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.

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2022

Цель работы.

Реализовать набор классов отвечающих за считывание команд пользователя, обрабатывающих их и изменяющих состояния программы (начать новую игру, завершить игру, сохраниться, управление игроком, и.т.д.). Команды/клавиши, определяющие управление должны считываться из файла.

Требования.

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.

Примечания.

Для реализации управления можно использовать цепочку обязанностей, команду, посредника, декоратор, мост, фасад.

Описание архитектурных решений и классов.

Новые классы.

Для реализации считывания команд, задаваемых пользователем, был создан абстрактный класс. Это позволит добавлять новые способы считывания команд, без изменения существующего кода. То есть от этого класса могут наследоваться классы, отвечающие за конфигурацию. В данной лабораторной работе реализован класс считывания из файла.

Класс *IConfig*: абстрактный класс, отвечающий за считывание команд.

У данного класса есть два поля типа *std::map* – ассоциативный контейнер, работающий по принципу ключ-значение. *command_default* задает управление игроком по умолчанию. Каждому ключу соответствует

стандартные кнопки клавиатуры, позволяющие осуществлять перемещение: *{"UP:", 'w'}, {"DOWN:", 's'}, {"LEFT:", 'a'}, {"RIGHT:", 'd'}, {"EXIT:", 'e'}, {"MENU:", 'm'}*. Контейнер по умолчанию нужен на случай, если пользователь неправильно задает конфигурацию. *commands* – ключи такие же, но значениями выступает пара, так как происходит считывание, и нужно соответствующие клавиши привязать к соответствующем действия игрока, поэтому значение *std::pair<chat, Player::Moves>*.

Метод *set_default()*. Внутри метода происходит вывод информации, что установлено управление по умолчанию. На случай выбора пользователем логирования происходит вызов метода класса *MessageFactory* сообщения с уровнем *Errors*. Так как при чтении заданных команд из файла, могла произойти ошибка чтения какой-либо команды, то ее надо установить по умолчанию. *For (auto& pair: commands)* проходим по контейнеру с командами. Так как произошла ошибка чтения, значит, значение по ключу отсутствует, его необходимо установить, то есть надо создать пару. Находим в *command_default* значение по ключу, которое было считано, и делаем пару с соответствующим Player::Moves, так *pair.second=std::make_pair(command_default.at(pair.first), pair.second.second).*

Метод *is_ok()*. Внутри метода происходит проверка, все ли команды установлены, если нет, то вызывается метод *set_default()* – установка команд по умолчанию.

Поля и два данных метода объявлены с модификатором доступа *protected*, так как доступ к ним необходим в классах наследниках.

Для того, чтобы считывать конфигурация создан чисто виртуальный метод *virtual std::map<std::string, std::pair<char, Player::Moves>> get_config() = 0*, который будет изменяться в классах наследниках.

Чисто виртуальный деструктор *virtual ~IConfig() = default*.

От данного класса можно наследовать другие различные классы, считывающие конфигурацию, при этом остальные классы менять не надо.

Класс *FileConfig*: класс считывания конфигурации из файла, является наследником абстрактного класса *IConfig*.

Объявлены приватные поля *std::fstream file* - название файла, *std::string used_chars* - поле, отвечающие за использованные символы.

Метод *bool check_used(char char_).* Данный метод отвечает за проверку, не используется же уже символ, то есть на один символ должна приходится одна команда в игре.

Конструктор от одного аргумента *explicit FileConfig(std::string& filename)*. Происходит открытие файла *file.open(filename, std::ios_base::in).* Благодаря *std::ios_base::in* происходит считывание из потока.

Переопределенный виртуальный метод *std::map<std::string, std::pair<char, Player::Moves>> GetConfig()* со спецификатором *final*, чтобы в наследуемых классах, если их создание будет необходимо, переопределение было невозможно. Реализована проверка, был ли открыт файл *file.is_open()*, если нет, то выводится сообщение об ошибке и вызывается метод *SetDefault()* – установка команд по умолчанию, и возвращается полученный контейнер *commands.* В случае удачного открытия файла работает цикл *while (getline(file, line))*, пока возможно считывание из файла. Осуществляется считывание: *stream >> command >> button*. В *command* записывается префикс команды, *button* – клавиша ему соответствующая. Далее осуществляются различные проверки:

1) Есть ли такая команда для управление игрой. Для этого в контейнере по умолчанию проверяется наличие элементов по такому ключу *commands.count(command).* При ошибке происходит вызов метода класса *MessageFactory* сообщения с уровнем *Errors.* Вызывается метод *SetDefault()* для установления команд по умолчанию.

2) Не отсутствует ли клавиша для управления button == '\0'. При ошибке происходит вызов метода класса *MessageFactory* сообщения с уровнем *Errors.* Вызывается метод *SetDefault()* для установления команд по умолчанию.

3) Не используется ли уже клавиша для управления. Так как одна клавиша соответствует одной команде осуществляем проверку: *CheckUsing(button)).* При ошибке происходит вызов метода класса *MessageFactory* сообщения с уровнем *Errors.* Вызывается метод *SetDefault()* для установления команд по умолчанию.

4) Для дальнейшей проверки используемых клавиш «расширяем строку» *using_chars* считанной переменной *button*. Осуществляем проверку, установлена ли считанная команда в пользовательском контейнере *commands.at(command).first*, если нет, то передаем считанную из потока клавишу для управления *commands.at(command).first = button*. При ошибке происходит вызов метода класса *MessageFactory* сообщения с уровнем *Errors.* Вызывается метод *SetDefault()* для установления команд по умолчанию.

В конце происходит вызов метода *IsOkey()* для контрольной проверки все ли команды для управления игрой были заданы. Метод описан выше.

Метод возвращает полученный пользовательский контейнер для управления игрой *commands.*

В деструкторе *FileConfig::~FileConfig()* происходит закрытие файла *file.close().*

Класс *ConfigReader:* класс считывания конфигураций. Позволяет создать контейнер где ключу клавише соответствует действие игрока, без названия команды, считанной из файла.

Приватное поле *std::map<char, Player::Moves> controls* – контейнер, отвечающий за управление игроком.

Реализованы конструктор по умолчанию и деструктор по умолчанию.

Метод *std::map<char, Player::Moves>& ReadConfig().* Внутри метода осуществляется считывание имени файла *std::string filename* и соответственно *std::cin >> filename.* Создается объект класса *FileConfig* и вызывается конструктор от аргумента – названия файла *cfg(filename).* Проходом по получившемуся контейнеру с командами *for (auto& pair : cfg.GetConfig())*

осуществляется добавление считанных клавиш и соответствующим им *Player::Moves*, то есть. *controls.insert(pair.second).* Метод возвращает получившийся контейнер *controls.*

Для реализации ввода команд создан интерфейс, от которого могут наследоваться классы, отвечающие за новый ввод, без изменения существующего кода.

Интерфейс *IController*: интерфейс ввода команд.

Представлен чистой виртуальной функцией, отвечающей за получение команды virtual *char GetCommand() = 0*.

Также реализован виртуальный деструктор *virtual ~IControler() = default*.

Класс *ConsoleController*: класс ввода команд с консоли. Наследник интерфейса *IController*.

Реализованы конструктор *ConsoleControler() = default* и деструктор по умолчанию *~ConsoleControler() final = default.*

В классе переопределен метод *char GetCommand()* с модификатором *final.* Внутри метода происходит считывание команды *char ch* с консоли *std::cin >> ch.* Метод возвращает считанную команду *return ch.*

Класс *ControlBridge*: связующий класс между считанными командами и осуществлением управлением внутри игры. Является наследником *MediatorObject*.

Поля класса *std::map<char, Player::Moves> command_interpretator* – контейнер, в котором будут интерпретированы считанные команды, *IControler* controler* – указатель на контролер, *Player::Moves step* – команды управления игроком.

Конструктор класса принимает в качестве аргументов объект типа *map*, отвечающий за интерпретацию команд пользователя и указатель на *IControler ControlBridge(std::map<char, Player::Moves> maps, IControler* contrl)*. По умолчанию поле *command_interpretator* такое же *std::move(maps)* – создание

данного контейнера было осуществлено в *ConfigReader, controler = contrl*, *step = Player::NOTHING* изначально игрок ничего не делает.

Метод *NextStep()*. Запрашивает у пользователя следующую команду *char ch = controler->GetCommand()*. Осуществляется проверка с помощью поиска в контейнере количества значений по ключу *command_interpretator.count(ch) == 0* на наличие такой команды, если нет, то *step = Player::Moves::NOTHING* , иначе получаем *step* по значению *command_interpretator.at(ch)*. Далее сообщается медиатору, что необходимо забрать команду *mediator->notify(this, IMediator::MEVENTS::STEP)*.

Метод *GetStep()*. Возвращает действие игрока – поле *step* типа *Player::Moves*.

В деструкторе *~ControlBridge()* происходит очищение путем удаления *delete controler.*

Изменения в классах.

Класс *Mediator*: Добавлено новое поле *ControlBridge* control* для осуществления связи между считанными командами и управлением игрой.

Метод *GameStart*. Создается объект класса *ConfigReader rcfg* для осуществления считывания конфигураций пользователя. Создается новый объект класса *ControlBridge* и вызывается соответствующий конструктор от двух аргументов: контейнера конфигураций и нового объекта класса *ConsoleControler*, то есть *control = new ControlBridge(rcfg.ReadConfig(), new ConsoleControler())*. Также устанавливается медиатор *control->SetMediator(this)* для осуществления работы методы *notify()*.

Также в некоторых методах добавлена конструкции *control->GetStep()* и *control->NextStep()*. Первая необходима для считывания команды и осуществления последующих действий относительно нее. Вторая для получения, считывания следующей команды.

Вывод.

Реализован набор классов, отвечающих за считывание команд пользователя, обрабатывающих их и изменяющих состояние программы.

ПРИЛОЖЕНИЕ.

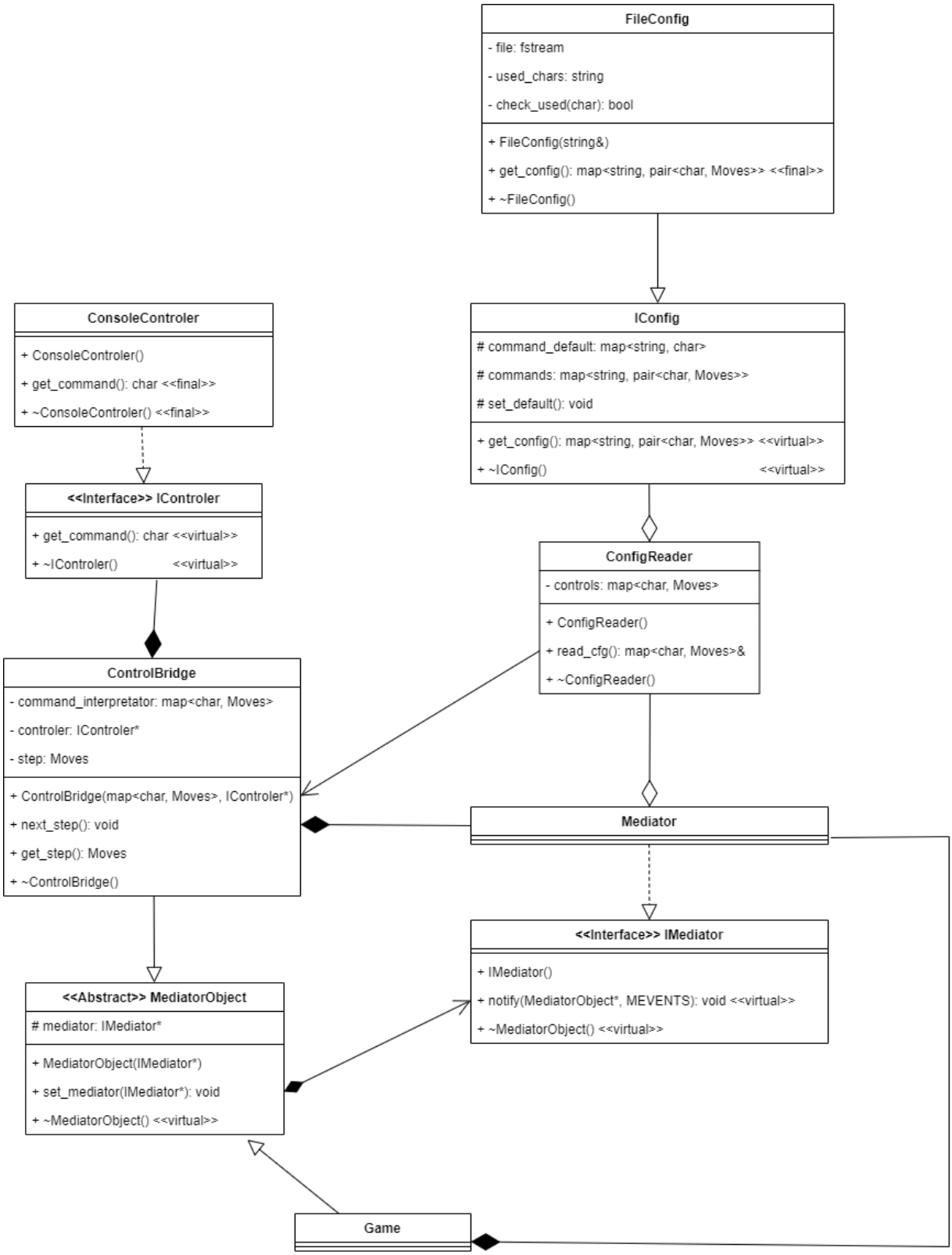

Рисунок 1 – UML-диаграмма межклассовых отношений.