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. 3344 _____ Бубякина Ю.В.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург
2024

Цель работы

Изучить связывание классов, путём усовершенствования программы из предыдущей лабораторной работы. Необходимо создать: класс игры и класс состояния игры.

Задание

0. Создать класс игры, который реализует следующий игровой цикл:
   0. Начало игры
      i. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
   ii. В случае проигрыша пользователь начинает новую игру
   iii. 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

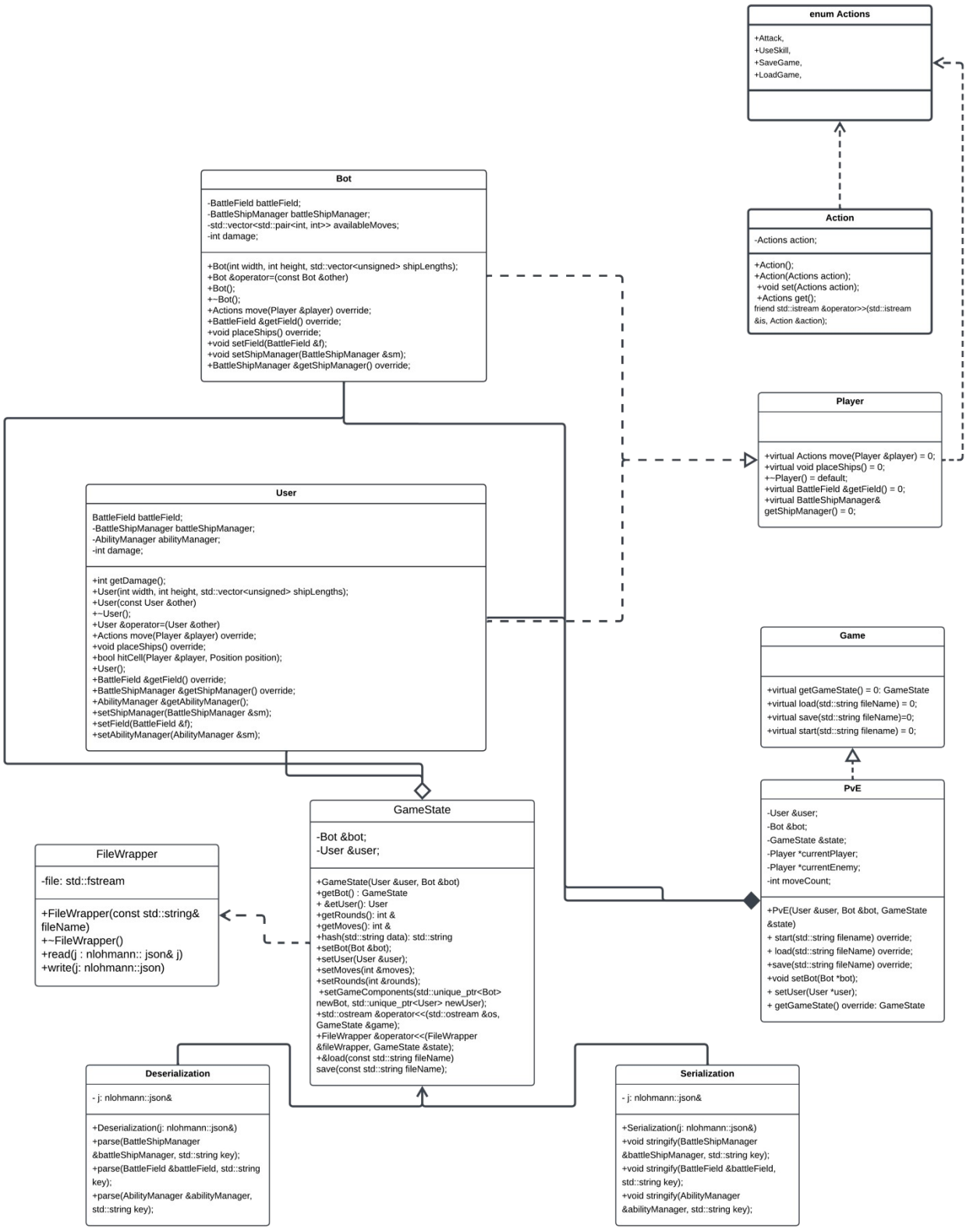

Рисунок 1 – UML-диаграмма классов

Код программы содержит реализацию классов: *Game*, *PvE*, *Player*, *User*, *Bot*, *Serialization*, *Deserialization*, *FileWrapper* и *GameState*.

Классы *Game* и *GameState* были добавлены согласно заданию. Game является абстрактным классом, от которого можно отнаследоваться и создать любой тип игры (в данном случае игра против бота *PvE*). Такая реализация позволит в будущем создавать другие типы игр такие как игра игрока против игрока и.т.д. *PvE* связывает классы и работает с ними, описывает игровой цикл и выполнение ходов. Класс *GameState* отвечает за связывание классов *Serialization*, *Deserialization* и *FileWrapper*, которые в сумме дают возможность работать с json файлом и совершать загрузку/сохранение игры. В нём также происходит хэширование json файла для его защиты от внешнего вмешательства.

Классы *Player, User* и *Bot* являются дата-классами, *Player* – абстрактный класс, который хранит общие для игрока и бота поля и методы; *User* и *Bot* – наследуемые от *Player* классы, представляющие собой игрока и бота соответственно, могут только возвращать значения полей.

Классы *Serialization* и *Deserialization* отвечают за считывание и запись из json файла. Прописаны методы для менеджера кораблей, поля и менеджера способностей, чтобы реализовать загрузку и сохранение игры. Обработка json файла организована с использованием библиотеки nlohmann/json.

Класс *FileWrapper* реализован как обёртка над файлом с использованием идиомы RAII для более удобной работы. В конструкторе происходит открытие файла, а в деструкторе его закрытие.

Помимо обозначенных классов, реализованы и интегрированы в код новые классы-исключения для обработки различных исключительных случаев работы с файлом и игрой.

Game является абстрактным классом для реализации логики игры. Он имеет следующие поля:

virtual GameState getGameState() = 0; - возвращает текущее состояние игры

virtual void load(std::string fileName) = 0 - загрузка игры

virtual void save(std::string fileName) = 0 - сохранение игры

virtual void start() = 0 - старт игры

PvEGame является классом, который был отнаследован от абстрактного класса Game. Его поля:

User& user - класс игрока

Bot& bot - класс бота

GameState& state класс состояния игры

Player currentPlayer* - текущий игрок в данном ходеж

Player currentEnemy* - текущий противник в данном ходе

И следующие методы:

void start () override - перегруженный метод Game

void load(std::string fileName)override - перегруженный метод Game

void save(std::string fileName) override - перегруженный метод Game

Класс Player является абстрактным классом для пользователя и бота. Он имеет следующие методы:

virtual BattleBattleField& getBattleBattleField() = 0 - возвращает поле игрока

virtual ShipManager& getShipManager() = 0 - возвращает менеджер кораблей игрока

virtual void placeShips() = 0 - расставляет корабли текущего игрока

virtual UserInputs move(Player& player) = 0 - ход игрока

Класс User является реализацией класса пользователя, который наследуется от класса *Player*. Он имеет следующие поля:

BattleBattleBattleField battleBattleBattleField - поле;

BattleShipManager battleShipManager - менеджер кораблей;

AbilityManager abilityManager; - менеджер способностей

int damage;

И следующие методы:

Actions move(Player &player) override - совершает действие пользователя

void placeShips() override - расставляет корабли пользователя

bool hitCell(Player &player,Position position) производит атаку пользователя

BattleField& getBattleField() override - возвращает поле пользователя

ShipManager& getShipManager() override возвращает менеджер кораблей пользователя

SkillManager& getSkillManager() - возвращает менеджер способностей пользователя

И соответсвующие им методы вставки.

Класс *Bot* является реализацией класса бота, он тоже наследуется от класса *Player*. Он имеет следующие поля:

- *BattleField& BattleField* – ссылка на поле
- BattleShipManager battleshipManager - ссылка на менеджер кораблей
- int damage - текущий урон бота
- std::vector<std::pair<int, int>> availableMoves - текущие доступные ходы бота

И следующие методы:

Action move(Player& player) override - действие бота

BattleField& getBattleField() override - получить ссылку на поле бота

void placeShips() override - рандомно расставить корабли бота

BattleShipManager& getBattleShipManager() - получить ссылку на менеджер кораблей бота

и соответсующие им сетеры.

Класс *Serialization* служит для записи информации в json файл с использованием библиотеки nlohmann/json. Он имеет следующее поле:

- *nlohmann::json& j* – ссылка на структуру данных для работы с json.

Он имеет три одинаковых по структуре метода (stringify) для подготовки к записи в файл менеджера кораблей, поля и менеджера способностей.

Класс *Deserialization* служит для загрузки информации из json файла. Он имеет следующее поле:

- *nlohmann::json& j* – ссылка на структуру данных для работы с json.

Он имеет три одинаковых по структуре метода (parse) для загрузки из файла менеджера кораблей, поля и менеджера способностей.

Класс *Wrapper* является обёрткой над файлом с использованием идиомы RAII. Он имеет следующее поле:

- *fstream file* – поток для работы с файлом.

И следующие методы:

- *read(nlohmann::json& j)* – записывает содержимое файла в структуру json.
- *write(nlohmann::json& j)* – записывает содержимое структуры json в файл.

Класс *GameState* является классом состояния для связывания других классов и для реализации полной логики загрузки/сохранения игры. Он имеет следующие поля:

- *Player& player* – ссылка на игрока.
- *Bot& bot* – ссылка на бота.

И следующие методы:

- FileWrapper& operator<<(FileWrapper& fileWrapper, GameState& state) – переопределяет оператор << следующим образом: сначала происходит сериализация и вся необходимая информация по кораблям, полям и способностям сохраняется в библиотечную структуру, которая потом переносится в обёртку и она возвращается.
- *Wrapper& operator>>(Wrapper& fileWrapper, GameState& state)* – переопределяет оператор >> следующим образом: сначала происходит считывание информации из обёртки в структуру json, затем десериализация, информация записывается в временные объекты и позже переносится на используемые, в конце возвращается обёртка.
- *void loadGame(const string& file)* – создаёт обертку и заполняет объект класса информацией из файла.
- void saveGame(const string file) – очищает файл, создаёт обёртку и загружает в неё информацию из объекта класса.
- *int& getCurrentDamage()* – возвращает урон.
- *void setCurrentDamage(int damage)* – выставляет урон.
- Bot &getBot() - возвращает ссылку на бота
- User &getUser() - возвращает ссылку на пользователя
- int &getRounds() - возвращает ссылку на количество раунд
- void setGameComponents(std::unique_ptr<Bot> newBot, std::unique_ptr<User> newUser) - устанавливает бота и пользователя в состояние игры

Тестирование:

Происходит симуляция игры между игроком (сверху) и ботом (снизу), для этого используется большая часть реализованных методов внутри классов. Поле игрока изначально открыто, а вражеское скрыто. В начале хода игрок может использовать одну случайную способность или сразу перейти к атаке вражеского поля.

В классе *Game* реализована логика игры, которая позволяет выбирать действия в зависимости от команд пользователя. Он может: запустить игру, реализовав игровой цикл, с возможностью выйти обратно после использования способности; загрузить игру, получив состояния кораблей, поля и способностей; сохранить игру, уже записав состояния игровых сущностей; выйти из игры.

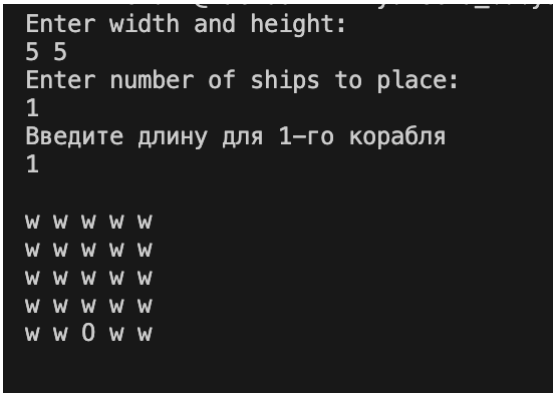

Рисунок 2 - ввод пользователем размеров поля, количества кораблей и длины кораблей

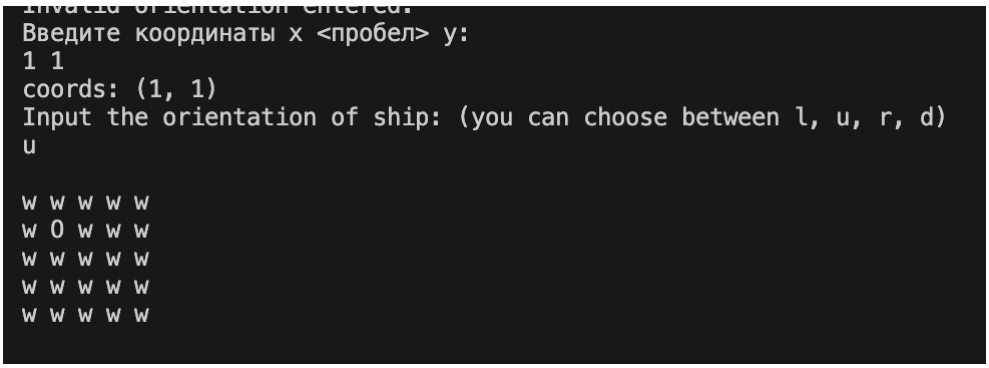

Рисунок 3 - ввод пользователем координат и направления корабля и расставление корабля в поле

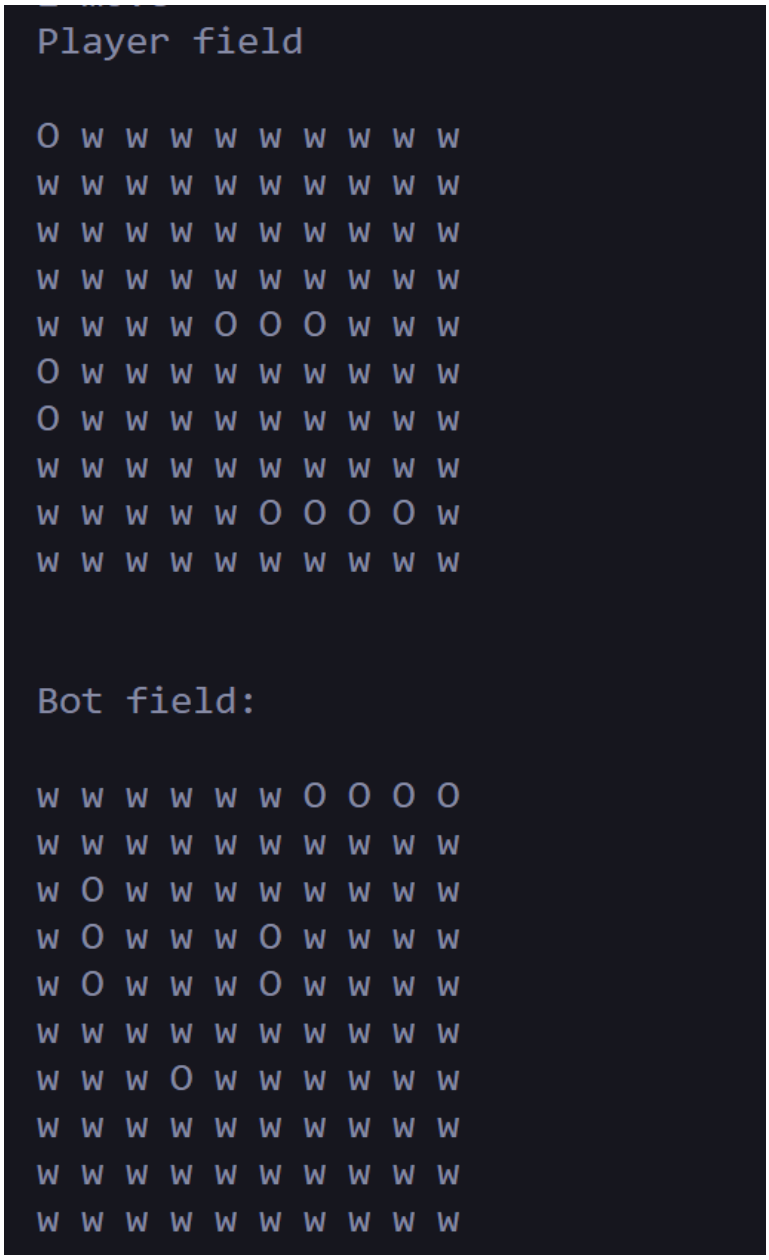

Рисунок 4 - игровые поля

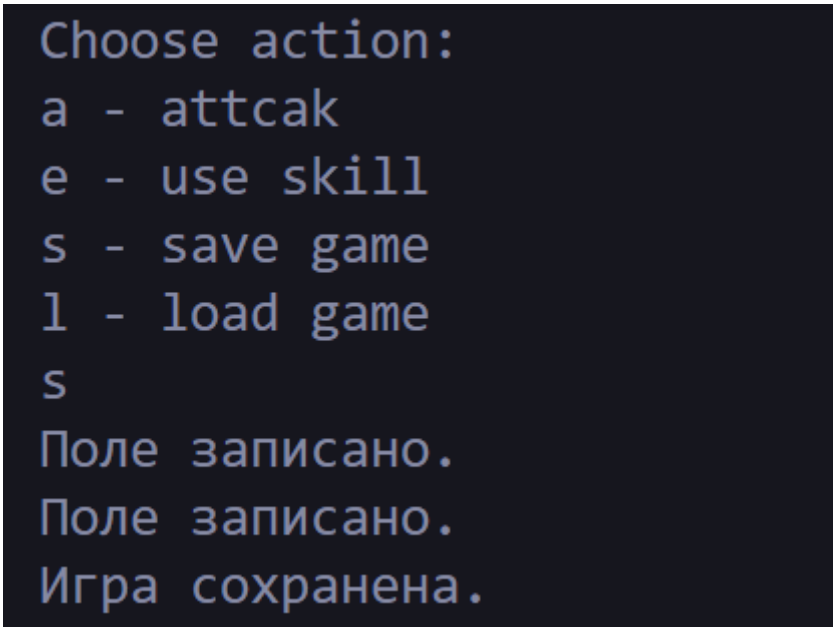

Рисунок 5 – Игра сохранена

Выводы

Во время выполнения лабораторной работы, было изучено связывание классов и созданные соответствующие заданию классы.