**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В. И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: Шаблонны классы**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Первалов П.И. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## **Цель работы**

Реализовать шаблонные классы для управления процессом игры.

## **Задание**

Создать шаблонный класс управления игрой. Данный класс должен содержать ссылку на игру. В качестве параметра шаблона должен указываться класс, который определяет способ ввода команда, и переводящий введенную информацию в команду. Класс управления игрой, должен получать команду для выполнения, и вызывать соответствующий метод класса игры.

Создать шаблонный класс отображения игры. Данный класс реагирует на изменения в игре, и производит отрисовку игры. То, как происходит отрисовка игры определяется классом переданном в качестве параметра шаблона.

Реализовать класс считывающий ввод пользователя из терминала и преобразующий ввод в команду. Соответствие команды введенному символу должно задаваться из файла. Если невозможно считать из файла, то управление задается по умолчанию.

Реализовать класс, отвечающий за отрисовку поля.

Примечание:

* Класс отслеживания и класс отрисовки рекомендуется делать отдельными сущностями. Таким образом, класс отслеживания инициализирует отрисовку, и при необходимости можно заменить отрисовку (например, на GUI) без изменения самого отслеживания
* После считывания клавиши, считанный символ должен сразу обрабатываться, и далее работа должна проводить с сущностью, которая представляет команду.
* Для представления команды можно разработать системы классов или использовать перечисление enum.
* Хорошей практикой является создание “прослойки” между считыванием/обработкой команды и классом игры, которая сопоставляет команду и вызываемым методом игры. Существуют альтернативные решения без явной “прослойки”
* При считывания управления необходимо делать проверку, что на все команды назначена клавиша, что на одну клавишу не назначено две команды, что на одну команду не назначено две клавиши.

## **Выполнение работы**

В процессе продолжения разработки игры "Морской бой" были реализованы классы для организации управления игровым процессом: GameController, GameDisplay, TerminalInputProcessor, TerminalRenderer.

**Класс GameController**

Класс предназначен для управления игровым процессом через обработку команд пользователя. Игрок вводит команды (например, атака, сохранение, загрузка или использование способности), которые обрабатываются игровым контроллером, а затем выполняются соответствующие действия в объекте игры.

Краткая реализация:

* Шаблонный класс GameController используется для связывания объекта игры и процессора ввода, что позволяет гибко задавать способ получения команд.
* Конструктор класса принимает указатель на объект игры и объект процессора ввода, инициализируя внутренние члены класса.
* Метод process\_command:

1. Получает команду от процессора ввода.
2. Выполняет соответствующее действие в объекте игры через вызов метода, например attack(), save() и т. д.
3. Если команда неизвестна, выводится сообщение об ошибке.

**Класс GameDisplay**

Класс GameDisplay отвечает за визуализацию состояния игры. Он связывает объект игры с рендерером, который отвечает за отрисовку. Это позволяет отделить игровую логику от процесса отображения, обеспечивая гибкость и модульность программы.

Краткая реализация:

* Класс шаблонный, параметризованный типом Renderer, что позволяет использовать разные способы визуализации.
* Конструктор принимает указатель на объект игры (Game) и экземпляр рендерера, инициализируя соответствующие члены.
* Метод render вызывает метод render у объекта рендерера, передавая ему указатель на игру для отображения текущего состояния.

**Класс TerminalInputProcessor**

Класс TerminalInputProcessor используется для обработки ввода с клавиатуры в игровом приложении. Он позволяет связывать нажатия клавиш с командами, загружая настройки из файла, либо используя предустановленные команды. Это обеспечивает гибкость и удобство настройки управления.

Краткая реализация:

* Класс содержит приватное поле command\_map, которое хранит сопоставление клавиш с командами.
* Метод load\_commands\_from\_file загружает настройки управления из указанного файла. Если файл недоступен, применяется стандартный набор команд (attack, load, save, use\_ability).
* Конструктор вызывает метод загрузки, инициализируя управление с учётом переданного файла настроек.
* Метод get\_command считывает символ с консоли, который интерпретируется как команда пользователя.
* В случае ошибок загрузки настроек из файла выводится предупреждение, и управление переключается на стандартные настройки.

**Класс TerminalRenderer**

Класс TerminalRenderer отвечает за отображение состояния игры в терминале. Он преобразует данные игры в визуальное представление для игрока, выводя информацию о текущем игровом процессе в консоль. Этот класс используется для реализации рендеринга в текстовом интерфейсе.

Краткая реализация:

* Класс реализует метод render, который принимает указатель на объект игры.
* В методе render выводится сообщение о рендеринге игрового поля. Затем вызывается метод display\_playing\_fields у объекта игры, который отображает текущее игровое поле. Это позволяет пользователю видеть состояние игры в терминале.
* Рендеринг ограничивается выводом информации в текстовом формате, соответствующем терминальному интерфейсу.

Класс ориентирован на простое текстовое отображение данных игры, обеспечивая вывод информации о текущем игровом процессе в консоль.

**Архитектурные решения**

Разделение ответственности (Separation of Concerns):

В проекте чётко разделены различные компоненты, каждый из которых отвечает за свою часть функционала:

GameController — управляет логикой взаимодействия с игрой, обрабатывая команды от пользователя.

GameDisplay — отображает состояние игры, разделяя логику игры и визуализацию, что способствует расширяемости.

TerminalInputProcessor — обрабатывает пользовательский ввод, предоставляя гибкую настройку управления через файл или с помощью стандартных команд.

TerminalRenderer — отвечает за отображение состояния игры в текстовом виде, позволяя выводить данные на экран.

Использование шаблонов:

Класс GameController является шаблонным и может работать с любым типом процессора ввода, что позволяет легко интегрировать различные способы получения команд. Это решение повышает гибкость и переиспользуемость кода.

Гибкость и конфигурируемость:

Класс TerminalInputProcessor позволяет загружать настройки управления из внешнего файла. Это даёт возможность изменять поведение игры без необходимости менять исходный код, а также использовать стандартные или пользовательские настройки управления. Подход с файлом конфигурации способствует лёгкой настройке и адаптации игры под разные предпочтения пользователей.

Модульность и расширяемость:

Каждый из классов выполняет отдельную задачу, и изменения в одном компоненте (например, добавление новых команд или изменение способа рендеринга) не влияют на остальные части программы. Это облегчает добавление новых функциональностей, например, новых команд или способов отображения.

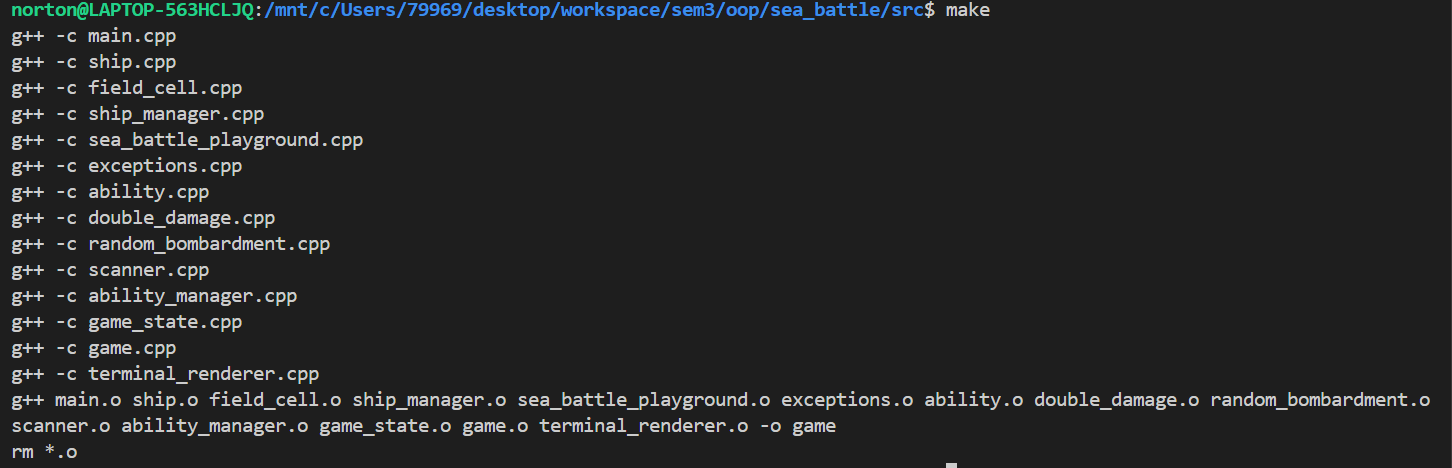
Обработка ошибок:

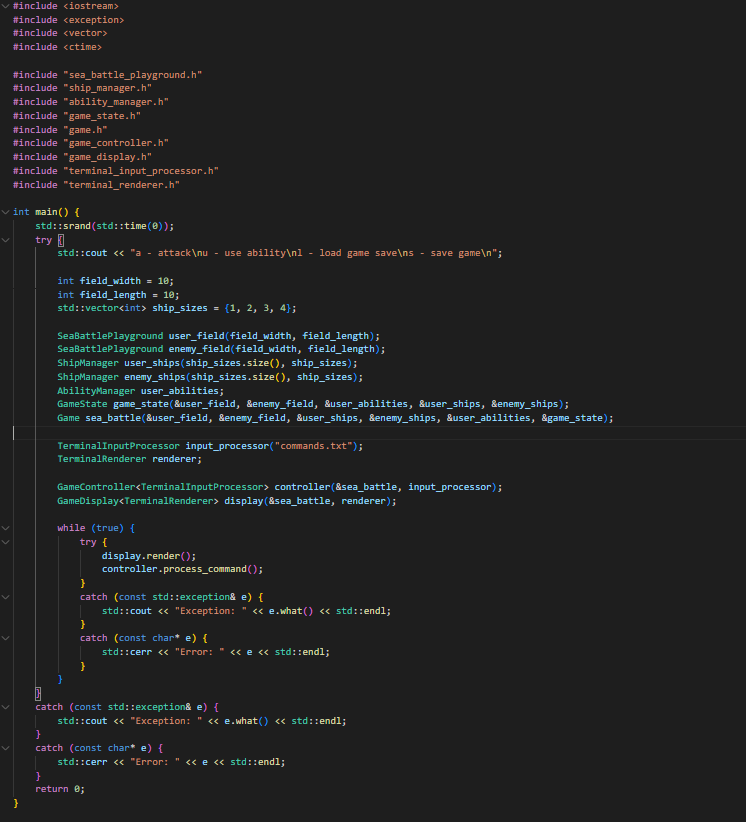
В классе TerminalInputProcessor предусмотрена обработка ошибок при открытии файла команд, что повышает надёжность приложения. В случае ошибки загружаются дефолтные команды, что гарантирует непрерывную работу программы.

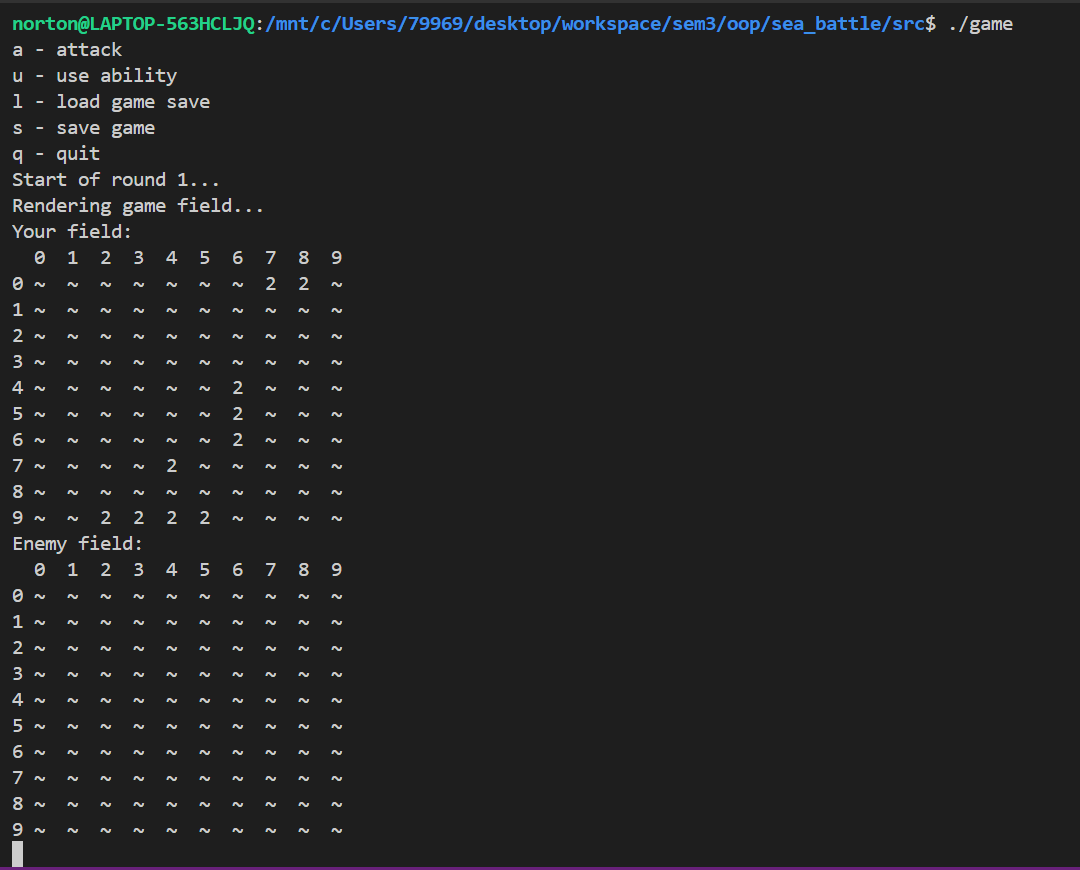
Текстовый интерфейс:

Все классы, работающие с визуализацией и вводом, ориентированы на текстовый интерфейс, что упрощает работу с терминалом и делает систему доступной для разработчиков с минимальными требованиями к графике.

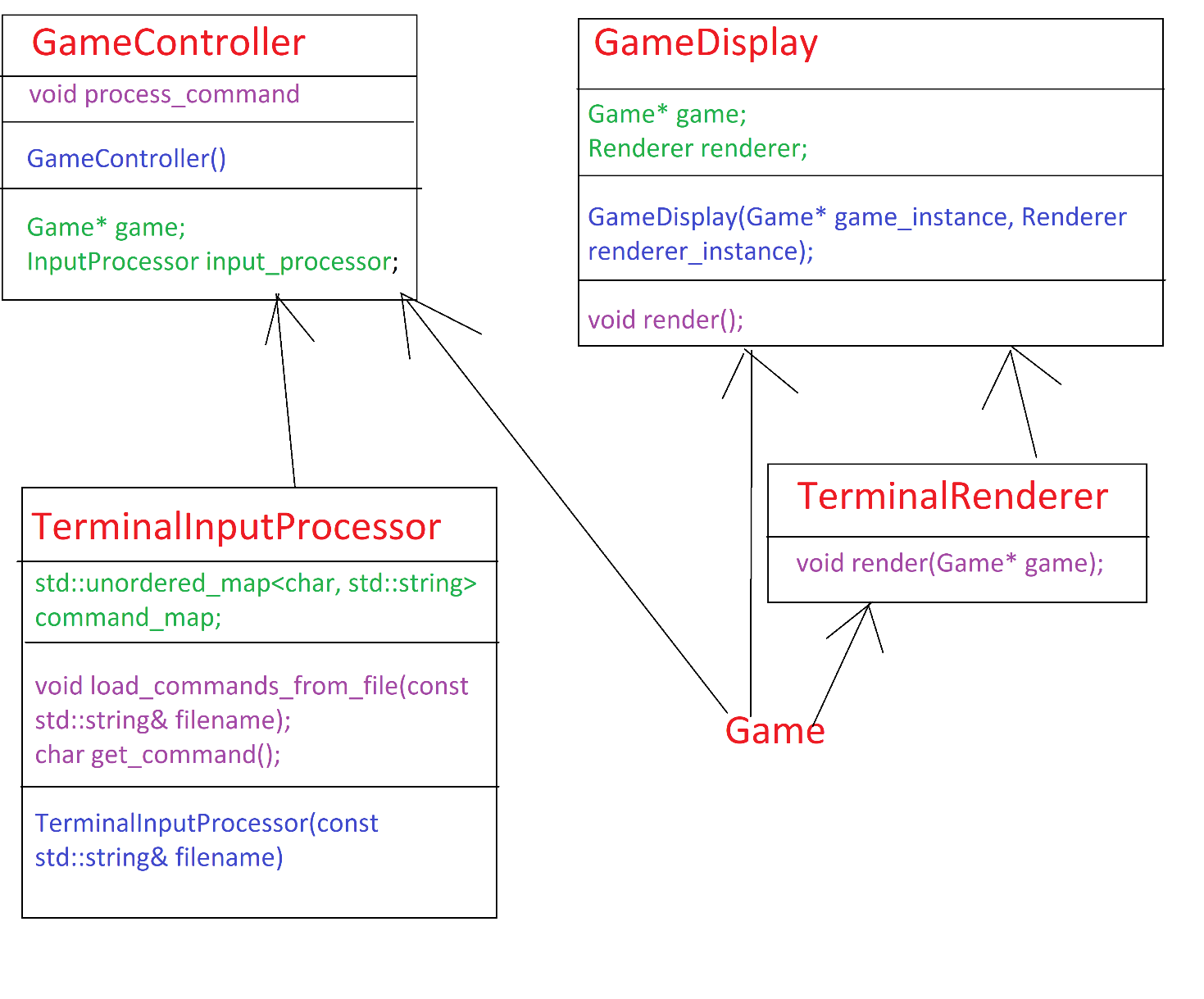
Интерфейсы и взаимодействие:

Классы взаимодействуют между собой через чётко определённые интерфейсы. Например, GameController использует TerminalInputProcessor для получения команд и GameDisplay для отображения игры, что упрощает понимание архитектуры и взаимодействия между компонентами. 





UML диаграмма классов отображена ниже. На ней зеленым цветом отображены поля классов, синим цветом конструкторы/деструкторы, а фиолетовым – методы. Также на диаграмме отображены связи между классами.



Разработанный программный код см. в приложении А.

## **Выводы**

В ходе разработки были созданы классы, которые помогли организовать управление игровым процессом.

# **Приложение А Исходный код программы**

Название файла: main.cpp

#include <iostream>

#include <exception>

#include <vector>

#include <ctime>

#include "sea\_battle\_playground.h"

#include "ship\_manager.h"

#include "ability\_manager.h"

#include "game\_state.h"

#include "game.h"

#include "game\_controller.h"

#include "game\_display.h"

#include "terminal\_input\_processor.h"

#include "terminal\_renderer.h"

int main() {

std::srand(std::time(0));

try {

std::cout << "a - attack\nu - use ability\nl - load game save\ns - save game\nq - quit\n";

int field\_width = 10;

int field\_length = 10;

std::vector<int> ship\_sizes = {1, 2, 3, 4};

SeaBattlePlayground user\_field(field\_width, field\_length);

SeaBattlePlayground enemy\_field(field\_width, field\_length);

ShipManager user\_ships(ship\_sizes.size(), ship\_sizes);

ShipManager enemy\_ships(ship\_sizes.size(), ship\_sizes);

AbilityManager user\_abilities;

GameState game\_state(&user\_field, &enemy\_field, &user\_abilities, &user\_ships, &enemy\_ships);

Game sea\_battle(&user\_field, &enemy\_field, &user\_ships, &enemy\_ships, &user\_abilities, &game\_state);

TerminalInputProcessor input\_processor("commands.txt");

TerminalRenderer renderer;

GameController<TerminalInputProcessor> controller(&sea\_battle, input\_processor);

GameDisplay<TerminalRenderer> display(&sea\_battle, renderer);

while (true) {

try {

display.render();

bool action = controller.process\_command();

if(!action) break;

}

catch (const std::exception& e) {

std::cout << "Exception: " << e.what() << std::endl;

}

catch (const char\* e) {

std::cerr << "Error: " << e << std::endl;

}

}

}

catch (const std::exception& e) {

std::cout << "Exception: " << e.what() << std::endl;

}

catch (const char\* e) {

std::cerr << "Error: " << e << std::endl;

}

return 0;

}

Название файла: game\_controller.h

#ifndef GAME\_CONTROLLER\_H

#define GAME\_CONTROLLER\_H

#include <iostream>

#include "game.h"

template<typename InputProcessor>

class GameController {

public:

Game\* game;

InputProcessor input\_processor;

GameController(Game\* game\_instance, InputProcessor processor) : game(game\_instance), input\_processor(processor) {}

bool process\_command() {

char command = input\_processor.get\_command();

switch (command) {

case 'a': game->attack(); return true;

case 's': game->save(); return true;

case 'l': game->load(); return true;

case 'u': game->use\_ability(); return true;

case 'q': return false;

default: std::cout << "Unknown command!" << std::endl;

}

return true;

}

};

#endif

Название файла: terminal\_input\_processor.h

#ifndef TERMINAL\_INPUT\_PROCESSOR\_H

#define TERMINAL\_INPUT\_PROCESSOR\_H

#include <iostream>

#include <fstream>

#include <unordered\_map>

class TerminalInputProcessor {

private:

std::unordered\_map<char, std::string> command\_map;

void load\_commands\_from\_file(const std::string& filename){

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "Error opening command file, using default controls." << std::endl;

command\_map = {{'a', "attack"}, {'l', "load"}, {'s', "save"}, {'u', "use\_ability"}, {'q', "quit"}};

return;

}

char key;

std::string command;

while (file >> key >> command) {

command\_map[key] = command;

}

}

public:

TerminalInputProcessor(const std::string& filename){

load\_commands\_from\_file(filename);

}

char get\_command(){

char input;

std::cin >> input;

return input;

}

};

#endif

Название файла: terminal\_renderer.cpp

#include "terminal\_renderer.h"

void TerminalRenderer::render(Game\* game) {

std::cout << "Rendering game field..." << std::endl;

game->display\_playing\_fields();

}

Название файла: terminal\_renderer.h

#ifndef TERMINAL\_RENDERER\_H

#define TERMINAL\_RENDERER\_H

#include "game.h"

class TerminalRenderer {

public:

void render(Game\* game);

};

#endif

Название файла: Makefile

all : game

main.o : main.cpp

g++ -c main.cpp

ship.o : ship.cpp

g++ -c ship.cpp

field\_cell.o : field\_cell.cpp

g++ -c field\_cell.cpp

ship\_manager.o : ship\_manager.cpp

g++ -c ship\_manager.cpp

sea\_battle\_playground.o : sea\_battle\_playground.cpp

g++ -c sea\_battle\_playground.cpp

exceptions.o : exceptions.cpp

g++ -c exceptions.cpp

ability.o : ability.cpp

g++ -c ability.cpp

double\_damage.o : double\_damage.cpp

g++ -c double\_damage.cpp

random\_bombardment.o : random\_bombardment.cpp

g++ -c random\_bombardment.cpp

scanner.o : scanner.cpp

g++ -c scanner.cpp

ability\_manager.o : ability\_manager.cpp

g++ -c ability\_manager.cpp

game\_state.o : game\_state.cpp

g++ -c game\_state.cpp

game.o : game.cpp

g++ -c game.cpp

terminal\_renderer.o : terminal\_renderer.cpp

g++ -c terminal\_renderer.cpp

game : main.o ship.o field\_cell.o ship\_manager.o sea\_battle\_playground.o exceptions.o ability.o double\_damage.o random\_bombardment.o scanner.o ability\_manager.o game\_state.o game.o terminal\_renderer.o

g++ main.o ship.o field\_cell.o ship\_manager.o sea\_battle\_playground.o exceptions.o ability.o double\_damage.o random\_bombardment.o scanner.o ability\_manager.o game\_state.o game.o terminal\_renderer.o -o game

rm \*.o