

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. 3344

Коняева М. В.

Преподаватель

Жангиров Т. Р.

Санкт-Петербург

2024

Цель работы.

Научиться организации взаимосвязи между классами. Получить опыт работы с вводом/выводом в файлы.

Задание.

Создать класс игры, который реализует следующий игровой цикл:

1. Начало игры
2. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
3. В случае проигрыша пользователь начинает новую игру
4. 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.

2. 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 самой игрой можно использовать обертки над командами
- При работе с файлом используйте идиому RAII.

Выполнение работы.

Класс IGameState

Класс IGameState представляет интерфейс для управления состоянием игры. Он абстрагирует работу с основными компонентами игровой логики, такими как управление полем, кораблями и сохранением данных. Его цель — упростить реализацию различных состояний игры (например, начального состояния, активного геймплея или завершения игры) и обеспечить единый контракт для работы с этими состояниями.

Методы класса *IGameState*:

- 1) void reset(): Сбрасывает состояние полей до начального.
- 2) ShipManager& getShipManager(): Возвращает ссылку на объект управления кораблями (ShipManager).
- 3) GameField& getGameField(): Возвращает ссылку на объект игрового поля (GameField).
- 4) void save (std::string name): Сохраняет текущее состояние игры в файл с указанным именем.
- 5) void load(std::string name): Загружает состояние игры из файла с указанным именем.

Класс GameStatePlayer

Класс - это конкретная реализация интерфейса IGameState. Он отвечает за управление состоянием игрока в игре, включая данные о поле, кораблях, способностях и другой сопутствующей информации.

Поля класса GameStatePlayer:

- 1) field (GameField): Игровое поле игрока, представляющее его состояние (размер, клетки и их содержимое).

- 2) shipManager (ShipManager): Объект для управления кораблями игрока. Содержит данные о позициях кораблей и их состоянии.
- 3) abilityManager (AbilityManager): Управляет активными способностями игрока.
- 4) abilityResponse (AbilityResponse): Содержит информацию о результате использования способностей.
- 5) coordHolder (CoordHolder): Утилитарный объект для работы с координатами на игровом поле.
- 6) numberOfWins (int): Количество побед игрока.
- 7) round (int): Текущий раунд игры.
- 8) needNewField (bool): Флаг, определяющий, требуется ли создать новое игровое поле.

Методы класса GameStatePlayer:

- 1) reset(): Сбрасывает состояние игры, возвращая его к начальному. Вызывает конструктор.
- 2) getAbilityManager(), getCoordHolder(): Возвращают ссылки на соответствующие компоненты (AbilityManager, CoordHolder), предоставляя доступ к их функционалу.
- 3) load (std::string name): Загружает состояние игрока из файла. Реализован через десериализацию JSON.
- 4) save(std::string name): Сохраняет текущее состояние игрока в файл. Реализован через сериализацию JSON.
- 5) toJson(): Конвертирует состояние игрока в объект nlohmann::json. Используется для сохранения состояния.
- 6) fromJson(nlohmann::json &j): Восстанавливает состояние игрока из JSON. Метод связывает корабли с клетками игрового поля.
- 7) getNumberOfWins(), setNumberOfWins(): Управляют количеством побед игрока.
- 8) getRound(), setRound(): Возвращают или увеличивают номер текущего раунда.

9) setNeedNewField(bool needNewField_), getNeedNewField():

Управляют флагом необходимости обновления игрового поля.

Поля класса GameStatePlayer:

- 9) field (GameField): Игровое поле игрока, представляющее его состояние (размер, клетки и их содержимое).
- 10) shipManager (ShipManager): Объект для управления кораблями игрока. Содержит данные о позициях кораблей и их состоянии.
- 11) abilityManager (AbilityManager): Управляет активными способностями игрока.
- 12) abilityResponse (AbilityResponse): Содержит информацию о результате использования способностей.
- 13) coordHolder (CoordHolder): Утилитарный объект для работы с координатами на игровом поле.
- 14) numberOfWins (int): Количество побед игрока.
- 15) round (int): Текущий раунд игры.
- 16) needNewField (bool): Флаг, определяющий, требуется ли создать новое игровое поле.

Класс GameStateEnemy

Класс GameStateEnemy - конкретная реализация интерфейса IGameState.

Он отвечает за управление состоянием врага в игре, включая данные о поле, кораблях. Основное отличие от состояния игрока в том, что он не содержит информацию о способностях или других элементах, специфичных для игрока.

Поля класса GameStateEnemy:

- 1) field (GameField): Игровое поле врага, представляющее его состояние (размер, клетки и их содержимое).
- 2) shipManager (ShipManager): Объект для управления кораблями врага. Содержит данные о позициях кораблей и их состоянии.

Методы класса GameStateEnemy:

- 1) `reset()`: Сбрасывает состояние игры, возвращая его к начальному. Вызывает конструктор.
- 2) `getShipManager()`: Возвращает ссылку на объект `ShipManager`, предоставляя доступ к данным о кораблях противника.
- 3) `getGameField()`: Возвращает ссылку на объект `GameField`, позволяя работать с состоянием поля противника.
- 4) `load (std::string name)`: Загружает состояние игрока из файла. Реализован через десериализацию JSON.
- 5) `save(std::string name)`: Сохраняет текущее состояние игрока в файл. Реализован через сериализацию JSON.
- 6) `toJson()`: Конвертирует состояние игрока в объект `nlohmann::json`. Используется для сохранения состояния.
- 7) `fromJson(nlohmann::json &j)`: Восстанавливает состояние игрока из JSON. Метод связывает корабли с клетками игрового поля.

Класс `GameManager` отвечает за управление игровым процессом. Этот класс действует как центральный контроллер, координирующий логику игры.

Поля класса `GameManager`:

- 1) `gameStatePlayer (GameStatePlayer&)`: Ссылка на объект состояния игрока, содержащий информацию о поле игрока, кораблях, способностях и текущем прогрессе.
- 2) `gameStateEnemy (GameStateEnemy&)`: Ссылка на объект состояния противника, включающий данные о поле противника и его кораблях.
- 3) `gameStatus (GameStatus)`: Текущее состояние игры (например, MENU, GAME, WIN). Необходимо для отрисовки в будущем

Методы класса `GameManager`:

- 1) `startNewGame()`: Сбрасывает состояния игрока и противника. Расставляет корабли для обеих сторон с использованием

ShipPlacer. Устанавливает статус игры в SHIP_PLACEMENT, а затем в GAME.

- 2) lose(): Если текущий статус игры LOSE, переводит игру в MENU.
- 3) goToNextLevel(): Сбрасывает состояние противника. Расставляет новые корабли для противника. Возвращает игру в статус GAME.
- 4) continueGame(): Позволяет продолжить игру, если игрок находится в MENU и не требуется новое поле.
- 5) backToMenu(): Переводит игру в статус MENU из состояния PAUSE.
- 6) attack(int x, int y): Обработывает атаку игрока по координатам (x, y). Если активирован эффект двойного урона, наносит двойной урон. При уничтожении корабля противника добавляет случайную способность игроку.
- 7) applyAbility(): Активирует текущую способность игрока через AbilityManager. Проверяет исход игры после применения способности.
- 8) move(int x, int y): Обработывает ход игрока и случайную атаку противника. Проверяет исход игры после каждого хода.
- 9) checkGameOutcome(): Проверяет, закончилась ли игра победой или поражением: если все корабли противника уничтожены, устанавливает статус WIN, если все корабли игрока уничтожены, устанавливает статус LOSE.
- 10) pause(): Переключает игру между состояниями PAUSE и GAME.
- 11) save(std::string name1, std::string name2): Сохраняет состояния игрока и противника в соответствующие файлы.
- 12) load(std::string name1, std::string name2): Загружает состояния игрока и противника из файлов.
- 13) setGameStatus(GameStatus gameStatus_): Устанавливает текущий статус игры.

Класс ShipPlacer

Класс ShipPlacer предназначен для размещения кораблей на игровом поле как для игрока, так и для врага. Размещение кораблей осуществляется с учетом правил игры, чтобы избежать пересечения кораблей или их выхода за пределы игрового поля.

Методы класса ShipPlacer:

- 1) `placeShipsForPlayer`: Эта функция отвечает за ручное размещение кораблей игроком. Для каждого корабля из менеджера (`shipManager`). Запрашивает у пользователя координаты и ориентацию (горизонтально/вертикально). Проверяет возможность размещения корабля на выбранных координатах с помощью метода `canPlaceShip` игрового поля. Если размещение возможно, вызывает метод `placeShip`, который обновляет состояние поля. Если размещение невозможно, предлагает пользователю повторить ввод.
- 2) `placeShipForEnemy`: Эта функция отвечает за автоматическое размещение кораблей для врага. Для каждого корабля из менеджера: генерирует случайные координаты и ориентацию (горизонтально/вертикально), проверяет возможность размещения корабля на выбранных координатах, если размещение возможно, вызывает метод `placeShip`, чтобы обновить состояние поля, если размещение невозможно, повторяет процесс до успешного размещения. После размещения всех кораблей выводит сообщение об успешной установке.

Тестирование

Пример с расположением у игрока и соперника по одному кораблю размера 4.

Каждый наносит урон по 1 единице, после чего игра сохраняется. Далее каждый снова наносит по 1 урону. Результат представлен на рис. 2. После этого происходит загрузка игры, результат на рис. 3.

```

int h = 10;
int w = 10;
std::vector<int> sizes = {4};
GameStatePlayer player = GameStatePlayer(w, h, sizes);
GameStateEnemy enemy = GameStateEnemy(w, h, sizes);
GameManager gameManager(player, enemy);
gameManager.setGameStatus(GameStatus::MENU);
gameManager.startNewGame();
gameManager.applyAbility();
gameManager.move(1, 1);
gameManager.getGameStatePlayer().getGameField().drawField(1);
gameManager.getGameStateEnemy().getGameField().drawField(1);
gameManager.save("player", "enemy");
gameManager.move(2, 2);
gameManager.getGameStatePlayer().getGameField().drawField(1);
gameManager.getGameStateEnemy().getGameField().drawField(1);
gameManager.load("player", "enemy");
gameManager.getGameStatePlayer().getGameField().drawField(1);
gameManager.getGameStateEnemy().getGameField().drawField(1);
return 0;

```

Рисунок 1 - Код тестирования

```

Save succes!
  0 1 2 3 4 5 6 7 8
A ~ ~ ~ ~ ~ ~ ~ ~
B ~ # ~ ~ ~ ~ ~ ~
C ~ # ~ ~ ~ ~ ~ ~
D ~ # ~ 0 ~ ~ ~ ~
E ~ + ~ ~ ~ ~ ~ ~
F ~ ~ ~ ~ ~ ~ ~ ~
G ~ ~ ~ ~ ~ ~ ~ ~
H ~ ~ ~ ~ ~ ~ ~ ~
I ~ ~ ~ ~ ~ ~ ~ ~
J ~ ~ 0 ~ ~ ~ ~ ~
  0 1 2 3 4 5 6 7 8
A ~ ~ ~ ~ ~ ~ ~ ~
B ~ 0 ~ ~ ~ ~ ~ ~
C ~ ~ 0 ~ ~ ~ ~ ~
D ~ ~ ~ ~ ~ ~ ~ ~
E ~ ~ ~ ~ ~ ~ ~ ~
F ~ ~ ~ ~ ~ ~ ~ ~
G ~ ~ ~ ~ ~ ~ ~ ~
H ~ ~ ~ ~ ~ ~ ~ ~
I ~ ~ ~ ~ ~ ~ ~ ~
J ~ # # # # ~ ~ ~

```

Резуль
таты
тестировани
я класса
представлен
в табл. 1.
Результат
проверки
работоспосо

Рисунок 2 - Поле игрока и
соперника после двух атак

```

Load succes!
  0 1 2 3 4 5 6 7 8
A ~ ~ ~ ~ ~ ~ ~ ~
B ~ # ~ ~ ~ ~ ~ ~
C ~ # ~ ~ ~ ~ ~ ~
D ~ # ~ ~ ~ ~ ~ ~
E ~ + ~ ~ ~ ~ ~ ~
F ~ ~ ~ ~ ~ ~ ~ ~
G ~ ~ ~ ~ ~ ~ ~ ~
H ~ ~ ~ ~ ~ ~ ~ ~
I ~ ~ ~ ~ ~ ~ ~ ~
J ~ ~ 0 ~ ~ ~ ~ ~
  0 1 2 3 4 5 6 7 8
A ~ ~ ~ ~ ~ ~ ~ ~
B ~ 0 ~ ~ ~ ~ ~ ~
C ~ ~ ~ ~ ~ ~ ~ ~
D ~ ~ ~ ~ ~ ~ ~ ~
E ~ ~ ~ ~ ~ ~ ~ ~
F ~ ~ ~ ~ ~ ~ ~ ~
G ~ ~ ~ ~ ~ ~ ~ ~
H ~ ~ ~ ~ ~ ~ ~ ~
I ~ ~ ~ ~ ~ ~ ~ ~
J ~ # # # # ~ ~ ~

```

Рисунок 3 - Поле игрока и
соперника после загрузки

программы представлен на рисунках 1 и 2.

Таблица 1 – Результаты тестирования классов

№ п/п	Входные данные	Выходные данные	Комментарии
----------	----------------	--------------------	-------------

1.	<pre> int h = 10; int w = 10; std::vector<int> sizes = {4}; GameStatePlayer player = GameStatePlayer(w, h, sizes); GameStateEnemy enemy = GameStateEnemy(w, h, sizes); GameManager gameManager(player, enemy); gameManager.setGameStatus(GameStatus::MENU); gameManager.startNewGame(); gameManager.applyAbility(); gameManager.move(1, 1); gameManager.getGameStatePlayer().getGameField() .drawField(1); gameManager.getGameStateEnemy().getGameField().drawField(1); gameManager.save("player", "enemy"); gameManager.move(2, 2); gameManager.load("player", "enemy"); gameManager.getGameStatePlayer().getGameField() .drawField(1); gameManager.getGameStateEnemy().getGameField().drawField(1); return 0; </pre>	-	Класс GameManager работает корректно
----	---	---	---

Выводы.

Получен опыт в связывании классов. Исследованы способы работы с файлами.

ПРИЛОЖЕНИЕ А

UML ДИАГРАММА КЛАССОВ

