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

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Объектно-Ориентированное программирование»
Тема: Связывание классов

Студент гр. 3341 _____ Бойцов В.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2024

Цель работы.

Целью работы является изучение основ объектно-ориентированного программирования, создание архитектуры игры, игровых состояний и системы игроков для написания игры «Морской бой» на основе разработанных ранее классов.

Для выполнения поставленной цели требуется:

- Разработать архитектуру классов игры, игровых состояний, игроков;
- Реализовать эти классы и построить связь между ними и написанными ранее классами.

Задание.

a. Создать класс игры, который реализует следующий игровой цикл:

i. Начало игры

ii. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.

iii. В случае проигрыша пользователь начинает новую игру

iv. В случае победы в раунде, начинается следующий раунд, причем состояние поля и способностей пользователя переносятся.

Класс игры должен содержать методы управления игрой, начало новой игры, выполнить ход, и т.д., чтобы в следующей лаб. работе можно было выполнять управление исходя из ввода игрока.

b. Реализовать класс состояния игры, и переопределить операторы ввода и вывода в поток для состояния игры. Реализовать сохранение и загрузку игры. Сохраняться и загружаться можно в любой момент, когда у пользователя приоритет в игре. Должна быть возможность загружать сохранение после перезапуска всей программы.

Примечание:

- Класс игры может знать о игровых сущностях, но не наоборот
- Игровые сущности не должны сами порождать объекты состояния
- Для управления самое игрой можно использовать обертки над командами
- При работе с файлом используйте идиому RAII.

Выполнение работы.

Перед описанием основных этапов выполнения работы необходимо отметить, что к создаваемой архитектуре было выдвинуто дополнительное требование, отражающееся на её масштабируемости: игра должна уметь поддерживать игровой процесс для любого количество игроков (или ботов), превышающего двух. Такое требование позволяет создавать более общую и масштабируемую архитектуру, при это всё так же удовлетворяя заданию.

Также необходимо отметить, что некоторые аспекты разработанной архитектуры созданы ради поддержания совместимости с заданием лабораторной работы 4, а некоторые аспекты будут изменены позднее, в 4 работе, т.к. на данный момент предусмотреть все моменты реализации представляется затруднительным.

Для представления абстрактного игрока была создана структура *Participant*, которая содержит в себе поля, представленные объектами классов основных игровых сущностей (например, игровое поле *Battlefield mField* или менеджер кораблей *ShipManager mShipManager*). Данная структура позволяет хранить игровые единицы участника игры в одном месте. От неё наследуются структуры *Bot* и *Player*, представляющие компьютерного игрока и реального игрока соответственно. Несмотря на то что класс игры, как будет показано далее, способен обрабатывать ввод участников независимо от того, являются ли они реальными игроками или ботами, такое разделение может оказаться полезным в ходе управления игровым процессом в дальнейшем. Тем самым, функционал этих структур может быть дополнен в будущем, но основная логика будет прописана в иных классах, например, контроллере игрока.

Для хранения информации о текущем состоянии игры была написана структура *GameInfo*, которая хранит в себе вектор указателей на абстрактных игроков, а также такую информацию как номер хода, номер раунда, количество ботов, игроков и т.п. эта структура применяется как в классе игровых состояний или самой игре, так и для реализации функционала сохранения/загрузки.

Понятно, что игра может находиться в разных состояниях: например, в состоянии расстановки кораблей, когда игроки не могут атаковать, или в состоянии атаки, когда игроки не могут ставить корабли. В этой связи были разработаны классы, отвечающие за данную механику и следующие паттерну «Состояние». Был создан полиморфный класс *GameState*, определяющий основные действия своими методами (атака – *attack*, расставить корабль-*placeShip* и т.д.), а также классы, отвечающие за эти игровые состояния – *ShipPosState* и *AttackState*, которые реализуют действия, предусмотренные данным состоянием. Это позволяет обобщить вызов методов игры и избавиться от множественных проверок.

Для реализации сохранения и загрузки были созданы два класса: *GameSaver* и *FileHandler*. *GameSaver* имеет полем ссылку на структуру *GameInfo* и определяет операторы ввода и вывода в поток. При выводе в поток класс записывает количество игроков, номер хода и т.д., а затем для каждого игрока записывает все его корабли, состояние менеджера кораблей, поля и т.д. при выгрузке из потока создаются новые объекты-корабли, игровое поле, в них записываются считанные из потока данные; затем на поле расставляются корабли, и эти данные передаются в конструктор бота или игрока. *FileHandler* отвечает за работу с файлом, его открытие/закрытие для чтения/записи, а также за запись в файл или выгрузку из него класса *GameSaver*.

Связующим звеном между данными классами служит класс *Game*, который посредством композиции реализует общую логику игры. Класс предоставляет методы для различного рода действий (атака, сохранение и загрузка, постановка корабля, применение способности и т.д.), а также управляет данными, содержащимися в *GameInfo*, и предоставляет методы для управления организацией игры и игровым процессом (добавление нового игрока, бота, условия окончания игры – количество оставшихся в живых игроков, индекс текущего активного игрока и др.)

Необходимо отметить, что за непосредственную организацию игрового процесса будут отвечать другие классы – контроллер игры, игрока, система

команд и т.д, однако они не являются частью данной лабораторной работы, пускай и некоторый экспериментальный их вариант был создан в целях тестирования и демонстрации возможностей данной архитектуры.

Диаграмма классов, разработанных в ходе выполнения лабораторной работы, представлена на рис.1.

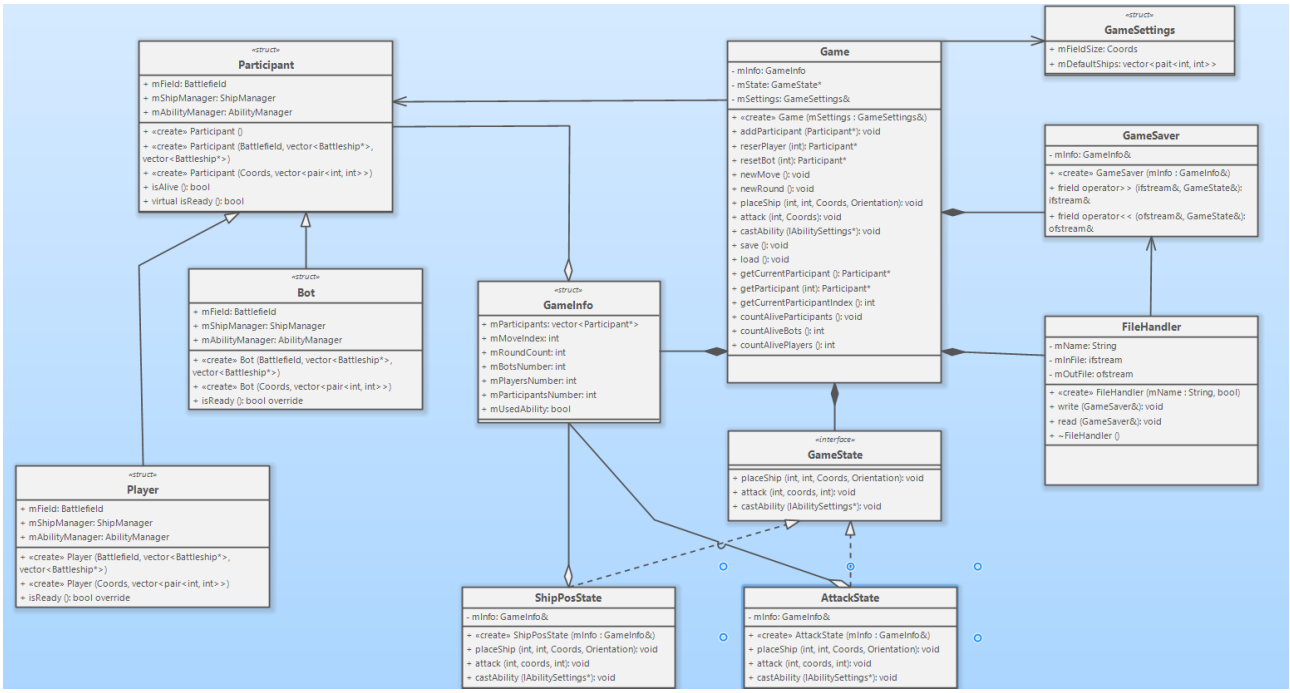

Рисунок 1 – диаграмма классов

Результаты тестирования см. в приложении А.

Выводы.

В результате выполнения лабораторной работы были изучены основы Объектно-ориентированного программирования на языке C++. Была придумана архитектура, объединяющая разработанную ранее систему классов для реализации игрового процесса, были написаны классы, позволяющие управлять состоянием игры, совершать игровые действия. Реализованная архитектура достаточно масштабируема и позволяет организовывать игровой процесс для произвольного количество игроков и компьютерных соперников.

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Тестирование написанных классов было выполнено в виде небольшой программы, результат работы которой представлен ниже:

```
Started ship placing phase

 Enter Command:
r 0 0 0 h
Init ship placement
Ended with 1
Init ship placement
Ended with 1
Ended ship placing phase

New round has started!

Move 0

Participant #0   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
----------
----------
----------
----------
----------
----------
----------


Participant #1   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
```

```
----------
----------
----------
----------
----------
----------
----------


Participant #2   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
----------
----------
----------
----------
----------
----------
----------


 Enter Command:
f 2 0 0

Move 1

Participant #0   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
----------
----------
----------
----------
----------
----------
----------
```

```
Participant #1   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
----------
----------
----------
----------
----------
----------
----------


Participant #2   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        1 2 2 2

x---------
----------
----------
----------
----------
----------
----------
----------
----------
----------


Move 2

Participant #0   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
```

```
----------
----*-----
----------
----------
----------
----------
----------


Participant #1   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
----------
----------
----------
----------
----------
----------
----------


Participant #2   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        1 2 2 2

x---------
----------
----------
----------
----------
----------
----------
----------
----------
----------


Move 3

Participant #0   1
Shelling
```

```
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
---*------
----*-----
----------
----------
----------
----------
----------


Participant #1   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        2 2 2 2

----------
----------
----------
----------
----------
----------
----------
----------
----------
----------


Participant #2   1
Shelling
Inactive ships in manager: 0
Active ships in manager: 1
Ship (0 , 0) condition:
        1 2 2 2

x---------
----------
----------
----------
----------
----------
----------
----------
```

