

**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. 3342

Иванов С.С.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2024

### **Цель работы**

Написать класс игры и класс состояния игры. Также реализовать сохранения состояния игры путём сериализации классов.

## Задание

- i) Создать класс игры, который реализует следующий игровой цикл:
  - a. Начало игры
  - b. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
  - c. В случае проигрыша пользователь начинает новую игру
  - d. В случае победы в раунде, начинается следующий раунд, причем состояние поля и способностей пользователя переносятся.
- i) 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.
- ii) 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.

## Выполнение работы

Класс Serializer:

Абстрактный класс сериализатора, что наследуется от Visitor.

Класс содержит следующие публичные методы:

- `void save(Owner *_ow);` - метод, отвечающий за сериализацию объектов;
- `const nlohmann::json& get() const noexcept;` - метод, который выдаёт сериализованный объект;
- `~Serializer() override = default;` - переопределённый деструктор;

Все наследники этого класса сериализуют объекты определённых классов, а именно:

- SkillManagerSerializer - для SkillManager;
- UnitSerializer - для Unit;
- FieldSizeSerializer - для Field::Size;
- ShipSerializer - для Ship;
- GameStateSerializer - для GameState;
- ShipManagerSerializer - для ShipManager;
- FieldSerializer - для Field;

Класс Loader:

Абстрактный класс загрузчика.

Класс содержит следующие методы:

- `virtual void load(const nlohmann::json &_json) = 0;` - метод, определяющий логику загрузки;
- `std::shared_ptr<Owner> get() const noexcept;` - метод, который выдаёт загруженный объект;
- `~Loader() override = default;` - переопределённый деструктор;

Все наследники этого класса загружают объекты определённых классов, а именно:

- SkillManagerLoader - для SkillManager;
- UnitLoader - для Unit;
- FieldSizeLoader - для Field::Size;
- ShipLoader - для Ship;
- GameStateLoader - для GameState;
- ShipManagerLoader - для ShipManager;
- FieldLoader - для Field;

Класс File:

Класс для указания и хранения имени файла.

Класс содержит следующие публичные методы:

- `const std::string& name(const std::string &name) noexcept;` - сеттер имени файла;
- `const std::string& name() const noexcept;` - геттер имени файла;

Класс GameState:

Класс состояния игры.

Класс содержит следующие публичные методы:

- `explicit GameState(const Field &_player_field, const Field &_enemy_field, const ShipManager &_player_ship_manager, const ShipManager &_enemy_ship_manager, const SkillManager &_player_skill_manager);`
- `explicit GameState(const Field &_player_field, const Field &_enemy_field);`
- `GameState();`
- `GameState(const GameState &other);`
- `GameState(GameState &&other) noexcept;`
- `~GameState() = default;`
- `GameState& operator=(const GameState& other);`
- `GameState& operator=(GameState&& other) noexcept;`
- `Field& get_player_field() noexcept;` - геттер поля игрока
- `Field& get_enemy_field() noexcept;` - геттер поля противника

- ShipManager& get\_player\_ship\_manager() noexcept; - геттер менеджера кораблей игрока
- ShipManager& get\_enemy\_ship\_manager() noexcept; - геттер менеджера кораблей врага
- SkillManager& get\_player\_skill\_manager() noexcept; - геттер менеджера скилов игрока
- void set\_player\_field(const Field &\_player\_field) noexcept; - сеттер поля игрока
- void set\_enemy\_field(const Field &\_enemy\_field) noexcept; - сеттер поля противника
- void set\_player\_ship\_manager(const ShipManager &\_player\_ship\_manager) noexcept; - сеттер менеджера кораблей игрока
- void set\_enemy\_ship\_manager(const ShipManager &\_enemy\_ship\_manager) noexcept; - сеттер менеджера кораблей врага
- void set\_player\_skill\_manager(const SkillManager &\_player\_skill\_manager) noexcept; - сеттер менеджера скилов игрока

Класс Game:

Класс игры.

Класс содержит следующие публичные методы:

- Game();
- void user\_attack(const Unit & \_unit); - атака пользователя
- SkillResultStatus user\_skill(const Unit &\_unit = Unit()); - использование скила, если он есть
- void load\_game(); - загрузка игры
- void save\_game(); - сохранение игры
- void new\_game(); - новая игра
- void bot\_attack(); - атака бота
- void setup\_bot(); - пересборка поля и менеджера кораблей бота
- GAME\_OVER\_FLAG is\_game\_over() noexcept; - проверка на победителя
- GameState& state() noexcept; - геттер состояния игры
- void set\_state(GameState & \_state) noexcept; - сеттер состояния игры
- ~Game() = default;

Архитектурные решения:

Были реализованы класс состояния игры, который отвечает за ресурсы игры и её состояния, также класс игры, который отвечает за процесс игры и реализует функционал, нужный для игрового цикла.

Также были записаны классы-сериализаторы и классы-лоадеры, которые отвечают за сохранение-загрузку данных игры. Было решено поместить их в отдельную динамическую библиотеку и подгружать их по необходимости, так как они не реализуют основной функционал игры.

Проверка работоспособности написанного кода:

```

int main()
{
    seagame::Game game_object;
    game_object.setup_bot();

    seagame::ShipManager &sm = game_object.state().get_player_ship_manager();
    seagame::Field &field = game_object.state().get_player_field();
    seagame::SkillManager &skill_manager = game_object.state().get_player_skill_manager();

    field.accept<seagame::SetupReactOfDestroyedShip>(
        std::make_shared<seagame::AddRandomSkill>(skill_manager)
    );

    std::uint64_t i;

    try
    {
        sm = seagame::ShipManager({
            seagame::Ship::Len::TWO,
            seagame::Ship::Len::FOUR
        });

        i = sm.new_ship(
            seagame::Ship::Len::THREE,
            seagame::Ship::Orientation::VERTICAL
        );
    }
    catch (const std::invalid_argument& err)
    {
        std::cerr << "Error: " << err.what() << std::endl;
        i = -1;
    }

    try
    {
        field = seagame::Field(5, 5);
    } catch (const std::invalid_argument& err)
    {
        std::cerr << "Error: " << err.what() << std::endl;
    }

    try
    {
        if (i != -1) field.add_ship(sm[i], Unit(2, 1));
        field.add_ship(sm[0], Unit(4, 2));
        field.add_ship(sm[1], Unit(2, 5));
    } catch (const seagame::PlacementError &err)
    {
        std::cerr << "Error: " << err.what() << std::endl;
    }
}

```



```

try
{
    game_object.user_attack(Unit(9, 6));
    game_object.user_attack(Unit(9, 6));
} catch (const std::invalid_argument &err)
{
    std::cerr << "Error: " << err.what() << std::endl;
}

try
{
    seagame::SkillResultStatus status = game_object.user_skill(Unit(3, 5));
    std::cout << "Skill result: " << status << std::endl;
} catch(const seagame::ExtractError&)
{
    std::cerr << "No skills" << std::endl;
}

game_object.bot_attack();

std::cout << "-----" << std::endl;

std::cout << sm[i].segments()[0] << ' ';
std::cout << sm[i].segments()[1] << ' ';
std::cout << sm[i].segments()[2] << std::endl;

std::cout << sm[0].segments()[0] << ' ';
std::cout << sm[0].segments()[1] << std::endl;

std::cout << sm[1].segments()[0] << ' ';
std::cout << sm[1].segments()[1] << ' ';
std::cout << sm[1].segments()[2] << ' ';
std::cout << sm[1].segments()[3] << std::endl;

std::cout << "-----" << std::endl;

std::cout << "skill_manager len: " << skill_manager.empty() << std::endl;

std::cout << "SERIALIZE TEST" << std::endl;

try
{
    game_object.save_game();
} catch(const std::exception&)
{
    std::cerr << "bad save" << std::endl;
}

std::cout << "-----" << std::endl;

std::cout << "LOAD TEST" << std::endl;

try
{
    game_object.load_game();
} catch(const std::exception&)
{
    std::cerr << "bad load" << std::endl;
}

```

Компиляция:

```

C:\soft\sea-battle\build (main -> origin)
λ build.cmd
-- The CXX compiler identification is Clang 19.1.3
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: D:/llvm-mingw-20241030-msvcrt-x86_64/bin/c++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- The C compiler identification is Clang 19.1.3
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: D:/llvm-mingw-20241030-msvcrt-x86_64/bin/cc.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done (1.3s)
-- Generating done (0.1s)
-- Build files have been written to: C:/soft/sea-battle/build
[ 6%] Built target utils
[ 58%] Built target seagame_lib
[ 60%] Building CXX object CMakeFiles/seagame.dir/main.cpp.obj
[ 62%] Linking CXX executable seagame.exe
[ 62%] Built target seagame
[100%] Built target serialization
C:\soft\sea-battle\build (main -> origin)
λ

```

Запуск:

```

C:\soft\sea-battle\build (main -> origin)
λ seagame
Skill result: Successful
-----
2 2 2
2 2
2 2 2 2
-----
skill_manager len: 0
SERIALIZE TEST
-----
LOAD TEST
-----
C:\soft\sea-battle\build (main -> origin)
λ

```

UML:



## **Выводы**

В ходе выполнения работы были написаны класс игры и класс состояния игры. Также реализовано сохранения состояния игры путём сериализации классов.