**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Программирование»**

Тема: **Отслеживание изменений**

|  |  |  |
| --- | --- | --- |
| Студент гр. 2381 |  | Рыжиков И.С. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2023

## Цель работы

Цель работы заключается в создании системы для эффективного управления игрой с явным разделением функциональности между отслеживанием событий в игре и их отображением. В рамках данной задачи предполагается разработка класса, взаимодействующего с игрой, чтобы отслеживать события, такие как перемещение игрока, происходящие события на поле и условия победы или поражения. Также планируется создание класса, отвечающего за отображение информации об игровом поле в терминале, с учетом различных символов для отображения игровых элементов, обеспечивая абстракцию от конкретных деталей отрисовки. Это разделение позволит легко заменять методы отображения, не затрагивая при этом логику отслеживания событий.

## Задание

а) Реализовать класс, который связывается с игрой, и отслеживает изменения в игре: перемещение игрока, победа или выигрыш, срабатывание событий. Данный класс должен реагировать на изменения и отрисовывать игровое поле, а также выводить информацию для игрока (например, предлагать начать новую игру).

б) При отрисовке поля должна считываться информация с поля и об игроке, и в зависимости от расположения происходит вывод представления поля в терминал. В представлении поля непроходимые клетки, игрок, события должны отображаться различными символами. Игрок, события, клетки и другие игровые сущности не должны знать ничего о том, каким символом они отрисовываются. За выбор символа отвечает класс выполняющий отрисовку

Примечания:

Класс отслеживания и класс отрисовки рекомендуется делать отдельными сущностями. Таким образом, класс отслеживания инициализирует отрисовку, и при необходимости можно заменить отрисовку (например, на GUI) без изменения самого отслеживания

При проверке типов события можно использовать dynamic\_cast / typeinfo, либо сделать связывание модель-представление

## Выполнение работы

### Класс Observable

**Observable**, предназначен для реализации паттерна "наблюдатель" (Observer). Вот краткое описание методов класса:

1. **void addObserver(IObserver \*observer)**
   * Добавляет наблюдателя (**observer**) в список наблюдателей данного объекта **Observable**.
2. **void notifyUpdate()**
   * Уведомляет все наблюдатели о том, что произошло обновление. Этот метод вызывает метод **update()** для каждого добавленного наблюдателя, оповещая их об изменениях.

### Класс IObserver

Интерфейс **IObserver** представляет собой наблюдателя, он определяет одну виртуальную функцию:

1. **virtual void update() = 0;**
   * Это чисто виртуальная функция, что делает интерфейс абстрактным. Все классы, реализующие этот интерфейс, должны предоставить свою собственную реализацию метода **update()**. Этот метод предназначен для обновления наблюдателя о произошедших изменениях в объекте, который он наблюдает.

### Класс FieldView

Класс **FieldView** представляет собой вид (визуализацию) игрового поля и реализует интерфейс **IObserver**. Вот краткое описание методов этого класса:

1. **explicit FieldView(Field &field, Player &player)**
   * Конструктор класса, который принимает объекты **Field** и **Player** в качестве параметров и регистрирует текущий объект в качестве наблюдателя за изменениями в поле и игроке.
2. **std::string renderCell(FieldCell &cell)**
   * Метод, который возвращает строковое представление ячейки игрового поля (**FieldCell**). В зависимости от типа события или наличия игрока в ячейке, возвращается соответствующий символ.
3. **std::string renderField()**
   * Метод, который генерирует строковое представление всего игрового поля, включая расположение игрока и событий. Используется для визуализации текущего состояния игрового поля.
4. **void update() override**
   * Реализация виртуальной функции **update** из интерфейса **IObserver**. Вызывается при изменениях в объектах **Field** и **Player**, обновляет строковое представление поля и уведомляет других наблюдателей.
5. **const std::string &getFieldString()**
   * Метод, возвращающий текущее строковое представление игрового поля.

Класс также содержит приватные члены, представляющие объекты **Field** и **Player**, а также строку **field\_str\_view** для хранения текущего состояния игрового поля в виде строки.

***Класс* PlayerView**

Класс **PlayerView** представляет собой визуализацию игрового персонажа (игрока) и также реализует интерфейс **IObserver**. Вот краткое описание методов этого класса:

1. **explicit PlayerView(Player &player)**
   * Конструктор класса, который принимает объект **Player** в качестве параметра и регистрирует текущий объект в качестве наблюдателя за изменениями в игроке.
2. **~PlayerView()**
   * Деструктор класса.
3. **const std::string &getPlayerString()**
   * Метод, возвращающий строковое представление игрового персонажа.
4. **void update() override**
   * Реализация виртуальной функции **update** из интерфейса **IObserver**. Вызывается при изменениях в объекте **Player**, обновляет строковое представление игрока и уведомляет других наблюдателей.
5. **std::string renderPlayer()**
   * Метод, который генерирует строковое представление игрового персонажа, включая его имя, координаты и текущее количество здоровья.

Класс также содержит приватные члены, представляющие объект **Player** и строку **player\_str\_view** для хранения текущего состояния игрового персонажа в виде строки.

### Класс FieldView

Класс **LevelView** представляет собой уровень игры, объединяя в себе виды игрового поля (**FieldView**) и игрока (**PlayerView**). Вот краткое описание методов этого класса:

1. **LevelView(FieldView &fieldView\_, PlayerView &playerView\_)**
   * Конструктор класса, который принимает объекты **FieldView** и **PlayerView** в качестве параметров и регистрирует текущий объект в качестве наблюдателя за изменениями в этих объектах.
2. **void showLevel()**
   * Метод, выводящий на экран текущее состояние игрового уровня, включая информацию об игроке и вид поля.
3. **void update() override**
   * Реализация виртуальной функции **update** из интерфейса **IObserver**. Вызывается при изменениях в объектах **FieldView** и **PlayerView**, обновляет экран с текущим состоянием уровня.
4. **void renderLevel()**
   * Метод, обновляющий состояние объектов **FieldView** и **PlayerView** с последующим вызовом **showLevel()** для отображения изменений на экране.

Класс содержит приватные члены **fieldView** и **playerView**, представляющие объекты **FieldView** и **PlayerView** соответственно.

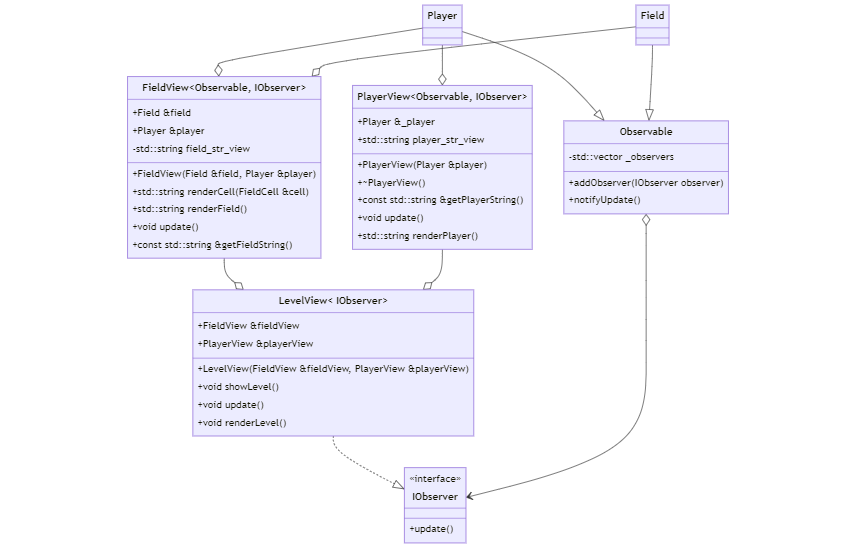
## Выводы

Добавлена визуализация в программу с использованием паттерна "наблюдатель". Классы `FieldView`, `PlayerView` и `LevelView` позволяют эффективно следить за изменениями в объектах `Field` и `Player`, а также предоставляет пользователю наглядное отображение текущего состояния игрового уровня.

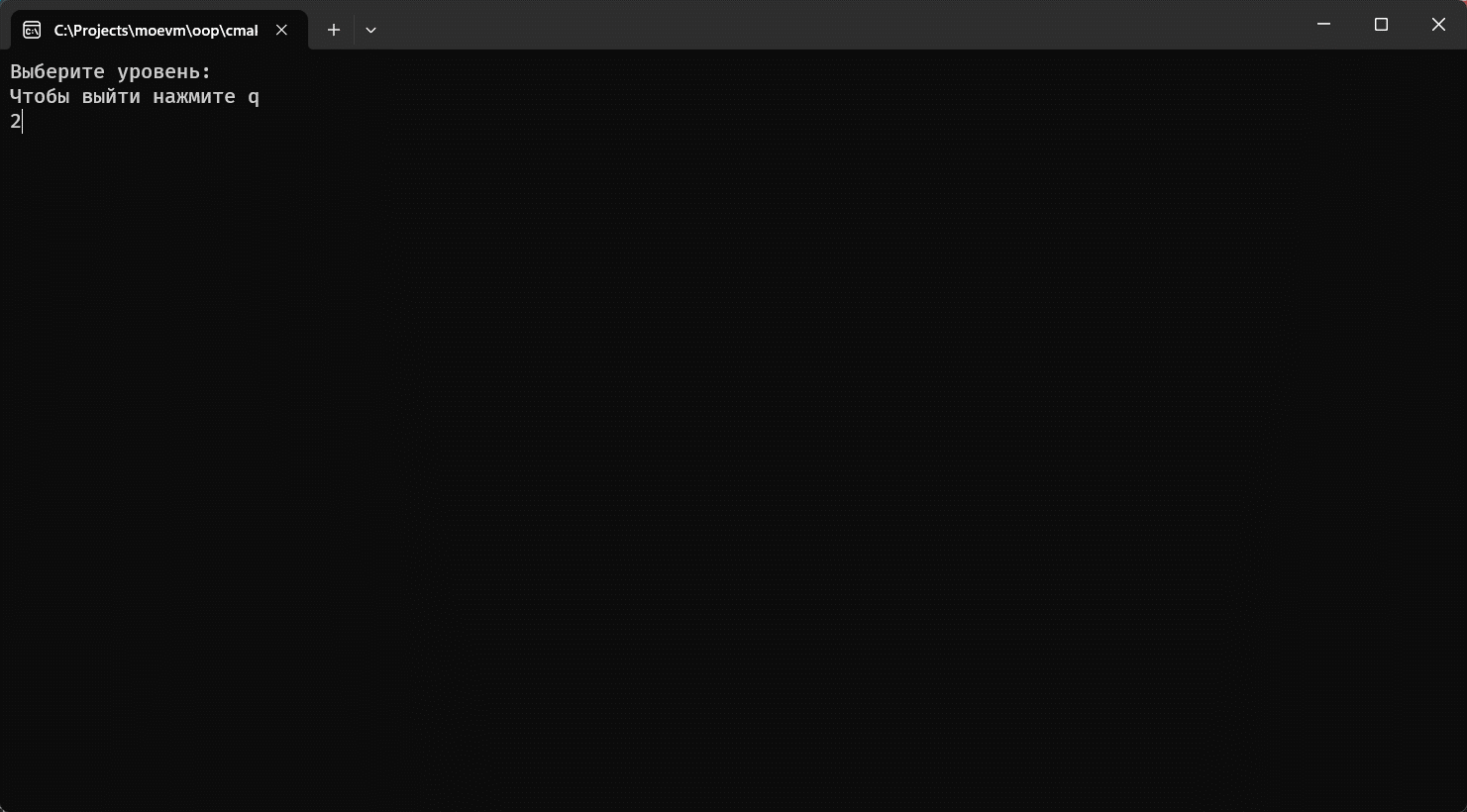
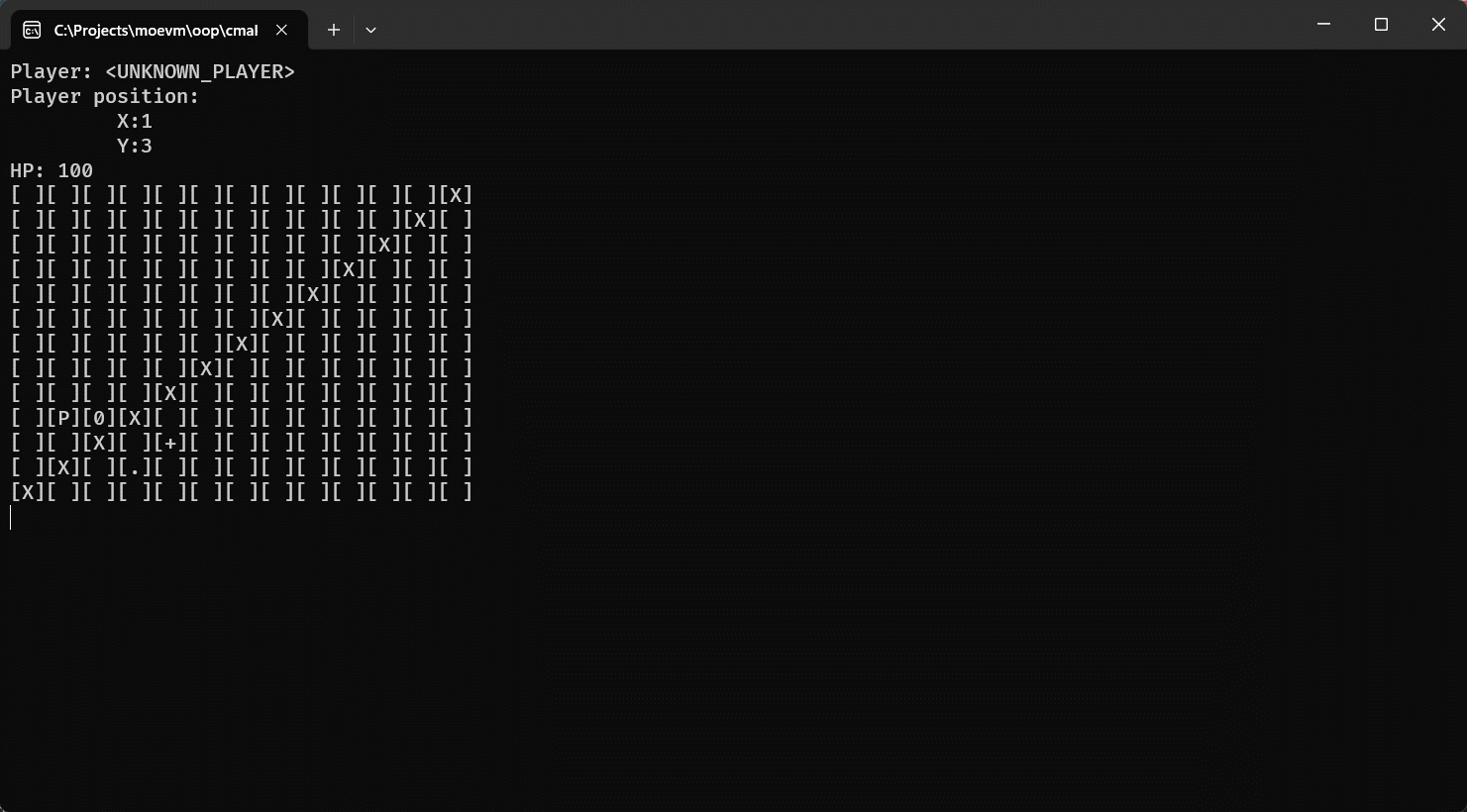
Реализация паттерна "наблюдатель" в виде классов `Observable` и `IObserver` позволяет достичь гибкости и разделения ответственности между компонентами. Объекты `Field` и `Player` выступают в роли наблюдаемых сущностей, а `FieldView` и `PlayerView` - в роли наблюдателей, отслеживая изменения и реагируя на них визуально. Это позволяет легко внедрять новые элементы в игру, сохраняя её легкость расширения и поддержки.

Пример работы программы (рис. 1) и диаграмма классов (рис. 2) представлена в Приложении.

# Приложение



1. Диаграмма классов

1. Пример работы программы