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: Уровни абстракции, управление игроком

Студент(ка) гр. 1381 _____ Денисова О.К.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2022

Цель работы

Реализовать набор классов отвечающих за считывание команд пользователя, обрабатывающих их и изменяющих состояния программы.

Общая формулировка задачи

- Реализован класс/набор классов обрабатывающие команды
- Управление задается из файла (определяет какая команда/нажатие клавиши отвечает за управление. Например, w - вверх, s - вниз, и.т.д)
- Реализованные классы позволяют добавить новый способ ввода команд без изменения существующего кода (например, получать команды из файла или по сети). По умолчанию, управление из терминала или через GUI, другие способы реализовывать не надо, но должна быть такая возможность.
- Из метода, считывающего команду, не должно быть “прямого” управления игроком

Выполнение работы

В ходе выполнения лабораторной работы был реализован паттерн Команда: в рамках паттерна создан интерфейс команды ICommand, и несколько классов, реализующих данный интерфейс. В качестве invoker’а создан класс Multipult, который отвечает за вызов команды для определенного запроса. В качестве reciever’а выступает класс Game, созданный в предыдущих лаб. работах. Также реализован класс Manager, отвечающий за установку управления в Multipult’е.

Класс Multipult отвечает за вызов команд, у него есть публичный метод pressOn, получающий в качестве аргумента значение кнопки, и выполняющий действие, соответствующее данной кнопке. Для вызова команд использовались два контейнера std::map, а также создан enum cmds, содержащий имена всех команд. Один из контейнеров

std::map<cmds, ICommand*> содержит указатели на интерфейс комманд, второй контейнер std::map<char, cmds> заполняется с помощью Manager’а запрошенными символами. Вызов команды происходит насквозь через оба контейнера: дважды обращаемся по ключу благодаря тому, что в первом контейнере cmds являются ключами, а во втором значениями. Данное решение обосновано тем, что мы можем быть уверены в том, что Multipult (далее – пульт) уже знает как выполнять все необходимые команды, поэтому мы можем не беспокоиться о том, что какой-то команды не хватает, остается только задать управление на конкретную клавишу.

Управление задает класс Manager, читающий символы из файла, и устанавливающий их в контейнер в пульте. Manager является дружественным классом для пульта, и использует приватный метод setKey пульта. Это сделано для того, чтобы управление мог задавать только Manager и никто другой. В методе manage() он считывает 6 символов, проверяет их на корректность и уникальность (уникальность проверяется с помощью добавления символов в std::set и проверки его длины). Если считанные символы корректны и уникальны, устанавливает их в пульт для управления, иначе устанавливает символы, заданные по умолчанию. Таким образом реализуется обеспечение корректного управления даже в случае некорректности файла.

Интерфейс ICommand содержит чисто виртуальный метод execute(), который переопределяется в конкретных реализациях: moveUp, moveDown, moveRight, moveLeft, gameOn, gameOff, а также чисто виртуальный деструктор. Классы команд агрегируют объект Game, и через него реализуют выполнение команды каждый по-своему.

Для считывания управления создан интерфейс CommandReader, который содержит чисто виртуальный метод read, переопределенный в реализации интерфейса – ConsoleCommandReader’е, и чисто виртуальный деструктор. Интерфейс создан для того, чтобы была возможность

добавления новых способов управления. ConsoleCommandReader агрегирует Multipult, в переопределенном методе read он считывает команду и подает запрос на выполнение в пульт. Пульт проверяет наличие такого ключа, если он существует – выполняет команду по ключу, иначе игнорирует. Игнорирование реализовано, чтобы при случайном вводе неправильной команды не падала вся игра, и можно было бы спокойно продолжать.

Пример работы:

1) Ввели в файл корректное управление – можем управлять игроком

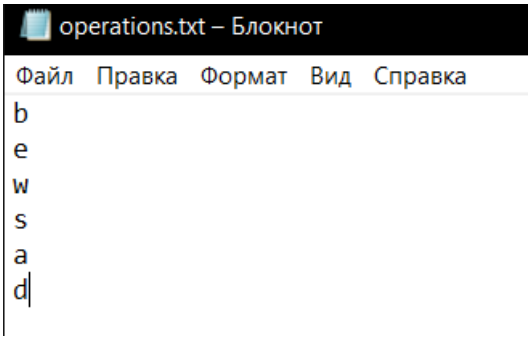

Рисунок 1. Содержание файла operations.txt, пример 1

Сделали шаг наверх, игрок переместился наверх

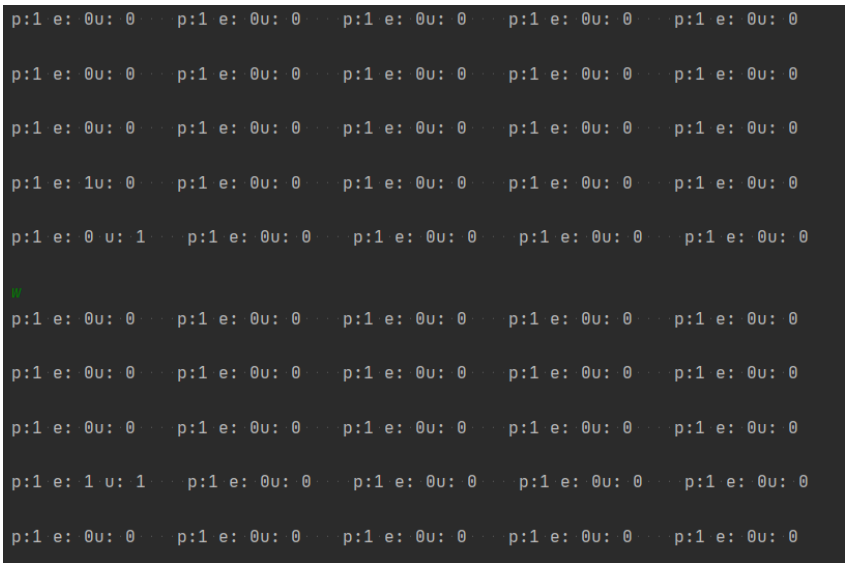

Рисунок 2. Вывод в консоль, пример 1

2) Ввели в файл некорректное управление – можем управлять игроком, т.к. срабатывает управление по умолчанию

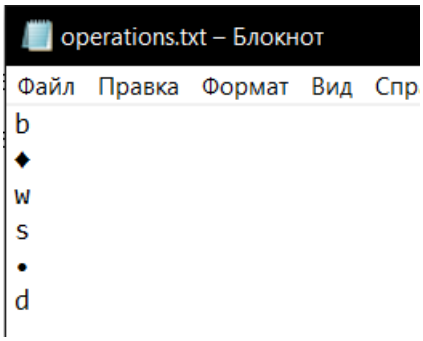

Рисунок 3. Содержание файла operations.txt, пример 2

Сделали шаг направо – игрок переместился вправо. Заметим, что первая команда была введена некорректно, поэтому сработало игнорирование и ничего не произошло.

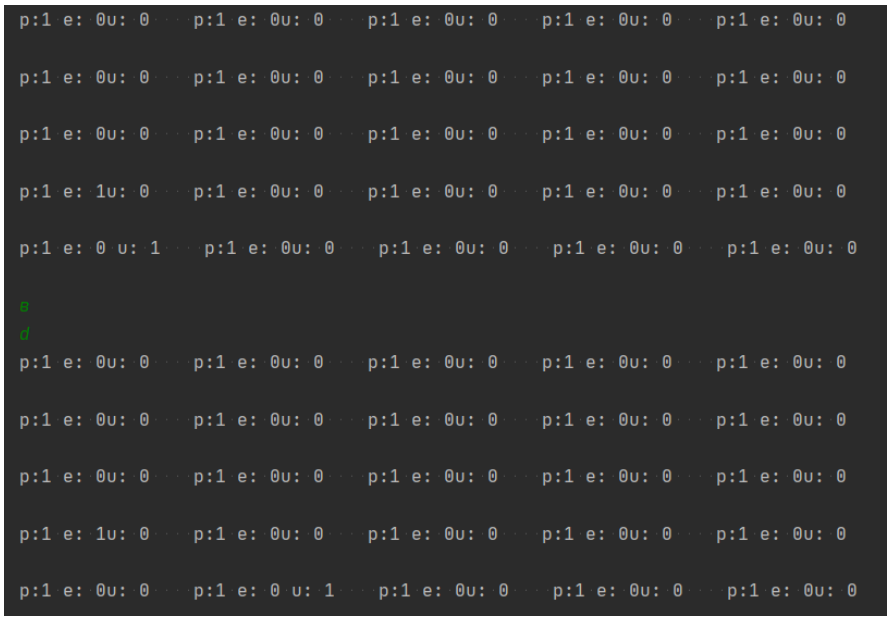

Рисунок 4. Вывод в консоль, пример 2

UML-диаграмма

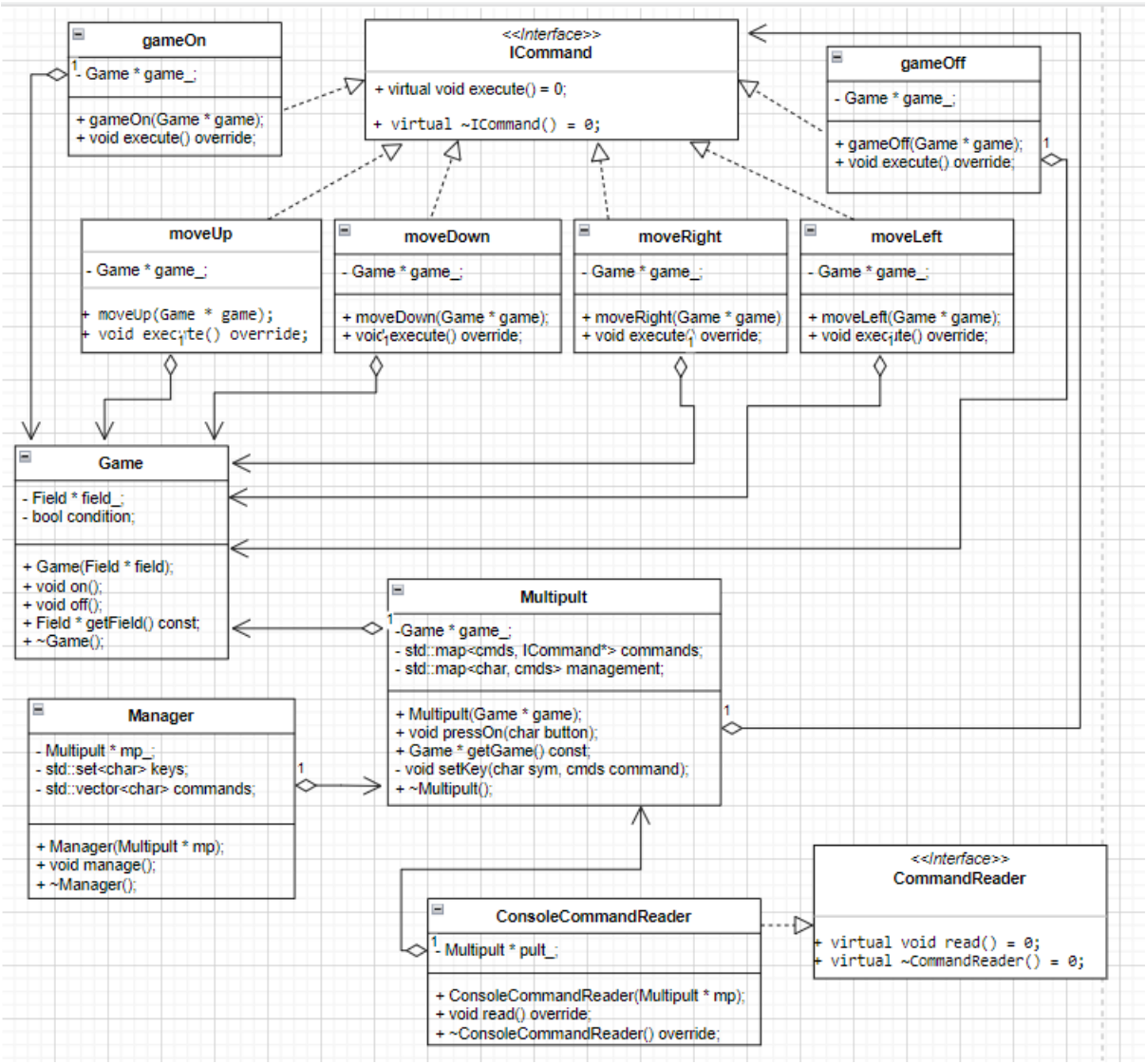

Рисунок 5. UML-диаграмма для лабораторной работы №4

Выводы

В ходе выполнения лабораторной работы было реализовано управление игроком с помощью создания набора классов, обрабатывающих запросы пользователя.