**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №4**

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

Тема: **Шаблонные классы**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3381 |  | Щеглов М.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## Цель работы.

Изучить и реализовать шаблонные классы для управления игрой и её отображения, обеспечив взаимодействие пользователя с игрой через терминал, настройку команд через файл и динамическую отрисовку игрового поля.

## Задание.

## Лабораторная работа №4 - Шаблонные классы

1. Создать шаблонный класс управления игрой. Данный класс должен содержать ссылку на игру. В качестве параметра шаблона должен указываться класс, который определяет способ ввода команда, и переводящий введенную информацию в команду. Класс управления игрой, должен получать команду для выполнения, и вызывать соответствующий метод класса игры.
2. Создать шаблонный класс отображения игры. Данный класс реагирует на изменения в игре, и производит отрисовку игры. То, как происходит отрисовка игры определяется классом переданном в качестве параметра шаблона.
3. Реализовать класс считывающий ввод пользователя из терминала и преобразующий ввод в команду. Соответствие команды введенному символу должно задаваться из файла. Если невозможно считать из файла, то управление задается по умолчанию.
4. Реализовать класс, отвечающий за отрисовку поля.

**Примечание:**

* 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
* После считывания клавиши, считанный символ должен сразу обрабатываться, и далее работа должна проводить с сущностью, которая представляет команду.
* Для представления команды можно разработать системы классов или использовать перечисление enum.
* Хорошей практикой является создание “прослойки” между считыванием/обработкой команды и классом игры, которая сопоставляет команду и вызываемым методом игры. Существуют альтернативные решения без явной “прослойки”
* При считывания управления необходимо делать проверку, что на все команды назначена клавиша, что на одну клавишу не назначено две команды, что на одну команду не назначено две клавиши.

**Выполнение работы**

Шаблонный класс **GameController** предназначен для управления игровым процессом, класс является принимает два шаблонных параметра:

* **InputMethod** — класс, который обрабатывает ввод команд.
* **Renderer** — класс, который отвечает за отображение игры.

**Основные элементы класса:**

Конструктор GameController принимает:

* Ссылку на объект игры (GameLoop), который управляет логикой игры.
* Объект InputMethod для обработки ввода.
* Объект Renderer для отображения игры.

Внутри конструктора вызываются два метода:

1. loadCommandsFromJson() — загрузка конфигурации команд из файла JSON.
2. initializeCommands() — связывание команд с соответствующими действиями.

**Методы:**

1. **loadCommandsFromJson** - этот метод пытается загрузить конфигурацию команд из JSON-файла.
2. **setDefaultCommands** - устанавливает базовые команды (например, "атака", "сохранение игры", "выход") и их соответствия символам, если пользовательский JSON недоступен.
3. **initializeCommands** - инициализирует функционал команд, связывая команды с действиями.
4. **handleInput** - обрабатывает ввод игрока, используя InputMethod. Читает первый символ, проверяет его в сопоставлении (карта commandMap) и возвращает соответствующую команду. Если команда неизвестна, возвращается пустая строка.
5. **executeCommand** - выполняет указанную команду, если она существует в списке зарегистрированных команд.
6. **Run** - основной игровой цикл:
   * Отображает приветственное сообщение.
   * Постоянно обрабатывает ввод пользователя с помощью handleInput.
   * Выполняет команды через executeCommand.

Шаблонный класс **GameRenderer**, который отвечает за отображение состояния игры и её элементов. Класс универсален, поскольку позволяет использовать разные реализации визуализации для игрового поля и текстовых сообщений.

**Структура класса:**

**GameRenderer** — это шаблонный класс, принимающий два параметра:

1. **FieldRenderer** — класс, отвечающий за визуализацию игрового поля (например, текстовое или графическое отображение).
2. **MessageRenderer** — класс, обрабатывающий отображение текстовых сообщений (например, подсказок или уведомлений).

**Методы класса:**

1. Конструктор принимает три аргумента:

* Ссылку на объект игры (GameLoop), который предоставляет данные для отрисовки.
* Объект FieldRenderer для визуализации игрового поля.
* Объект MessageRenderer для отображения текстовых сообщений.

1. **renderField()** - этот метод отвечает за отрисовку игрового поля.
2. **displayMessage(const std::string& message)** - отвечает за отображение текстовых сообщений.
3. **displayHelp()** - показывает справочную информацию или подсказки для игрока:
4. **update()** - этот метод обновляет визуализацию игры:

**Связь с другими классами:**

* **FieldRenderer** предоставляет реализацию визуализации игрового поля, которая может быть заменена без изменения логики GameRenderer.
* **MessageRenderer** аналогично обрабатывает сообщения и может быть подменён для изменения способа отображения (например, переход от текстового вывода к GUI).

Класс **FieldRenderer** отвечает за визуализацию игровых полей. Метод отображает состояние как поля игрока, так и поля врага, используя разные символы для обозначения различных статусов клеток и сегментов кораблей.

**Основная логика метода drawField**

1. **Входные параметры**:

* playerField — объект игрового поля игрока.
* enemyField — объект игрового поля врага.

1. **Отрисовка игрового поля**: Метод разделён на две части: отрисовка поля игрока и отрисовка поля врага.

**Отрисовка поля игрока:**

1. **Проверка статуса клетки**: метод getCellStatus(x, y) возвращает состояние клетки. Возможные состояния:
   * + CellStatus::Empty (вода) — отображается символом x.
     + CellStatus::Ship (клетка с сегментом корабля) — требуется дополнительная обработка.
     + CellStatus::Unknown (область была атакована, но не попали) — отображается символом ~.
2. **Обработка сегмента корабля**: Если клетка содержит сегмент корабля, вызывается метод getSegmentStatus(x, y), который возвращает состояние сегмента.

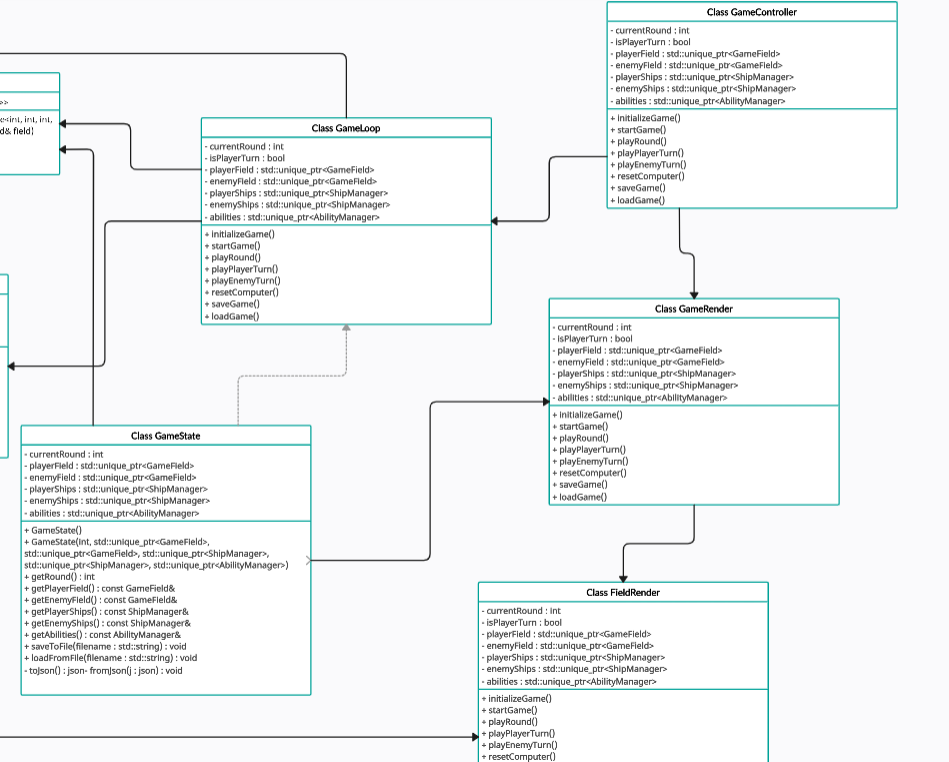
**Отрисовка поля врага:**

1. **Проверка статуса клетки**:
   * Неизвестная клетка — отображается символом **#.**
   * Вода — отображается символом **~**.
   * Клетка с сегментом корабля — требуется дополнительная обработка.
2. **Обработка сегмента корабля**: если клетка содержит сегмент корабля:
   * целый сегмент — отображается символом # (скрытый от игрока).
   * повреждённый сегмент — отображается символом 1.
   * уничтоженный сегмент — отображается символом ~.

**Символы, используемые для визуализации:**

* x — вода (пустая клетка на поле игрока).
* ~ — промах, волна, или уничтоженный сегмент.
* 2 — целый сегмент корабля игрока.
* 1 — повреждённый сегмент корабля.
* # — неизвестная клетка на поле врага или скрытый целый сегмент.

Ниже представлена UML-диаграмма реализованных в данной работе классов:



Разработанный программный код см. в приложении А.

Тестирование программы см. в приложении Б.

## Выводы

В ходе работы были реализованы шаблонные классы управления и отображения игры. Класс управления обрабатывает команды пользователя и связывает их с методами игры, а класс отображения визуализирует поля и сообщения. Архитектура получилась гибкой и модульной, с возможностью легко добавлять новые способы ввода и визуализации. Работа завершена с соблюдением всех требований задания.