МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. Шухова»
(БГТУ им. В. Г. Шухова)**

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Кафедра программного обеспечения вычислительной
техники и автоматизированных систем

Курсовая работа

по дисциплине: «Объектно-ориентированное программирование»
тема: Программная реализация алгоритмов
игры «Морской бой»

Выполнил: студент группы ПВ-233
Алексеенко Андрей Евгеньевич

_____

Принял: преподаватель
Морозов Данила Александрович

_____

Белгород, 2025 г.

Содержание

Введение

Целью данной курсовой работы является разработка программы, реализующей механику и алгоритмы классической игры «Морской бой». Эта игра, основанная на стратегии и логическом мышлении, предоставляет игрокам возможность развивать навыки планирования, анализа и предвидения действий соперника. Проект включает в себя создание программной платформы, которая симулирует игру с её правилами, графическим интерфейсом и возможностью взаимодействия с пользователем.

В ходе работы над проектом разработаны алгоритмы для размещения кораблей на игровом поле, обработки ходов, а также проверки условий победы. Разработан интуитивно понятный интерфейс, который в ходе игры позволяет сосредоточиться на стратегии и тактике, а не на сложностях управления.

Таким образом, данная курсовая работа не только позволила углубить знания в области объектно-ориентированного программирования и разработки игр, но и предоставит возможность создать увлекательный и интеллектуальный продукт.

Постановка задачи

Задачей курсовой работы является разработка программы, реализующей механику и алгоритмы головоломки «Морской бой», основанной на стратегическом размещении кораблей и логическом анализе ходов.

Программа позволяет пользователю играть в игру, включая размещение кораблей, выбор ходов и проверку попаданий. Программа отображает игровое поле, включая текущее состояние игры.

Техническая информация

Программа разработана с использованием объектно-ориентированного языка программирования C++ и фреймворка Qt, что обеспечило гибкость и удобство в создании графического интерфейса.

Правила игры

Цель: уничтожить все корабли противника (бота) до того, как это сделает он.

Правила, в соответствии с которыми необходимо развернуть флот:

- Корабли не должны пересекаться.
- Корабли не должны касаться сторонами.
- Корабли не должны касаться углами.

Состав флота:

- Однопалубные корабли: 4 ед.
- Двухпалубные корабли: 3 ед.
- Трёхпалубные корабли: 2 ед.
- Четырёхпалубный корабль: 1 ед.

Основная часть игры состоит из чередующихся ходов игрока и бота. Игрок открывает огонь по полю противника. В случае попадания он получает возможность сделать ещё один ход, а на клетку противника наносится метка. В противном случае ход переходит к боту. Игра завершается, когда все корабли одной из сторон уничтожены.

Объектная декомпозиция

Объектная декомпозиция включает 19 объектов. Пользователь получает информацию на экране и, основываясь на этих данных, отправляет запрос контроллеру, который выступает в роли промежуточного звена между визуальным представлением и программным обеспечением.

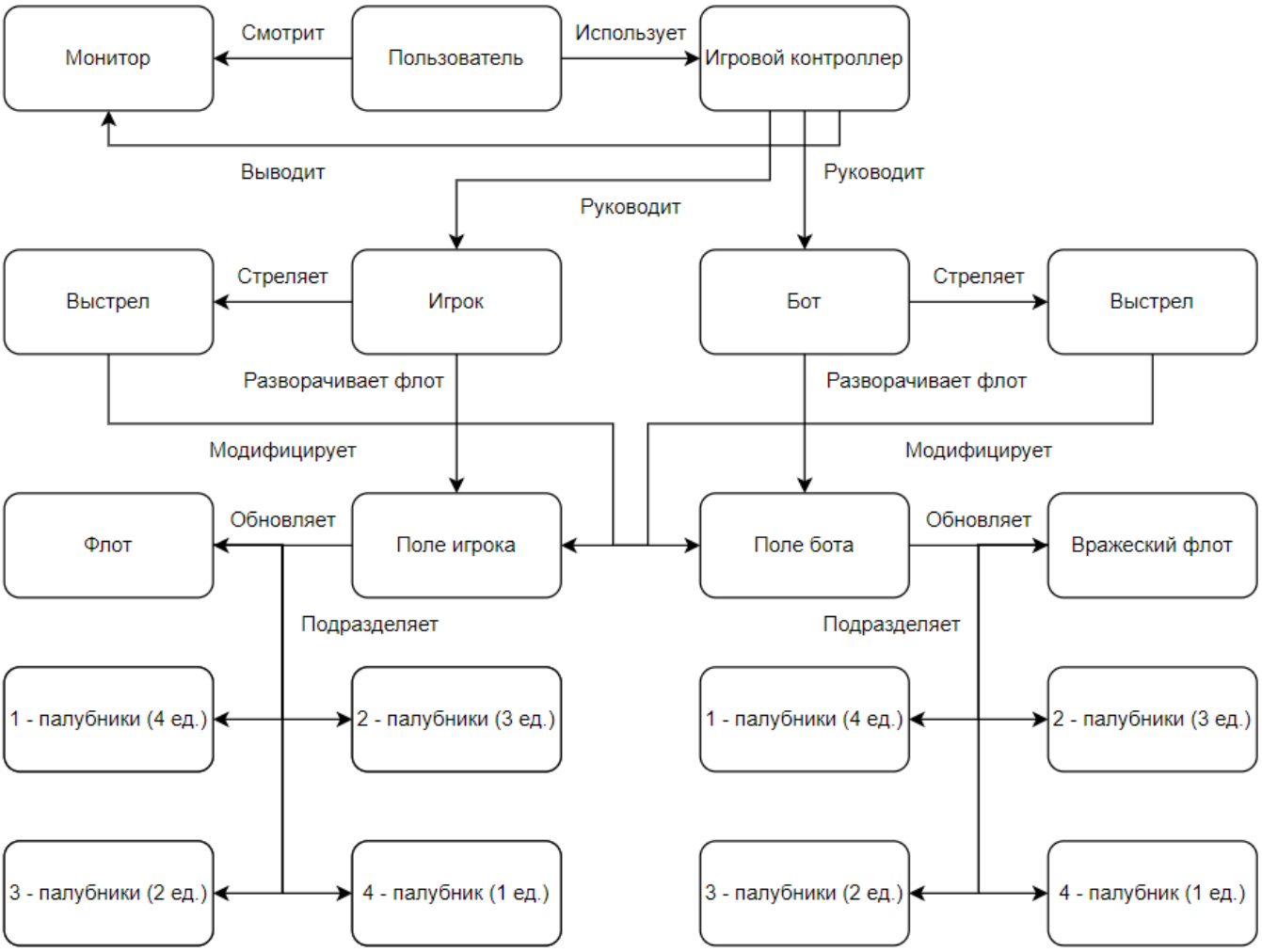
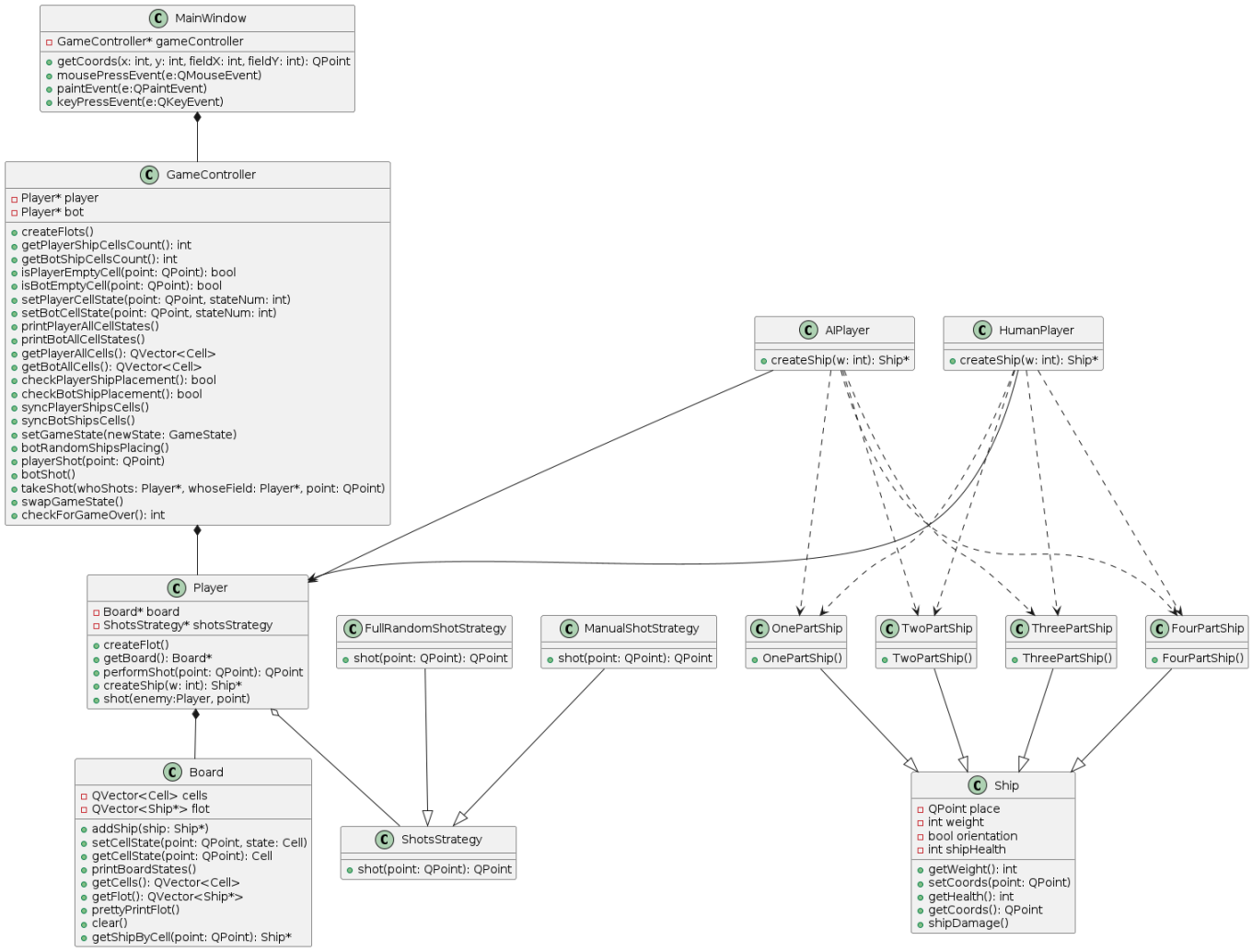


Диаграмма классов



Применено 2 паттерна проектирования:

– Фабричный метод (порождающий паттерн)
– Стратегия (поведенческий паттерн)

- Factory Method – определяет интерфейс для создания объектов и позволяет подклассам решать, какой класс инстанцировать. Этот паттерн делегирует создание объектов подклассам, что способствует инверсии зависимостей. В результате класс Player не зависит от конкретных реализаций класса Ship, что делает архитектуру слабо связанной. Класс Player объявляет фабричный метод, который реализуется в классах HumanPlayer и AIPlayer.
- Strategy – определяет группу алгоритмов, инкапсулирует их и делает взаимозаменяемыми. Этот паттерн позволяет изменять алгоритмы независимо от клиентов, которые их используют. Класс Player содержит ссылку на объект абстрактного класса ShotsStrategy (агрегация), который реализуется конкретными классами ManualShotStrategy и FullRandomShotStrategy. В каждом из наследников класса Player определяется свой конкретный алгоритм. Таким образом, классы типа Player не имеют информации о внутреннем устройстве переданных им алгоритмов. Это обеспечивает слабую связанность и гибкость спроектированной архитектуры, так как в любой момент можно изменить поведение классов Player, не изменяя сами классы.

Программная реализация

Ссылка на репозиторий

https://github.com/andrey5e/KR

Заголовочные файлы

ship.h:

```
#ifndef SHIP_H
#define SHIP_H

#include <QDebug>
#include <QPoint>

class Ship {
protected:
        int weight; // Вес (размер) корабля
        QPoint coords; // Координаты расположения корабля на игровом поле
        int shipHealth; // Здоровье корабля (количество оставшихся палуб)
        bool orientation; // Ориентация корабля (горизонтальная или вертикальная)
public:
        int getWeight(); // Получение веса корабля
        void setCoords(QPoint point); // Установка координат корабля
        int getHealth(); // Получение текущего здоровья корабля
        QPoint getCoords(); // Получение координат корабля
        void shipDamage(); // Нанесения урона кораблю
};

// Однопалубный корабль
class OnePartShip : public Ship {
public:
        OnePartShip() {
            weight = 1; // Вес корабля
            shipHealth = weight; // Здоровье = вес
        }
};

// Двухпалубный корабль
class TwoPartShip : public Ship {
public:
        TwoPartShip() {
            weight = 2; // Вес корабля
            shipHealth = weight; // Здоровье = вес
        }
};

// Трёхпалубный корабль
class ThreePartShip : public Ship {
public:
        ThreePartShip() {
            weight = 3; // Вес корабля
            shipHealth = weight; // Здоровье = вес
        }
};

// Четырёхпалубный корабль
class FourPartShip : public Ship {
public:
        FourPartShip() {
            weight = 4; // Вес корабля
            shipHealth = weight; // Здоровье = вес
        }
```

```
};

#endif // SHIP_H
```

shot.h:

```
#ifndef SHOT_H
#define SHOT_H

#include <QPoint>
#include <QDebug>

class ShotsStrategy {
public:
        virtual QPoint shot(QPoint point = QPoint(0, 0)) = 0;  // Выполнение выстрела
};

class ManualShotStrategy : public ShotsStrategy {
public:
        QPoint shot(QPoint point = QPoint(0, 0)) override;  // Ручной выстрел
};

class FullRandomShotStrategy : public ShotsStrategy {
public:
        QPoint shot(QPoint point = QPoint(0, 0)) override;  // Случайный выстрел
};

#endif // SHOT_H
```

defines.h:

```
#ifndef DEFINES_H
#define DEFINES_H

// Размеры игрового поля для игрока
const int MYFIELD_X = 41; // Ширина
const int MYFIELD_Y = 41; // Высота

// Половинные размеры игрового поля для центрования
const int MYFIELD_HALF_X = 148;
const int MYFIELD_HALF_Y = 148;

// Размеры игрового поля для противника
const int ENEMYFIELD_X = 323; // Ширина
const int ENEMYFIELD_Y = 42;  // Высота

// Половинные размеры игрового поля противника для центрования
const int ENEMYFIELD_HALF_X = 430;
const int ENEMYFIELD_HALF_Y = 148;

// Общие размеры игрового поля
const int FIELD_WIDTH = 216; // Ширина
const int FIELD_HEIGHT = 217; // Высота

// Размер ячейки на игровом поле (ширина / 10)
const int CELL_SIZE = FIELD_WIDTH / 10;

// Количество кораблей
const int SHIPS4COUNT = 1; // 4 - палубник
const int SHIPS3COUNT = 2; // 3 - палубники
const int SHIPS2COUNT = 3; // 2 - палубники
const int SHIPS1COUNT = 4; // 1 - палубники

// Задержка для тестов
const float FOR_TEST_BOT_DELAY = 0.5;
```

```
// Задержка для бота
const int REAL_BOT_DELAY = 1;

#endif // DEFINES_H
```

field.h:

```
#ifndef FIELD_H
#define FIELD_H

#include <QVector>
#include "ship.h"

enum Cell {
    EMPTY,
    DOT,
    SHIP,
    DEAD,
    DAMAGED
};

class Board {
public:
        Board();
        void addShip(Ship* ship); // Добавление корабля в флот
        void setCellState(QPoint point, Cell state); // Установка состояния ячейки на
игровом поле
        Cell getCellState(QPoint point); // Получение состояния ячейки на игровом
поле
        void printBoardStates(); // Вывод состояния игрового поля в виде матрицы
        QVector<Cell> getCells(); // Получение всех ячеек игрового поля
        QVector<Ship *> getFlot(); // Получения флота (списка кораблей)
        void prettyPrintFlot(); // Вывод информации о кораблях
        void clear(); // Очистка игрового поля, устанавливая все ячейки в состояние
EMPTY
        Ship *getShipByCell(QPoint point); // Получение корабля по координатам ячейки
private:
        QVector<Cell> cells;
        QVector<Ship*> flot;
};

#endif // FIELD_H
```

player.h:

```
#ifndef PLAYER_H
#define PLAYER_H

#include "field.h"
#include "defines.h"
#include "shot.h"

class Player {
public:
        Player();
        virtual ~Player();
        virtual Ship* createShip(int w) = 0;
        void createFlot(); // Создание флота
        Board *getBoard(); // Получение игрового поля игрока
        QPoint performShot(QPoint point = QPoint(-1, -1)); // Выстрел в заданную
точку
protected:
        Board* board;
        ShotsStrategy* shotS;
};
```

```
class HumanPlayer : public Player {
public:
        HumanPlayer() {
            shotS = new ManualShotStrategy();
        };
        Ship* createShip(int w) override; // Создание корабля
};

class AIPlayer : public Player {
public:
        AIPlayer() {
            shotS = new FullRandomShotStrategy();
        };
        Ship* createShip(int w) override; // Создание вражеского корабля
};

#endif // PLAYER_H
```

controller.h:

```
#ifndef CONTROLLER_H
#define CONTROLLER_H

#include "player.h"
#include <unistd.h>
#include <QLabel>

enum GameState {
    SHIPS_PLACING,
    PLAYER_TURN,
    ENEMY_TURN,
    GAMEOVER
};

class GameController {
public:
        GameController();
        ~GameController();
        GameState getGameState(); // Получение текущего состояния игры
        void createFlots(); // Создание флотов для игрока и бота
        int getPlayerShipCellsCount(); // Получение количества ячеек с кораблями у
игрока
        int getBotShipCellsCount(); // Получение количества ячеек с кораблями у бота
        bool isPlayerEmptyCell(QPoint point); // Проверка, является ли ячейка пустой
у игрока
        bool isBotEmptyCell(QPoint point); // Проверка, является ли ячейка пустой у
бота
        void setPlayerCellState(QPoint point, int stateNum); // Установка состояния
ячейки на доске у игрока
        void setBotCellState(QPoint point, int stateNum); // Установка состояния
ячейки на доске у бота
        Cell getPlayerCellState(QPoint point); // Получение состояния ячейки на доске
у игрока
        Cell getBotCellState(QPoint point); // Получение состояния ячейки на доске у
бота
        void printPlayerAllCellStates(); // Печать всех состояний ячеек на доске у
игрока
        void printBotAllCellStates(); // Печать всех состояний ячеек на доске у бота
        QVector<Cell> getPlayerAllCells(); // Получение всех ячеек на доске у игрока
        QVector<Cell> getBotAllCells(); // Получение всех ячеек на доске у бота
        bool checkPlayerShipPlacement(); // Проверка размещения кораблей у игрока
        bool checkBotShipPlacement(); // Проверка размещения кораблей у бота
        void syncPlayerShipsCells(); // Синхронизация ячеек кораблей у игрока
        void syncBotShipsCells(); // Синхронизация ячеек кораблей у бота
        void setGameState(GameState newState); // Установка нового состояния игры
        void botRandomShipsPlacing(); // Размещение кораблей бота случайным образом
```

```
        void playerShot(QPoint point); // Обработка выстрела игрока
        void botShot(); // Обработка выстрела бота
        QLabel *infoLabel;
        void takeShot(Player* whoShots, Player* whoseField, QPoint point); //
Обработка выстрела
        void swapGameState(); // Переключение состояния игры между ходами игрока и
бота
        int checkForGameOver(); // Проверка на окончание игры
private:
        GameState gameState;
        Player* player;
        Player* bot;
};

#endif // CONTROLLER_H
```

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QMouseEvent>
#include <QPixMap>
#include <QPainter>
#include <QImage>
#include <QVector>
#include <QKeyEvent>
#include <QApplication>
#include "controller.h"

class MainWindow : public QMainWindow{
        Q_OBJECT
public:
        MainWindow(QWidget *parent = nullptr);
        ~MainWindow();
        QPoint getCoords(int x, int y, int fieldX, int fieldY); // Получение
координат
private:
        GameController* gameController;
        protected:
        void mousePressEvent(QMouseEvent *event) override; // Обработка нажатия
правой кнопкой мыши
        void paintEvent(QPaintEvent *event) override; // Визуализация игрового поля
        void keyPressEvent(QKeyEvent *event) override; // Обработка нажатий клавиш
};

#endif // MAINWINDOW_H
```

Файлы реализации

ship.cpp:

```
#include "ship.h"

// Получение веса корабля
int Ship::getWeight() {
    return weight;
}

// Установка координат корабля
void Ship::setCoords(QPoint point) {
    coords = point;
}

// Получение текущего здоровья корабля
int Ship::getHealth() {
    return shipHealth;
}

// Получение координат корабля
QPoint Ship::getCoords() {
    return coords;
}

// Нанесение урона кораблю
void Ship::shipDamage() {
    shipHealth--;
}
```

shot.cpp:

```
#include "shot.h"

// Ручной выстрел
QPoint ManualShotStrategy::shot(QPoint point) {
    return point;
}

// Случайный выстрел
QPoint FullRandomShotStrategy::shot(QPoint point) {
    srand(time(NULL));
    return QPoint(rand() % 10, rand() % 10);
}
```

field.cpp:

```
#include "field.h"

Board::Board() {
    cells.fill(EMPTY, 100); // Инициализация игрового поля 10x10, заполняя все ячейки
состоянием EMPTY
}

// Добавление корабля в флот
void Board::addShip(Ship *ship) {
    flot.push_back(ship);
}

// Установка состояния ячейки на игровом поле
void Board::setCellState(QPoint point, Cell state) {
    if (point.x() < 10 && point.x() >= 0 && point.y() < 10 && point.y() >= 0)
        cells[point.x() + 10 * point.y()] = state;
}

// Получение состояния ячейки на игровом поле
```

```
Cell Board::getCellState(QPoint point) {
    if (point.x() > 9 || point.y() > 9 || point.x() < 0 || point.y() < 0) {
        return Cell::EMPTY;
    }
    return cells[point.x() + 10 * point.y()];
}

// Вывод состояния игрового поля в виде матрицы
void Board::printBoardStates() {
    for (int i = 0; i < 10; i++) {
        QString str = "";
        for (int j {0}; j < 10; j++) {
            str += QString::number(cells[i * 10 + j]) + " ";
        }
        qDebug() << str;
    }
    qDebug() << "\n";
}

// Получение всех ячеек игрового поля
QVector<Cell> Board::getCells() {
    return cells;
}

// Получения флота (списка кораблей)
QVector<Ship *> Board::getFlot() {
    return flot;
}

// Вывод информации о кораблях
void Board::prettyPrintFlot() {
    for (Ship* ship : flot) {
        qDebug() << "Point: " << ship->getWeight() << ", " << ship->getCoords();
    }
}

// Очистка игрового поля, устанавливая все ячейки в состояние EMPTY
void Board::clear() {
    for (Cell &cell : cells) {
        cell = Cell::EMPTY;
    }
}

// Получение корабля по координатам ячейки
Ship *Board::getShipByCell(QPoint point) {
    if (getCellState(point) != Cell::SHIP) {
        qDebug() << "В функцию getShipByCell передана клетка без корабля.";
        return nullptr;
    }
    for (Ship* ship : flot) {
        if (ship->getCoords() == point)
            return ship;
    }
    if (getCellState(QPoint(point.x() - 1, point.y())) == Cell::SHIP ||
        getCellState(QPoint(point.x() - 1, point.y())) == Cell::DAMAGED) {
        while (true) {
            point = QPoint(point.x() - 1, point.y());
            for (Ship* ship : flot) {
                if (ship->getCoords() == point)
                    return ship;
            }
        }
    } else {
        while (true) {
            point = QPoint(point.x(), point.y() - 1);
            for (Ship* ship : flot) {
                if (ship->getCoords() == point)
```

```
                    return ship;
                }
            }
        }
        qDebug() << "Корабль не найден.";
        return nullptr;
}
```

player.cpp:

```
#include "player.h"

Player::Player() {
    board = new Board(); // Формирование игрового поля
}

Player::~Player() {
    delete board; // Освобождение памяти, занятой игровым полем
}

// Создание флота
void Player::createFlot() {
    for (int i {0}; i < SHIPS4COUNT; i++) {
        board->addShip(createShip(4));
    }
    for (int i {0}; i < SHIPS3COUNT; i++) {
        board->addShip(createShip(3));
    }
    for (int i {0}; i < SHIPS2COUNT; i++) {
        board->addShip(createShip(2));
    }
    for (int i {0}; i < SHIPS1COUNT; i++) {
        board->addShip(createShip(1));
    }
}

// Получение игрового поля игрока
Board *Player::getBoard() {
    return board;
}

// Выстрел в заданную точку
QPoint Player::performShot(QPoint point) {
    return shotS->shot(point);
}

// Создание корабля
Ship* HumanPlayer::createShip(int w) {
    if (w == 1) {
        return new OnePartShip();
    } else if (w == 2) {
        return new TwoPartShip();
    } else if (w == 3) {
        return new ThreePartShip();
    } else if (w == 4) {
        return new FourPartShip();
    } else {
        qDebug() << "Некорректный вес корабля.";
        return nullptr;
    }
}

// Создание вражеского корабля
Ship* AIPlayer::createShip(int w) {
    if (w == 1) {
        return new OnePartShip();
```

```
    } else if (w == 2) {
        return new TwoPartShip();
    } else if (w == 3) {
        return new ThreePartShip();
    } else if (w == 4) {
        return new FourPartShip();
    } else {
        qDebug() << "Некорректный вес корабля.";
        return nullptr;
    }
}
```

controller.cpp:

```
#include "controller.h"

GameController::GameController() {
    gameState = GameState::SHIPS_PLACING;
    player = new HumanPlayer(); // Создание игрока
    bot = new AIPlayer(); // Создание бота
    createFlots(); // Создание флотов для игрока и бота
}

GameController::~GameController() {
    delete player; // Удаление игрока
    delete bot; // Удаление бота
}

// Получение текущего состояния игры
GameState GameController::getGameState() {
    return gameState;
}

// Создание флотов для игрока и бота
void GameController::createFlots() {
    player->createFlot();
    bot->createFlot();
}

// Подсчёт количества ячеек с кораблями на доске игрока
int getShipCellsCount(Player* somePlayer) {
    int count = 0;
    Board* board = somePlayer->getBoard();
    for (int i {0}; i < 10; i++) {
        for (int j {0}; j < 10; j++) {
            if (board->getCellState(QPoint(j, i)) == Cell::SHIP)
                count++;
        }
    }
    return count;
}

// Получение количества ячеек с кораблями у игрока
int GameController::getPlayerShipCellsCount() {
    return getShipCellsCount(player);
}

// Получение количества ячеек с кораблями у бота
int GameController::getBotShipCellsCount() {
    return getShipCellsCount(bot);
}

// Проверка, является ли ячейка пустой
bool isEmptyCell(Player* somePlayer, QPoint point) {
    Board* board = somePlayer->getBoard();
    return board->getCellState(point) == Cell::EMPTY;
```

```
}

// Проверка, является ли ячейка пустой у игрока
bool GameController::isPlayerEmptyCell(QPoint point) {
    return isEmptyCell(player, point);
}

// Проверка, является ли ячейка пустой у бота
bool GameController::isBotEmptyCell(QPoint point) {
    return isEmptyCell(bot, point);
}

// Установка состояния ячейки на доске
void setCellState(Player* somePlayer, QPoint point, int stateNum) {
    Board* board = somePlayer->getBoard();
    board->setCellState(point, static_cast<Cell>(stateNum));
}

// Установка состояния ячейки на доске у бота
void GameController::setBotCellState(QPoint point, int stateNum) {
    return setCellState(bot, point, stateNum);
}

// Получение состояния ячейки на доске
Cell getCellState(QPoint point, Player* somePlayer) {
    Board *board = somePlayer->getBoard();
    return board->getCellState(point);
}

// Получение состояния ячейки на доске у бота
Cell GameController::getBotCellState(QPoint point) {
    return getCellState(point, bot);
}

// Получение состояния ячейки на доске у игрока
Cell GameController::getPlayerCellState(QPoint point) {
    return getCellState(point, player);
}

// Установка состояния ячейки на доске у игрока
void GameController::setPlayerCellState(QPoint point, int stateNum) {
    return setCellState(player, point, stateNum);
}

// Печать всех состояний ячеек на доске
void printAllCellStates(Player* somePlayer) {
    somePlayer->getBoard()->printBoardStates();
}

// Печать всех состояний ячеек на доске у игрока
void GameController::printPlayerAllCellStates() {
    printAllCellStates(player);
}

// Печать всех состояний ячеек на доске у бота
void GameController::printBotAllCellStates() {
    printAllCellStates(bot);
}

// Получение всех ячеек на доске
QVector<Cell> getAllCells(Player* somePlayer) {
    return somePlayer->getBoard()->getCells();
}

// Получение всех ячеек на доске у игрока
QVector<Cell> GameController::getPlayerAllCells() {
    return getAllCells(player);
```

```
}

// Получение всех ячеек на доске у бота
QVector<Cell> GameController::getBotAllCells() {
    return getAllCells(bot);
}

// Проверка, находится ли корабль в заданной позиции
bool isShip(Player* somePlayer, int size, int x, int y) {
    Board* board = somePlayer->getBoard();
    if (x > 0 && board->getCellState(QPoint(x - 1, y)) != Cell::EMPTY)
        return false;
    if (y > 0 && board->getCellState(QPoint(x, y - 1)) != Cell::EMPTY)
        return false;
    if (board->getCellState(QPoint(x, y)) == Cell::EMPTY)
        return false;
    int tmp = x;
    int num = 0;
    while (board->getCellState(QPoint(tmp, y)) != Cell::EMPTY && tmp < 10) {
        tmp++;
        num++;
    }
    if (num == size) {
        if (board->getCellState(QPoint(x, y + 1)) != Cell::EMPTY)
            return false;
        if (board->getCellState(QPoint(x - 1, y - 1)) != Cell::EMPTY ||
            board->getCellState(QPoint(x - 1, y + 1)) != Cell::EMPTY ||
            board->getCellState(QPoint(x + size, y - 1)) != Cell::EMPTY ||
            board->getCellState(QPoint(x + size, y + 1)) != Cell::EMPTY)
            return false;
        return true;
    }
    tmp = y;
    num = 0;
    while (board->getCellState(QPoint(x, tmp)) != Cell::EMPTY && tmp < 10) {
        tmp++;
        num++;
    }
    if (num == size) {
        if (board->getCellState(QPoint(x + 1, y)) != Cell::EMPTY)
            return false;
        if (board->getCellState(QPoint(x - 1, y - 1)) != Cell::EMPTY ||
            board->getCellState(QPoint(x + 1, y - 1)) != Cell::EMPTY ||
            board->getCellState(QPoint(x - 1, y + size)) != Cell::EMPTY ||
            board->getCellState(QPoint(x + 1, y + size)) != Cell::EMPTY)
            return false;
        return true;
    }
    return false;
}

// Подсчёт количества кораблей заданного размера на доске
int shipNum(Player* somePlayer, int size) {
    int shipNumber = 0;
    for(int i = 0; i < 10; i++)
        for(int j = 0; j < 10; j++)
            if(isShip(somePlayer, size, j, i))
                shipNumber++;
    return shipNumber;
}

// Проверка размещения кораблей на доске (20)
bool checkShipPlacement(Player* somePlayer) {
    if (getShipCellsCount(somePlayer) == 20) {
        if (shipNum(somePlayer, 1) == 4 &&
            shipNum(somePlayer, 2) == 3 &&
            shipNum(somePlayer, 3) == 2 &&
```

```
            shipNum(somePlayer, 4) == 1) {
            return 1;
        } else {
            return 0;
        }
    } else {
        return 0;
    }
}

// Проверка размещения кораблей у игрока
bool GameController::checkPlayerShipPlacement() {
    return checkShipPlacement(player);
}

// Проверка размещения кораблей у бота
bool GameController::checkBotShipPlacement() {
    return checkShipPlacement(bot);
}

// Синхронизация ячеек кораблей на доске
void syncShipsCells(Player* somePlayer) {
    Board* board = somePlayer->getBoard();
    QPoint fourPartShip;
    for (int i = 0; i < 10; i++)
        for (int j = 0; j < 10; j++)
            if (isShip(somePlayer, 4, j, i)) {
                fourPartShip.setX(j);
                fourPartShip.setY(i);
            }
    QPoint threePartShip1(-1, -1);
    QPoint threePartShip2(-1, -1);
    for (int i = 0; i < 10; i++)
        for (int j = 0; j < 10; j++)
            if (isShip(somePlayer, 3, j, i)) {
                if (threePartShip1.x() == -1) {
                    threePartShip1.setX(j);
                    threePartShip1.setY(i);
                } else {
                    threePartShip2.setX(j);
                    threePartShip2.setY(i);
                }
            }
    QPoint twoPartShip1(-1, -1);
    QPoint twoPartShip2(-1, -1);
    QPoint twoPartShip3(-1, -1);
    for (int i = 0; i < 10; i++)
        for (int j = 0; j < 10; j++)
            if (isShip(somePlayer, 2, j, i)) {
                if (twoPartShip1.x() == -1) {
                    twoPartShip1.setX(j);
                    twoPartShip1.setY(i);
                } else if (twoPartShip2.x() == -1) {
                    twoPartShip2.setX(j);
                    twoPartShip2.setY(i);
                } else {
                    twoPartShip3.setX(j);
                    twoPartShip3.setY(i);
                }
            }
    QPoint onePartShip1(-1, -1);
    QPoint onePartShip2(-1, -1);
    QPoint onePartShip3(-1, -1);
    QPoint onePartShip4(-1, -1);
    for (int i = 0; i < 10; i++)
        for (int j = 0; j < 10; j++)
            if (isShip(somePlayer, 1, j, i)) {
```

```
                if (onePartShip1.x() == -1) {
                    onePartShip1.setX(j);
                    onePartShip1.setY(i);
                } else if (onePartShip2.x() == -1) {
                    onePartShip2.setX(j);
                    onePartShip2.setY(i);
                } else if (onePartShip3.x() == -1){
                    onePartShip3.setX(j);
                    onePartShip3.setY(i);
                } else {
                    onePartShip4.setX(j);
                    onePartShip4.setY(i);
                }
            }
    // Синхронизация координат кораблей на доске
    for (Ship* ship : board->getFlot()) {
        if (ship->getWeight() == 4) {
            ship->setCoords(fourPartShip);
        } else if (ship->getWeight() == 3) {
            if (threePartShip1.x() != -1) {
                ship->setCoords(threePartShip1);
                threePartShip1.setX(-1);
            } else {
                ship->setCoords(threePartShip2);
            }
        } else if (ship->getWeight() == 2) {
            if (twoPartShip1.x() != -1) {
                ship->setCoords(twoPartShip1);
                twoPartShip1.setX(-1);
            } else if (twoPartShip2.x() != -1) {
                ship->setCoords(twoPartShip2);
                twoPartShip2.setX(-1);
            } else {
                ship->setCoords(twoPartShip3);
            }
        } else {
            if (onePartShip1.x() != -1) {
                ship->setCoords(onePartShip1);
                onePartShip1.setX(-1);
            } else if (onePartShip2.x() != -1) {
                ship->setCoords(onePartShip2);
                onePartShip2.setX(-1);
            } else if (onePartShip3.x() != -1) {
                ship->setCoords(onePartShip3);
                onePartShip3.setX(-1);
            } else {
                ship->setCoords(onePartShip4);
            }
        }
    }
    board->prettyPrintFlot();
}

// Синхронизация ячеек кораблей у игрока
void GameController::syncPlayerShipsCells() {
    return syncShipsCells(player);
}

// Синхронизация ячеек кораблей у бота
void GameController::syncBotShipsCells() {
    return syncShipsCells(bot);
}

// Установка нового состояния игры
void GameController::setGameState(GameState newState) {
    gameState = newState;
}
```

```
// Размещение кораблей бота случайным образом (только горизонтальная ориентация)
void GameController::botRandomShipsPlacing() {
    Board* board = bot->getBoard();
    srand(time(NULL));
    QPoint fourPartShip;
    while (true) {
        fourPartShip.setX(rand() % 10);
        fourPartShip.setY(rand() % 10);
        if (fourPartShip.x() < 7) {
            board->setCellState(QPoint(fourPartShip.x(), fourPartShip.y()),
Cell::SHIP);
            board->setCellState(QPoint(fourPartShip.x() + 1, fourPartShip.y()),
Cell::SHIP);
            board->setCellState(QPoint(fourPartShip.x() + 2, fourPartShip.y()),
Cell::SHIP);
            board->setCellState(QPoint(fourPartShip.x() + 3, fourPartShip.y()),
Cell::SHIP);
            if (isShip(bot, 4, fourPartShip.x(), fourPartShip.y()))
                break;
            else {
                board->setCellState(QPoint(fourPartShip.x(), fourPartShip.y()),
Cell::EMPTY);
                board->setCellState(QPoint(fourPartShip.x() + 1, fourPartShip.y()),
Cell::EMPTY);
                board->setCellState(QPoint(fourPartShip.x() + 2, fourPartShip.y()),
Cell::EMPTY);
                board->setCellState(QPoint(fourPartShip.x() + 3, fourPartShip.y()),
Cell::EMPTY);
            }
        }
    }
    // Генерация и размещение двух трехпалубных кораблей
    QPoint threePartShip1(-1, -1);
    QPoint threePartShip2(-1, -1);
    for (int i {0}; i < 2; i++) {
        while (true) {
            if (threePartShip1.x() == -1) {
                threePartShip1.setX(rand() % 10);
                threePartShip1.setY(rand() % 10);
                if (threePartShip1.x() < 8 && board->getCellState(threePartShip1) ==
Cell::EMPTY
                    && board->getCellState(QPoint(threePartShip1.x() + 1,
threePartShip1.y())) == Cell::EMPTY
                    && board->getCellState(QPoint(threePartShip1.x() + 2,
threePartShip1.y())) == Cell::EMPTY) {
                    board->setCellState(QPoint(threePartShip1.x(),
threePartShip1.y()), Cell::SHIP);
                    board->setCellState(QPoint(threePartShip1.x() + 1,
threePartShip1.y()), Cell::SHIP);
                    board->setCellState(QPoint(threePartShip1.x() + 2,
threePartShip1.y()), Cell::SHIP);
                    if (isShip(bot, 3, threePartShip1.x(), threePartShip1.y())) {
                        break;
                    }
                    else {
                        board->setCellState(QPoint(threePartShip1.x(),
threePartShip1.y()), Cell::EMPTY);
                        board->setCellState(QPoint(threePartShip1.x() + 1,
threePartShip1.y()), Cell::EMPTY);
                        board->setCellState(QPoint(threePartShip1.x() + 2,
threePartShip1.y()), Cell::EMPTY);
                    }
                }
                threePartShip1.setX(-1);
            } else {
                threePartShip2.setX(rand() % 10);
```

```
                threePartShip2.setY(rand() % 10);
                if (threePartShip2.x() < 8 && board->getCellState(threePartShip2) ==
Cell::EMPTY
                    && board->getCellState(QPoint(threePartShip2.x() + 1,
threePartShip2.y())) == Cell::EMPTY
                    && board->getCellState(QPoint(threePartShip2.x() + 2,
threePartShip2.y())) == Cell::EMPTY) {
                    board->setCellState(QPoint(threePartShip2.x(),
threePartShip2.y()), Cell::SHIP);
                    board->setCellState(QPoint(threePartShip2.x() + 1,
threePartShip2.y()), Cell::SHIP);
                    board->setCellState(QPoint(threePartShip2.x() + 2,
threePartShip2.y()), Cell::SHIP);
                    if (isShip(bot, 3, threePartShip2.x(), threePartShip2.y())) {
                        break;
                    }
                    else {
                        board->setCellState(QPoint(threePartShip2.x(),
threePartShip2.y()), Cell::EMPTY);
                        board->setCellState(QPoint(threePartShip2.x() + 1,
threePartShip2.y()), Cell::EMPTY);
                        board->setCellState(QPoint(threePartShip2.x() + 2,
threePartShip2.y()), Cell::EMPTY);
                    }
                }
                threePartShip2.setX(-1);
            }
        }
    }
    // Генерация и размещение трех двухпалубных кораблей
    QPoint twoPartShip1(-1, -1);
    QPoint twoPartShip2(-1, -1);
    QPoint twoPartShip3(-1, -1);
    for (int i {0}; i < 3; i++) {
        while (true) {
            if (twoPartShip1.x() == -1) {
                twoPartShip1.setX(rand() % 10);
                twoPartShip1.setY(rand() % 10);
                if (twoPartShip1.x() < 9 && board->getCellState(twoPartShip1) ==
Cell::EMPTY
                    && board->getCellState(QPoint(twoPartShip1.x() + 1,
twoPartShip1.y())) == Cell::EMPTY) {
                    board->setCellState(QPoint(twoPartShip1.x(), twoPartShip1.y()),
Cell::SHIP);
                    board->setCellState(QPoint(twoPartShip1.x() + 1,
twoPartShip1.y()), Cell::SHIP);
                    if (isShip(bot, 2, twoPartShip1.x(), twoPartShip1.y())) {
                        break;
                    }
                    else {
                        board->setCellState(QPoint(twoPartShip1.x(),
twoPartShip1.y()), Cell::EMPTY);
                        board->setCellState(QPoint(twoPartShip1.x() + 1,
twoPartShip1.y()), Cell::EMPTY);
                    }
                }
                twoPartShip1.setX(-1);
            } else if (twoPartShip2.x() == - 1) {
                twoPartShip2.setX(rand() % 10);
                twoPartShip2.setY(rand() % 10);
                if (twoPartShip2.x() < 9 && board->getCellState(twoPartShip2) ==
Cell::EMPTY
                    && board->getCellState(QPoint(twoPartShip2.x() + 1,
twoPartShip2.y())) == Cell::EMPTY) {
                    board->setCellState(QPoint(twoPartShip2.x(), twoPartShip2.y()),
Cell::SHIP);
                    board->setCellState(QPoint(twoPartShip2.x() + 1,
```

```
twoPartShip2.y()), Cell::SHIP);
                    if (isShip(bot, 2, twoPartShip2.x(), twoPartShip2.y())) {
                        break;
                    }
                    else {
                        board->setCellState(QPoint(twoPartShip2.x(),
twoPartShip2.y()), Cell::EMPTY);
                        board->setCellState(QPoint(twoPartShip2.x() + 1,
twoPartShip2.y()), Cell::EMPTY);
                    }
                }
                twoPartShip2.setX(-1);
            } else {
                twoPartShip3.setX(rand() % 10);
                twoPartShip3.setY(rand() % 10);
                if (twoPartShip3.x() < 9 && board->getCellState(twoPartShip3) ==
Cell::EMPTY
                    && board->getCellState(QPoint(twoPartShip3.x() + 1,
twoPartShip3.y())) == Cell::EMPTY) {
                    board->setCellState(QPoint(twoPartShip3.x(), twoPartShip3.y()),
Cell::SHIP);
                    board->setCellState(QPoint(twoPartShip3.x() + 1,
twoPartShip3.y()), Cell::SHIP);
                    if (isShip(bot, 2, twoPartShip3.x(), twoPartShip3.y())) {
                        break;
                    }
                    else {
                        board->setCellState(QPoint(twoPartShip3.x(),
twoPartShip3.y()), Cell::EMPTY);
                        board->setCellState(QPoint(twoPartShip3.x() + 1,
twoPartShip3.y()), Cell::EMPTY);
                    }
                }
                twoPartShip3.setX(-1);
            }
        }
    }
    // Генерация однопалубников - используем другой метод: простой перебор
    while (!checkBotShipPlacement()) {
        QPoint onePartShip(rand() % 10, rand() % 10);
        if (board->getCellState(onePartShip) != Cell::EMPTY)
            continue;
        board->setCellState(onePartShip, Cell::SHIP);
        if (!isShip(bot, 1, onePartShip.x(), onePartShip.y()))
            board->setCellState(onePartShip, Cell::EMPTY);
    }
    syncBotShipsCells();
}

// Переключение состояния игры между ходами игрока и бота
void GameController::swapGameState() {
    if (getGameState() == GameState::ENEMY_TURN)
        setGameState(GameState::PLAYER_TURN);
    else if (getGameState() == GameState::PLAYER_TURN){
        setGameState(GameState::ENEMY_TURN);
    }
}

// Проверка на окончание игры
int GameController::checkForGameOver() {
    Board* playerBoard = player->getBoard();
    Board* botBoard = bot->getBoard();
    QVector<Ship*> botFlot = botBoard->getFlot();
    QVector<Ship*> playerFlot = playerBoard->getFlot();
    int playerSummuryHelth = 0;
    for (Ship* ship : playerFlot) {
        playerSummuryHelth += ship->getHealth();
```

```
    }
    if (playerSummuryHelth == 0) {
        return 1;
    }
    int botSummuryHelth = 0;
    for (Ship* ship : botFlot) {
        botSummuryHelth += ship->getHealth();
    }
    if (botSummuryHelth == 0) {
        return 2;
    }
    return 0;
}

// Обработка выстрела
void GameController::takeShot(Player* whoShots, Player* whoseField, QPoint point) {
    Board* board = whoseField->getBoard();
    QPoint shotedPoint = whoShots->performShot(point);
    if (board->getCellState(shotedPoint) == Cell::EMPTY) {
        board->setCellState(shotedPoint, Cell::DOT);
        swapGameState();
        if (getGameState() == GameState::ENEMY_TURN)
            infoLabel->setText("Ход противника!");
        else {
            infoLabel->setText("Ваш ход!");
        }
    } else if (board->getCellState(shotedPoint) == Cell::SHIP) {
        Ship* shotedShip = board->getShipByCell(shotedPoint);
        shotedShip->shipDamage();
        int shipHealth = shotedShip->getHealth();
        board->setCellState(shotedPoint, Cell::DAMAGED);
        if (shipHealth != 0) {
            int d = 0;
        } else {
            if (board->getCellState(QPoint(shotedShip->getCoords().x() + 1,
shotedShip->getCoords().y())) == Cell::DAMAGED || shotedShip->getWeight() == 1) {
                for (int i {0}; i < shotedShip->getWeight(); i++) {
                    board->setCellState(QPoint(shotedShip->getCoords().x() + i,
shotedShip->getCoords().y()), Cell::DEAD);
                }
                for (int i {-1}; i < shotedShip->getWeight() + 1; i++) {
                    board->setCellState(QPoint(shotedShip->getCoords().x() + i,
shotedShip->getCoords().y() + 1), Cell::DOT);
                    board->setCellState(QPoint(shotedShip->getCoords().x() + i,
shotedShip->getCoords().y() - 1), Cell::DOT);
                }
                board->setCellState(QPoint(shotedShip->getCoords().x() - 1,
shotedShip->getCoords().y()), Cell::DOT);
                board->setCellState(QPoint(shotedShip->getCoords().x() + shotedShip-
>getWeight(), shotedShip->getCoords().y()), Cell::DOT);
            } else {
                for (int i {0}; i < shotedShip->getWeight(); i++) {
                    board->setCellState(QPoint(shotedShip->getCoords().x(),
shotedShip->getCoords().y() + i), Cell::DEAD);
                }
                for (int i {-1}; i < shotedShip->getWeight() + 1; i++) {
                    board->setCellState(QPoint(shotedShip->getCoords().x() + 1,
shotedShip->getCoords().y() + i), Cell::DOT);
                    board->setCellState(QPoint(shotedShip->getCoords().x() - 1,
shotedShip->getCoords().y() + i), Cell::DOT);
                }
                board->setCellState(QPoint(shotedShip->getCoords().x(), shotedShip-
>getCoords().y() - 1), Cell::DOT);
                board->setCellState(QPoint(shotedShip->getCoords().x(), shotedShip-
>getCoords().y() + shotedShip->getWeight()), Cell::DOT);
            }
            qDebug() << "Уничтожен" << shotedShip->getWeight() << "- палубник.";
```

```
        }
    }
}

// Обработка выстрела игрока
void GameController::playerShot(QPoint point) {
    takeShot(player, bot, point);
}

// Обработка выстрела бота
void GameController::botShot() {
    takeShot(bot, player, QPoint(-1, -1));
    sleep(FOR_TEST_BOT_DELAY);
}
```

mainwindow.cpp:

```
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent) {
        setFixedSize(555, 289); // Установка размера главного окна
        gameController = new GameController(); // Инициализация контроллера игры
        qDebug() << "Главное окно успешно создано. Контроллер инициализирован."; //
Отладка
        QPalette pal;
        pal.setBrush(QPalette::Active, QPalette::Window,
QBrush(QPixmap("C:/LR/field.png"))); // Установка фона главного окна
        this->setPalette(pal);
        gameController->infoLabel = new QLabel(this);
        gameController->infoLabel->move(200, 260);
        gameController->infoLabel->setText("Разверните флот!");
        gameController->infoLabel->setFixedSize(150, 28);
        gameController->infoLabel->setStyleSheet("font-weight: bold; border-style:
outset; border-width: 1px; border-radius: 5px;");
        gameController->infoLabel->setAlignment(Qt::AlignCenter);
}

MainWindow::~MainWindow() {
    delete gameController; // Освобождение памяти при уничтожении главного окна
}

// Получение координат
QPoint MainWindow::getCoords(int x, int y, int fieldX, int fieldY) {
    QPoint res;
    res.setX(-1);
    res.setY(-1);
    if (x < fieldX || x > (fieldX + FIELD_WIDTH) || y < fieldY || y > (fieldY +
FIELD_HEIGHT))
        return res;
    double cfx = 1.0 * FIELD_WIDTH / 10.0;
    double cfy = 1.0 * FIELD_HEIGHT / 10.0;
    res.setX(1.0 * (x - fieldX) / cfx);
    res.setY(1.0 * (y - fieldY) / cfy);
    return res;
}

// Обработка нажатия кнопки мыши
void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::RightButton) {
        QPoint pos = event->pos(); // Получение позиции курсора
        qDebug() << "Mouse click at (" << pos.x() << "," << pos.y() << ")"; //
Отладка
        qDebug() << "\nПоле бота:";
        gameController->printBotAllCellStates();
    }
    else if (event->button() == Qt::LeftButton) {
```

```
        if (gameController->getGameState() == GameState::SHIPS_PLACING) {
            QPointF pos = event->position();
            if (pos.x() >= MYFIELD_X &&
                pos.x() <= FIELD_WIDTH + MYFIELD_X &&
                pos.y() >= MYFIELD_Y &&
                pos.y() <= FIELD_HEIGHT + MYFIELD_Y) {
                QPoint qp = getCoords(pos.x(), pos.y(), MYFIELD_X, MYFIELD_Y);
                if (qp.x() == 10)
                    qp.setX(9);
                if (qp.y() == 10)
                    qp.setY(9);
                if (gameController->isPlayerEmptyCell(qp)) {
                    if (gameController->getPlayerShipCellsCount() < 20) {
                        gameController->setPlayerCellState(qp, 2);
                        if (gameController->checkPlayerShipPlacement()) {
                            gameController->infoLabel->setText("Начать игру");
                        }
                    }
                } else {
                    gameController->setPlayerCellState(qp, 0);
                    gameController->infoLabel->setText("Разверните флот!");
                }
                update();
            }
        }
        else if (gameController->getGameState() == GameState::PLAYER_TURN) {
            QPointF pos = event->position();
            if (pos.x() >= ENEMYFIELD_X &&
                pos.x() <= FIELD_WIDTH + ENEMYFIELD_X &&
                pos.y() >= ENEMYFIELD_Y &&
                pos.y() <= FIELD_HEIGHT + ENEMYFIELD_Y) {
                QPoint qp = getCoords(pos.x(), pos.y(), ENEMYFIELD_X, ENEMYFIELD_Y);
                if (qp.x() == 10)
                    qp.setX(9);
                if (qp.y() == 10)
                    qp.setY(9);
                gameController->playerShot(qp);
                repaint();
                QApplication::processEvents();
                // Проверка на завершение игры
                int gO_status = gameController->checkForGameOver();
                if (gameController->checkForGameOver() != 0) {
                    if (gO_status == 2) {
                        gameController->setGameState(GameState::GAMEOVER);
                        gameController->infoLabel->setText("Победа!");
                    } else {
                        gameController->setGameState(GameState::GAMEOVER);
                        gameController->infoLabel->setText("Поражение!");
                    }
                }
                // Ход бота
                while (gameController->getGameState() == GameState::ENEMY_TURN) {
                    gameController->botShot(); // Выстрел
                    repaint();
                    QApplication::processEvents();
                    // Проверка на завершение игры
                    int gO_status = gameController->checkForGameOver();
                    if (gameController->checkForGameOver() != 0) {
                        if (gO_status == 2) {
                            gameController->setGameState(GameState::GAMEOVER);
                            gameController->infoLabel->setText("Победа!");
                        } else {
                            gameController->setGameState(GameState::GAMEOVER);
                            gameController->infoLabel->setText("Поражение!");
                        }
                    }
                }
```

```
            }
        }
    }
}

// Визуализация игрового поля
void MainWindow::paintEvent(QPaintEvent *event) {
    Q_UNUSED(event);
    QPainter painter(this);
    QVector<Cell> currentCellsState = gameController->getPlayerAllCells();
    for (int i {0}; i < currentCellsState.size(); i++) {
        QPoint drawPoint;
        int x = i % 10;
        int y = i / 10;
        if (x < 5 && y < 5) {
            drawPoint.setX(MYFIELD_X + (x * CELL_SIZE));
            drawPoint.setY(MYFIELD_Y + (y * CELL_SIZE));
        } else {
            drawPoint.setX(MYFIELD_HALF_X + ((x - 5) * CELL_SIZE));
            drawPoint.setY(MYFIELD_HALF_Y + ((y - 5) * CELL_SIZE));
        }
        if (currentCellsState[i] == Cell::SHIP) {
            painter.drawPixmap(drawPoint, QPixmap("C:/LR/MyK.jpg"));
        } else if (currentCellsState[i] == Cell::DAMAGED) {
            painter.drawPixmap(drawPoint, QPixmap("C:/LR/RedK2.jpg"));
        } else if (currentCellsState[i] == Cell::DEAD) {
            painter.drawPixmap(drawPoint, QPixmap("C:/LR/RedK.jpg"));
        } else if (currentCellsState[i] == Cell::DOT) {
            painter.drawPixmap(drawPoint, QPixmap("C:/LR/T.png"));
        }
    }
    QVector<Cell> currentCellsStateBot = gameController->getBotAllCells();
    for (int i {0}; i < currentCellsStateBot.size(); i++) {
        QPoint drawPoint;
        int x = i % 10;
        int y = i / 10;
        if (x < 5 && y < 5) {
            drawPoint.setX(ENEMYFIELD_X + (x * CELL_SIZE));
            drawPoint.setY(ENEMYFIELD_Y + (y * CELL_SIZE));
        } else {
            drawPoint.setX(ENEMYFIELD_HALF_X + ((x - 5) * CELL_SIZE));
            drawPoint.setY(ENEMYFIELD_HALF_Y + ((y - 5) * CELL_SIZE));
        }
        if (currentCellsStateBot[i] == Cell::DOT) {
            painter.drawPixmap(drawPoint, QPixmap("C:/LR/T.png"));
        } else if (currentCellsStateBot[i] == Cell::DAMAGED) {
            painter.drawPixmap(drawPoint, QPixmap("C:/LR/MyK2"));
        } else if (currentCellsStateBot[i] == Cell::DEAD) {
            painter.drawPixmap(drawPoint, QPixmap("C:/LR/MyK"));
        }
    }
}

// Обработка нажатий клавиш
void MainWindow::keyPressEvent(QKeyEvent *event) {
    if (gameController->getGameState() == GameState::SHIPS_PLACING) {
        if (event->key() == Qt::Key_Space) {
            if (gameController->checkPlayerShipPlacement()) {
                gameController->syncPlayerShipsCells();
                gameController->botRandomShipsPlacing();
                gameController->setGameState(GameState::PLAYER_TURN);
                gameController->infoLabel->setText("Ваш ход!");
            }
        }
    }
}
```

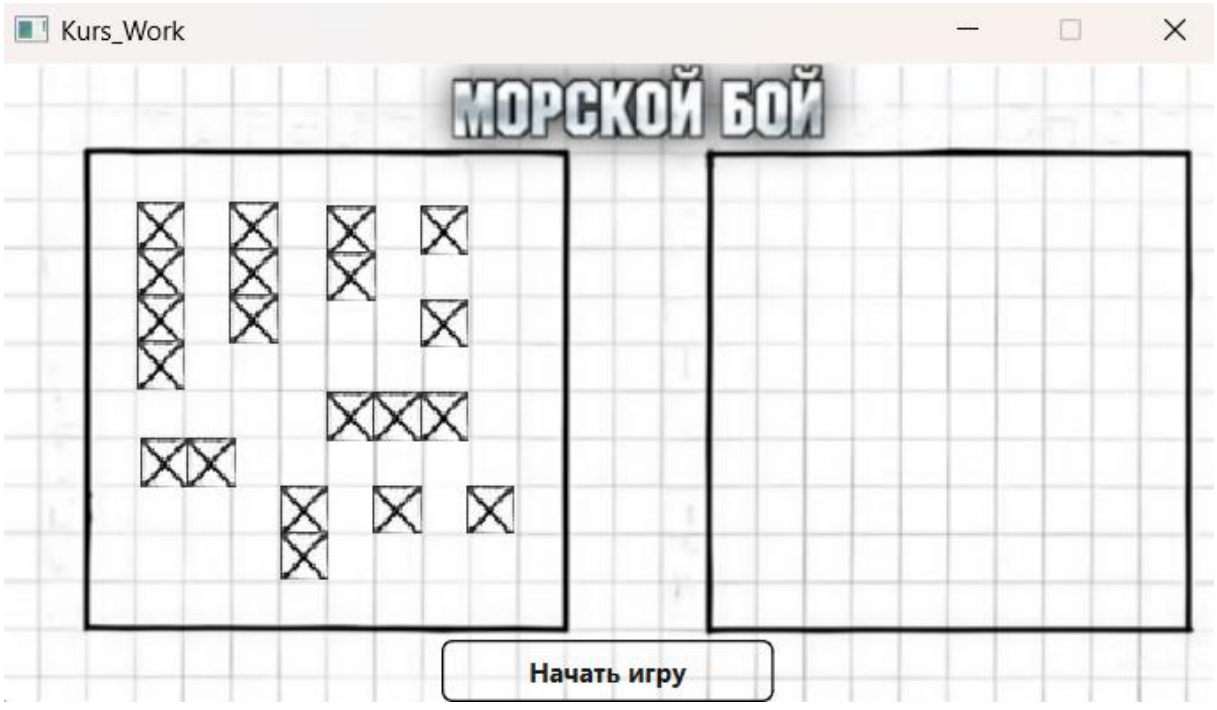
main.cpp:

```
#include "mainwindow.h"

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

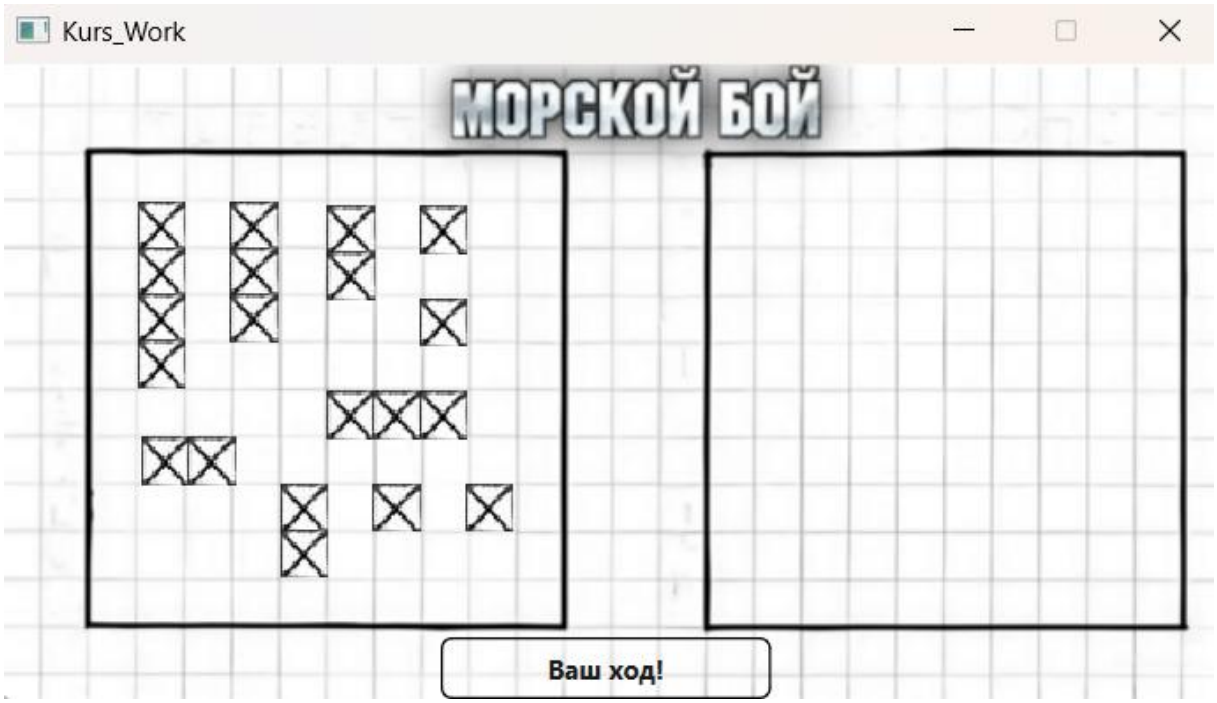
Результат работы программы

1) Расстановка кораблей.

2) Начало боя.



3) Ход игрока.



4) Ход противника.

5) Завершение игры.

Заключение

Выполняя курсовую работу, я усовершенствовал навыки объектно-ориентированного программирования на языке C++. Реализовал игру "Морской бой" с использованием фреймворка Qt, что позволило глубже понять основные принципы разработки. Применение паттернов проектирования оказало ключевое влияние в создании архитектуры программы. Сделал вывод: правильный выбор архитектурных решений может существенно повлиять на качество конечного продукта и упростить процесс разработки.

Список использованной литературы

- «Объектно-ориентированный анализ и проектирование с примерами приложений на C++», Г. Буч.
- «Паттерны Объектно-Ориентированного проектирования», Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес.
- «Игровой программист. Базовый уровень», Джон Плата.
- «Qt. Программирование на C++», Макс Шлее.