

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. 3383

Солдунова Е.П.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы

Изучение связывания классов в объектно-ориентированном программировании для реализации модифицированной игры “Морской бой” в соответствии с её принципами на языке C++.

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 самой игрой можно использовать обертки над командами
- При работе с файлом используйте идиому RAII.

Выполнение работы

1. Подключение необходимых библиотек

- `#include <fstream>` – библиотека для чтения и записи данных в файл.

2. Описание реализованных классов и их методов

а. Класс *Game*:

Класс *Game* содержит в полях основные атрибуты игры, такие как поля игрока и противника, корабли игрока и противника, неатакованные клетки и способности игрока.

Класс имеет только одно поле, которое хранит объект класса *GameState*, через который оперирует всеми составляющими игры.

Методы класса:

- Конструктор *Game()*:

Инициализирует объект класса *Game* с переданным начальным состоянием игры *GameState*. Проверяет, нужно ли загружать игру из сохранения или начать новую: если игра загружается из сохранения, то загружает состояние игры и выводит сообщение об успешной загрузке. Иначе происходит инициализация игрового поля игрока, добавление кораблей игрока, инициализация менеджера способностей в соответствии с вводимыми пользователем данными. Также вызывается метод *EnemyInitialization()* для инициализации поля и кораблей противника.

- Метод *StartGame()*:

Метод, который запускает игру. Выводит информацию о начале игры и правилах ввода команд. Затем вызывает метод *Round()* для проведения игрового раунда. Если результатом раунда является победа, вызывается метод *EnemyInitialization()* для новой инициализации полей противника. Если результатом раунда является поражение, создается новый объект

класса *GameState* и *Game* запускается новая игра. Цикл выполняется до тех пор, пока пользователь не запросит выход из игры.

- Метод *Round()*:

Выполняет ход игрока и противника поочередно, пока один из них не победит. Возвращает результат раунда (победа игрока, поражение игрока или выход).

- Метод *PlayerMove()*:

Предлагает игроку использовать способность: если игрок соглашается, применяет способность к полю противника.

Производит атаку клетки поля противника, введенную игроком. После каждого хода ожидает ввод от игрока команды загрузки, сохранения, сохранения с выходом, выхода или продолжения игры.

- Метод *EnemyMove()*:

Выбирает случайные координаты для атаки на поле игрока. Проверяет, не была ли выбранная клетка атакована ранее. Выполняет атаку на поле игрока и обрабатывает результат.

- Метод *InputInts(int num)*:

Запрашивает ввод целочисленных значений от пользователя и возвращает введенные значения в виде вектора. Используется для избегания дублирования кода при вводе различных данных пользователем. Метод используется для получения координат клеток, размеров кораблей в других методах.

- Метод *EnemyInitialization()*:

Инициализирует новое поле и корабли противника, в размере и количестве поля и кораблей игрока. Размещает корабли на поле противника случайным образом. Инициализирует массив неатакованных клеток.

б. Класс *GameState*:

Класс *GameState* представляет игровое состояние, включающее в себя информацию о расстановке кораблей, полях, способностях.

Поля класса:

- *string saving* – строковое поле, хранит состояние сохранения игры: *empty*, если сохранений нет, и *saving* если в файле существует сохранение.
- *ShipManager player_manager* – объект типа *ShipManager*, который отвечает за управление кораблями игрока.
- *ShipManager enemy_manager* – объект типа *ShipManager*, который отвечает за управление кораблями противника.
- *Field player_field* – объект типа *Field*, представляющий игровое поле игрока.
- *Field enemy_field* – объект типа *Field*, представляющий игровое поле противника.
- *AbilityManager* abilities* – указатель на объект типа *AbilityManager*, который управляет способностями, доступными игрокам.
- *vector<vector <int>> unattacked_cells* – двумерный вектор, который представляет собой список ячеек поля, которые еще не были атакованы противником.
- *string filename = "saving"* – строковая переменная, которая по умолчанию содержит значение *"saving"*. Предназначена для указания имени файла, где будет сохраняться состояние игры.

Методы класса:

- Конструктор класса *GameState*:

Принимает объект типа *Game* и инициализирует менеджеры кораблей для игрока и врага, игровые поля для игрока и врага, статус

сохранения, а также менеджер способностей и список неатакованных клеток.

- Метод *save(const string& filename):*

Сохраняет текущее состояние игры в файл с указанным именем. Метод открывает файл для записи, записывает информацию о состоянии игры в файл и закрывает файл.

- Метод *load(const string& filename):*

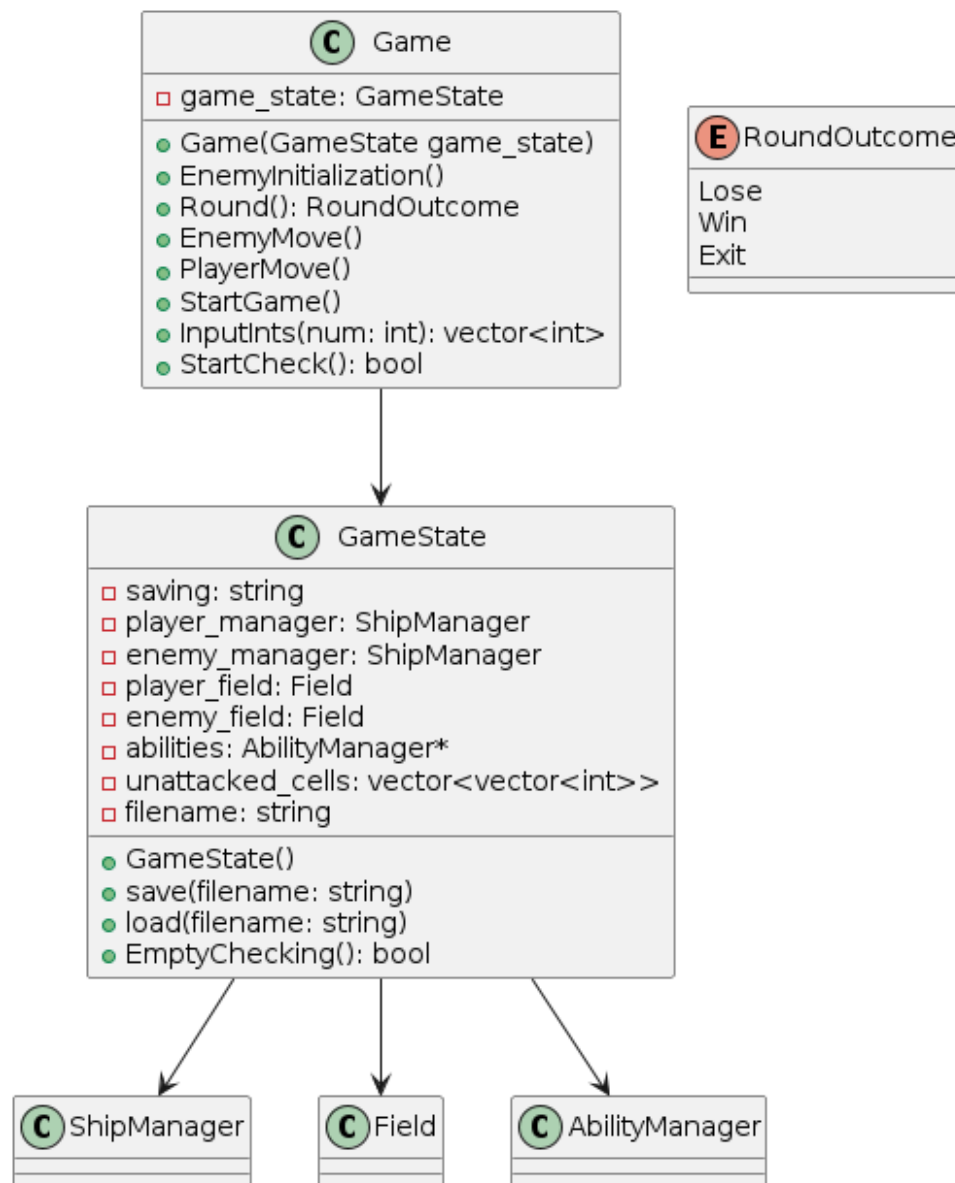
Загружает игровое состояние из файла с указанным именем. Метод открывает файл для чтения, считывает информацию о состоянии игры из файла и закрывает файл.

- Метод *bool GameState::EmptyChecking():*

Проверяет, есть ли сохранение в текущем объекте *GameState*. Если сохранение пустое, метод выводит сообщение об отсутствии сохранения и возвращает *false*, иначе возвращает *true*.

Операторы `<<` и `>>` перегружены для сериализации и десериализации объекта *GameState*. Оператор `<<` записывает информацию о состоянии игры в поток вывода, включая размеры полей, расстановку кораблей, статус клеток на поле и список неатакованных клеток. Оператор `>>` считывает информацию из потока ввода и создает объект *GameState* на основе этой информации.

UML реализованных классов



Вывод

В ходе выполнения лабораторной работы было изучено и реализовано связывание классов для создания модифицированной игры “Морской бой” на языке C++.

ПРИЛОЖЕНИЕ А

Исходный код программы

Game.h

```
#ifndef GAME_H
#define GAME_H

#include "GameState.h"

enum class RoundOutcome {Lose, Win, Exit};

class Game{
public:
    GameState game_state;
    Game(GameState game_state);
    void EnemyInitialization();
    RoundOutcome Round();
    void EnemyMove();
    void PlayerMove();
    void StartGame();
    vector<int> InputInts(int num);
    bool StartCheck();
};

#endif
```

Game.cpp

```
#include "Game.h"

Game::Game(GameState game):game_state(game){
    if (StartCheck()){
        cout << "Hello! Before starting the new game, set the height
        and width of the fields:" << endl;
```

```

vector<int> sizes = InputInts(2);
while (true){
    try{
        Field new_field(sizes[0], sizes[1]);
        break;
    }
    catch (const FieldSizeException& ex){
        cout << ex.getMessage() << endl;
        sizes = InputInts(2);
    }
}

game_state.player_field = Field(sizes[0], sizes[1],
FieldType::Users);
cout << "The creation of the field was successful!" << endl;

cout << "Enter the number of ships:" << endl;
int quantity = InputInts(1)[0];
while((quantity<1)|| (quantity>(game_state.player_field.GetWidth()
*game_state.player_field.GetHeight()/2))){
    cout << "The quantity of the ships cannot be negative or zero,
please try again: ";
    quantity = InputInts(1)[0];
}
game_state.player_field.Draw();

cout << "Enter sizes of ships. After each size, determine the
position of the ship: enter v if you want the ship to be
positioned vertically or h if horizontally" << endl;
vector<int> ships(quantity);
vector<char> orient(quantity);
int i = 0;

```

```

game_state.player_manager = ShipManager(0, {});

while (i < quantity){
ships[i] = InputInts(1)[0];
if (ships[i] < 1){cout << "The length of the ship cannot be
negative or zero, please try again: " << endl;}
else {
while(!(cin >> orient[i])){
cout << "The entered data is
incorrect, please try again: ";
cin.clear();

cin.ignore(numeric_limits<streamsize>::max(), '\n');
}
if
(orient[i]=='v'){game_state.player_manager.AddShip(ships[i]);}
if
(orient[i]=='h'){game_state.player_manager.AddShip(ships[i],
Orientation::Horizontal);}
if ((orient[i]!='v')&&(orient[i]!='h')){
cout << "You entered an incorrect orientation symbol, so the
ship is set vertically by default." << endl;
game_state.player_manager.AddShip(ships[i]);
}
i++;
}
}

cout << "For each of the created ships, set the coordinates of
its heads:" << endl;
i = 0;
int flag = 0;

```

```

vector<int> coordinates;
while (i < quantity){
coordinates = InputInts(2);
try{
game_state.player_field.PutShip(coordinates[0],
coordinates[1], game_state.player_manager.GetShip(i));
i++;
}
catch (const IncorrectPlacementException& ex){
cout << ex.getMessage() << endl;
flag++;
if (flag == 3){
game_state.player_manager.DeleteShip(i);
cout << "You have placed the ship incorrectly three times.
Perhaps it was too big, so it was deleted. You can continue to
place the remaining ships." << endl;
flag = 0;
i++;
}
}
}
game_state.player_field.Draw();
game_state.abilities = new AbilityManager;
this->EnemyInitialization();
}
}

void Game::StartGame(){
cout << "The new game has started!" << endl;
cout << "Before each round, you will be asked to enter a
command:" << endl;

```

```

cout << "-if you want to end the game with saving, enter
'save_exit';" << endl;
cout << "-if you want to exit without saving, enter 'exit';"
<< endl;
cout << "-if you want to save the game and continue, enter
'save';" << endl;
cout << "-if you want to load a previous save, enter 'load';"
<< endl;
cout << "-if you want to continue the game, enter any line."
<< endl;
RoundOutcome outcome;
outcome = this->Round();
while (outcome != RoundOutcome::Exit){
if (outcome == RoundOutcome::Win){
this->EnemyInitialization();
outcome = this->Round();
} else if (outcome == RoundOutcome::Lose){
GameState new_state;
Game new_game(new_state);
new_game.StartGame();
}
}

void Game::EnemyInitialization(){
Field          new_field(game_state.player_field.GetHeight(),
game_state.player_field.GetWidth(), FieldType::Enemy);
game_state.enemy_field = new_field;
game_state.enemy_field.Draw();
vector<int>
enemy_ships_size(game_state.player_manager.GetShipsNumber());

```

```

for (int i = 0; i <
game_state.player_manager.GetShipsNumber();
i++){enemy_ships_size[i] =
(*game_state.player_manager.GetShip(i)).GetLenght();}
ShipManager new_ships =
ShipManager(game_state.player_manager.GetShipsNumber(),
enemy_ships_size);
game_state.enemy_manager = new_ships;
int x;
int y;
int i = 0;
int flag = 0;
while (i < game_state.enemy_manager.GetShipsNumber()){
x = rand() % game_state.enemy_field.GetHeight();
y = rand() % game_state.enemy_field.GetWidth();
try{
game_state.enemy_field.PutShip(x, y,
game_state.enemy_manager.GetShip(i));
i++;
flag = 0;
}
catch (const IncorrectPlacementException& ex){
flag++;
if (flag == 10){
game_state.enemy_manager.DeleteShip(i);
flag = 0;
i++;
}
}
}
vector<vector<int>> cells(game_state.player_field.GetHeight(),
vector<int>(game_state.player_field.GetWidth(), 1));

```

```

game_state.unattacked_cells = cells;
}

RoundOutcome Game::Round() {
while (true) {
PlayerMove();
game_state.enemy_field.Draw();
string command;
cout << "Enter a command: " << endl;
cin >> command;
if (command == "save_exit") {
game_state.save("saving");
return RoundOutcome::Exit;
}
if (command == "exit") {return RoundOutcome::Exit;}
if (command == "save") {game_state.save("saving");}
if (command == "load") {game_state.load("saving");}
if (!game_state.enemy_manager.GetFleetState()) {return
RoundOutcome::Win;}
EnemyMove();
game_state.player_field.Draw();
if (!game_state.player_manager.GetFleetState()) {return
RoundOutcome::Lose;}
}
}

void Game::PlayerMove() {
string ability;
cout << "Do you want to apply the ability? Yes - enter yes, No
- enter any line: " << endl;
cin >> ability;
if (ability == "yes") {

```



```

try{
if
 ((*game_state.abilities).ApplicationOfAbilities(game_state.enemy_field,
game_state.enemy_manager))
(*game_state.abilities).AddAbility();
}
catch (const NoAbilityException& ex){cout << ex.getMessage()
<< endl;}
}
cout << "Enter the coordinates of the cell you want to
attack:" << endl;
vector<int> coordinates = InputInts(2);
int x = coordinates[0];
int y = coordinates[1];
try{
if(game_state.enemy_field.Attack(x, y)){
if(game_state.enemy_field.FieldGetShipStatus(x,
y)==ShipStatus::Destroyed){(*game_state.abilities).AddAbility(
);}
}
}
catch (const OutsideAttackException& ex){cout <<
ex.getMessage() << endl;}
}

void Game::EnemyMove(){
int attack_y = rand() % game_state.player_field.GetHeight();
int attack_x = rand() % game_state.player_field.GetWidth();
while (game_state.unattacked_cells[attack_y][attack_x] != 1){
attack_y = rand() % game_state.player_field.GetHeight();
attack_x = rand() % game_state.player_field.GetWidth();
}
}

```

```

if (game_state.player_field.Attack(attack_x, attack_y)){
if      (game_state.player_field.FieldGetShipStatus(attack_x,
attack_y)                                     ==
ShipStatus::Destroyed){game_state.unattacked_cells[attack_y][a
ttack_x] = 0;}
}
if (game_state.player_field.GetCellStatus(attack_x,  attack_y)
==
Status::Empty){game_state.unattacked_cells[attack_y][attack_x]
= 0;}
}

```

```

vector<int> Game::InputInts(int num){
vector<int> values(num);
if (num == 2){
while(!(cin >> values[0] >> values[1])){
cout << "The entered data is incorrect, please try again: ";
cin.clear();
cin.ignore(numeric_limits<streamsize>::max(), '\n');
}
}
if (num == 1){
while(!(cin >> values[0])){
cout << "The entered data is incorrect, please try again: ";
cin.clear();
cin.ignore(numeric_limits<streamsize>::max(), '\n');
}
}
return values;
}

```

```

bool Game::StartCheck(){

```

```

string loading;
cout << "Do you want to load the game? Yes - enter yes, No -
enter any line: " << endl;
cin >> loading;
if (loading == "yes"){
game_state.load("saving");
if (game_state.EmptyChecking()){
cout << "Your game is succesfully load!" << endl;
return 0;
}
}
return 1;
}

```

GameState.h

```

#ifndef GAMESTATE_H
#define GAMESTATE_H

#include <fstream>
#include "Ship.h"
#include "ShipManager.h"
#include "Field.h"
#include "Ability.h"
#include "AbilityManager.h"
#include "Exceptions.h"

class GameState{
public:
string saving;
ShipManager player_manager;
ShipManager enemy_manager;
Field player_field;

```

```

Field enemy_field;
AbilityManager* abilities;
vector<vector <int>> unattacked_cells;
string filename = "saving";

GameState();
void save(const string& filename);
void load(const string& filename);
friend ostream& operator<<(ostream& os, GameState& gameState);
friend istream& operator>>(istream& is, GameState& gameState);
bool EmptyChecking();
};

#endif

```

GameState.cpp

```

#include "GameState.h"

GameState::GameState(): saving("empty"), player_field(2, 2),
player_manager(0, {}), enemy_field(2, 2), enemy_manager(0,
{}), unattacked_cells({}), abilities(new AbilityManager){}

void GameState::save(const string& filename) {
ofstream outFile(filename);
    if (outFile.is_open()) {
        outFile << *this;
        outFile.close();
    }
    else {
        cerr << "File not open!" << endl;
    }
}

```

```

void GameState::load(const string& filename) {
    ifstream inFile(filename);
    if (inFile.is_open()) {
        inFile >> *this;
        inFile.close();
    } else {
        cerr << "File not open!" << endl;
    }
}

ostream& operator<<(ostream& os, GameState& gameState) {
    os << "saving" << "\n";
    os <<   gameState.player_field.GetHeight()   <<   "   "   <<
gameState.player_field.GetWidth() << "\n";
    os << gameState.player_manager.GetShipsNumber() << "\n";
    for (int i = 0; i < gameState.player_manager.GetShipsNumber();
i++){
        os <<   gameState.player_field.heads[i][0]   <<   "   "   <<
gameState.player_field.heads[i][1] << " ";
        if ((*gameState.player_manager.GetShip(i)).GetOrientation() ==
Orientation::Vertical){os << "v "};
        else {os << "h "};
        os << (*gameState.player_manager.GetShip(i)).GetLenght() << "
";
        for          (int          j          =          0;          j          <
(*gameState.player_manager.GetShip(i)).GetLenght(); j++){
            os
            <<
            (*gameState.player_manager.GetShip(i)).GetSegmentStatus(j) <<
" ";
        }
        os << "\n";
    }
}

```

```

}
os << (*gameState.abilities).GetLength() << "\n";
os << gameState.enemy_manager.GetShipsNumber() << "\n";
for (int i = 0; i < gameState.enemy_manager.GetShipsNumber();
i++){
os << gameState.enemy_field.heads[i][0] << " " <<
gameState.enemy_field.heads[i][1] << " ";
if ((*gameState.enemy_manager.GetShip(i)).GetOrientation() ==
Orientation::Vertical){os << "v "};
else {os << "h "};
os << (*gameState.enemy_manager.GetShip(i)).GetLenght() << "
";
for (int j = 0; j <
(*gameState.enemy_manager.GetShip(i)).GetLenght(); j++){os <<
(*gameState.enemy_manager.GetShip(i)).GetSegmentStatus(j) << "
";}
os << "\n";
}
for (int i = 0; i < gameState.enemy_field.GetHeight(); i++){
for (int j = 0; j < gameState.enemy_field.GetWidth(); j++){
if (gameState.enemy_field.GetCellStatus(j, i) ==
Status::Unknown){os << "? "};
else {os << "+ "};
}
os << "\n";
}
for (int i = 0; i < gameState.unattacked_cells.size(); i++){
for (int j = 0; j < gameState.unattacked_cells[0].size();
j++){os << gameState.unattacked_cells[i][j] << " ";}
}
return os;
}

```

```

istream& operator>>(istream& is, GameState& gameState) {
    is >> gameState.saving;
    int height;
    int width;
    is >> height >> width;
    gameState.player_field = Field(height, width,
    FieldType::Users);
    gameState.enemy_field = Field(height, width,
    FieldType::Users);

    int player_ships_number;
    is >> player_ships_number;

    gameState.player_manager = ShipManager({0, {}});
    char orientation;
    int ship_len;
    vector<int> head(2);

    for (int i = 0; i < player_ships_number; i++){
        is >> head[0] >> head[1];
        is >> orientation;
        is >> ship_len;
        vector<int> ship_segments(ship_len);
        for (int j = 0; j < ship_len; j++){is >> ship_segments[j];}
        if (orientation ==
        'v'){gameState.player_manager.AddShip(ship_len,
        Orientation::Vertical, ship_segments);} //сегменты
        копируются как то треш
        else{gameState.player_manager.AddShip(ship_len,
        Orientation::Horizontal, ship_segments);}
    }
}

```

```

gameState.player_field.PutShip(head[0], head[1],
gameState.player_manager.GetShip(i));
}

int ability_quantity;
is >> ability_quantity;
gameState.abilities = new AbilityManager;
    if (ability_quantity < 3){
        for (size_t i = 3; i > ability_quantity; i--)
    }{(*gameState.abilities).DeleteAbility();}
    }
    if (ability_quantity > 3){
        for (int i = 3; i < ability_quantity;
i++){(*gameState.abilities).AddAbility();}
    }

int enemy_ships_number;
is >> enemy_ships_number;

gameState.enemy_manager = ShipManager({0, {}});

for (int i = 0; i < enemy_ships_number; i++){
is >> head[0] >> head[1];
is >> orientation;
is >> ship_len;
vector<int> ship_segments(ship_len);
for (int j = 0; j < ship_len; j++){is >> ship_segments[j];}
if (orientation ==
'v'){gameState.enemy_manager.AddShip(ship_len,
Orientation::Vertical, ship_segments);}
else{gameState.enemy_manager.AddShip(ship_len,
Orientation::Horizontal, ship_segments);}

```



```

gameState.enemy_field.PutShip(head[0], head[1],
gameState.enemy_manager.GetShip(i));
}

for (int i = 0; i < gameState.enemy_field.GetHeight(); i++){
for (int j = 0; j < gameState.enemy_field.GetWidth(); j++){
char status;
is >> status;
if (status ==
'?'){gameState.enemy_field.ChangeStatusUnknown(j, i);}
}
}
gameState.enemy_field.ChangeFieldType();

vector<vector<int>>
new_unattacked_cells(gameState.enemy_field.GetHeight(),
vector<int>(gameState.enemy_field.GetWidth()));
for (int i = 0; i < gameState.enemy_field.GetHeight(); i++){
for (int j = 0; j < gameState.enemy_field.GetWidth();
j++){is >> new_unattacked_cells[i][j];}
}
gameState.unattacked_cells = new_unattacked_cells;

return is;
}

bool GameState::EmptyChecking(){
if (saving == "empty"){
cout << "Your don't have any saves.\n";
return 0;
}
return 1;
}

```

}