МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Объектно-ориентированное программирование»
Тема: управление, разделение на уровни абстракции

Студент гр.0382 _____ Литягин С.М.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2021

Цель работы.

Реализовать управление игрой, используя разделение на уровни абстракции.

Задание.

Необходимо организовать управление игрой (номинально через CLI). При управлении игрой с клавиатуры должна считываться нажатая клавиша, после чего происходит перемещение игрок или его взаимодействия с другими элементами поля.

Требования:

- Реализовать управление игрой. Считывание нажатий клавиш не должно происходить в классе игры, а должно происходить в отдельном наборе классов.
- Клавиши управления не должны жестко определяться в коде. Например, это можно определить в отдельном классе.
- Классы управления игрой не должны напрямую взаимодействовать с элементами игры (поле, клетки, элементы на клетках)
- Игру можно запустить и пройти.

Потенциальные паттерны проектирования, которые можно использовать:

- *Команда (Command) - передача команд с информацией через единый интерфей, помещение команд в очередь*
- *Посредник (Mediator) - организация взаимодействия различных модулей*

Выполнение работы.

Для управления игрой было написано два класса: Manager и Commander. Класс Commander создан для считывания нажатий клавиш при управлении игрой вне класса игры Game. Как это происходит? Поскольку используется библиотека SFML, то мы можем обрабатывать событие event, пока открыто окно. У него есть несколько типов, но нам важно всего пара штук: Closed, KeyPressed, KeyReleased, MouseButtonPressed и MouseButtonReleased. Соответственно, событие закрытия окна, события нажатия и отжатия клавиши, события нажатия и отжатия кнопки мыши. Во время работы игры, мы постоянно отслеживаем event. Чтобы отследить события, перечисленные ранее, был написан метод Command Moving(sf::Event event). Его суть – как раз выдавать команды, согласно типу события. Для команд мы также создали перечисление Command: moveUp, moveDown, moveRight, moveLeft, close, objectAction, nothing, win, death. Он также имеет два поля: sf::Keyboard::Key direction[4] (хранит 4 клавиши управления; по стандарту управление на WASD, но можно задать другие при создании объекта; благодаря этому, нет жесткого определения клавиш в коде) и bool pressed (переменная-флаг, что нужна для блокировки зажатия кнопки). И все же ближе к сути, созданы два варианта управления (доступны одновременно). Результат данного метода зависит от типа события. Собственно, если событие KeyPressed, то он сравнивает нажатую клавишу с клавишами управления (direction). Если есть совпадение – возвращает одну из команд: moveLeft, moveUp, moveRight, moveDown (команды по перемещения главного героя). Если событие Closed, то вернет команду close (команда закрытие игры). Если событие MouseButtonPressed, то сравнивает координату, куда произошел клик левой кнопкой мыши (если по одной из нарисованных кнопок, то вернет соответствующую команду по перемещению игрока, как и при событии KeyPressed). Если же событие KeyReleased или MouseButtonReleased, то вернет команду objectAction (команда по перемещению

всех остальных юнитов на поле). Если никакого из перечисленных событий не было, то вернет команду nothing(т.е. команду ничего-неделания).

Экземпляр данного класса game создается в Game. И когда открыто окно игры, т.е. идет процесс игры и у нас обновляются события event, вместе с обновлением мы вызываем ему метод Moving(), тем самым и получая команду для дальнейшего управления игрой.

Как ранее уже был упомянуто, для управления был еще написан класс Manager. Экземпляр класса создается в конструкторе Game, который его также потом будет хранить в поле Manager* manager. Класс имеет несколько полей: ObjectAction* objAct, HeroAction* heroAct (о них позже), sf::RenderWindow* window (нужно при обработке команд close, death, win). Это класс для обработки команд, большая часть из которых получается с использованием класса Commander. Для этой цели был написан метод DoIteration(Command command). Он как раз и вызывается после получения команды в основном процессе игры (также после этого специально вызывается еще два раза этот метод с командами death и win для проверки, не умер ли игрок или не победил ли игрок). Согласно команде, с помощью switch, выбирается то, что произойдет с игрой. Тут все же стоит прервать на короткое объяснение классов ObjectAction и HeroAction.

Эти классы были написаны, чтобы управление игрой напрямую не взаимодействовало с элементами игры. ObjectAction отвечает за все игровые объекты, кроме игрока. Для этого ему нужно хранить массив противников, массив вещей, их количества, указатель на игровое поле и указатель на логгер. Его методы ранее находились в классе Game, поэтому вдаваться в подробности не буду: CheckObject() (проверяет, нет ли на поле уже убитых/использованных объектов; если есть – убирает их) и MoveEnemy() (двигает противников). HeroAction отвечает за передвижение игрока. Для этого ему нужно хранить указатель на игрока, указатель на игровое поле. У класса, конечно же, есть методы для перемещения игрока в 4 стороны (MoveHeroUp() и т.п.), метод

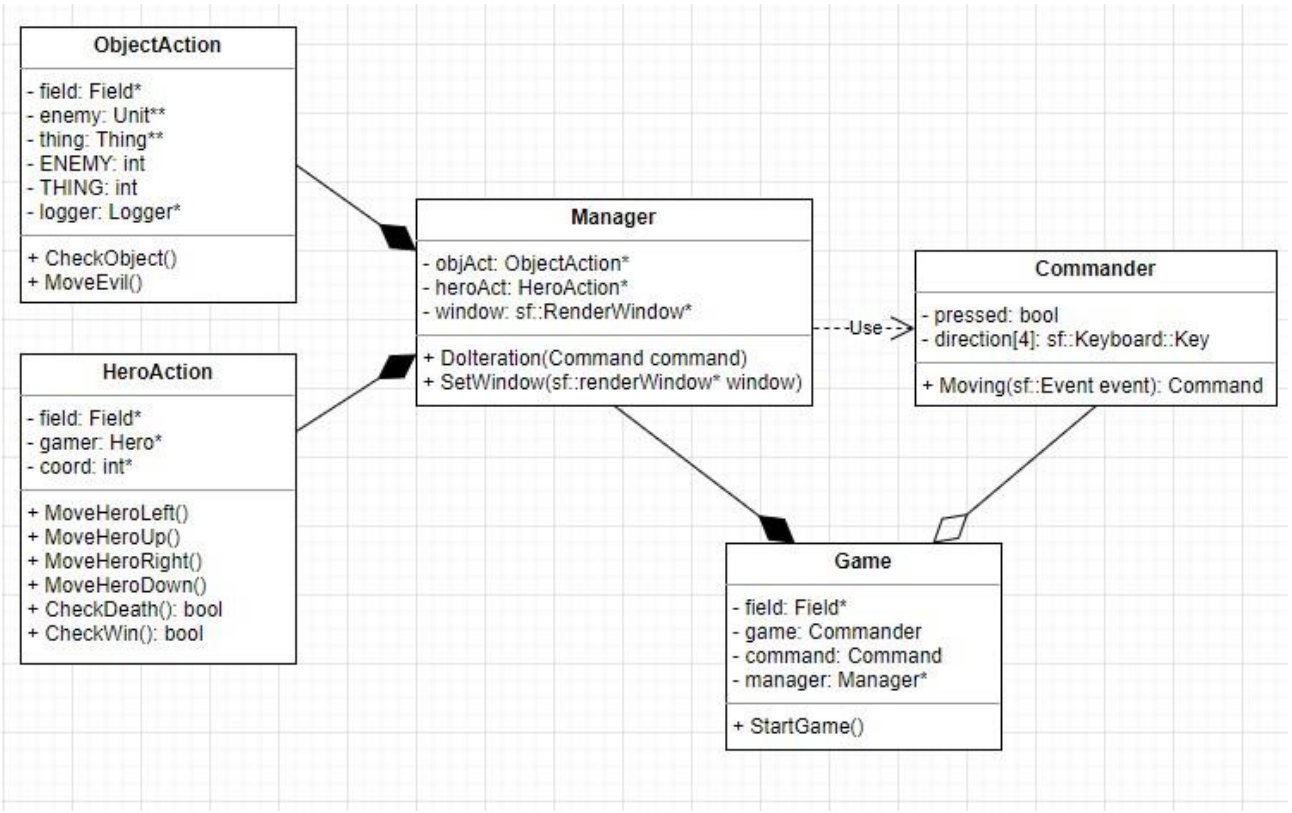
CheckDeath() (для проверки, умер ли игрок) и метод CheckWin() (для проверки, дошел ли игрок до финиша).

Возвращаясь к методу Manager. Согласно полученной команде, switch выбирает, что именно должно произойти. Если команда objectAction, то вызываются методы objAct для проверки объектов на поле и передвижения противников. Если команды moveRight, moveLeft, moveUp, moveDown, то вызывается соответствующий метод для передвижения игрока у объекта heroAct. Если команда close, то выводится в консоль сообщение о закрытии игры, и для окна window вызывается метод close(). Если команда death, то вызывается метод для проверки, жив ли игрок, объекта heroAct; если игрок мертв – выводится об этом сообщение в консоль и окно закрывается. Если команда win, то вызывается метод для проверки, дошел ли игрок до финиша; если дошел – выводится об этом сообщение в консоль и окно закрывается.

При закрытии окна, завершается и игра. Таким образом, мы реализовали управление игрой со считыванием клавиш в отдельном классе и избежали прямого воздействия классов управления на элементы игры.

UML-диаграмма классов представлена на рис. 1.

Рисунок 1 – UML-диаграмма классов.



Выводы.

В ходе работы было изучено разделение работы на уровни абстракции; было написано управление, работа которого как раз и разбивается на уровни абстракции.