и загрузки состояния игры.

Поля класса:

- *int moveCount* - поле для хранения текущего количества ходов;
- *int currentLevel* - поле для хранения текущего уровня;
- *bool levelCompleted* - поле значения прохождения уровня.

Методы класса:

void generateLevel(int level) — приватный метод для генерации нового уровня. Создаёт поле необходимо размера, вызывает метод подготовки игрока к новому уровню, создаёт стартового врага, вражеское здание и вражескую башню для нового уровня.

Принимаемые аргументы:

- *level: int* - уровень, который необходимо подготовить.

void setupPlayerForLevel(int level) — приватный метод для подготовки игрока к новому уровню. Ставит игрока на стартовую позицию (0,0) Увеличивает здоровье до максимального, обнуляет очки и обновляет заклинания.

Принимаемые аргументы:

- *level: int* - уровень, к которому необходимо подготовить игрока.

fightType convertFightType(const std::string& fightTypeStr) — приватный метод для конвертации типа боя из строки в *fightType*, необходим для загрузки сохранённых данных игры.

Принимаемые аргументы:

- *fightTypeStr: const std::string&* - строка типа боя.

Возвращаемое значение:

- *fightType* - тип боя.

moveType convertMove(const std::string& moveStr) — приватный метод для конвертации способности передвигаться из строки в *moveType*, необходим для загрузки сохранённых данных игры.

Принимаемые аргументы:

- *moveStr: const std::string&* - строка способности перемещаться.

Возвращаемое значение:

- *moveType* - способность игрока перемещаться.

void nextLevel(int newLevel) — метод для создания нового уровня. Обновляется значение текущего уровня, флаг прохождения уровня ставиться *false*, количество ходов обнуляется. Вызывается метод удаления половины заклинаний и метод генерации нового уровня.

Принимаемые аргументы:

- *newLevel: int* - новый уровень.

void setCurrentLevel(int level) — метод для задания текущего уровня. Значение текущего уровня устанавливается равным *level.*

Принимаемые аргументы:

- *level: int* - текущий уровень.

bool applyUpgrade(UpgradeType type) — метод применения улучшений. В зависимости от выбранного типа улучшения, улучшается соответствующий параметр у игрока, либо вызывается метод улучшения случайного заклинания.

Принимаемые аргументы:

- *type: UpgradeType* - выбранный тип улучшения.

Возвращаемое значение:

- *true/false: bool* - выполнено ли улучшение (может быть не выполнено в случае выбора несуществующего типа).

void removeHalfSpells() — метод удаления половины заклинаний в руке. Удаляется половина из находящихся в руке заклинаний, заклинания для удаления выбираются случайным образом. Если в руке нечётное число заклинаний удаляется на 1 меньше половины.

void upgradeRandomSpell() — метод улучшения случайного заклинания. Случайным образом выбирается заклинание, у которого увеличивается урон.

int getCurrentLevel() const — метод получения текущего уровня.

Возвращаемое значение:

- *currentLevel: int* - текущий уровень.

int getMoveCount() const — метод получения текущего количества ходов.

Возвращаемое значение:

- *moveCount: int* - текущее количество ходов.

bool isLevelCompleted() const — метод получения флага прохождения уровня.

Возвращаемое значение:

- *levelCompleted: bool* - флаг прохождения уровня.

SaveData getSaveData() const — метод формирования данных для сохранения. Создаётся объект структуры *SaveData*, в который записываются данные игры для сохранения.

Возвращаемое значение:

- *data: SaveData* - данные для сохранения.

void loadFromSaveData(const SaveData& data) — метод загрузки сохранённых данных. Создаёт объекты игры, на основе сохранённых данных.

Принимаемые аргументы:

- *data: const SaveData&* - структура сохранённых данных.

struct SaveData — структура для сохранения данных игры.

Поля:

- *int level* - поле для сохранения уровня (по умолчанию 1);
- *int moveCount* - поле для сохранения числа ходов (по умолчанию 0);
- *bool levelCompleted* - флаг пройденности уровня;
- *int savedState* - поле для сохранения текущего состояния игры;
- *int playerHealth = 0* - поле для сохранения здоровья игрока;
- *int playerMaxHealth = 0* - поле для сохранения максимального здоровья игрока;
- *int playerDamage = 0* - поле для сохранения урона игрока;
- *int playerMana = 0* - поле для сохранения количества монет;
- *int playerPoints = 0* - поле для сохранения очков игрока;
- *std::string playerFightType = "NEAR"* - поле для сохранения типа боя (по умолчанию *NEAR)*;
- *std::string playerMove = "CAN"* - поле для сохранения способности перемещаться (по умолчанию *CAN)*;
- *std::pair <int, int> playerPosition = {0, 0}* - поле для сохранения координат игрока (по умолчанию *{0, 0})*;
- *int fieldWidth* - поле для сохранения ширины поля (по умолчанию 0*)*;
- *int fieldHeight* - поле для сохранения высоты поля (по умолчанию 0*)*;
- *std::vector <std::vector <CellData>> fieldCells* - вектор для сохранения данных клеток поля;

- *std::vector <SpellData> playerSpells* - вектор для сохранения данных заклинаний в руке игрока;
- *std::vector <EnemyData> enemies* - вектор для сохранения данных врагов;
- *std::vector <AllyData> allies* - вектор для сохранения данных союзников;
- *std::pair <int, int> buildingPosition* - поле для сохранения координат вражеского здания (по умолчанию *{0, 0}*);
- *int buildingHealth* - поле для сохранения силы вражеского здания (по умолчанию 0);
- *std::pair <int, int> towerPosition* - поле для сохранения координат вражеской башни (по умолчанию *{0, 0}*);

Методы:

json toJson() const — метод записи структуры в *json*-файл.

static SaveData fromJson(const json& j) — метод извлечения данных из *json*-файла в структуру.

Возвращаемое значение:

- *data: SaveData* - извлечённые данные.

class SaveException — класс ошибок работы с файлами. Наследуется от *std::runtime_error*.

Методы класса:

explicit SaveException(const std::string& message) — конструктор. Создаёт сохранение ошибки *std::runtime_error*.

class SaveManager — класс для сохранения и загрузки данных.

Поля класса:

- *std::string saveFileName* - поле для хранения имени файла сохранения.

Методы класса:

SaveManager() — конструктор. Проверяет наличие директории для сохранения, при необходимости создаёт её.

void saveGame(const SaveData& data) — функция записи данных игры в файл. В файл записываются данные игры и метаданные, в случае ошибок в файле, вызываются исключения.

Принимаемые аргументы:

- *data: const SaveData&* - сохраняемые данные.

bool loadGame(SaveData& data) — функция извлечения данных игры из файла. Проверяется существование и корректность файла сохранения, в случае проблем в стандартный поток вывода выводится сообщение о проблеме. Данные извлекаются из файла.

Принимаемые аргументы:

- *data: SaveData&* - объект для извлечения данных.

Возвращаемое значение:

- *true/false: bool* - извлечены ли данные.

struct GameState — структура для хранения текущего состояния игры.

Поля:

- *State state* - поле для хранения состояния игры (главное меню, непосредственно игра, переход на следующий уровень и т.д.);
- *bool showingAttackRadius* - флаг отображения радиуса атаки;
- *bool castingSpell* - флаг применения заклинания;
- *bool showingEnemyInfo* - флаг отображения информации о враге;
- *bool showingSpellRadius* - флаг отображения радиуса заклинания;
- *int selectedSpellIndex* - индекс выбранного заклинания;

- *std::pair <int, int> spellTarget* - координаты выбранной цели заклинания;
- *Enemy* selectedEnemy* - выбранный враг (для отображения информации врага).

Методы:

GameState(State initialState = MAIN_MENU) — конструктор. Принимает стартовое состояние, устанавливает флаги *false*, индекс заклинания -1 и выбранного врага *nullptr*.

Принимаемые аргументы:

- *initialState: State* - начальное состояние (по умолчанию показ главного меню).

class Game — класс игры.

Поля класса:

- *sf::RenderWindow window* - поле игрового окна;
- *std::unique_ptr<GameGUI> gameGUI* - объект для работы с *GUI*;
- *std::unique_ptr<GameManager> gameManager* - игровой менеджер;
- *SaveManager saveManager* - менеджер сохранения;
- *GameState currentState* - поле текущего состояния игры;
- *int currentLevel* - поле текущего уровня;
- *bool gameRunning* - флаг течения игры.

Методы класса:

void executeGameTurn() — приватный метод для выполнения действий союзников, врагов и вражеских построек. Вызывает необходимые методы действий из *gameManager*, выполняет проверку окончания игры.

void handleEvents() — приватный метод обработки действий пользователя. В случае нажатия на «крестик» закрывается окно игры, при

нажатии на какую-либо кнопку клавиатуры вызывается метод обработки ключей клавиатуры, при нажатии на левую клавишу мыши сохраняется позиция нажатия.

void processInput(sf::Keyboard::Key key) — приватный метод для обработки нажатия на кнопки клавиатуры. В зависимости от текущего состояния игры вызываются конкретные обработчики действий.

Принимаемые аргументы:

- *key: sf::Keyboard::Key* - нажатая клавиша.

void handleMainMenuInput(sf::Keyboard::Key key) — приватный метод для обработки нажатия на кнопки клавиатуры при работе с главным меню. При нажатии клавиши «1» - вызов метода запуска новой игры, переход в состояние *GameState::PLAYING*, «2» - вызов метода загрузки сохранённой игры, «3» - прекращение игры.

Принимаемые аргументы:

- *key: sf::Keyboard::Key* - нажатая клавиша.

void handlePlayingInput(sf::Keyboard::Key key) — приватный метод для обработки нажатия на кнопки клавиатуры при ходе игрока во время игры. При нажатии на стрелки - вызов метода перемещения игрока, «F» - смена типа боя, «A» - показ радиуса атаки, «B» - переход к покупке заклинания, «S» - сохранение текущего состояния игры, «1»-«6» - применение заклинание, «Esc» - сброс атаки или применения заклинания. В случае свершения действия вызывается метод действия остальных игровых сущностей.

Принимаемые аргументы:

- *key: sf::Keyboard::Key* - нажатая клавиша.

void handleUpgradeInput(sf::Keyboard::Key key) — приватный метод для обработки выбора улучшения при переходе на новый уровень. При нажатии

клавиши «S» сохраняется текущее состояние, «1»-«4» - вызывается метод применения улучшения, после чего выполняется запуск нового уровня.

Принимаемые аргументы:

- *key: sf::Keyboard::Key* - нажатая клавиша.

void handleGameOverInput(sf::Keyboard::Key key) — приватный метод для обработки случая проигрыша. При нажатии клавиши «1» выполняется переход к главному меню, «2» - завершение игры.

Принимаемые аргументы:

- *key: sf::Keyboard::Key* - нажатая клавиша.

void handleBuyingSpellInput(sf::Keyboard::Key key) — приватный метод для обработки покупки заклинания. При нажатии клавиш «1»-«5» - вызывается метод добавление необходимого заклинания, «Esc» - выход из состояния покупки.

Принимаемые аргументы:

- *key: sf::Keyboard::Key* - нажатая клавиша.

void handleMouseClick(const sf::Vector2i& position) — приватный метод для обработки нажатия на кнопку мыши. Если текущее состояние — показ радиуса атаки — выполняется вызов функции атаки. Если текущее состояние — применение заклинания, показ радиуса заклинания — вызывается применение заклинания. После заклинания и атаки вызывается метод действия остальных игровых сущностей. Если клик выполнен по врагу запускается отображение информации о враге.

Принимаемые аргументы:

- *position: const sf::Vector2i&* - позиция клика.

void startNewGame() — приватный метод для запуска новой игры. Создаётся новый *GameManager,* текущий уровень устанавливается равным 1.

void startNewLevel() — приватный метод для создания нового уровня. Увеличивается текущий уровень, вызывается метод созданий нового уровня.

void loadGame() — приватный метод для запуска сохранённого состояния игры. Выполняется загрузка сохранённых данных игры. В случае невозможности создаётся новая игра.

void saveGame() — приватный метод для сохранения текущего состояния игры. Вызывается метод подготовки данных к сохранению, после чего вызывается метод сохранения данных в файл.

Game() — конструктор. Создаётся игровое окно, игровой менеджер. Текущее состояние устанавливается на показ главного меню.

void run() — метод для «ведения» игры. Пока открыто игровое окно и игра продолжается, вызывается метод обработки действий пользователя, вызывается проверка продолжения игры и отрисовка текущего состояния 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. Созданы классы, реализующие основной цикл игры. При прохождении уровня игрок выбирает, перейти на следующий уровень или завершить игру. Реализована системы сохранения и загрузки игры, обработка исключительных ситуаций при работе с файлами сохранения.