**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. 3388 |  | Еникеев А.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## Цель работы

Создать класс игры, который реализует игровой цикл, включая чередование ходов между игроком и компьютерным противником. Также требуется реализовать сохранение и загрузку состояния игры с возможностью восстановления после перезапуска программы.

## Задание

1. Создать класс игры, который реализует следующий игровой цикл:
2. Начало игры
3. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
4. В случае проигрыша пользователь начинает новую игру
5. 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.

1. 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

Были реализованы два основных класса:

- *Game* — основной класс, управляющий игровым процессом.

- *GameState* — класс, отвечающий за сохранение и восстановление состояния игры.

**Описание методов класса *Game***

- Краткое описание: класс хранит умные указатели на все поля, необходимые для управления игрой (игровые поля, менеджеры кораблей) и некоторые вспомогательные поля (менеджер способностей, номер хода, флаг хода пользователя), поля заполняются и изменяются специальными методами, также есть реализация раунда, где текущий игрок делает ход. Игру можно сохранить и восстановить. Кроме того, есть различные вспомогательные методы, например, для проверки доступности способности или сбрасывание игры в начало с новой расстановкой *(reset).*

- Конструктор *Game::Game();*

Инициализирует игровые доски для игрока и компьютера, менеджеры кораблей и способностей, а также устанавливает флаг хода игрока и номер раунда.

- *playRound()*

Выполняет один раунд игры:

- Если ход компьютера, атакует доску игрока.

- Если ход игрока, проверяет тип команды:

- Если используется способность, применяет её.

- Если производится атака, атакует выбранную клетку.

- Увеличивает номер раунда и переключает флаг хода.

- Возвращает результат применения способности.

- *placeShips()*

Размещает корабли: ему передается вектор структур с данными о расположении корабля *(x, y, id, orientation)*. Использует переданный флаг *isPlayerBoard* для выбора доски.

- Метод *isComputerLost()*

Проверяет, уничтожены ли все корабли компьютера. Возвращает *true*, если все корабли уничтожены, иначе *false*.

- *isPlayerLost()*

Проверяет, уничтожены ли все корабли игрока. Возвращает *true,* если все корабли уничтожены, иначе *false*.

- *getRoundNumber()*

Возвращает текущий номер раунда.

- *saveGame ()*

Создает объект GameState (сост. игры), записывает его в файл с помощью метода *saveToFile* (класса GameState).

- *loadGame()*

Загружает объект *GameState* из файла, обновляет текущие данные игры (доски, менеджеры и т.д.) на основе загруженного состояния.

- resetGame()

Принимает векторы расположения кораблей для пользователя и компьютера, обнуляет все поля, устанавливает корабли на поле. Метод не изменяет размеры полей.

- hasAbility()

Возвращает true если пользователю имеет доступна способность, иначе false.

- getAvailableAbility()

Возвращает тип способности, доступной к применения в данный момент, если таких нет, выбрасывает исключение.

- computerBoardRecovery()

Восстанавливает игровое поле компьютера и менеджер, принимает вектор позиций кораблей компьютера и расставляет их.

Также есть два метода геттера, которые возвращают ссылки на игровое поля компьютера и пользователя, это необходимо для дальнейшей отрисовки полей.

**Описание методов класса GameState**

- Краткое описание класса: класс состояния игры в качестве полей хранит все данные, необходимые для цикла игры, эти данные можно сохранить в файл и восстановить из файла. Причем восстанавливать сохранение можно после перезапуска программы, если файл сохранения был изменен вручную, то будет выброшено исключение и считать данные не удастся. Переопределены операторы ввода и вывода.

- Конструктор *GameState();*

Сохраняет состояние всех объектов игры, включая доски, менеджеры кораблей и способностей, номер раунда и текущий ход. Принимает эти объекты в качестве аргумента.

- Конструктор по умолчанию GameState();

Создает объект с минимальным состоянием (пустые доски, менеджеры без кораблей и способностей, начальный номер раунда и ход). Необходим инициализации перед заполнением полей из файла.

- *saveToStream()*

Записывает состояние игры в поток:

- Номер раунда и текущий ход.

- Состояния менеджеров кораблей.

- Состояния игровых досок.

- Очередь способностей игрока.

- Метод *loadFromFile()*

Читает состояние игры из потока. Обновляет все элементы состояния (доски, менеджеры и способности, доп. поля):

- Считывается номер раунда и флаг хода, восстанавливаются

- Считываются данные менеджеров кораблей, они

восстанавливается.

- Происходит восстановление игровых полей на основе

менеджеров: игровые поля должны ссылаться на те же корабли,

что и менеджеры.

- Происходит восстановление очереди способностей игрока.

**Методы сохранения и загрузки игровых объектов**

- Сохранение менеджера кораблей *saveShipManager();*

Записывает количество кораблей, их размеры, *ID* и состояние сегментов.

Сохраняет следующий порядок байт: {кол-во кораблей, для каждого корабля {длина, индекс в менеджере (*ID*), состояние всех сегментов}}

- Загрузка менеджера кораблей *loadShipManager()*;

Читает данные кораблей из файла, создает новый ShipManager и восстанавливает состояние каждого корабля.

Сохранение игровой доски *saveGameBoard();*

Записывает размеры доски и состояние каждой клетки, включая информацию о кораблях (*ID* и индекс сегмента).

Сохраняет следующий порядок байт: {ширина, высота, для каждой клетки поля {статус клетки, *ID* корабля и индекс сегмента корабля (если в клетке корабль)}

- Загрузка игровой доски *loadUserGameBoard()* и l*oadCompGameBoard();*

Читает размеры доски и восстанавливает состояние клеток. Если в клетке находится часть корабля, восстанавливает ссылку на соответствующий объект корабля.

- Сохранение и загрузка способностей *saveAbilityManager()* и *loadAbilityManager();*

Сохраняет и восстанавливает очередь способностей игрока.

- Переопределенные операторы ввода и вывода используют реализованные методы для управления потоком состояния игры.

UML-диаграмму классов см. в прил. A.

## Тестирование

1. Для тестирования класса игры и класса состояния будем использовать структуры команд, тем самым управляя игровым процессом, для наглядности выводились игровые поля, см. в табл. 1.

Табл. 1

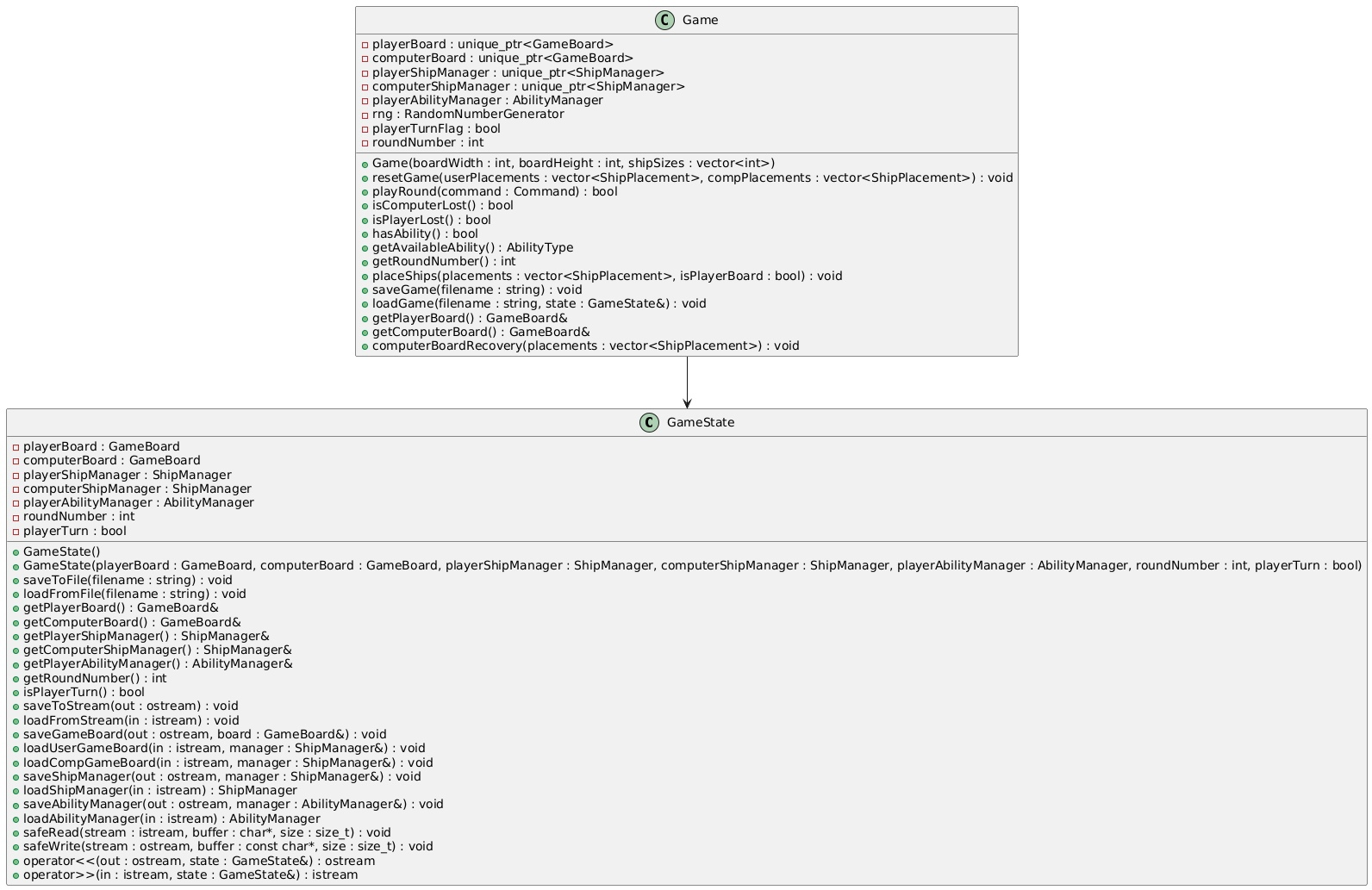
|  |  |
| --- | --- |
| Тестирующий код | Вывод |
| void testGameStateSaveLoad() {  int boardWidth = 10;  int boardHeight = 10;  std::vector<int> shipSizes = {3, 2, 1}; // Размеры кораблей  // Создаем игру  Game game(boardWidth, boardHeight, shipSizes);  // Веторы расположения кораблей  std::vector<ShipPlacement> placementsPlayer = {  {0, 0, 0, Orientation::Horizontal},  {1, 2, 3, Orientation::Vertical},  {2, 4, 1, Orientation::Horizontal}  };  std::vector<ShipPlacement> placementsComp = {  {0, 1, 1, Orientation::Horizontal},  {1, 6, 6, Orientation::Vertical},  {2, 9, 9, Orientation::Horizontal}  };  game.placeShips(placementsPlayer, true); // Размещение кораблей на доске игрока  game.placeShips(placementsComp, false); // Размещение кораблей на доске компа  GameBoard& compBoard = game.getComputerBoard();  std::cout << "Игровое поле компьюетра после расстановки" << std::endl;  compBoard.printBoard();  // Выполняем "комманды"  game.playRound(Command{CommandType::Attack, 2, 1}); // Игрок атакует 1  game.playRound(Command{CommandType::Attack, 0, 0}); // Компьютер атакует 2  std::cout << "Игровое поле компьюетра после 1 атаки пользователем (2, 1)" << std::endl;  compBoard.printBoard();  // Сохраняем игру  const std::string saveFile = "game\_save\_test.dat";  game.saveGame(saveFile);  std::cout << "Выполняем несколько атак по полю компьютера" << std::endl;  game.playRound(Command{CommandType::Attack, 9, 9}); // Игрок атакует 3  game.playRound(Command{CommandType::Attack, 0, 0}); // Компьютер атакует 4  game.playRound(Command{CommandType::Attack, 9, 9}); // Игрок атакует 5  game.playRound(Command{CommandType::Attack, 0, 0}); // Компьютер атакует 6  compBoard.printBoard();  // Загружаем игру  GameState state;  Game gameLoaded(state.getPlayerBoard().getWidth(), state.getPlayerBoard().getHeight(), state.getPlayerShipManager().getShipSizes());  gameLoaded.loadGame(saveFile, state);  GameBoard& compLoadedBoard = gameLoaded.getComputerBoard();  std::cout << "Игровое поле компьюетра после восстановления" << std::endl;  compLoadedBoard.printBoard();  // Атака для проверки  gameLoaded.playRound(Command{CommandType::Attack, 2, 3}); // Игрок атакует 3  gameLoaded.playRound(Command{CommandType::Attack, 0, 0}); // Компьютер атакует 4  std::cout << "Игровое поле компьюетра после 1 атаки пользователем (2, 3)" << std::endl;  compLoadedBoard.printBoard();  // Проверяем восстановление состояний  if (gameLoaded.getRoundNumber() == 5){  std::cout << "сохранение раунда: OK" << std::endl;  }  if (gameLoaded.isPlayerLost() == false){  std::cout << "Проверка окончания пользователем игры: OK" << std::endl;  }  if(gameLoaded.isComputerLost() == false){  std::cout << "Проверка окончания игры комп.: ОК" << std::endl;  }  } | Будет использован файл: game\_save\_test.dat  Игра загружена из файла: game\_save\_test.dat  Игровое поле компьюетра  0 1 2 3 4 5 6 7 8 9  0 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  1 ~ X D X ~ ~ ~ ~ ~ ~  2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  3 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  6 ~ ~ ~ ~ ~ ~ X ~ ~ ~  7 ~ ~ ~ ~ ~ ~ X ~ ~ ~  8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  9 ~ ~ ~ ~ ~ ~ ~ ~ ~ X  Игровое поле игрока  0 1 2 3 4 5 6 7 8 9  0 D X X ~ ~ ~ ~ ~ ~ ~  1 ~ ~ ~ ~ X ~ ~ ~ ~ ~  2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  3 ~ ~ X ~ ~ ~ ~ ~ ~ ~  4 ~ ~ X ~ ~ ~ ~ ~ ~ ~  5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  6 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  7 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  9 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  aomine@aomine-VirtualBox:~/OOP/Enikeev\_Anton\_lb1\_$ ./bin/game  Выберите тест для запуска:  1. Test GameState Save/Load  Тестируем сохранение и загрузку  2. Test End Game  Тестируем окончание игры  3. Test Reset Game  Тестирует восстановление игры  4. loadingGameNewProgram  Тестируеv загрузку игры (в директории игра должна быть уже сохранена)  Прежде воспользуйтесь тестом 1  Введите номер теста: 1  Игровое поле компьюетра после расстановки  0 1 2 3 4 5 6 7 8 9  0 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  1 ~ X X X ~ ~ ~ ~ ~ ~  2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  3 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  6 ~ ~ ~ ~ ~ ~ X ~ ~ ~  7 ~ ~ ~ ~ ~ ~ X ~ ~ ~  8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  9 ~ ~ ~ ~ ~ ~ ~ ~ ~ X  Игровое поле компьюетра после 1 атаки пользователем (2, 1)  0 1 2 3 4 5 6 7 8 9  0 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  1 ~ X D X ~ ~ ~ ~ ~ ~  2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  3 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  6 ~ ~ ~ ~ ~ ~ X ~ ~ ~  7 ~ ~ ~ ~ ~ ~ X ~ ~ ~  8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  9 ~ ~ ~ ~ ~ ~ ~ ~ ~ X  Игра сохранена в файл: game\_save\_test.dat  Выполняем несколько атак по полю компьютера  0 1 2 3 4 5 6 7 8 9  0 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  1 ~ X D X ~ ~ ~ ~ ~ ~  2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  3 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  6 ~ ~ ~ ~ ~ ~ X ~ ~ ~  7 ~ ~ ~ ~ ~ ~ X ~ ~ ~  8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  9 ~ ~ ~ ~ ~ ~ ~ ~ ~ !  Игра загружена из файла: game\_save\_test.dat  Игровое поле компьюетра после восстановления  0 1 2 3 4 5 6 7 8 9  0 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  1 ~ X D X ~ ~ ~ ~ ~ ~  2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  3 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  6 ~ ~ ~ ~ ~ ~ X ~ ~ ~  7 ~ ~ ~ ~ ~ ~ X ~ ~ ~  8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  9 ~ ~ ~ ~ ~ ~ ~ ~ ~ X  Игровое поле компьюетра после 1 атаки пользователем (2, 3)  0 1 2 3 4 5 6 7 8 9  0 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  1 ~ X D X ~ ~ ~ ~ ~ ~  2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  3 ~ ~ O ~ ~ ~ ~ ~ ~ ~  4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  6 ~ ~ ~ ~ ~ ~ X ~ ~ ~  7 ~ ~ ~ ~ ~ ~ X ~ ~ ~  8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~  9 ~ ~ ~ ~ ~ ~ ~ ~ ~ X  сохранение раунда: OK  Проверка окончания пользователем игры: OK  Проверка окончания игры комп.: ОК |

## Выводы

В ходе выполнения лабораторной работы был реализован класс игры, который управляет игровым процессом, включая чередование ходов игрока и компьютерного врага. Также реализован класс состояния игры. Основное внимание было уделено сохранению и загрузке состояния игры. В дальнейшем управление игрой будет происходить через вызовы методов класса игры, который зависит от класса состояния игры.

# Приложение А UML-диаграммА классов

*UML*-диаграмму классов смотри на рис. 1.

Рисунок 1 - UML-диаграмма классов