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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Объектно-ориентированное программирование»

Тема: Интерфейсы, динамический полиморфизм.

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2022

Цель работы.

Реализовать систему событий. Событие - сущность, которая срабатывает при взаимодействии с игроком. Должен быть разработан класс интерфейс общий для всех событий, поддерживающий взаимодействие с игроком. Необходимо создать несколько групп разных событий реализуя унаследованные от интерфейса события (например, враг, который проверяет условие, будет ли воздействовать на игрока или нет; ловушка, которая безусловно воздействует на игрока; событие, которое меняет карту; и.т.д.). Для каждой группы реализовать конкретные события, которые по-разному воздействуют на игрока (например, какое-то событие заставляет передвинуться игрока в определенную сторону, а другое меняет характеристики игрока). Также, необходимо предусмотреть событие “Победа/Выход”, которое срабатывает при соблюдении определенного набора условий.

Реализовать ситуацию проигрыша (например, потери всего здоровья игрока) и выигрыша игрока (добрался и активировал событие “Победа/Выход”)

Требования.

Разработан интерфейс события с необходимым описанием методов.

Реализовано минимум 2 группы событий (2 абстрактных класса наследников события).

Для каждой группы реализовано минимум 2 конкретных события (наследники от группы события).

Реализовано минимум одно условное и безусловное событие (условное - проверяет выполнение условий, безусловное - не проверяет).

Реализовано минимум одно событие, которое меняет карту (меняет события на клетках или открывает расположение выхода или делает какие-то клетки проходимыми (на них необходимо добавить события) или не непроходимыми.

Игрок в гарантированно имеет возможность дойти до выхода.

Примечания.

Классы событий не должны хранить никакой информации о типе события (никаких переменных и функций, дающих информацию о типе события).

Для создания события можно применять абстрактную фабрику/прототип/строитель.

Описание архитектурных решений и классов.

Новые классы.

Интерфейс события *IEvent:* реализован интерфейс события, в котором определены чисто виртуальные методы, от данного интерфейса будут наследоваться конкретные события – абстрактные классы. Виртуальный метод *virtual void reaction(Player& player) = 0* принимает в качестве аргумента игрока по ссылке, так как все изменения будут происходить с полями игрока. Для очищения памяти, удаления объектов, также прописан виртуальный деструктор *virtual ~IEvent().*

Абстрактный класс *GoodEvent:* реализован абстрактный класс-наследник от интерфейса *IEvent*, в котором определен чисто виртуальный метод *virtual void addGood(Player& player) = 0*, который принимает в качестве аргумента игрока по ссылке и будет переопределен в конкретных событиях для воздействия на игрока. Также для того чтобы класс стал абстрактным определен обычный метод *void addPlayerScore(Player& player),* то есть по умолчанию любое хорошее событие будет добавлять игроку 3 очка.

Абстрактный класс *BadEvent:* реализован абстрактный класс-наследник от интерфейса *IEvent*, в котором определен чисто виртуальный метод *virtual void addBad(Player& player) = 0*, который принимает в качестве аргумента игрока по ссылке и будет переопределен в конкретных событиях для воздействия на игрока. Также для того чтобы класс стал абстрактным

определен обычный метод *void cutPlayerScore(Player& player),* то есть по умолчанию любое плохое событие будет отнимать у игрока 4 очка.

Класс *Cave:* реализован класс-наследник от абстрактного класса *GoodEvent.* В нем реализованы конструктор по умолчанию *Cave() = default* и деструктор *~Cave()* со спецификатором *final*, в случае создания наследников, чтобы они не смогли переопределить данный метод. Переопределены два метода - *void reaction(Player& player)* интерфейса события и *void addGood(Player& player) final* абстрактного класса, от которого происходит наследование. Первый метод вызывает переопределенный в данном классе метод *addGood(player)*, который добавляет игроку здоровья и еды в количестве 2 и 1 соответственно, и вызывает метод класса-родителя *addPlayerScore(player)* для начисления игроку очков за попадание на хорошее событие.

Класс *Exit:* реализован класс-наследник от абстрактного класса *GoodEvent.* В нем реализованы конструктор по умолчанию *Exit() = default* и деструктор *~Exit()* со спецификатором *final*, в случае создания наследников, чтобы они не смогли переопределить данный метод. Переопределены два метода - *void reaction(Player& player)* интерфейса события и *void addGood(Player& player) final* абстрактного класса, от которого происходит наследование. Первый метод обращается к методу игрока *setWinner()*, для установления игроку статуса победитель, так как он дошел до выхода.

Класс *Resource:* реализован класс-наследник от абстрактного класса *GoodEvent.* В нем реализованы конструктор по умолчанию *Resource() = default* и деструктор *~Resource()* со спецификатором *final*, в случае создания наследников, чтобы они не смогли переопределить данный метод. Переопределены два метода - *void reaction(Player& player)* интерфейса события и *void addGood(Player& player) final* абстрактного класса, от которого происходит наследование. Первый метод вызывает переопределенный в данном классе метод *addGood(player)*, который добавляет игроку ресурса в

количестве 3 единиц, и вызывает метод класса-родителя *addPlayerScore(player)* для начисления игроку очков за попадание на хорошее событие. Также это событие условное, в нем происходит проверка: собрал ли игрок ресурсов больше 5(для проверки вызывается соответствующий метод игрока *getResource())* и не открыт ли выход уже(для проверки вызывается соответствующий метод игрока *getOpenExit()).* В случае выполнения двух условий вызывается метод игрока *setOpenExit()* и открывается выход.

Класс *Clan:* реализован класс-наследник от абстрактного класса *BadEvent.* В нем реализованы конструктор по умолчанию *Clan() = default* и деструктор *~Clan()* со спецификатором *final*, в случае создания наследников, чтобы они не смогли переопределить данный метод. Переопределены два метода - *void reaction(Player& player)* интерфейса события и *void addBad(Player& player) final* абстрактного класса, от которого происходит наследование. Первый метод вызывает переопределенный в данном классе метод *addBad(player)*, который отнимает у игрока здоровье в количестве 2 единиц, и вызывает метод класса-родителя *CutPlayerScore(player)* для убавления игроку очков за попадание на плохое событие. Также вызывается метод игрока *setDead()* для проверки, не умер ли игрок, чтобы завершить игру в данном случае.

Класс *Hungry:* реализован класс-наследник от абстрактного класса *BadEvent.* В нем реализованы конструктор по умолчанию *Hungry() = default* и деструктор *~Hungry()* со спецификатором *final*, в случае создания наследников, чтобы они не смогли переопределить данный метод. Переопределены два метода - *void reaction(Player& player)* интерфейса события и *void addBad(Player& player) final* абстрактного класса, от которого происходит наследование. Первый метод вызывает переопределенный в данном классе метод *addBad(player)*, который отнимает у игрока еду в количестве 2 единиц, и вызывает метод класса-родителя *CutPlayerScore(player)* для убавления игроку очков за попадание на плохое событие. Также вызывается метод игрока

setDead() для проверки, не умер ли игрок, чтобы завершить игру в данном случае.

Порождающий паттерн. Абстрактная фабрика.

Для создания целых семейств связанных продуктов – событий, без указания конкретных классов продуктов, реализуем паттерн абстрактная фабрика.

Абстрактная фабрика задаёт интерфейс создания всех доступных типов продуктов, а каждая конкретная реализация фабрики порождает продукты одной из вариаций. Клиентский код вызывает методы фабрики для получения продуктов вместо самостоятельного создания с помощью оператора *new*. При этом фабрика сама следит за тем, чтобы создать продукт нужной вариации.

Интерфейс фабрики *FactoryEvent*: реализован интерфейс фабрики, в котором определены чисто виртуальные методы, от данного интерфейса будут наследоваться конкретные фабрики-классы. Виртуальный метод *virtual IEvent* createEvent() = 0* будет переопределён в классах наследниках и будет возвращать указатель на *IEvent*, так как на данном этапе мы не работаем с конкретным событием. Для очищения памяти, удаления объектов, также прописан виртуальный деструктор *virtual ~FactoryEvent()*.

Класс *FactoryCave*: реализован класс-наследник от интерфейса *FactoryEvent*, в котором переопределен виртуальный метод *IEvent* createEvent*() со спецификатором *final*, чтобы в дальнейшем его переопределение было невозможно. Реализация метода заключается в том, что создается объект класса *Cave.*

Класс *FactoryClan*: реализован класс-наследник от интерфейса *FactoryEvent*, в котором переопределен виртуальный метод *IEvent* createEvent*() со спецификатором *final*, чтобы в дальнейшем его переопределение было невозможно. Реализация метода заключается в том, что создается объект класса *Clan.*

Класс *FactoryExit*: реализован класс-наследник от интерфейса *FactoryEvent*, в котором переопределен виртуальный метод *IEvent* createEvent*() со спецификатором *final*, чтобы в дальнейшем его переопределение было невозможно. Реализация метода заключается в том, что создается объект класса *Exit.*

Класс *FactoryHungry*: реализован класс-наследник от интерфейса *FactoryEvent*, в котором переопределен виртуальный метод *IEvent* createEvent*() со спецификатором *final*, чтобы в дальнейшем его переопределение было невозможно. Реализация метода заключается в том, что создается объект класса *Hungry.*

Класс *FactoryResource*: реализован класс-наследник от интерфейса *FactoryEvent*, в котором переопределен виртуальный метод *IEvent* createEvent*() со спецификатором *final*, чтобы в дальнейшем его переопределение было невозможно. Реализация метода заключается в том, что создается объект класса *Resource.*

Изменения в классах.

Класс *Player*: добавлены методы *addHealth(), addFood(), addScore(), addResource(),* которые увеличивают соответствующие поля игрока на определенное значение. Методы *cutHealth(), cutFood(), cutScore(),* которые уменьшают соответствующие поля на определённое значение. Добавлены поля *winner, dead и open_exit* и соответственные методы для них. Поля добавлены в конструктор и их значение по умолчанию – *false*. Геттеры, для получения значения этих полей, и сеттеры, для установления значения. Для установления значения поля *winner* в *true* происходит проверка: набрано ли ресурсов больше 5. Для установления значения поля *dead* в *true* происходит проверка: значение полей здоровья или еды не меньше ли 0. Сеттер для поля *open_exit* устанавливает его значение в *true*.

Класс *Cell*: добавлено поле *IEvent* event* – в поле будет храниться указатель на событие в зависимости от того, которое в нем будет определено.

Реализован геттер, который возвращает значение *event* соответствующей клетки. В соответствующем сеттере в данное поле устанавливается *create_event*. Так как после наступления на клетку и влияния события на игрока событие из клетки удаляется, определён метод *deleteEvent()*, который полю event присваивает значение *nullptr*. В деструкторе добавлено очищение клетки от *event*, находящегося на ней, даже если игрок не взаимодействовал с этой клеткой.

Класс Field: в методах, отвечающих за передвижение игрока *moveUp/Down/Right/Left* добавлено удаление события из клетки, на которой на данный момент находиться игрок, так как событие уже было использовано. Проверяется есть ли на клетки событие, в случае выполнения условия вызывается метод *reaction(player)* для воздействия конкретного события на игрока. Также добавлена проверка: открыл ли уже игрок выход (это позволяет сделать метод *getOpenExit())* и отсутствует ли в самой нижней правой клетки событие, при выполнении каждого условия устанавливается выход методом, определенном в классе *Field::setExit(player).* Для помещения события на клетку реализуется метод *setEventOnCell(FactoryEvent* factory),* так как в фабрике происходит создание конкретного события в метод передается указатель на объект данного класса. Каждого вида событий на карте будет 2, поэтому запускается цикл *while (count < 2)* и случайным образом с помощью *rand()* выбирается значения *y* и *x*, для установления в конкретную клетку события. Проверка: не является ли данная клетка по умолчанию уже непроходимой и нет ли в ней уже *event*. При выполнении условий вызывается метод клетки *setEvent()* и устанавливается событие, то есть в зависимости от того, какая именно фабрика-события поступает функции на обработку в клетке создается соответствующий класс конкретного события. Метод *setFieldEvents()* реализует добавление конкретных событий на поле: создается указатель на фабрику-события, одно из 4, создается класс соответствующего типа, затем вызывается *setEventOnCell()* для установления этого события на

поле, затем фабрика очищается, так как она нам нужна только для создания объекта и дальше не пригодится. Для изменения карты – появления выхода реализуется метод *setExit(),* который проверяет доступен ли игроку на данном этапе выход значение *getOpenExit()* должно быть *true*, в этом случае как в предыдущем описании создания event на клетках создается фабрика-выхода, на клетку в самом нижнем правом углу устанавливается данное событие, фабрика удаляется.

Класс *CellView:* в методе *printCell()* добавлены условия, что будет выводиться на экран в случае, если на клетке есть конкретное событие. Проверка происходит за счет того, можно ли возвращаемое значение, получаемое методом *getEvent(),* с помощью *dynamic_cast<>* преобразовать к типу указатель на класс конкретного события.

Класс *Application*: в методе *start()* добавление вызывание *field.setFieldEvents()* для расстановки событий на игровом поле.

ПРИЛОЖЕНИЕ.

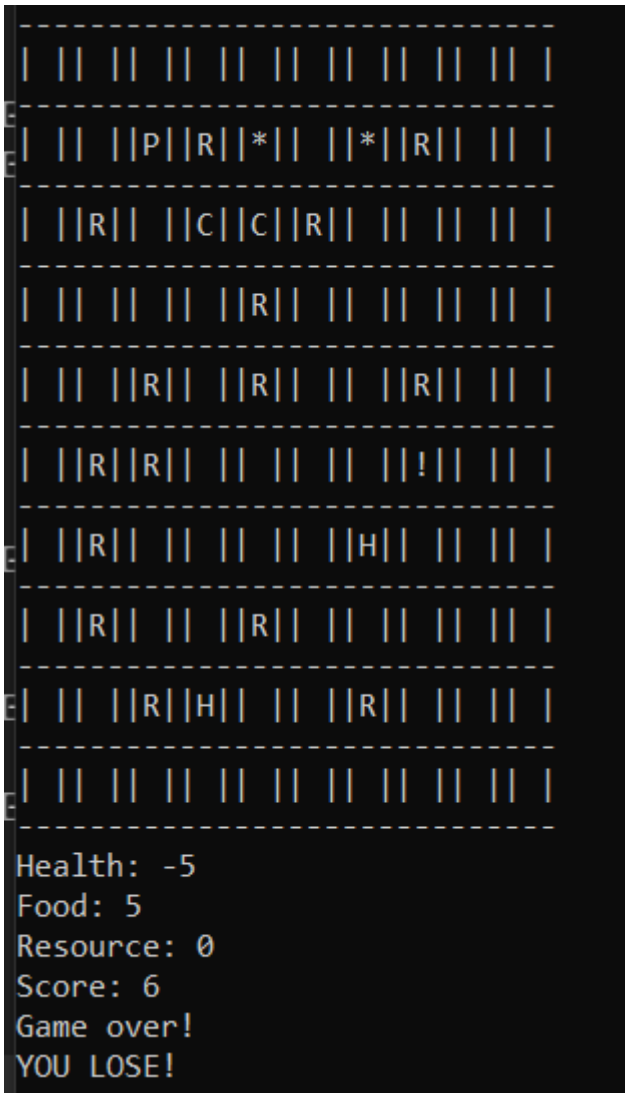

Рисунок 1 – работа программы

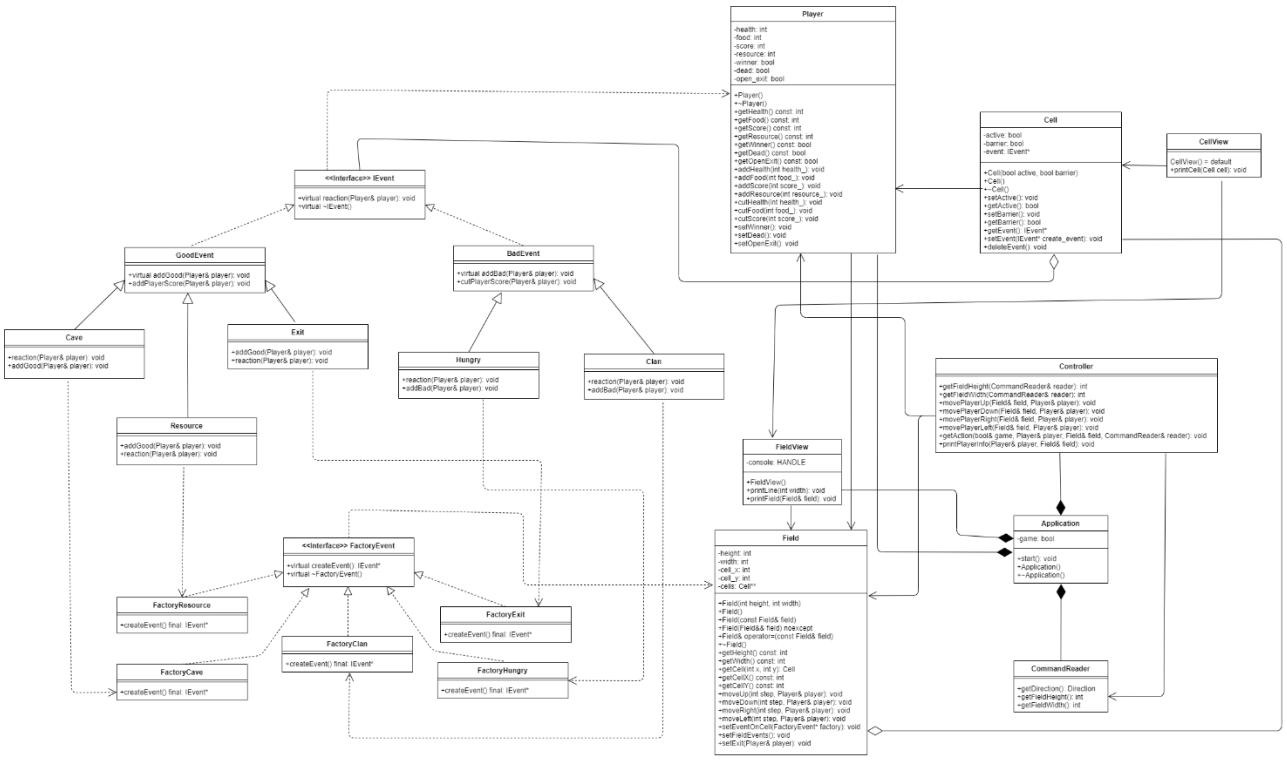

Рисунок 2 – UML-диаграмма