**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. 3383 |  | Галкин А.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

**Цель работы.**

Изучение программной идиомы RAII (Resource Acquisition Is Initialization) на примере работы с файлами для загрузки/сохранения игры «Морской бой». Изучение работы с переопределением операторов ввода в поток/вывода из потока и процесса сериализации/десериализации. Связывание написанных в предыдущих лабораторных работах классов игровых объектов в единую игру – класс Game.

**Задание на лабораторную работу**

1. Создать класс игры, который реализует следующий игровой цикл:
   1. Начало игры
   2. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
   3. В случае проигрыша пользователь начинает новую игру
   4. 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.

1. 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.**

Во время выполнения лабораторной работы разработаны классы Game и GameState, реализующие логику класса игры и класса состояния игры, соответственно.

Класс Game. В приватных полях содержит указатели на игровые сущности – игровые поля, менеджеры кораблей игрока и противника, менеджер способностей игрока. Также содержится объект класса состояния игры GameState state\_ и указатель на компьютерного оппонента – ComputeEnemy \*enemy\_.

Метод начала новой игры StartNewGame(int fieldx, int fieldy). Реализует логику начала игры. Началом игры называется создание нового игрового поля поданных в метод размеров (fieldx, fieldy), создание нового менеджера кораблей и способностей.

Метод добавления на поле корабля AddShip(int x, int y, int size, Orientation orientation). Принимает на вход координаты, ориентацию и размер корабля, который необходимо добавить на поле. Вызывает метод AddShipToGameField класса ShipManager, принимающий ссылку на игровое поле и характеристики поданного корабля. В свою очередь менеджер кораблей создает новый корабль с поданными характеристиками и добавляет его в свое отслеживание (в вектор всех кораблей), после чего вызывает метод AddShip класса GameField, добавляющий корабль на поле. В случае получения от метода GameField::AddShip ошибки (пересечение/выход за границы поля), данная ошибка будет выброшена выше. Координаты поставленного корабля и его характеристики запоминаются в классе игры, что необходимо для сохранения/загрузки состояния игры.

Метод StartNewRound(). Реализует начало нового раунда. Началом нового раунда обозначается создание компьютерного врага, его игрового поля и менеджера кораблей. Игровое поле врага случайным образом заполняется кораблями с размерами кораблей игрока. Вражеское поле покрывается «туманом войны» и состояние кораблей становится невидимо для игрока. Также происходит сериализация – обновляется объект класса состояния игры.

Метод Serialize(). Реализует сериализацию всех игровых объектов путем вызова соответствующих методов сериализации у каждого объекта класса Game. Полученные сериализованные данные передаются и сохраняются в объекте класса GameState state\_.

Метод SaveGame(std::string filename). При помощи стандартной библиотеки <fstream>, переопределенного оператора вывода (<<) класса GameState и класса std::ofstream производится сохранение состояния игры в файл с названием, поданным в качестве аргумента filename. Класс ofstream необходим для реализации идиомы RAII, так как после выхода из зоны видимости, объект разрушится и автоматически закроется файл. В случае, если файл на сохранение открыть не удалось, выбрасывается ошибка. Иначе, в файл записываются данные, сохраненные в классе состояния игры state\_ (сериализованные объекты класса Game).

Метод LoadGame(std::string filename). Аналогично методу сохранения игры, используется класс из библиотеки <fstream>, реализующий идиому RAII – std::ifstream. Если файл для загрузки игры открыть не удалось, выбрасывается ошибка. Иначе, происходит десериализация данных, считанных в объект GameState state\_. В случае, если какие-то данные десериализовать не удалось, выбрасывается ошибка, иначе новые объекты, считанные из файла, становятся игровыми объектами и игра продолжается с ними.

Метод DoTurn(TurnType type, int x, int y). Метод необходим для совершения хода игроком. Тип хода определяется аргументом TurnType и может принимать значение SKILL – способность или ATTACK – атака. В случае, если ход – атака, вызывается метод игрового поля Attack с аргументами – координатами x и y. Иначе, вызывается метод менеджера способностей UseOwnedSkill. После вызовов проверяется, не были ли выброшены ошибки. В случае получения ошибки, она выбрасывается из метода. Иначе проверяется, был ли уничтожен корабль. В случае уничтожения корабля выдается способность и проверяется окончание игры методом checkForWin(). В случае, если метод проверки окончания раунда вернул true, возвращается Result::WIN, что означет победу в раунде и должно быть обработано для начала нового раунда.

Метод ComputerTurn(int &x, int &y). Метод совершает вражескую атаку по полю игрока. В аргументы x, y записываются координаты, по которым компьютерный враг enemy\_ совершил атаку методом DoAttack(). После этого проверяется окончание игры методом checkForLose(). Если данный метод вернул true, из метода ComputerTurn() возвращается Result::LOSE, обозначающее конец игры и победу компьютерного противника. После получения значения LOSE из данного метода, игра начинается заново.

Класс GameState. В приватных полях хранит сериализованные объекты игры в формате строк. Помимо объектов игры, хранит сериализованные данные, необходимые для восстановления связи игрового поля и менеджера кораблей – массив данных о кораблях (индекс корабля в менеджере – его координаты на поле).

Методы класса GameState представляют собой набор геттеров и сеттеров для извлечения/записи сериализованных объектов в класс.

Для обозначения игрока, которому принадлежат данные создан enum PlayerType, способный принимать значения Player и Computer.

Для того, чтобы после записи в файл, игрок не смог изменить себе состояние игры (например, можно легко понять по обычной сериализации, чем обозначается состояние сегмента корабля итд), реализована зашифровка и соответствующая дешифровка. Шифрование – метод crypt(std::string str). Проходится по буквам строки и совершает операцию XOR с буквой заданного ключа game\_key\_. Операция XOR может использоваться и для шифровки, и для последующей дешифровки аналогичным способом.

В классе GameState для записи в файл и чтения из файла переопределены и обозначены дружественными операторы ввода и вывода в поток.

Friend std::ostream& operator<<(std::ostream& out, const GameState &state). Реализует вывод путем последовательной записи в поток out строк – приватных полей GameState (сериализованных объектов) и символа перевода строки ‘\n’.

Friend std::istream& operator>>(std::istream& out, GameState &state). Ввод реализован путем последовательного чтения строк при помощи метода getline. После этого, каждая считанная строка, записывается в соответствующее поле объекта state.

Класс ComputerEnemy. В приватных полях хранит ассоциативный массив done\_, очередь attack\_queue\_ и пару, prev\_destroy\_. Done\_ необходимо для обозначения совершенных атак, чтобы бот не атаковал трижды одну и ту же клетку.

Метод DoAttack(GameField &field, int &xcoord, int &ycoord). В цикле for подбирает случайные координаты, если очередь необходимых атак пуста, до момента, пока не будет опробовано 4\*X\*Y случаев. Если координаты подобраны, они записываются в переданные аргументы xcoord и ycoord (необходимо в последующем для отрисовки), после чего совершается атака и, в зависимости от ее результата, в очередь атаки записываются необходимые координаты (добить сегмент/обойти сегмент в поиске следующего итд). Результат атаки возвращается из метода.

Результаты тестирования и UML-диаграмма реализованных классов представлены в Приложении А и Приложении Б, соответственно.

**Выводы.**

В ходе выполнения лабораторной работы, была изучена работа с файлами в языке программирования С++, а также идиома RAII, использованная для закрытия файлов при уничтожении объекта, обозначающего файл. В результате лабораторной работы связаны все написанные в предыдущих работах классы, реализована полноценная логика игры «Морской бой», описанная в условии задания.

**ПРИЛОЖЕНИЕ А**

**РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ**

Проверка корректности начала игры, начала раунда и игрового цикла

|  |  |  |
| --- | --- | --- |
| Ввод | Вывод программы | Комментарий |
|  |  | ОК |

Проверка сохранения состояния игры в файл и последующей загрузки из файла

|  |  |  |
| --- | --- | --- |
| Ввод | Вывод программы | Комментарий |
|  | - сохраненное состояние  - произведено несколько ходов  - загружено сохранение | ОК |

Проверка начала нового раунда при победе

|  |  |  |
| --- | --- | --- |
| Ввод | Вывод программы | Комментарий |
|  | - состояние перед победой (остался один ход игроку)  - после победы начался новый раунд с сохранением поля и способностей | ОК |

**ПРИЛОЖЕНИЕ Б**

**UML-ДИАГРАММА**

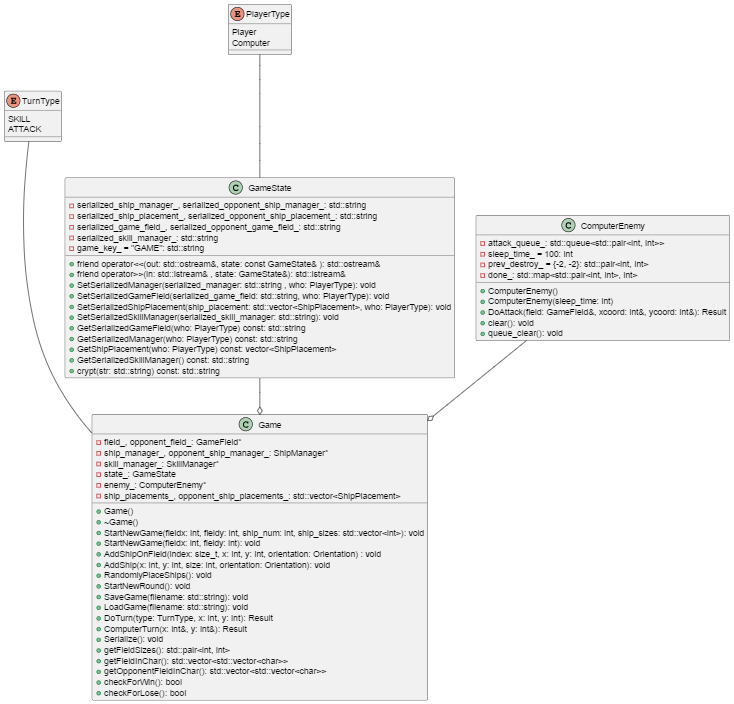


Рисунок 1 - UML-диаграмма реализованных классов