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

Студентка гр. 3388 _____ Сурова Е.Г.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы

Цель работы - разработка программы, реализующей простую игру с чередованием ходов игрока и компьютерного противника. Основные задачи лабораторной работы:

1. Создание классов и их взаимодействие: Разработка класса Game, управляющего игровым процессом, включая начало игры, выполнение ходов, определение победителя и запуск новых раундов. Реализация класса GameState, представляющего состояние игры (здоровье игрока и врага, использованные способности и т.д.), и переопределение операторов ввода/вывода для удобного отображения и сохранения состояния.

2. Реализация игрового цикла: Организация чередования ходов игрока и компьютера. Выполнение атак и применение способностей. Определение победителя раунда и начало нового раунда с сохранением состояния.

3. Сохранение и загрузка игры: Реализация механизма сохранения и загрузки состояния игры в любой момент, когда у игрока есть приоритет. Обеспечение возможности загрузки сохранения после перезапуска программы. Дополнительные задачи: Разработка удобного интерфейса для взаимодействия с игрой. Обеспечение читаемости и структурированности кода. Проведение тестирования и отладки программы. Результат работы: В результате выполнения лабораторной работы будет разработана программа, демонстрирующая базовые принципы объектно-ориентированного программирования, взаимодействие классов, а также работу с состоянием и сохранением данных.

Задание

0. Создать класс игры, который реализует следующий игровой цикл:

a. Начало игры

b. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.

c. В случае проигрыша пользователь начинает новую игру

d. В случае победы в раунде, начинается следующий раунд, причем состояние поля и способностей пользователя переносятся.

e. 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.

f. 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.

Примечания:

- 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

Был реализован класс игры, который отвечает за весь цикл игры. Имеет поле GameState, а так же методы:

void setPlayerShips() – ставит корабли игрока на игровое поле

void setComputerShips() – ставит корабли компьютера на игровое поле

void startRoundCycle() – начинает цикл раунда

void playerAttack() – вызывает атаку игрока по коорд

void computerAttack() – вызывает атаку компьютера по координатам

void clearPlayerMemory() – очищает память экземпляров класса, относящихся к игроку

void clearComputerMemory() – очищает память экземпляров класса, относящихся к компьютеру

void nextTurn() – совершение следующего хода в раунде

void initComputer() – инициализирует экземпляры класса, относящиеся к компьютеру

void initNewGame() – инициализирует экземпляры класса, необходимые для цикла игры

void continueGame() – после конца раунда, продолжает раунд, пересоздав классы компьютера

void startGameCycle(bool isSave) – запускает цикл игры

А так же конструктор Game(int width, int height,std::vector<int> sizes), который принимает размеры поля и размеры кораблей.

Так же в работе реализован класс состояния игры, содержит в себе:

GameBoard* playerGameBoard – игровое поле игрока

GameBoard* computerGameBoard – игровое поле компьютера

ShipManager* playerShipManager – менеджер кораблей игрока

ShipManager* computerShipManager – менеджер кораблей компьютера

AbilityManager* abilityManager – менеджер способностей

int boardWidth - ширина поля

int boardHeight – высота поля

std::vector<int> shipSizes – размеры кораблей

bool isDoubleDamage – флаг, показывающий будет ли нанесен игроком двойной урон

bool isPlayerWon – флаг, показывающий на то, победил ли игрок в раунде

int shipCount – количество кораблей с каждой из сторон

Так же реализованы методы:

std::vector<std::string> splitString(const std::string& str) – разделяет строку на построки

friend std::ostream& operator<<(std::ostream& out, const GameState& state) – переопределение оператора вывода в поток

friend std::istream& operator>>(std::istream& in, GameState& state) – переопределение оператора ввода из потока

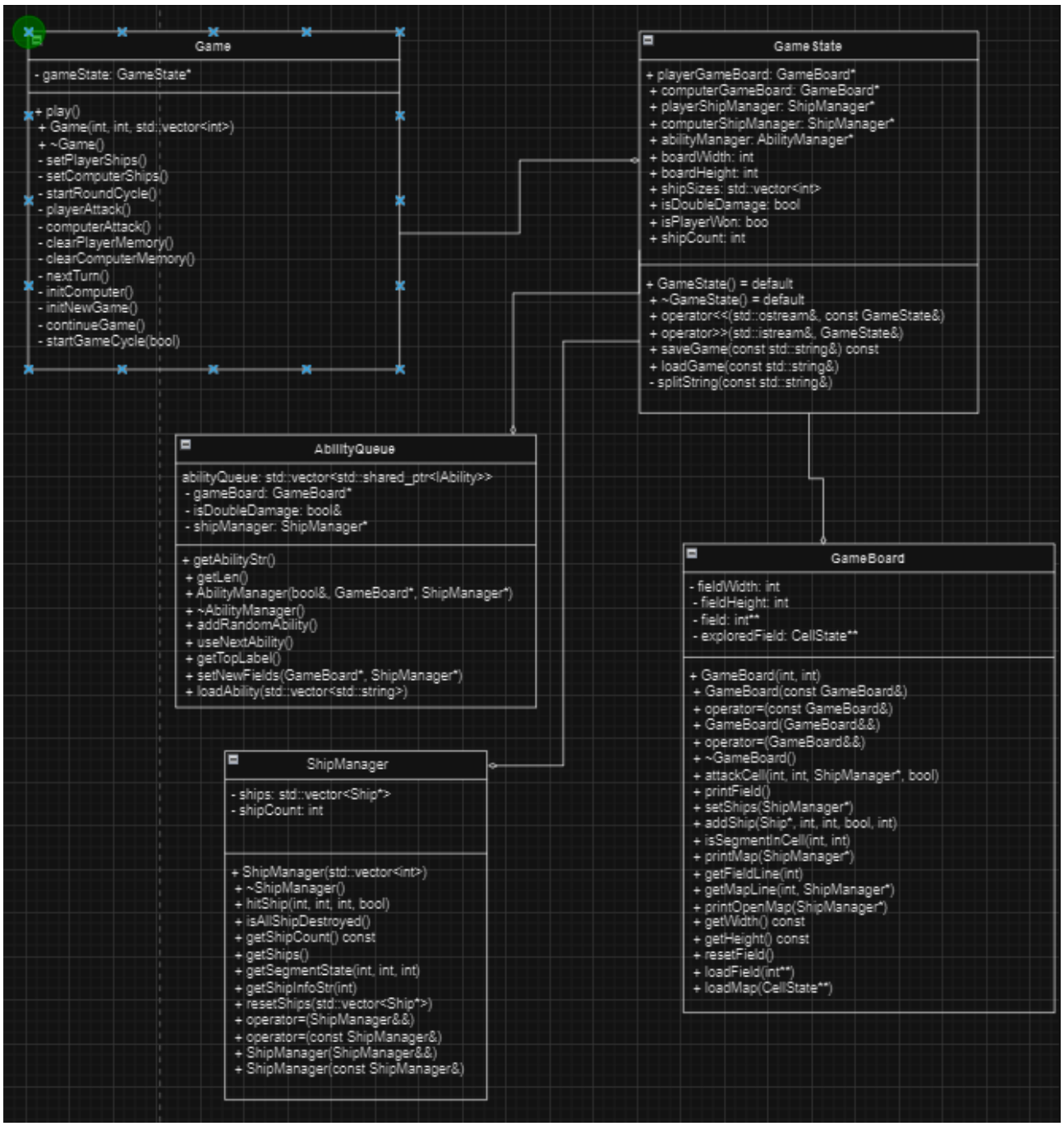
void saveGame(const std::string& fileName) const – сохраняет игру

void loadGame(const std::string& filename) – загружает игру

Разработанный программный код см. в приложении А.

Тестирование

Для проверки корректной работы программы, была проверена работоспособность класса игры и класса состояния игры. Было протестировано несколько раундов игры с загрузкой и сохранением.


Game
- gameState: GameState*
+ play()
+ Game(int, int, std::vector<int>)
+ ~Game()
- setPlayerShips()
- setComputerShips()
- startRoundCycle()
- playerAttack()
- computerAttack()
- clearPlayerMemory()
- clearComputerMemory()
- nextTurn()
- initComputer()
- initNewGame()
- continueGame()
- startGameCycle(bool)
Game State
+ playerGameBoard: GameBoard*
+ computerGameBoard: GameBoard*
+ playerShipManager: ShipManager*
+ computerShipManager: ShipManager*
+ abilityManager: AbilityManager*
+ boardWidth: int
+ boardHeight: int
+ shipSizes: std::vector<int>
+ isDoubleDamage: bool
+ isPlayerWon: boo
+ shipCount: int
+ GameState() = default
+ ~GameState() = default
+ operator<<(std::ostream&, const GameState&)
+ operator>>(std::istream&, GameState&)
+ saveGame(const std::string&) const
+ loadGame(const std::string&)
- splitString(const std::string&)
AbilityQueue
abilityQueue: std::vector<std::shared_ptr<IAbility>>
- gameBoard: GameBoard*
- isDoubleDamage: bool&
- shipManager: ShipManager*
+ getAbilityStr()
+ getLen()
+ AbilityManager(bool&, GameBoard*, ShipManager*)
+ ~AbilityManager()
+ addRandomAbility()
+ useNextAbility()
+ getTopLabel()
+ setNewFields(GameBoard*, ShipManager*)
+ loadAbility(std::vector<std::string>)
GameBoard
- fieldWidth: int
- fieldHeight: int
- field: int**
- exploredField: CellState**
+ GameBoard(int, int)
+ GameBoard(const GameBoard&)
+ operator=(const GameBoard&)
+ GameBoard(GameBoard&&)
+ operator=(GameBoard&&)
+ ~GameBoard()
+ attackCell(int, int, ShipManager*, bool)
+ printField()
+ setShips(ShipManager*)
+ addShip(Ship*, int, int, bool, int)
+ isSegmentInCell(int, int)
+ printMap(ShipManager*)
+ getFieldLine(int)
+ getMapLine(int, ShipManager*)
+ printOpenMap(ShipManager*)
+ getWidth() const
+ getHeight() const
+ resetField()
+ loadField(int**)
+ loadMap(CellState**)
ShipManager
- ships: std::vector<Ship*>
- shipCount: int
+ ShipManager(std::vector<int>)
+ ~ShipManager()
+ hitShip(int, int, int, bool)
+ isAllShipDestroyed()
+ getShipCount() const
+ getShips()
+ getSegmentState(int, int, int)
+ getShipInfoStr(int)
+ resetShips(std::vector<Ship*>)
+ operator=(ShipManager&&)
+ operator=(const ShipManager&)
+ ShipManager(ShipManager&&)
+ ShipManager(const ShipManager&)

Выводы

В ходе лабораторной работы №3 была разработана архитектура игры с пошаговым игровым процессом, чередующим ходы игрока и компьютера. Основными задачами стали создание и взаимодействие классов, реализация игрового цикла, а также механизмы сохранения и загрузки состояния игры.

Был разработан класс `Game`, управляющий игровым процессом. Он инициализирует игру, управляет ходами и раундами, выполняет действия игрока и компьютера, а также определяет победителя. В классе `Game` используется объект `GameState`, представляющий текущее состояние игры, включая игровые доски, менеджеры кораблей и способностей, а также параметры игры. `GameState` поддерживает операторы ввода/вывода для удобства отображения и сохранения.

Результатом работы стала программа, демонстрирующая основные принципы объектно-ориентированного программирования, взаимодействие классов и работу с сохранением данных. Разработанное приложение успешно реализует игровой процесс и может послужить основой для создания более сложных игр.