**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

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Павлов А. Р. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## Цель работы

Целью данной лабораторной работы является разработка игрового класса, реализующего игровой процесс с учетом состояния игры и возможностью сохранения и загрузки прогресса.

## Задание

a) Создать класс игры, который реализует следующий игровой цикл:

1. Начало игры
2. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
3. В случае проигрыша пользователь начинает новую игру
4. 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.

b) 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

### 1. **Класс** InputProcessor

Поля:

* std::unordered\_map<flagType, bool> flags: флаги для отслеживания состояния различных опций игры.

Методы:

* static std::pair<size\_t, size\_t> readCoords(): статический метод для чтения координат.
* GameSettings\* readGameSettings(): метод для чтения режима игры.
* bool readStartNewGame(): метод для проверки, следует ли начать новую игру.
* Orientation readShipOrientation(): метод для чтения ориентации корабля.
* Option readOption(): метод для чтения выбора опции игрока.
* bool& getFlag(flagType flagName): метод для получения ссылки на флаг по имени.
* void resetFlags(): метод для сброса флагов.

### 2. **Класс** OutputProcessor

Методы:

* void drawBoard(GameBoard& board) const: метод для отображения игрового поля.
* void drawPlayerShips(ShipManager& shipManager) const: метод для отображения кораблей игрока.
* void drawBoards(GameBoard& userBoard, GameBoard& aiBoard) const: метод для отображения двух полей (пользователя и компьютера).
* void drawShips(ShipManager& userShipManager, ShipManager& aiShipManager) const: метод для отображения кораблей пользователя и компьютера.
* static void showMessage(const std::string& message): статический метод для вывода сообщения.

### 3. **Класс** GameController

Поля:

* OutputProcessor\* OutputProcessor: указатель на объект OutputProcessor.
* InputProcessor\* InputProcessor: указатель на объект InputProcessor.
* bool userTurn: флаг, указывающий, чей ход (пользователя или компьютера).
* GameStatus gameStatus: статус игры (в процессе, окончена, выход).
* GameState\* gameState: указатель на объект GameState.

Методы:

* GameController(): конструктор класса.
* ~GameController(): деструктор класса.
* void startNewGame(): метод для начала новой игры.
* void playRound(): метод для выполнения одного раунда игры.
* void makeMove(): метод для выполнения одного хода.

### 4. **Класс** GameState

Поля:

* size\_t roundNumber: номер текущего раунда.
* GameSettings\* GameSettings: указатель на объект GameSettings.
* InputProcessor\* InputProcessor: указатель на объект InputProcessor.
* UserPlayer\* user: указатель на объект UserPlayer.
* ComputerPlayer\* ai: указатель на объект ComputerPlayer.

Методы:

* GameState(GameSettings\* GameSettings, InputProcessor\* InputProcessor): конструктор класса.
* ~GameState(): деструктор класса.
* void incrementRoundNumber(): метод для увеличения номера раунда.
* void resetRoundNumber(): метод для сброса номера раунда.
* size\_t getRoundNumber() const: метод для получения номера текущего раунда.
* GameSettings& getGameSettings(): метод для получения объекта GameSettings.
* UserPlayer& getUser(): метод для получения объекта UserPlayer.
* ComputerPlayer& getAI(): метод для получения объекта ComputerPlayer.
* void saveGame(): метод для сохранения состояния игры.
* void loadGame(): метод для загрузки состояния игры.
* friend std::ostream& operator<<(std::ostream& os, const GameState& state): оператор вывода для записи состояния игры в поток.
* friend std::istream& operator>>(std::istream& is, GameState& state): оператор ввода для чтения состояния игры из потока.

### 5. **Класс** ComputerPlayer

Поля:

* GameBoard\* aiBoard: указатель на игровое поле компьютера.
* ShipManager\* aiShipManager: указатель на менеджер кораблей компьютера.

Методы:

* ComputerPlayer() = default: конструктор класса.
* ~ComputerPlayer(): деструктор класса.
* void createShips(const GameSettings& GameSettings): метод для создания кораблей для компьютера.
* void makeMove(GameBoard& enemyBoard): метод для выполнения хода компьютером.
* GameBoard& getGameBoard(): метод для получения игрового поля компьютера.
* ShipManager& getShipManager(): метод для получения менеджера кораблей компьютера.
* bool isDefeated(): метод для проверки, проиграл ли компьютер.
* friend std::ostream& operator<<(std::ostream& os, ComputerPlayer& player): оператор вывода для записи объекта ComputerPlayer в поток.
* friend std::istream& operator>>(std::istream& is, ComputerPlayer& player): оператор ввода для чтения объекта ComputerPlayer из потока.

### 6. **Класс** UserPlayer

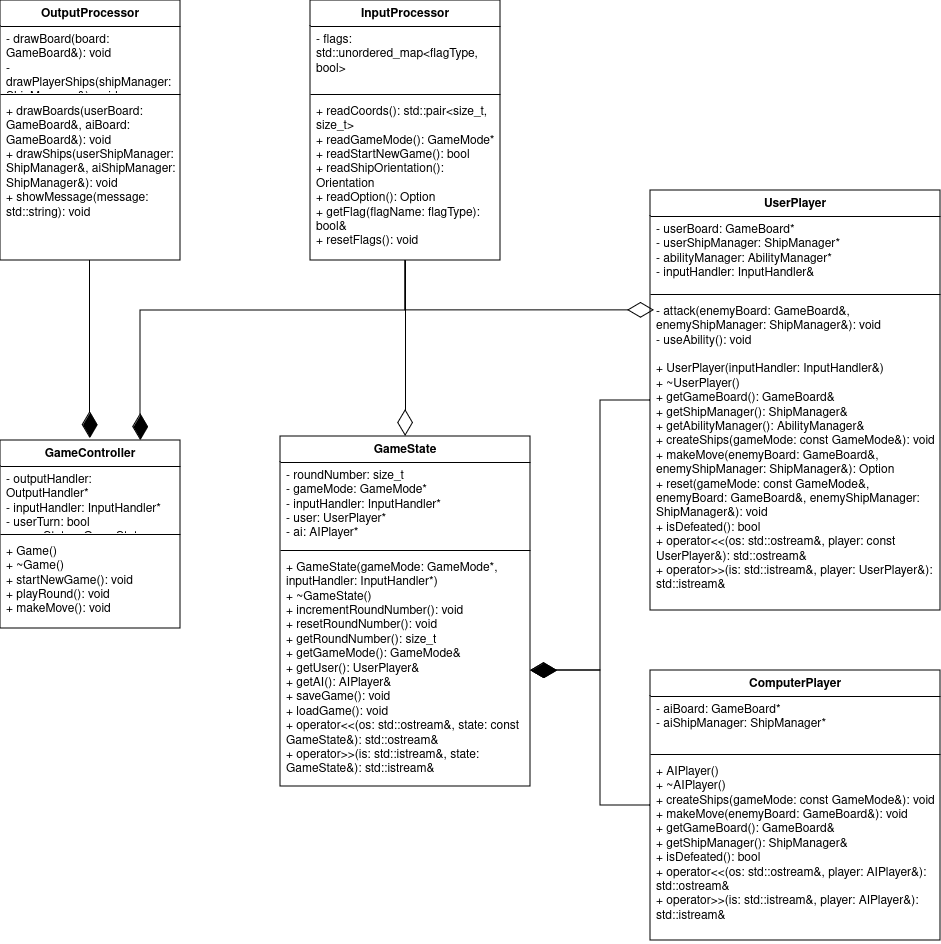
Поля:

* GameBoard\* userBoard: указатель на игровое поле пользователя.
* ShipManager\* userShipManager: указатель на менеджер кораблей пользователя.
* SkillManager\* abilityManager: указатель на менеджер способностей пользователя.
* InputProcessor& InputProcessor: ссылка на объект InputProcessor.

Методы:

* UserPlayer(InputProcessor& InputProcessor): конструктор класса.
* ~UserPlayer(): деструктор класса.
* GameBoard& getGameBoard(): метод для получения игрового поля пользователя.
* ShipManager& getShipManager(): метод для получения менеджера кораблей пользователя.
* SkillManager& getSkillManager(): метод для получения менеджера способностей пользователя.
* void createShips(const GameSettings& GameSettings): метод для создания кораблей пользователя.
* Option makeMove(GameBoard& enemyBoard, ShipManager& enemyShipManager): метод для выполнения хода пользователя.
* void reset(const GameSettings &GameSettings, GameBoard &enemyBoard, ShipManager &enemyShipManager): метод для сброса состояния игрока.
* bool isDefeated(): метод для проверки, проиграл ли пользователь.
* friend std::ostream& operator<<(std::ostream& os, const UserPlayer& player): оператор вывода для записи объекта UserPlayer в поток.
* friend std::istream& operator>>(std::istream& is, UserPlayer& player): оператор ввода для чтения объекта UserPlayer из потока.

## UML-диаграмма классов



## Выводы

В ходе выполнения работы был разработан класс игры, который позволяет управлять игровым процессом, сохраняя и загружая состояния игры. Реализованный игровой цикл соответствует описанным требованиям: игрок чередуется с компьютером, есть возможность сохранять и загружать прогресс, а в случае поражения начинается новая игра.

Использование класса состояния игры позволило выделить логику сохранения данных, что улучшило читаемость и масштабируемость кода. Переопределение операторов ввода/вывода для состояния игры упростило работу с файлами, а применение идиомы RAII обеспечило безопасное управление ресурсами.