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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Объектно-ориентированное программирование»

Тема: Связывание классов

Студентка гр. 3388 _____ Басик В.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы

Цель задания — создать класс игры, который реализует игровой цикл с чередующимися ходами игрока и компьютерного врага. Игрок может использовать способности и атаковать, а враг только атакует. При поражении игрока начинается новая игра, а при победе продолжается следующий раунд с сохранением состояния поля и способностей. Класс должен включать методы для управления игрой, начала новой игры, выполнения ходов и сохранения/загрузки игры. Также необходимо переопределить операторы ввода/вывода для состояния игры и реализовать сохранение, которое можно загрузить после перезапуска программы, используя идиому RAII для работы с файлами.

Задание

Создать класс игры, который реализует следующий игровой цикл:

Начало игры

Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.

В случае проигрыша пользователь начинает новую игру

В случае победы в раунде, начинается следующий раунд, причем состояние поля и способностей пользователя переносятся.

Класс игры должен содержать методы управления игрой, начало новой игры, выполнить ход, и т.д., чтобы в следующей лаб. работе можно было выполнять управление исходя из ввода игрока.

Реализовать класс состояния игры, и переопределить операторы ввода и вывода в поток для состояния игры. Реализовать сохранение и загрузку игры. Сохраняться и загружаться можно в любой момент, когда у пользователя приоритет в игре. Должна быть возможность загружать сохранение после перезапуска всей программы.

Примечание:

Класс игры может знать о игровых сущностях, но не наоборот

Игровые сущности не должны сами порождать объекты состояния

Для управления самое игрой можно использовать обертки над командами

При работе с файлом используйте идиому RAII.

Выполнение работы

Класс Game

Класс Game представляет собой основной класс для управления игровой логикой, включая чередование ходов игрока и компьютерного врага, а также сохранение и загрузку состояния игры. Он взаимодействует с игровыми полями, менеджерами кораблей и системой способностей.

Поля класса Game:

- game_state: Объект класса GameState, который хранит текущее состояние игры, включая поля, корабли, способности и информацию о текущем ходе.

Методы класса Game:

- Game(shared_ptr<Battleground> player_field, shared_ptr<Battleground> enemy_field, shared_ptr<ShipsManager> player_ships, shared_ptr<ShipsManager> enemy_ships): Конструктор, инициализирующий поля и менеджеры кораблей игрока и врага.
- void start(): Запускает игру, устанавливая начальные значения флагов состояния игры.
- void playerTurn(size_t x, size_t y, bool use_skill=false, size_t skill_x=0, size_t skill_y=0): Выполняет ход игрока, включая использование способностей и атаку.
- void enemyTurn(): Выполняет ход врага, атакуя случайную клетку на поле игрока.
- bool isPlayerWin(): Проверяет, выиграл ли игрок, то есть все корабли врага уничтожены.
- bool isEnemyWin(): Проверяет, выиграл ли враг, то есть все корабли игрока уничтожены.

- void reload_enemy(shared_ptr<Battleground> battleground, shared_ptr<ShipsManager> ships_manager): Перезагружает состояние поля и менеджера кораблей врага.
- void reload_game(shared_ptr<Battleground> player_field, shared_ptr<Battleground> enemy_field, shared_ptr<ShipsManager> player_ships, shared_ptr<ShipsManager> enemy_ships): Перезагружает полное состояние игры, включая поля, корабли и способности.
- bool getIsPlayerTurn(): Возвращает текущий статус хода игрока.
- bool getIsGameStarted(): Возвращает текущий статус начала игры.
- void check_game_status(bool reverse=false): Проверяет статус игры и выбрасывает исключение, если текущий ход игрока или игры неверен.
- void save(string filename): Сохраняет текущее состояние игры в файл.
- void load(string filename): Загружает состояние игры из файла.

Класс GameState

Класс GameState представляет собой структуру данных, которая хранит все ключевые элементы состояния игры, включая поля, корабли, способности и информацию о текущем ходе. Этот класс используется для управления состоянием игры, его сериализации и десериализации, а также для обеспечения сохранения и восстановления игры.

Поля класса GameState:

- m_player_field: Умный указатель на объект поля игрока (тип Battleground).
- m_enemy_field: Умный указатель на объект поля врага (тип Battleground).
- m_player_ships: Умный указатель на менеджер кораблей игрока (тип ShipsManager).
- m_enemy_ships: Умный указатель на менеджер кораблей врага (тип ShipsManager).

- m_info_holder: Объект класса InfoHolder, который содержит дополнительную информацию для управления игрой, включая координаты для применения способностей.
- m_ability_manager: Менеджер способностей, отвечающий за обработку и использование способностей в игре (тип AbilityManager).
- m_is_player_turn: Флаг, указывающий, чей сейчас ход (игрока или врага).
- m_is_game_started: Флаг, который указывает, началась ли игра.

Методы класса GameState:

- GameState(shared_ptr<Battleground> player_field, shared_ptr<Battleground> enemy_field, shared_ptr<ShipsManager> player_ships, shared_ptr<ShipsManager> enemy_ships): Конструктор, инициализирующий поля, менеджеры кораблей и объект информации для игры.
- vector<pair<size_t, size_t>> getShipPosition(Ship& ship, Battleground& field) const: Метод для получения позиций корабля на игровом поле.
- string serializeShips(ShipsManager& ships_manager, Battleground& field) const: Метод для сериализации кораблей и их положения на поле в строку.
- string serializeField(Battleground& field) const: Метод для сериализации поля в строку.
- string serializeSkills(const AbilityManager& ability_manager) const: Метод для сериализации навыков (способностей) в строку.
- vector<string> split(const string &s, char delim): Метод для разделения строки по разделителю.
- vector<tuple<size_t, Ship::Orientation, vector<tuple<size_t, size_t, Segment::State>>>> readShips(vector<string>& lines, size_t& j): Метод для чтения и десериализации данных о кораблях из строки.
- void updateFieldWithShips(Battleground& field, ShipsManager& ships_manager, vector<tuple<size_t, Ship::Orientation, vector<tuple<size_t,

size_t, Segment::State>>>>& ships, vector<string>& lines, size_t& j): Метод для обновления поля с кораблями на основе десериализованных данных.

Операторы ввода/вывода:

- ostream& operator<<(ostream& os, const GameState& game_state): Метод для записи состояния игры в поток (например, в файл).
- istream& operator>>(istream& is, GameState& game_state): Метод для чтения состояния игры из потока и его восстановления.

Класс MD5

Этот класс реализует алгоритм хеширования MD5, который принимает входные данные и вычисляет их хеш в виде 128-битного значения. Класс включает методы для обновления состояния хеширования, завершения хеширования и получения результата в удобном формате.

Поля класса:

- state: Массив из 4 целых чисел, представляющих текущее состояние хеша (результат промежуточных вычислений).
- count: Массив из 2 целых чисел, отслеживающий количество обработанных бит данных.
- buffer: Буфер для хранения входных данных перед их обработкой.
- digest: Массив из 16 байт, который хранит итоговый хеш.
- BLOCK_SIZE: Константа, определяющая размер блока данных (64 байта).
- OUTPUT_SIZE: Константа, определяющая размер итогового хеша (16 байт, 128 бит).

Методы класса:

- Конструктор MD5(): Инициализирует начальное состояние хеширования, устанавливает значения для state, count и других переменных, подготавливая их к работе.

- update(const unsigned char* input, unsigned int length): Обновляет хеш с помощью новых данных. Данные добавляются в буфер и, если это необходимо, блоки данных обрабатываются.
- finalize(): Завершает процесс хеширования, добавляя специальные данные для завершения (паддинг и длину данных), а затем вычисляет итоговый хеш.
- hexdigest(): Возвращает результат хеширования в виде строки из 32 символов в шестнадцатеричном формате.
- reset(): Сбрасывает все поля, восстанавливая состояние до первоначального, чтобы класс можно было использовать для нового хеширования.

Вспомогательные методы:

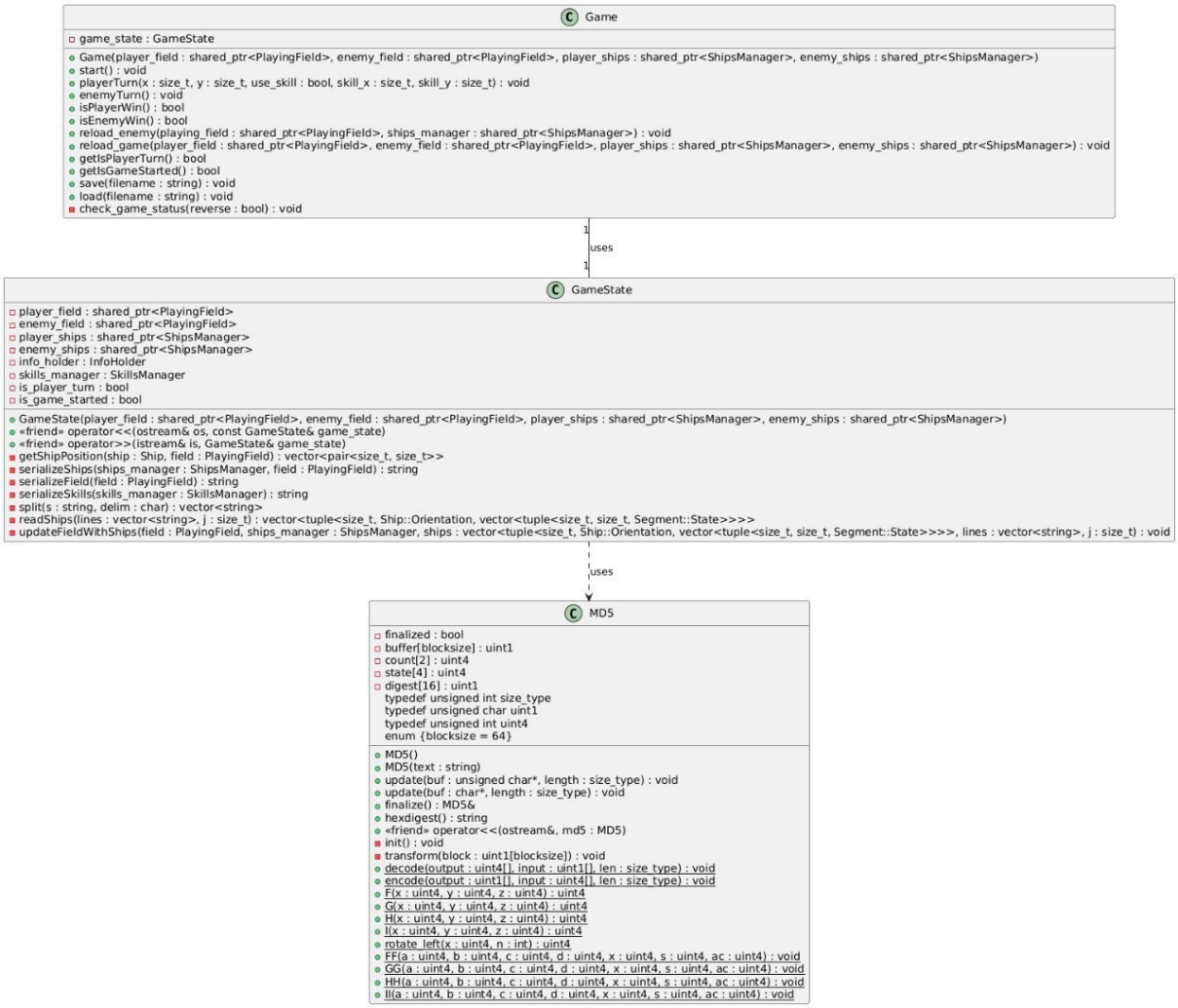
- F(), G(), H(), I(): Основные логические функции, используемые в процессе обработки данных.
- rotate_left(): Функция для циклического сдвига влево, используемая при преобразовании данных.
- transform(): Выполняет основную обработку данных (блоков) с использованием логических операций и сдвигов, обновляя состояние хеша.
- encode() и decode(): Функции для преобразования данных между форматами байтов и 32-битных слов.

main()

В программе создается игра, в которой два игрока (человек и враг) размещают свои корабли на поле, а затем поочередно делают ходы. Для каждого игрока создаются менеджеры кораблей и поля боя. Корабли размещаются на поле с помощью метода addShip(). Вся игра управляется через класс Game, который запускает игровой процесс, выполняет ходы игрока и врага, а также предоставляет функциональность для сохранения и загрузки состояния игры. Используются умные указатели для управления

ресурсами и предотвращения утечек памяти. В конце программы выводится результат игры (победа игрока или врага).

UML-диаграмма классов



Выводы

Была разработана система управления игрой для "Морского боя", реализующая игровой цикл с чередующимися ходами игрока и врага. Игрок имеет возможность использовать различные способности и атаковать, в то время как враг ограничен только атакующими действиями. В случае поражения игрока игра начинается заново, а при победе продолжается новый раунд с сохранением состояния поля и активных способностей.

Класс игры включает методы для управления игровыми ходами, начала новой игры, сохранения и загрузки состояния. Реализованы операторы ввода/вывода для удобного сохранения и восстановления состояния игры, что позволяет продолжить игру после перезапуска программы. Для работы с файлами применена идиома RAII, что гарантирует корректное управление ресурсами. Всё это способствует надежности и гибкости игрового процесса, обеспечивая удобство для пользователя и возможность дальнейших расширений системы.