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. 3384 _____ Поздеев В.Д

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург
2024

Цель работы.

Связать созданные в предыдущей лабораторной работы классы. Создать класс игры. Реализовать класс состояния игры и переопределить операторы ввода и вывода в поток для состояния игры. Реализовать сохранение и загрузку игры.

Задание.

Создать класс-интерфейс способности, которую игрок может применять. Через наследование создать 3 разные способности:

Двойной урон - следующая атак при попадании по кораблю нанесет сразу 2 урона (уничтожит сегмент).

Сканер - позволяет проверить участок поля 2x2 клетки и узнать, есть ли там сегмент корабля. Клетки не меняют свой статус.

Обстрел - наносит 1 урон случайному сегменту случайного корабля. Клетки не меняют свой статус.

Создать класс менеджер-способностей. Который хранит очередь способностей, изначально игроку доступно по 1 способности в случайном порядке. Реализовать метод применения способности.

Реализовать функционал получения одной случайной способности при уничтожении вражеского корабля.

Реализуйте набор классов-исключений и их обработку для следующих ситуаций (можно добавить собственные):

Попытка применить способность, когда их нет

Размещение корабля вплотную или на пересечении с другим кораблем

Атака за границы поля

Примечания:

Интерфейс события должен быть унифицирован, чтобы их можно было единообразно использовать через интерфейс

Не должно быть явных проверок на тип данныхСоздать 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 самое игрой можно использовать обертки над командами

При работе с файлом используйте идиому RAII.

Выполнение работы.

Была создана система сообщений и реализован паттерн цепочка обязанностей.

Сообщения

Создан абстрактный класс Message. Создана виртуальная функция *clone*, которая возвращает копию сообщения ввиде *unique_ptr<Message>*, данный метод нужно переопределять в наследуемых классах.

Создан класс keyMessage, конструктор которого принимает *enum Key info* и хранит его в поле. Сообщение клонируется с помощью *std::make_unique<keyMessage>(*this)*.

Создан класс *playFieldMessage*, который в конструкторе принимает название поля(*std::string field_name)*, ссылку на игровое поле *playField &, enum fieldPosition,* флаг *fog* и флаг *drawPointer,* который по умолчанию равен *false* и записывает в поля. Переопределён метод *clone*, который копирует сообщение.

Создан класс *pointerMessage,* который имеет *default* конструктор, а также второй конструктор, который принимает *box2d pointer_area* и *point2d coordinates* и записывает в поля класса. Создан метод копирования *clone()*.

Создан класс *textMessage.* Внутри себя содержит *SDL_Color color, std::string msg, textPosition position.* В конструкторе принимает эти объекты и записывает в поля. Переопределен метод копирования *clone()*.

Принимающие сообщения

Создан виртуальный класс *messageHandler,* который слушит опорой для паттерна цепочка обязанностей. Внутри созданы два виртуальных публичных метода *Handle(std::unique_ptr<Message> message)* и *setNext(messageHandler * handler)*. То есть данный класс принимает сообщения и обрабатывает их.

Класс GUIOutput

Создан класс *GUIOutput,* который наследуется от *messageHandler.* Внутри создан *enum fontSize,* который хранит в себе размеры текста для вывода.

Поля класса:

*SDL_Window * window* — окно в которое выводится изображение.

*SDL_Renderer * renderer* — выводит изображение в окно.

*TTF_Font * big_font, medium_font, small_font* — создает объект шрифта, который используется для рендера.

textMessage title

std::vector<std::string> instruction — вектор строк(инструкций), которые выводятся на экран с помощью *drawInstructions()*

textMessage log[seabattle::LOG_LENGTH] — массив, который затем выводится в окно с помощью *drawLog()*

pointerMessage pointer.

*MessageHandler * handler = nullptr* — указывает на следующий объект в цепочке(пока что никуда не указывает).

Конструктор *GUIOutput* инициализирует *SDL* с помощью *SDL_INIT(SDL_INIT_VIDEO),* инициализирует *TTF* с помощью *TTF_Init().*

Создается окно с помощью *SDL_CreateWindow("Sea Battle",*
SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED,
seabattle::WIDTH, seabattle::HEIGHT,
SDL_WINDOW_SHOWN);

Создается *renderer* с помощью *SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);* Вызывается *SDL_SetRenderDrawBlendMode(renderer, SDL_BLENDMODE_BLEND)* для того, чтобы можно было использовать альфа канал при рендеринге. Затем создаются с помощью *TTF_OpenFont(seabattle::FONT_DIR, seabattle::BIG_FONT_SIZE)* создаются шрифты описанные в поле класса(меняется только размер шрифта из настроек(*settings.h*)).

Функция *drawField* класса *GUIOutput* начинает с определения входных параметров, включая имя поля, ссылку на объект игрового поля, позицию (центр, слева или справа), флаг для отображения тумана войны и флаг для указателя. Затем вычисляются координаты верхнего левого угла области, где будет отрисовываться игровое поле, с учетом заданной позиции и отступа сверху. После этого устанавливается цвет фона для области игрового поля, и создается прямоугольник, который заполняется этим цветом, чтобы создать фон. Далее определяется размер ячеек на основе общего размера области игрового поля и количества ячеек по горизонтали и вертикали. Затем создается прямоугольник, обрамляющий игровое поле, который заполняется цветом для создания визуального эффекта границы. Внутри функции используются вложенные циклы для перебора каждой ячейки игрового поля. Для каждой ячейки определяется её состояние в зависимости от значения флага *fog*. Если туман войны включен, отображаются только состояния "неизвестно", "пусто" и различные состояния сегментов кораблей. Если туман войны выключен, отображаются состояния ячеек, включая атаки на них. Устанавливается цвет для каждой ячейки на основе её состояния, и рисуется прямоугольник с помощью функции отрисовки. Также рисуется контур ячейки, чтобы выделить её на фоне. В результате функция визуализирует состояние игрового поля, учитывая различные состояния ячеек и возможность отображения тумана войны.

Затем под полем рисуется его название и вызывается метод отрисовки указателя игрока(если это нужно).

Функция *drawPointer* отрисовывает указатель на экране с использованием *SDL*. Она принимает размер ячейки *size_cell*, координаты начала отрисовки coordinates и размер игрового поля *field_size* в качестве входных данных. Вложенные циклы перебирают все ячейки области указателя pointer.area, определенной точками *min_point* и *max_point*. В каждом цикле вычисляются экранные координаты ячейки *cell_coordinates* путем сложения координат указателя *pointer.coordinates* и текущих координат цикла, умноженных на

size_cell. Затем создается прямоугольник cell с использованием вычисленных координат и размера ячейки. Функция *SDL_SetRenderDrawColor* устанавливает цвет отрисовки в цвет указателя *seabattle::POINTER_COLOR*. Функция *SDL_RenderFillRect* заполняет прямоугольник *cell* этим цветом. После этого цвет меняется на темно-серый *(32, 32, 32, 255)*, и функция *SDL_RenderDrawRect* отрисовывает рамку вокруг прямоугольника *cell*. Таким образом, каждая ячейка указателя заполняется цветом указателя и обводится темно-серой рамкой.

Функция *drawText* отрисовывает текст на экране используя *SDL_ttf*. Сначала она выбирает шрифт *font* в зависимости от переданного размера *font_size*. Если текст не пустой, она рендерит текст в поверхность textSurface используя функцию *TTF_RenderText_Solid*, обрабатывая возможные ошибки. Затем создается текстура *textTexture* из поверхности *SDL_CreateTextureFromSurface*, с обработкой ошибок. Создается прямоугольник renderQuad с координатами, шириной и высотой, взятыми из *textSurface*. Если *is_centered* истинно, координата x сдвигается на половину ширины текста для центрирования. Функция *SDL_RenderCopy* копирует текстуру на экран. Наконец, текстура и поверхность освобождаются, и функция возвращает *renderQuad*. Если текст пустой, выбрасывается исключение.

Функция *redirectText* принимает сообщение *text* и обрабатывает его в зависимости от позиции text.position. Если позиция *title*, сообщение сохраняется в переменную *title*. Если позиция *log*, элементы массива *log* сдвигаются на одну позицию назад, начиная с последнего элемента, и новое сообщение *text* записывается в начало массива *log*, тем самым реализуя логирование с ограничением длины *seabattle::LOG_LENGTH*.

Функция *drawTitle* отрисовывает заголовок на экране. Она определяет начальные координаты *coordinates*, добавляя отступ *top_indent* к центру экрана по горизонтали и верхнему краю по вертикали. Затем вызывает функцию

drawText для отрисовки текста из поля *title.msg* в рассчитанных координатах, используя большой размер шрифта *big*, цвет из *title.color* и центрирование по горизонтали *true*.

Функция *drawLog* отрисовывает лог сообщений. Она устанавливает начальные координаты *coordinates* в нижней части экрана, с отступом *indent*. Цикл перебирает массив *log*. Если сообщение *log[i].msg* не пустое, рисуется полупрозрачный черный прямоугольник outline как фон для сообщения. Затем вызывается *drawText* для отрисовки сообщения с использованием маленького шрифта *small* и цвета из *log[i].color*. Координаты *coordinates* обновляются, смещаясь вверх на высоту отрисованного текста *renderQuad.h* для следующего сообщения.

Функция *drawInstructions* отрисовывает инструкции на экране. Она устанавливает начальные координаты *coordinates* в нижней части экрана, с отступом *indent*. Рисуется полупрозрачный белый прямоугольник *outline* как фон для инструкций. Затем вызывается функция *drawText* многократно для отрисовки каждой строки инструкций с использованием маленького шрифта *small* и черного цвета, с отступом *text_indent* между строками. Инструкции заключены между разделительными линиями.

Функция *update* обновляет отображение игры. Она последовательно вызывает функции *drawTitle*, *drawLog* и *drawInstructions* для отрисовки заголовка, лога и инструкций. Затем *SDL_RenderPresent* отображает все нарисованное на экране. После этого устанавливается фоновый цвет *seabattle::BACKGROUND_COLOR* с помощью *SDL_SetRenderDrawColor*, и *SDL_RenderClear* очищает экран для следующего кадра, готовя его к отрисовке. Обратите внимание, что порядок вызова функций отрисовки и очистки экрана важен: сначала отрисовываются элементы, а затем очищается экран для следующего цикла отрисовки.

Функция *Handle* обрабатывает входящие сообщения *message*. Она проверяет тип сообщения и выполняет соответствующие действия. Если сообщение типа *textMessage*, оно передается функции *redirectText* для обработки. Если сообщение типа *playFieldMessage*, вызывается функция *drawField* для отрисовки игрового поля. Если сообщение типа *pointerMessage*, данные указателя обновляются, присваивая содержимое сообщения *tr_msg* переменной *pointer*. В каждом случае используется *dynamic_cast* для безопасного приведения указателя на базовый класс *Message* к соответствующему производному типу.

Деструктор *GUIOutput* освобождает ресурсы, используемые графическим интерфейсом. Он последовательно уничтожает рендерер renderer, окно window, шрифты *big_font*, *medium_font*, *small_font* и завершает работу библиотек *SDL_ttf* и *SDL*.

Класс GUIInput

Класс *GUIInput* наследуется от *messageHandler* и предназначен для обработки ввода с клавиатуры. Он содержит указатель *handler* на следующий обработчик сообщений.

Функция *transformKey* преобразует код клавиши *SDL* в перечисление *Key*. Она принимает код клавиши *key* и возвращает соответствующее значение *Key* в зависимости от нажатой клавиши: клавиши *W, A, S, D* сопоставляются с перемещением указателя, *Enter* — с основным действием, *E* и *Q* — с дополнительными действиями, *1* и *2* — с сохранением и загрузкой соответственно. Если клавиша не обрабатывается, возвращается значение *Key::null*.

Функция *update* обрабатывает события *SDL*. Она использует *SDL_PollEvent* для получения событий из очереди. Если событие типа

SDL_QUIT, создается и отправляется сообщение *keyMessage* с типом *Key::quit*. Если событие типа *SDL_KEYDOWN*, код клавиши преобразуется с помощью transformKey, и создается и отправляется сообщение *keyMessage* с соответствующим типом *Key*.

Функция *Handle* перенаправляет полученное сообщение message следующему обработчику *handler* в цепочке обработки, используя *std::move* для передачи владения сообщением.

Класс Game

Класс *Game* наследуется от *messageHandler* и представляет собой основную игровую логику. Он содержит указатель на текущее игровое состояние *state*, указатель на следующий обработчик сообщений *handler*, флаг running для управления циклом игры, экземпляр игрока-человека *player* и экземпляр игрока-бота *bot*.

Конструктор класса *Game* инициализирует игру. Он устанавливает флаг *running* в *true*, создает объект *setupFieldState* и устанавливает его в качестве текущего состояния игры. Затем он устанавливает обработчики сообщений для игрока и бота, а также устанавливает текущее состояние игры в качестве следующего обработчика сообщений.

Метод *setState* меняет текущее игровое состояние. Он удаляет текущий объект состояния *this->state*, устанавливает новый объект *state* и обновляет цепочку обработчиков сообщений, устанавливая новый объект состояния в качестве следующего обработчика.

Метод *execute* запускает выполнение текущего игрового состояния, вызывая метод *execute* объекта *state*.

Метод *Handle* обрабатывает сообщения. Если сообщение — *keyMessage*, проверяется тип нажатой клавиши. *Key::quit* останавливает игру, устанавливая *running* в *false*. *Key::save_action* вызывает сохранение игры и отправляет сообщение об успехе в лог. *Key::load_action* вызывает загрузку игры, обрабатывая возможные исключения при чтении файла сохранения и отправляя сообщения об успехе или ошибке в лог. В остальных случаях сообщение передаётся следующему обработчику в цепочке (*handler*).

Метод *setNext* устанавливает следующий обработчик сообщений в цепочке обработки, передавая указатель на него в переменную *handler*.

Метод *save* сохраняет текущее состояние игры в файл. Он определяет текущее состояние игры, используя *typeid(*state).name()*, и сериализует его в *JSON* с помощью библиотеки nlohmann::json. В зависимости от типа состояния *(setupFieldState, setupShipState, playState, endGameState)*, он выполняет соответствующее приведение типа и сериализует данные состояния, а также данные игроков. Затем он создает *JSON*-объект *full_data*, содержащий сериализованные данные и хэш-код для проверки целостности, и сохраняет его с помощью *fileWrite* в указанный каталог.

Метод *load* загружает состояние игры из файла. Сначала он очищает данные игроков, создавая новые *shipManager*. Затем он считывает данные из файла используя *fileRead*, проверяет хэш-код для обнаружения повреждений данных и, в зависимости от имени сохраненного состояния (*state_name*), создает соответствующий объект состояния *(setupFieldState, setupShipState, playState, endGameState)*, десериализует данные из *JSON* в объект и устанавливает его как текущее состояние игры с помощью *setState*. Также он загружает данные игроков из сохраненных данных. Если хэш-коды не совпадают, выбрасывается исключение.

Деструктор класса *Game* освобождает память, занятую текущим объектом игрового состояния *state*.

Состояния игры

Абстрактный базовый класс *gameState* представляет собой состояние игры. Он содержит указатель на обработчик сообщений *handler* и указатель на объект игры *game*. Конструктор инициализирует указатель на игру. Метод *setGame* устанавливает указатель на объект игры. Виртуальный метод *execute* определяет поведение состояния, *Handle* — обработку сообщений. Деструктор виртуальный. Метод *setNext* устанавливает следующий обработчик сообщений. Класс *Game* объявлен как дружественный, предоставляя ему доступ к защищенным членам.

Состояние setupFieldState

Класс *setupFieldState* наследуется от *gameState* и представляет состояние настройки игрового поля. Он содержит поля *size_x* и *size_y* для размеров поля, и поле *play_field* для представления самого поля.

Метод *execute* создает игровое поле *play_field* с размерами *size_x* и *size_y*. Он обрабатывает исключение *invalidFieldSize*, если размеры поля некорректны, выводит сообщение об ошибке и корректирует размеры поля. После успешного или неудачного создания поля он отправляет сообщение *playFieldMessage* для отрисовки поля.

Метод *Handle* обрабатывает сообщения. Если сообщение — *keyMessage*, он изменяет размеры поля (*size_x*, *size_y*) в зависимости от нажатой клавиши. *Key::main_action* завершает текущее состояние с помощью end(). *Key::extra_action_0* меняет местами *size_x* и *size_y*. *Key::extra_action_1* устанавливает размеры поля в *8x8*. В остальных случаях сообщение передаётся следующему обработчику.

Метод *end* завершает текущее состояние *setupFieldState*. Он устанавливает созданное игровое поле *play_field* для обоих игроков (*game->player; game->bot*) и переключает игровое состояние на *setupShipState*, создавая новый объект этого класса.

Оператор << перегружен для сериализации объекта *setupFieldState* в *JSON*. Он принимает ссылку на объект *JSON data* и ссылку на объект *game_state*. Он добавляет поля *size_x* и *size_y* объекта *game_state* в объект *JSON data* и возвращает измененный объект *JSON*.

Оператор >> перегружен для десериализации объекта *setupFieldState* из *JSON*. Он принимает ссылку на объект *JSON data* и ссылку на объект *game_state*. Он извлекает значения полей *size_x* и *size_y* из объекта *JSON data* и присваивает их соответствующим полям объекта *game_state*. Функция возвращает измененный объект *JSON data*.

Состояние setupShipState

Класс *setupShipState* наследуется от *gameState* и представляет состояние расстановки кораблей. Он содержит массив *ships* для хранения количества кораблей каждой длины (предположительно от 1 до 4), флаг *is_vertical* для указания ориентации кораблей, ссылку на игровое поле *field*, ссылку на координаты указателя *pointer*, ссылку на область указателя *pointer_area*, и длину корабля *length*.

Конструктор *setupShipState* инициализирует состояние расстановки кораблей. Он получает указатели на объект игры *game* и следующий обработчик сообщений *next*, и устанавливает ссылки на игровое поле *field* и указатель pointer игрока. Если флаг *place_ships* установлен, он инициализирует указатель в (0, 0), вызывает функцию callculateShips для определения количества кораблей

и функцию *placeShipsRandomly* для автоматической расстановки кораблей бота. Он также отправляет сообщение в заголовок.

Функция *enoughShips* возвращает *true*, если количество кораблей всех типов не равно нулю, и *false* в противном случае.

Функция *execute* управляет процессом расстановки кораблей. Если достаточно кораблей всех типов (*enoughShips()* возвращает *true*), вызывается функция *end()* для перехода к следующему этапу. В противном случае, в зависимости от наличия кораблей определенной длины, устанавливается размер области указателя *pointer_area.max_point* и длина корабля *length*. Ориентация корабля меняется в зависимости от флага *is_vertical*, и если область указателя выходит за пределы поля, ориентация меняется на противоположную. Затем отправляются сообщения *pointerMessage* и *playFieldMessage* для обновления отображения.

Функция *placeShip* размещает корабль на игровом поле. Она создает объект *Ship* с заданными параметрами и пытается разместить его на поле игрока с помощью *game->player.placeShip*. В зависимости от длины корабля уменьшается количество кораблей соответствующей длины в массиве *ships*. Если свободных клеток на поле нет, вызывается функция *end()* для перехода к следующему этапу. Функция обрабатывает исключения *invalidShipPosition* и *objectOutOfBounds*, выводя сообщения об ошибках.

Функция *end* завершает текущее состояние *setupShipState* и переключает игровое состояние на *playState*, создавая новый экземпляр этого класса.

Метод *Handle* обрабатывает сообщения. Если сообщение — *keyMessage*, он обрабатывает нажатия клавиш для перемещения указателя (*Key::pointer_up*, *Key::pointer_down*, *Key::pointer_right*, *Key::pointer_left*), при этом проверяя, не

выходит ли указатель за границы поля. *Key::main_action* вызывает размещение корабля с помощью *placeShip()*. *Key::extra_action_0* меняет ориентацию корабля. *Key::extra_action_1* выполняет автоматическую расстановку кораблей и завершает текущее состояние. В остальных случаях сообщение передается следующему обработчику.

Оператор << перегружен для сериализации объекта *setupShipState* в *JSON*. Он добавляет в *JSON*-объект data поля *ships* (массив), *is_vertical*, и *length* из объекта *game_state*. Возвращается измененный *JSON*-объект.

Оператор >> перегружен для десериализации объекта *setupShipState* из *JSON*. Он извлекает значения полей *ships*, *is_vertical*, и *length* из *JSON*-объекта data и присваивает их соответствующим полям объекта *game_state*. Возвращается измененный *JSON*-объект.

Состояние playState

Класс *playState* наследуется от *gameState* и представляет игровое состояние во время игры. Он содержит ссылки на указатель *pointer* и его область *pointer_area*, номер раунда *round_number*, и флаг *input*.

Конструктор *playState* инициализирует игровое состояние. Он устанавливает ссылки на указатель и его область, номер раунда, и устанавливает связь между игроком и ботом, используя *getOpponent*. Он также отправляет сообщения в заголовок и лог, указывая начало раунда.

Метод *execute* проверяет, не уничтожены ли все корабли у игрока или бота. Если да, вызывается функция *end()* с соответствующим результатом. Если флаг *input* установлен, вызывается функция *usingAbility()*. В противном случае отправляются сообщения для обновления отображения игрового поля и указателя.

Функция *usingAbility* обновляет отображение игрового поля и указателя. Она отправляет сообщения для отрисовки игрового поля игрока и бота, а также обновляет положение указателя.

Метод *Handle* обрабатывает сообщения. Если сообщение — *keyMessage*, обрабатываются клавиши управления указателем, аналогично *setupShipState*. *Key::main_action* вызывает атаку игрока и бота. *Key::extra_action_0* пытается активировать способность игрока, обрабатывая исключение *noAbilitiesException*. Если способность активирована, меняется заголовок и выводится сообщение. В остальных случаях сообщение передается следующему обработчику.

Метод *end* завершает текущее состояние *playState* и переключает игровое состояние на *endGameState*, передавая информацию о победе/поражении (*lost*) и номер раунда.

Оператор << перегружен для сериализации объекта *playState* в *JSON*. Он добавляет поля *input* и *round_number* объекта *game_state* в *JSON*-объект data и возвращает модифицированный *JSON*-объект.

Оператор >> перегружен для десериализации объекта *playState* из *JSON*. Он извлекает значения полей *input* и *round_number* из *JSON*-объекта data и присваивает их соответствующим полям объекта *game_state*. Возвращается модифицированный *JSON*-объект.

Состояние endGameState

Класс *endGameState* наследуется от *gameState* и представляет конечное состояние игры. Он содержит булево значение *lost*, указывающее на поражение, и номер раунда *round_number*.

Конструктор *endGameState* инициализирует конечное состояние игры. Он устанавливает флаг *lost*, указывающий на поражение, и номер раунда. В зависимости от значения *lost*, отправляются сообщения в заголовок и лог, сообщая о победе или поражении и предлагая нажать *Enter* для перезапуска или перехода к следующему раунду.

Метод *execute* отображает итоговые игровые поля игрока и бота. Он отправляет сообщения для отрисовки полей без тумана войны.

Метод *Handle* обрабатывает сообщения. Если сообщение — *keyMessage* и нажата клавиша *Key::main_action* (*Enter*), вызывается метод *end()* для перехода к следующему состоянию. В остальных случаях сообщение передается следующему обработчику.

Метод *end* завершает текущее состояние и переходит к следующему. Если игра была проиграна (*lost*), ресурсы игроков освобождаются и игра перезапускается с *setupFieldState*. Если игра была выиграна, ресурсы бота освобождаются, корабли бота расставляются заново, и игра переходит к следующему раунду с *playState*.

Оператор << перегружен для сериализации объекта *endGameState* в *JSON*. Он добавляет поля *round_number* и *lost* объекта game_state в *JSON*-объект *data* и возвращает модифицированный *JSON*-объект.

Оператор >> перегружен для десериализации объекта *endGameState* из *JSON*. Он извлекает значения полей *round_number* и *lost* из *JSON*-объекта *data* и присваивает их соответствующим полям объекта *game_state*. Возвращается модифицированный *JSON*-объект.

Класс игрока

Класс *Player* наследуется от *messageHandler* и представляет игрока. Он содержит указатель на игровое поле противника *opponent_play_field*, указатель на менеджер кораблей противника *opponent_ship_manager*, собственное игровое поле *play_field* и менеджер кораблей *ship_manager*. Также присутствует указатель на следующий обработчик сообщений *handler*.

Функция *setField* устанавливает игровое поле *play_field* игрока, принимая в качестве аргумента объект *field*.

Функция *placeShip* размещает корабль на игровом поле игрока. Она принимает указатель на корабль *ship* и использует менеджер кораблей *ship_manager* для размещения его на *play_field*.

Функция *Handle* перенаправляет полученное сообщение message следующему обработчику сообщений *handler* в цепочке обработки, передавая ему владение сообщением.

Функция *setNext* устанавливает следующий обработчик сообщений в цепочке, присваивая указатель *handler* полю *this->handler*.

Функция *free* очищает игровое поле и менеджер кораблей игрока. Она создает новый пустой менеджер кораблей и создает новое игровое поле с теми же размерами, что и предыдущее.

Функция *placeShipsRandomly* случайным образом расставляет корабли на игровом поле. Она использует генератор случайных чисел для выбора координат и ориентации кораблей. Цикл продолжается до тех пор, пока не будут размещены все корабли. При размещении корабля проверяется, не пересекается ли он с другими кораблями и не выходит ли за границы поля. Если

корабль успешно размещен, соответствующее количество кораблей уменьшается в массиве *bot_ships*. Если свободных клеток нет, функция завершается.

Функция *callculateShips* вычисляет количество кораблей каждой длины для игрока, в зависимости от размера игрового поля. Она определяет общее количество клеток поля и приблизительное количество кораблей, которое должно быть размещено. Затем, используя вложенные циклы, она ищет комбинацию количества кораблей каждой длины, которая в сумме дает нужное количество кораблей, и при этом количество кораблей убывает с увеличением длины. Результат записывается в массив *ships*.

Оператор << перегружен для сериализации объекта *Player* в *JSON*. Он сериализует поля *ship_manager* и *play_field* объекта player в *JSON*, используя их собственные методы сериализации (*toJson*), и добавляет результат в *JSON*-объект *data*. Возвращается модифицированный *JSON*-объект.

Оператор >> перегружен для десериализации объекта *Player* из *JSON*. Он извлекает данные из *JSON*-объекта *data* для полей *ship_manager* и play_field, создавая объекты этих классов из *JSON*-данных, используя их собственные методы десериализации. Затем он загружает корабли в игровое поле с помощью *loadShips*. Возвращается модифицированный *JSON*-объект.

Класс игрока пользователя

Класс *humanPlayer* наследуется от *Player* и представляет собой игрока-человека. Он содержит менеджер способностей *abilities_manager*, флаг *double_damage*, указатель на текущую способность *current_ability*, а также координаты указателя *pointer* и его область *pointer_area*.

Функция *getOpponent* устанавливает указатели на игровое поле и менеджер кораблей противника. Она принимает указатель на объект Player и присваивает его поля *play_field* и *ship_manager* полям *opponent_play_field* и *opponent_ship_manager* текущего объекта.

Функция *getAbility* пытается получить способность от *abilities_manager*. Если способность требует ввода координат (*info.need_input*), устанавливается размер области указателя и возвращается *true*. В противном случае, способность применяется с помощью *useAbility()* и возвращается *false*.

Функция *useAbility* применяет текущую способность *current_ability* к игроку, используя метод *apply*. После применения способность обнуляется, и область указателя сбрасывается.

Функция *Attack* выполняет атаку противника. Если установлен флаг *double_damage*, выполняется атака с удвоенным уроном, после чего флаг сбрасывается. Затем выполняется обычная атака в координатах *pointer*. Если в результате атаки были уничтожены корабли, создается случайная способность, и отправляется соответствующее сообщение в лог.

Функция *free* очищает ресурсы игрока. Она создает новый пустой менеджер кораблей и создает новое игровое поле с теми же размерами, что и предыдущее. Также она очищает менеджер способностей.

Функция *areaInField* проверяет, находится ли заданная область area в пределах игрового поля *play_field* после смещения на координаты *coordinates*. Она смещает границы области на заданные координаты и затем проверяет, содержит ли область игрового поля смещенную область. Возвращает *true*, если область целиком находится в пределах поля, и *false* в противном случае.

Оператор << перегружен для сериализации объекта *humanPlayer* в *JSON*. Он сериализует поля *play_field*, *ship_manager*, *pointer*, *pointer_area*, и *abilities_manager* объекта player в *JSON*, используя их собственные методы сериализации (*toJson*), и добавляет результат в *JSON*-объект *data*. Возвращается модифицированный *JSON*-объект.

Оператор >> перегружен для десериализации объекта *humanPlayer* из *JSON*. Он извлекает данные из *JSON*-объекта data для полей *ship_manager*, *play_field*, *pointer*, *pointer_area*, и *abilities_manager*, создавая объекты этих классов из *JSON*-данных, используя их собственные методы десериализации. Затем он загружает корабли в игровое поле с помощью *loadShips*. Возвращается модифицированный *JSON*-объект.

Класс бота

Класс *botPlayer* наследуется от *Player* и представляет игрока-бота. Он объявляет методы *Attack*, *setField*, и *getOpponent* для атаки, установки игрового поля и получения информации о противнике соответственно.

Функция *setField* устанавливает игровое поле *play_field* для бота, принимая в качестве аргумента объект *play_field.*

Функция *getOpponent* устанавливает указатели на игровое поле и менеджер кораблей противника для бота. Она принимает указатель на объект Player и присваивает его поля *play_field* и *ship_manager* полям *opponent_play_field* и *opponent_ship_manager* текущего объекта.

Функция *Attack* выполняет атаку бота. Она использует генератор случайных чисел для выбора координат атаки и вызывает метод *Attack* у игрового поля противника. Этот метод реализует очень простой, случайный *AI* бота.

Класс для чтения файлов JSON

Класс *fileRead* предназначен для чтения данных из файла. В приватной секции объявлен поток ввода *reader*. В публичной секции объявлен конструктор *fileRead*, метод read для чтения *JSON*-данных и деструктор *~fileRead*.

Конструктор *fileRead* открывает файл, указанный в строке *fname*, для чтения. Если файл не может быть открыт, выбрасывается исключение *std::runtime_error*.

Метод *read* считывает *JSON*-данные из открытого файла *reader* и записывает их в объект *j*. Если файл не открыт, выбрасывается исключение *std::runtime_error*.

Деструктор *~fileRead* закрывает файл *reader*, если он открыт.

Класс для записи в файлы JSON

Класс *fileRead* предназначен для чтения данных из файла. В приватной секции объявлен поток ввода *reader*. В публичной секции объявлен конструктор *fileRead*, метод *read* для чтения *JSON*-данных и деструктор *~fileRead*.

Конструктор открывает файл с именем *fname* в режиме записи. Если файл не может быть открыт, выбрасывается исключение *std::runtime_error*.

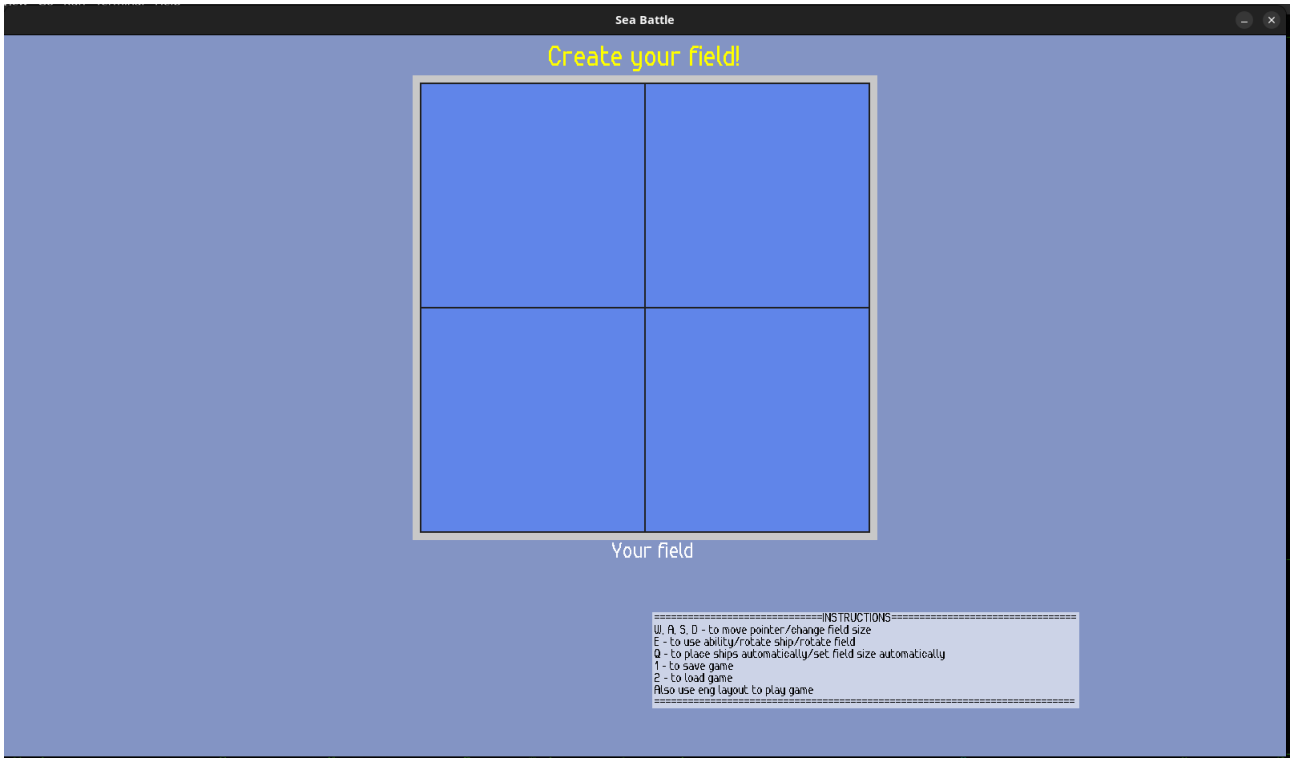
Функция *write* записывает отформатированные *JSON* данные в открытый файл. Если writer открыт, *j.dump(4)* сериализует *JSON* данные в отформатированную строку с отступом в *4* пробела, которая затем записывается в файл. В противном случае, выбрасывается исключение *std::runtime_error*.

Деструктор *~fileWrite* гарантирует закрытие файла *writer* перед уничтожением объекта *fileWrite*. Проверка *writer.is_open()* предотвращает попытки закрыть уже закрытый файл.

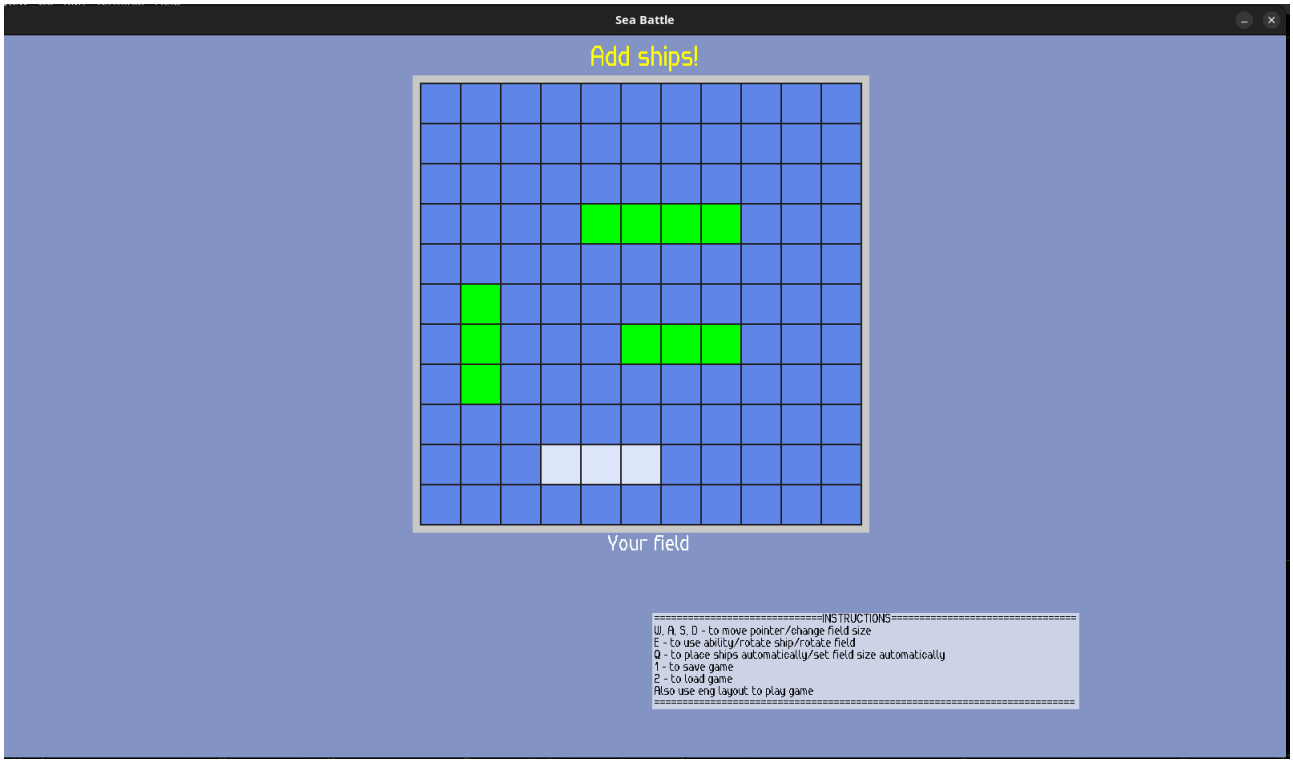
Диаграммы классов.

Тестирование.

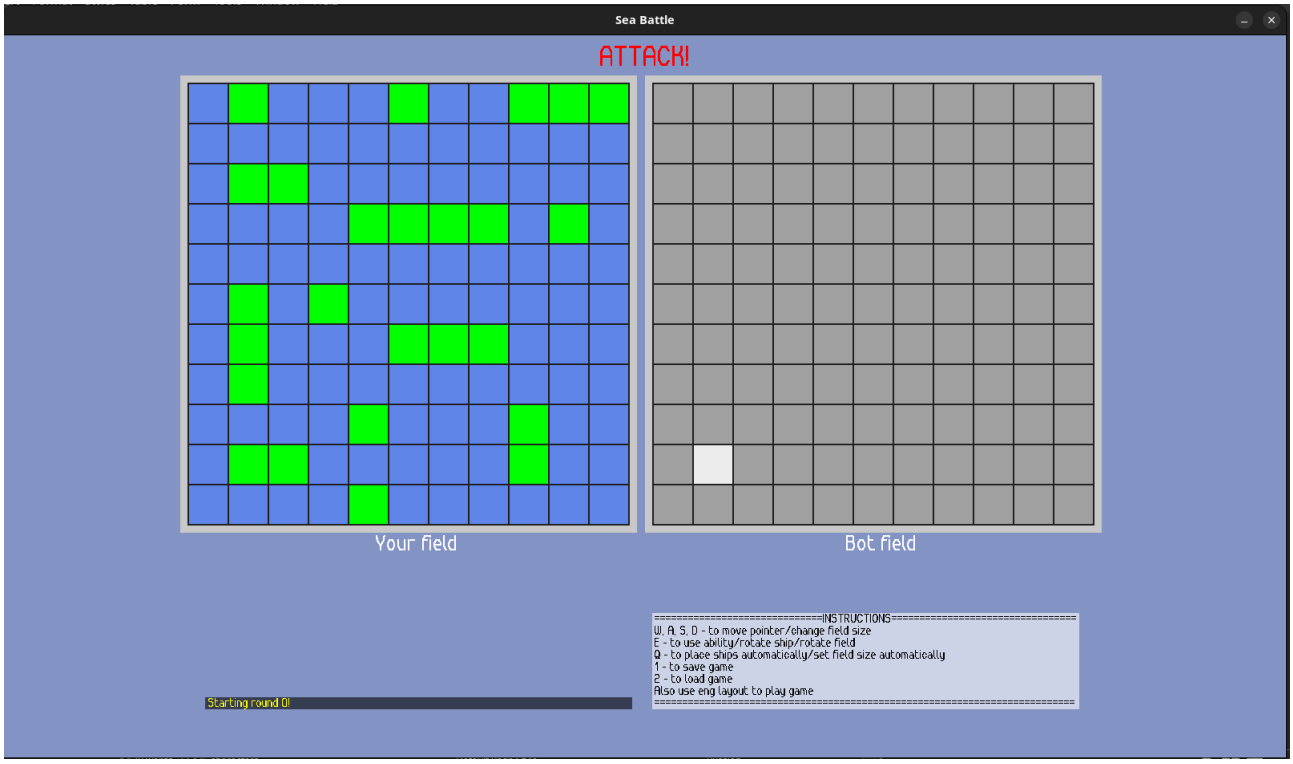
Запуск игры



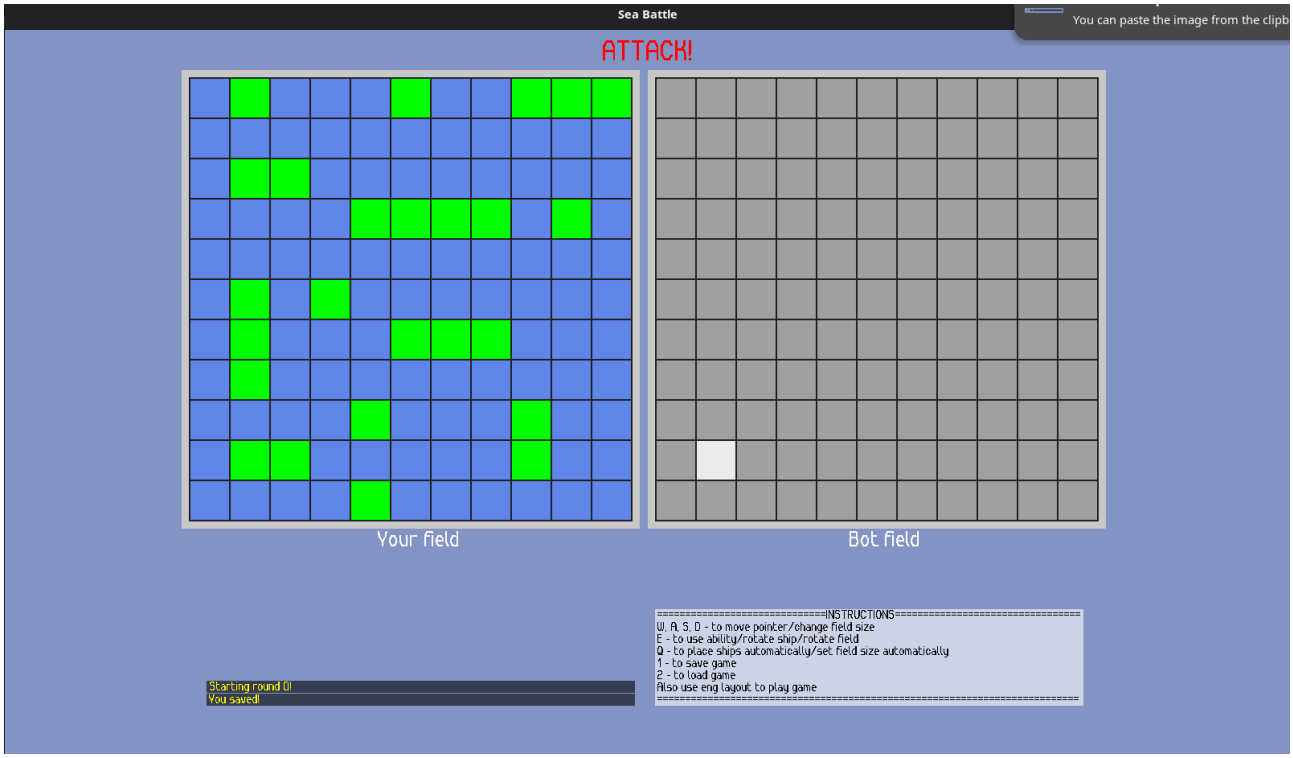
Расстановка кораблей



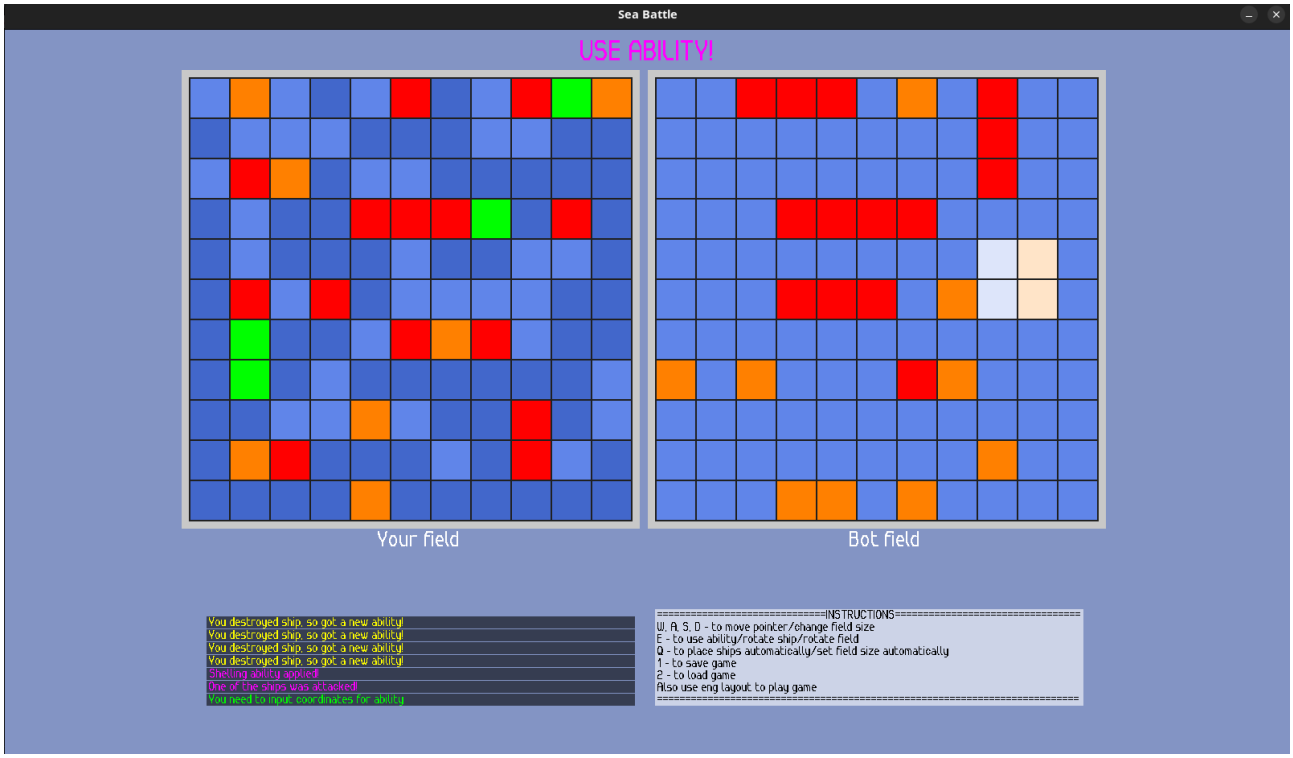
Игра



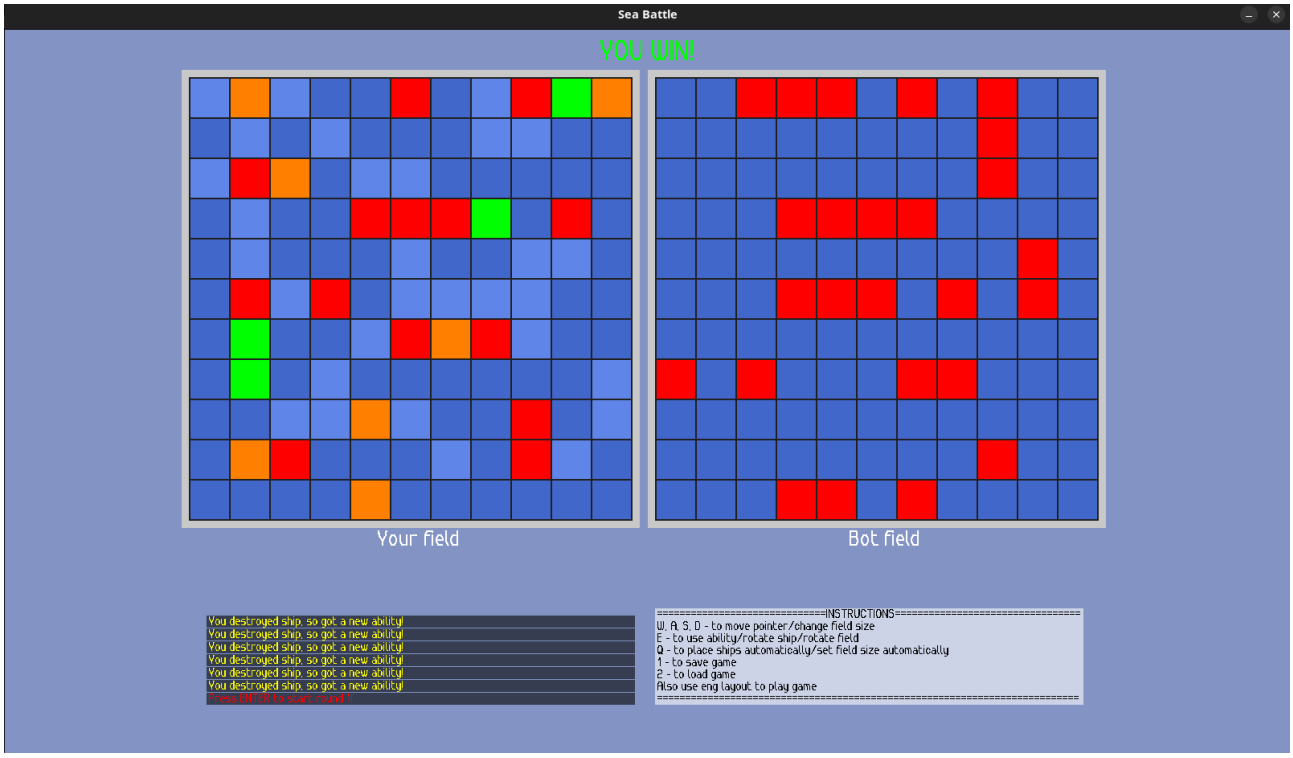
Сохранение



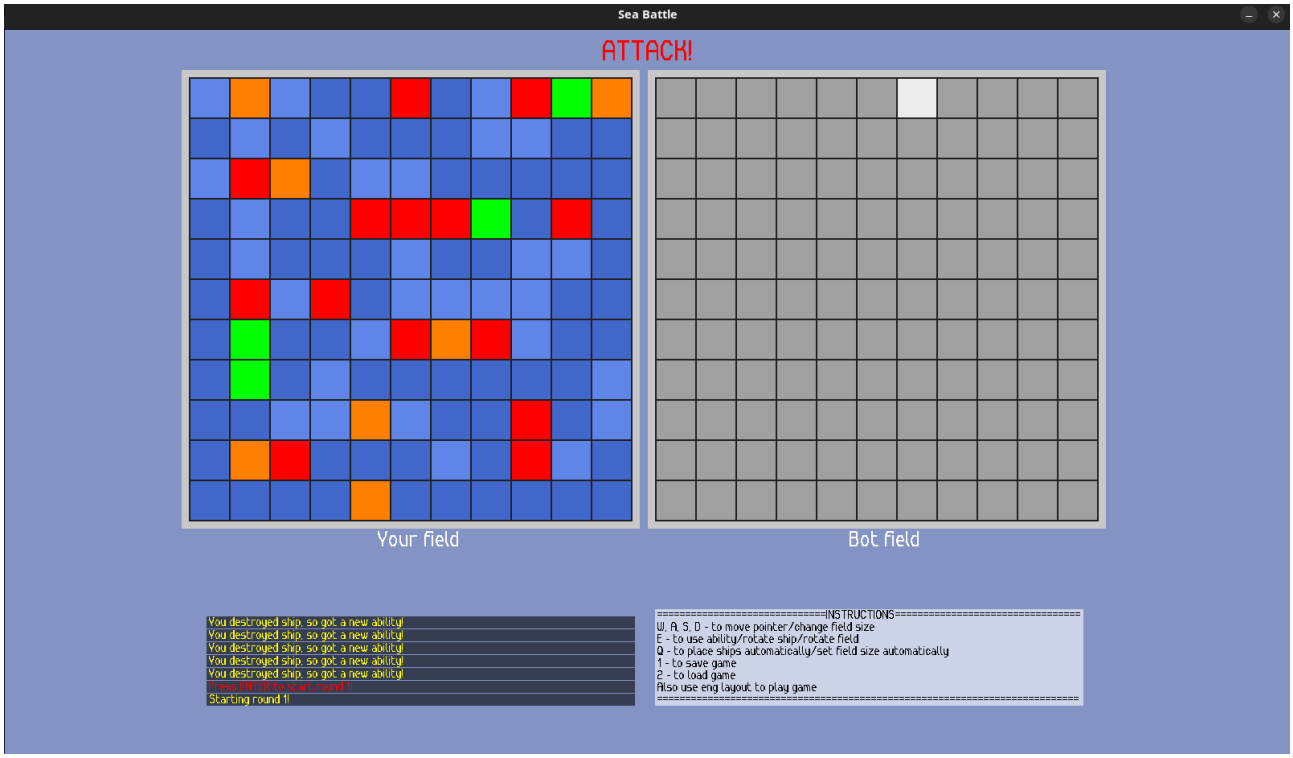
Игра 2



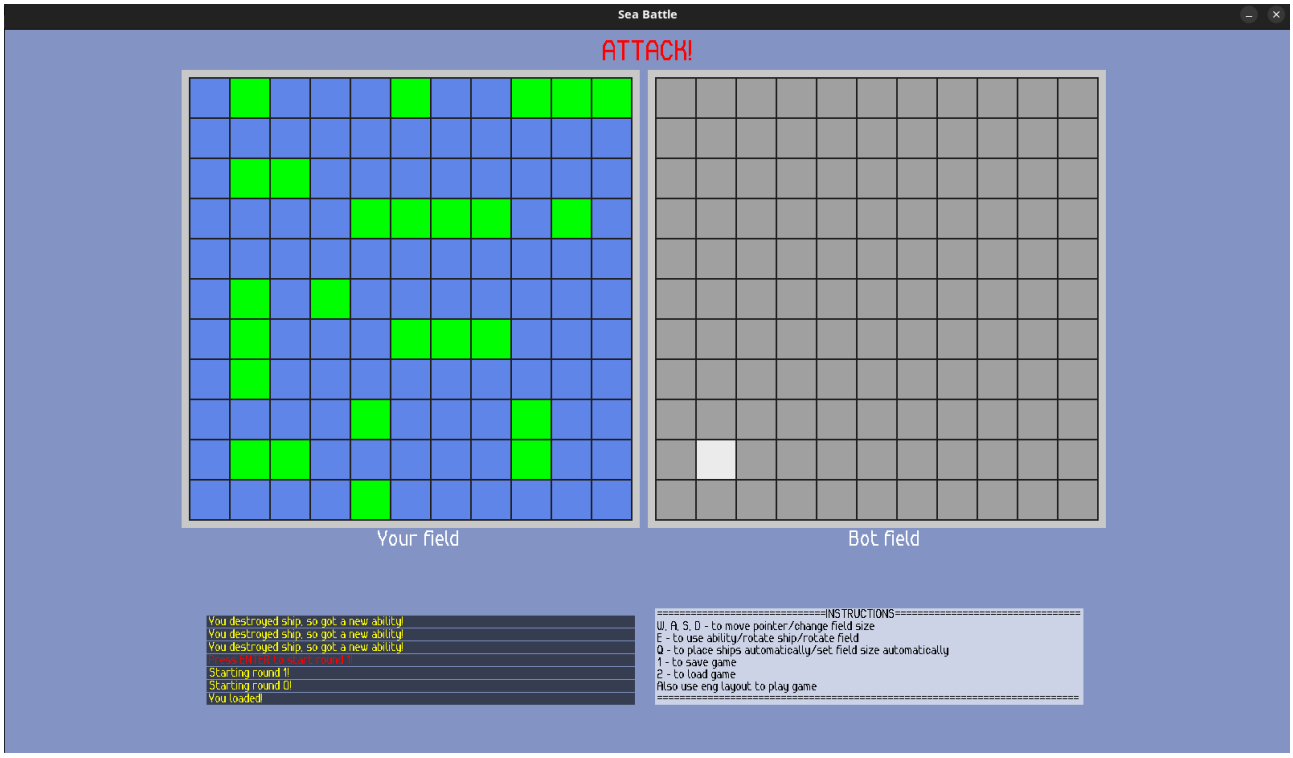
Выигрыш



Второй раунд



Загрузка



Выводы.

Были созданы классы для управления состоянием игры (*gameState*), обеспечивающие переключение между различными этапами игры (настройка поля, расстановка кораблей, ход игры, завершение игры). Для удобства работы с данными, были перегружены операторы потокового ввода и вывода для сериализации/десериализации состояния игры в формате *JSON*. Наконец, реализованы функции сохранения и загрузки состояния игры в файл, позволяющие возобновить игру с места остановки. В результате создана работоспособная игровая система с поддержкой сохранений.