**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра математического обеспечения и применения ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: Связывание классов**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3385 |  | Закиров И.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

**Цель работы.**

Изучение принципов связывания классов и взаимодействия объектов в программировании через реализацию игрового цикла с сохранением и загрузкой состояния игры. Закрепление навыков проектирования классов с учетом принципов инкапсуляции, слабой связанности, и использования идиомы RAII при работе с файлами.

**Основные теоретические положения.**

### Назначение класса Game

Класс **Game** является основным классом, управляющим игровым процессом. Его основные задачи:

* Координация всех игровых компонентов, таких как игроки (**Player**), поле (**Field**), корабли (**ShipManager**), и взаимодействие с вводом/выводом (**IOManager**).
* Обеспечение взаимодействия между пользователем и игровыми механиками через меню и игровые команды.
* Управление сохранением и загрузкой игрового состояния.
* Организация логики смены ходов и выполнения действий игроками и ботами.
* Реализация игрового цикла.

Этот класс представляет основную точку входа в игру и координирует работу всех других модулей.

### 2. Архитектурные решения

### 2.1. Игроки (players\_)

**Описание:**

* Вектор players\_ хранит двух игроков (человек и бот).
* Игроки инициализируются методами CreatePlayer и CreateBot.

**Причины выбора:**

* Использование вектора позволяет легко расширить количество игроков, если потребуется.
* Каждый игрок представлен объектом класса Player, который инкапсулирует состояние игрока, его поле и способности.

### 2.2. Менеджер ввода/вывода (io\_manager\_)

**Описание:**

* Экземпляр IOManager используется для взаимодействия с пользователем через текстовый интерфейс.
* Он отображает игровое поле, доступные команды, и обрабатывает пользовательский ввод.

**Причины выбора:**

* Централизация логики ввода/вывода в отдельном классе упрощает поддержание и модификацию пользовательского интерфейса.
* Позволяет отделить логику обработки ввода/вывода от игровой логики.

### 2.3. Состояние игры (game\_is\_start\_ и stage\_)

**Описание:**

* Переменная game\_is\_start\_ отслеживает, началась ли игра.
* Переменная stage\_ определяет текущий уровень или этап игры, который влияет на сложность (например, точность бота).

**Причины выбора:**

* Легкое управление состоянием игры через булевую переменную.
* Использование уровня stage\_ позволяет динамически менять сложность игрового процесса.

### 2.4. Меню и управление процессом

**Описание:**

* Метод MainMenu предоставляет начальное меню с возможностью создания новой игры, загрузки сохраненной игры или выхода.
* Метод Process организует основной игровой цикл, в котором обрабатываются команды пользователя и выполняются действия ботов.

**Причины выбора:**

* Структурированное управление игровым процессом через меню и игровой цикл упрощает восприятие и поддержку кода.
* Отделение меню (MainMenu) от игрового цикла (Process) делает код более читаемым.

### 2.5. Сохранение и загрузка игры

**Описание:**

* Методы Save и Load отвечают за сохранение и восстановление состояния игры.
* Для каждого игрока сохраняется его поле, способности и состояние кораблей.
* Логика сериализации и десериализации реализована в методах SavePlayer, LoadPlayer, SaveField, LoadField и других вспомогательных функциях.

**Причины выбора:**

* Подробное сохранение состояния каждого компонента игры (игроков, поля, кораблей) позволяет точно восстановить игру в любой момент.
* Логика сохранения/загрузки разделена на модули (игроки, поле, корабли), что упрощает добавление новых компонентов.

### 2.6. Боты

**Описание:**

* Бот реализован через игрока (Player), для которого is\_bot\_ == true.
* Метод BotMove определяет действия бота, используя случайные значения и параметры точности (bot\_accuracity\_).

**Причины выбора:**

* Логика бота интегрирована в общий механизм управления игроками, что снижает сложность реализации.
* Использование вероятностных подходов упрощает создание искусственного интеллекта.

### 2.7. Смена хода

**Описание:**

* Метод ChangeTurn переключает флаг хода у обоих игроков.
* Логика хода игрока/бота реализована через флаги player\_turn\_ и is\_it\_bot\_.

**Причины выбора:**

* Простое управление очередностью ходов через булевые флаги.
* Легкая интеграция с игровым циклом.

### 3. Основные методы класса

### Конструктор

1. **Game()**
   * Инициализирует состояние игры, создавая объект IOManager и вызывая MainMenu.
   * Удобен для запуска игрового процесса с начального меню.

### Методы управления процессом

1. **Process**
   * Организует основной игровой цикл, обрабатывая команды пользователя и действия ботов.
   * Включает взаимодействие с IOManager для отображения состояния игры и получения ввода.
2. **MainMenu**
   * Отображает начальное меню и позволяет выбрать действие: начать новую игру, загрузить сохранение или выйти.
3. **ExitGame**
   * Обеспечивает корректный выход из игры, предлагая сохранить прогресс.
4. **ChangeTurn**
   * Меняет текущего игрока, переключая флаг хода.
5. **BotMove**
   * Реализует ход бота с учетом его точности.

### Методы управления игроками

1. **CreatePlayer**
   * Добавляет игрока в список players\_.
2. **CreateBot**
   * Создает бота, генерируя его поле и размещая корабли.
3. **NewGame**
   * Инициализирует новую игру, создавая игроков и их поля.

### Методы сохранения и загрузки

1. **Save**
   * Сохраняет текущее состояние игры в файл.
2. **Load**
   * Загружает состояние игры из файла.
3. **SaveField и LoadField**
   * Сохраняют и загружают состояние поля игрока.
4. **SavePlayer и LoadPlayer**
   * Сохраняют и загружают состояние игрока.

### 4. Связь с другими классами

1. **Player**
   * Класс Game управляет игроками через вектор players\_.
2. **Field**
   * Используется для управления состоянием игрового поля игроков.
3. **IOManager**
   * Обеспечивает взаимодействие с пользователем.
4. **Saver**
   * Используется для сохранения и загрузки состояния игры.
5. **ShipManager**
   * Управляет кораблями игроков.

### 2. Назначение класса Player

Класс Player представляет игрока в игре (человека или бота) и служит для управления его игровыми данными и состоянием. Основные функции:

* Хранение состояния игрового поля (Field) игрока.
* Управление доступными способностями игрока (AbilityManager).
* Управление кораблями игрока через (ShipManager).
* Отслеживание текущего хода.
* Управление логикой бота, если игрок — бот.

Player инкапсулирует ключевые элементы, необходимые для работы одного игрока в игре, и служит основным интерфейсом между игровым процессом и конкретным участником (человеком или ботом).

### 2. Архитектурные решения

### 2.1. Тип игрока (is\_bot\_)

**Описание:**

* Переменная is\_bot\_ определяет, является ли игрок ботом или человеком.
* На основе этого флага определяется, как будет обрабатываться поведение игрока (например, случайные ходы для бота или ввод команды для человека).

**Причины выбора:**

* Четкое разделение логики для бота и человека позволяет легко добавлять или изменять функциональность каждого типа игрока.
* Флаг используется для активации специфичных для бота функций (например, bot\_accuracity\_).

### 2.2. Игровое поле (field\_)

**Описание:**

* Поле игрока хранится в объекте класса Field.
* Управление полем осуществляется через методы set\_field\_ и get\_field\_.

**Причины выбора:**

* Использование отдельного класса Field позволяет:
  + Уменьшить сложность кода в Player.
  + Сосредоточить логику, связанную с состоянием поля, внутри Field.

### 2.3. Менеджер способностей (ability\_manager\_)

**Описание:**

* AbilityManager управляет доступными способностями игрока, такими как "атаковать", "использовать способность", "восстановить здоровье" и т.д.
* Управление осуществляется через методы set\_ability\_manager\_ и get\_ability\_manager\_.

**Причины выбора:**

* Удобство и модульность: изменения в механике способностей можно вносить, редактируя только класс AbilityManager.
* Централизованное управление делает код масштабируемым и упрощает добавление новых способностей.

### 2.4. Менеджер кораблей (ship\_manager\_)

**Описание:**

* ShipManager управляет всеми кораблями игрока, их созданием, размещением и состоянием.
* Доступ к менеджеру осуществляется через методы set\_ship\_manager и get\_ship\_manager.

**Причины выбора:**

* Централизованное управление кораблями снижает сложность логики класса Player.
* Такой подход упрощает расширение, например, добавление новых типов кораблей.

### 2.5. Смена хода (player\_turn\_)

**Описание:**

* Переменная player\_turn\_ определяет, чей сейчас ход (данного игрока или нет).
* Метод ChangeTurn() переключает флаг player\_turn\_ между истиной и ложью.

**Причины выбора:**

* Простая логика управления ходами через булевый флаг.
* Легкая интеграция с игровым циклом (например, проверка if (player.get\_player\_turn\_())).

### 3. Основные методы класса

### Конструкторы

1. **Player()**
   * Создает игрока-человека с пустым менеджером способностей и без кораблей.
   * Полезен для начальной инициализации без дополнительных параметров.
2. **Player(bool is\_bot, size\_t stage)**
   * Создает игрока-бота или человека. Для ботов рассчитывается начальная точность.
   * Удобен для создания игроков с конкретными настройками.

### Методы управления

1. **set\_field\_ и get\_field\_**
   * Управляют состоянием игрового поля игрока.
2. **set\_ability\_manager\_ и get\_ability\_manager\_**
   * Позволяют устанавливать и получать менеджер способностей.
3. **set\_ship\_manager и get\_ship\_manager**
   * Позволяют устанавливать и получать менеджер кораблей.
4. **set\_turn\_ и get\_player\_turn\_**
   * Управляют состоянием хода игрока.
5. **ChangeTurn**
   * Переключает флаг хода на противоположный.
6. **get\_bot\_accuracity\_**
   * Возвращает текущую точность бота.

### 4. Связь с другими классами

1. **Field**
   * Управляет состоянием игрового поля, связан с игроком через переменную field\_.
2. **AbilityManager**
   * Предоставляет интерфейс для работы с возможностями игрока.
3. **ShipManager**
   * Управляет всеми кораблями игрока.
4. **Игровой процесс**
   * Класс интегрируется в игровой цикл и обеспечивает управление состоянием одного игрока.

### 3. Назначение класса Saver

Класс Saver предоставляет функциональность для сохранения и загрузки данных в/из файла. Его основные задачи:

* Упрощение процесса сериализации данных.
* Инкапсуляция логики работы с файловой системой.
* Обеспечение универсального интерфейса для записи и чтения данных различных типов.

Этот класс служит связующим звеном между игровыми компонентами и файловой системой, предоставляя простой способ сохранять и восстанавливать данные.

### 2. Архитектурные решения

### 2.1. Использование шаблонов для универсальности

**Описание:**

* Методы saveData и loadData реализованы как шаблоны, что позволяет работать с данными любого типа.
* Код шаблонных методов находится в заголовочном файле, что требуется для шаблонных функций.

**Причины выбора:**

* Универсальность позволяет использовать класс для работы с любыми типами данных, избегая дублирования кода.
* Это особенно полезно для игры, где нужно сохранять и загружать разнообразные типы данных (целые числа, строки, сложные структуры).

### 2.2. Простой интерфейс для работы с файлами

**Описание:**

* Конструктор класса принимает имя файла, который используется для сохранения и загрузки данных.
* Методы:
  + saveData для записи данных.
  + loadData для чтения данных.

**Причины выбора:**

* Простой интерфейс снижает сложность использования класса.
* Пользователи класса могут сосредоточиться на логике игры, не вникая в детали работы с файлами.

### 2.3. Автоматическое управление файлами

**Описание:**

* Каждый вызов saveData и loadData открывает и закрывает файл.
* Конструктор проверяет доступность файла, создавая его при необходимости.

**Причины выбора:**

* Избежание необходимости вручную открывать и закрывать файлы.
* Повышение надежности, так как файл автоматически закрывается после операции.

### 2.4. Обработка ошибок

**Описание:**

* Если файл не удается открыть, генерируется исключение std::ios\_base::failure.

**Причины выбора:**

* Обеспечение стабильности работы программы.
* Уведомление об ошибках, связанных с файловой системой, чтобы их можно было обработать на уровне игрового кода.

### 3. Основные методы класса

### Конструктор

1. **Saver(const std::string& filename)**
   * Инициализирует объект с указанным именем файла.
   * Проверяет доступность файла, создавая его, если он не существует.

### Шаблонные методы

1. **saveData(const T& data)**
   * Записывает данные в файл, добавляя новую строку.
   * Открывает файл в режиме std::ios::app, что позволяет дописывать данные без удаления существующего содержимого.
2. **loadData<T>()**
   * Считывает данные из файла.
   * Открывает файл в режиме std::ios::in, чтобы гарантировать чтение.

### 4. Связь с другими классами

1. **Игровой процесс (Game)**
   * Используется для сохранения и загрузки состояния игры, включая игроков, поля и корабли.
2. **Игрок (Player)**
   * Сохраняет и восстанавливает состояние игрока, его поле, способности и корабли.
3. **Поле (Field)**
   * Позволяет сохранять и загружать состояние ячеек поля.
4. **Способности (AbilityManager)**
   * Сохраняет и загружает очередь способностей игрока.

**Выводы.**

Проект демонстрирует грамотное использование ООП и проектирования. Архитектура поддерживает расширение, упрощает управление игровыми механиками и повышает надежность кода.