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. 3342		Мохамед М.Х.
Преподаватель		Жангиров Т. Р.

Санкт-Петербург

2024

Цель работы

Изучить связывание классов, путём усовершенствования программы из предыдущей лабораторной работы. Необходимо создать: класс игры и класс состояния игры.

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.

- 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

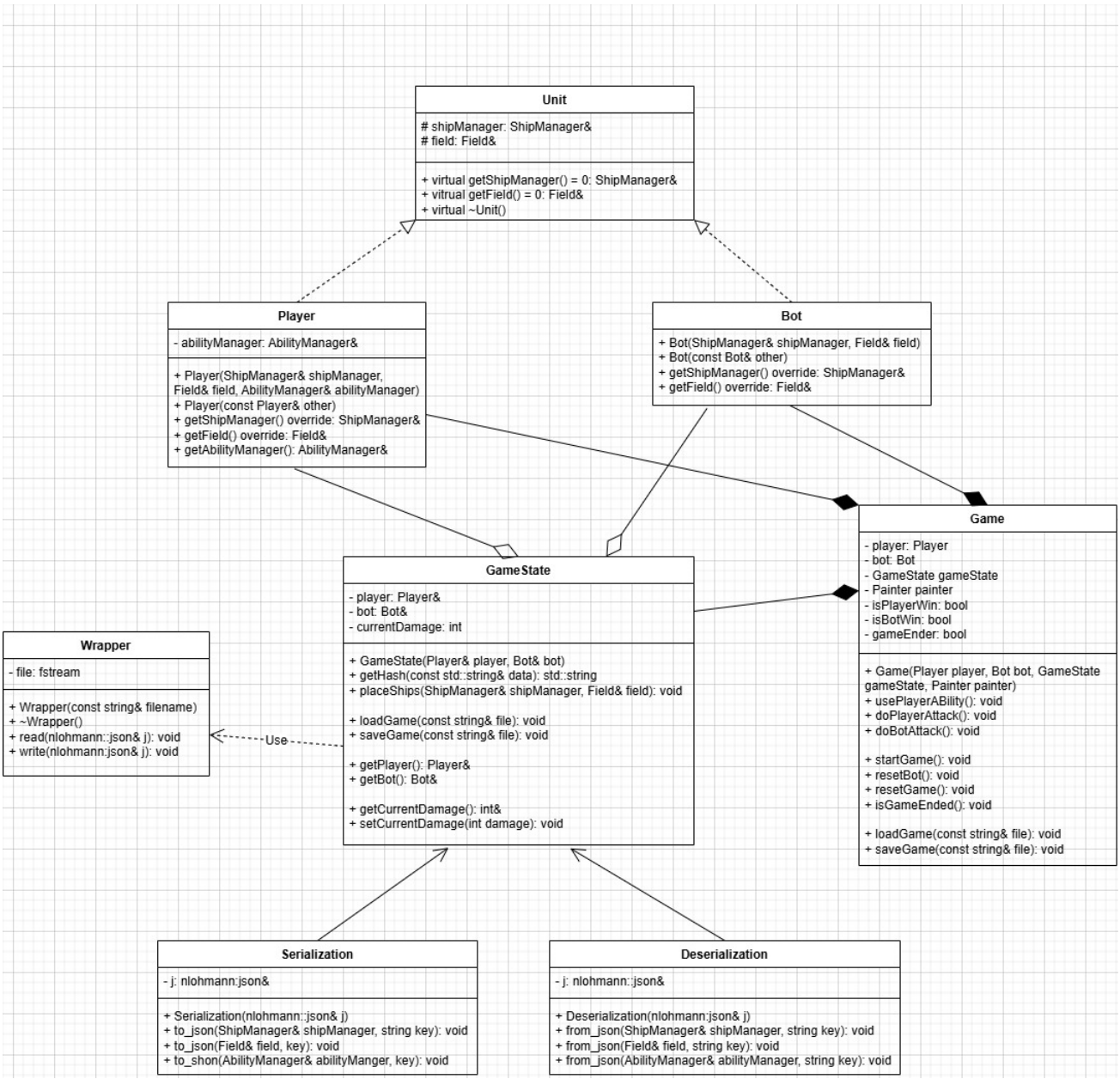

Рисунок 1 – UML-диаграмма классов

Код программы содержит реализацию классов: *Game*, *Unit*, *Player*, *Bot*, *Serialization*, *Deserialization*, *Wrapper* и *GameStatus*.

Классы *Game* и *GameState* были добавлены согласно заданию. *Game* связывает классы и работает с ними, описывает игровой цикл и выполнение ходов. Класс *GameState* отвечает за связывание классов *Serialization*,

Deserialization и *Wrapper*, которые в сумме дают возможность работать с json файлом и совершать загрузку/сохранение игры. В нём также происходит хэширование json файла для его защиты от внешнего вмешательства.

Классы *Unit, Player* и *Bot* являются дата-классами, *Unit* – абстрактный класс, который хранит общие для игрока и бота поля и методы; *Player* и *Bot* – наследуемые от *Unit* классы, представляющие собой игрока и бота соответственно, могут только возвращать значения полей.

Классы *Serialization* и *Deserialization* отвечают за считывание и запись из json файла. Прописаны методы для менеджера кораблей, поля и менеджера способностей, чтобы реализовать загрузку и сохранение игры. Обработка json файла организована с использованием библиотеки nlohmann/json.

Класс *Wrapper* реализован как обёртка над файлом с использованием идиомы RAII для более удобной работы. В конструкторе происходит открытие файла, а в деструкторе его закрытие.

Помимо обозначенных классов, реализованы и интегрированы в код новые классы-исключения для обработки различных исключительных случаев работы с файлом и игрой.

Ability является классом для реализации логики игры. Он имеет следующие поля:

- *Player player* – класс игрока.
- *Bot bot* – класс бота.
- *GameState gameState* – класс состояния игры.
- *Painter painter* – класс отрисовщика для исключений и поля.
- *bool isPlayerWin* – выиграл ли игрок.
- *bool isBotWin* – выиграл ли бот.
- *bool gameEnder* – закончилась ли игра.

И следующие методы:

- *void usePlayerAbility()* – использовать способность игрока.
- *void doPlayerAttack()* – провести атаку игрока по полю бота.
- *void doBotAttack()* – провести атаку бота по полю игрока.
- *void StartGame()* – метод начала игры, в котором, в зависимости от решения игрока, происходит непосредственно игра, загрузка/сохранение или выход.
- *void resetBot()* – обнуление бота (после победы над ним и при желании продолжить игру).
- *void resetGame()* – обнуление всей игры (после проигрыша и при желании продолжить игру).
- *void isGameEnded()* – проверка, завершилась ли игра и требуется ли продолжение.
- *void loadGame()* – вызов загрузки игры у класса состояния.
- *void saveGame()* – вызов сохранения игры у класса состояния.

Класс *Unit* является абстрактным классом для игрока и бота. Он имеет следующие protected поля:

- *ShipManager& shipManager* – ссылка на менеджер кораблей.
- *Field& field* – ссылка на поле.

И следующие виртуальные методы:

- *virtual ShipManager& getShipManager() = 0* – возвращает ссылку на менеджер кораблей.
- *virtual Field& getField() = 0* – возвращает ссылку на поле.

Класс *Player* является реализацией дата-класса игрока, который наследуется от класса *Unit*. Он имеет следующие поля:

- *ShipManager& shipManager* – ссылка на менеджер кораблей, наследуется от *Unit*.

- *Field& field* – ссылка на поле, наследуется от *Unit*.
- *AbilityManager& abilityManager* – ссылка на менеджер способностей.

И соответствующие методы для получения полей.

Класс *Bot* является реализацией дата-класса бота, он тоже наследуется от класса *Unit*. Он имеет следующие поля:

- *Field& field* – ссылка на поле, наследуется от *Unit*.

И соответствующие методы для получения полей.

Класс *Serialization* служит для записи информации в json файл с использованием библиотеки nlohmann/json. Он имеет следующее поле:

- *nlohmann::json& j* – ссылка на структуру данных для работы с json.

Он имеет три одинаковых по структуре метода (*to_json*) для подготовки к записи в файл менеджера кораблей, поля и менеджера способностей.

Класс *Deserialization* служит для загрузки информации из json файла. Он имеет следующее поле:

- *nlohmann::json& j* – ссылка на структуру данных для работы с json.

Он имеет три одинаковых по структуре метода (*from_json*) для загрузки из файла менеджера кораблей, поля и менеджера способностей.

Класс *Wrapper* является обёрткой над файлом с использованием идиомы RAII. Он имеет следующее поле:

- *fstream file* – поток для работы с файлом.

И следующие методы:

- *read(nlohmann::json& j)* – записывает содержимое файла в структуру json.

- *write(nlohmann::json& j)* – записывает содержимое структуры json в файл.

Класс *GameState* является классом состояния для связывания других классов и для реализации полной логики загрузки/сохранения игры. Он имеет следующие поля:

- *Player& player* – ссылка на игрока.
- *Bot& bot* – ссылка на бота.
- *bool isAbilityUsed* – была ли использована способность (случай сохранения до атаки).
- *int currentDamage* – текущий урон (случай сохранения до атаки для двойного урона).

И следующие методы:

- Wrapper& operator<<(Wrapper& fileWrapper, GameState& state) – переопределяет оператор << следующим образом: сначала происходит сериализация и вся необходимая информация по кораблям, полям и способностям сохраняется в библиотечную структуру, которая потом переносится в обёртку и она возвращается.
- *Wrapper& operator>>(Wrapper& fileWrapper, GameState& state)* – переопределяет оператор >> следующим образом: сначала происходит считывание информации из обёртки в структуру json, затем десериализация, информация записывается в временные объекты и позже переносится на используемые, в конце возвращается обёртка.
- *void placeShips(ShipManager& shipManager, Field& field)* – расставляет корабли обратно после загрузки из файла.
- *void loadGame(const string& file)* – создаёт обертку и заполняет объект класса информацией из файла.

- void saveGame(const string file) – очищает файл, создаёт обёртку и загружает в неё информацию из объекта класса.
- *bool& getIsAbilityUsed()* – возвращает информацию о том, была ли использована способность.
- *void setIsAbilityUsed(bool value)* – выставляет информацию о использовании способности.
- *int& getCurrentDamage()* – возвращает урон.
- *void setCurrentDamage(int damage)* – выставляет урон.

Тестирование:

Происходит симуляция игры между игроком (слева) и ботом (справа), для этого используется большая часть реализованных методов внутри классов. Поле игрока изначально открыто, а вражеское скрыто. В начале хода игрок может использовать одну случайную способность или сразу перейти к атаке вражеского поля.

В классе *Game* реализована логика игры, которая позволяет выбирать действия в зависимости от команд пользователя. Он может: запустить игру, реализовав игровой цикл, с возможностью выйти обратно после использования способности; загрузить игру, получив состояния кораблей, поля и способностей; сохранить игру, уже записав состояния игровых сущностей; выйти из игры.

При победе игроку предлагается продолжить игру с сохранением его поля и с новым противником. В случае победы бота, игру можно продолжить, обнулив вообще всё.

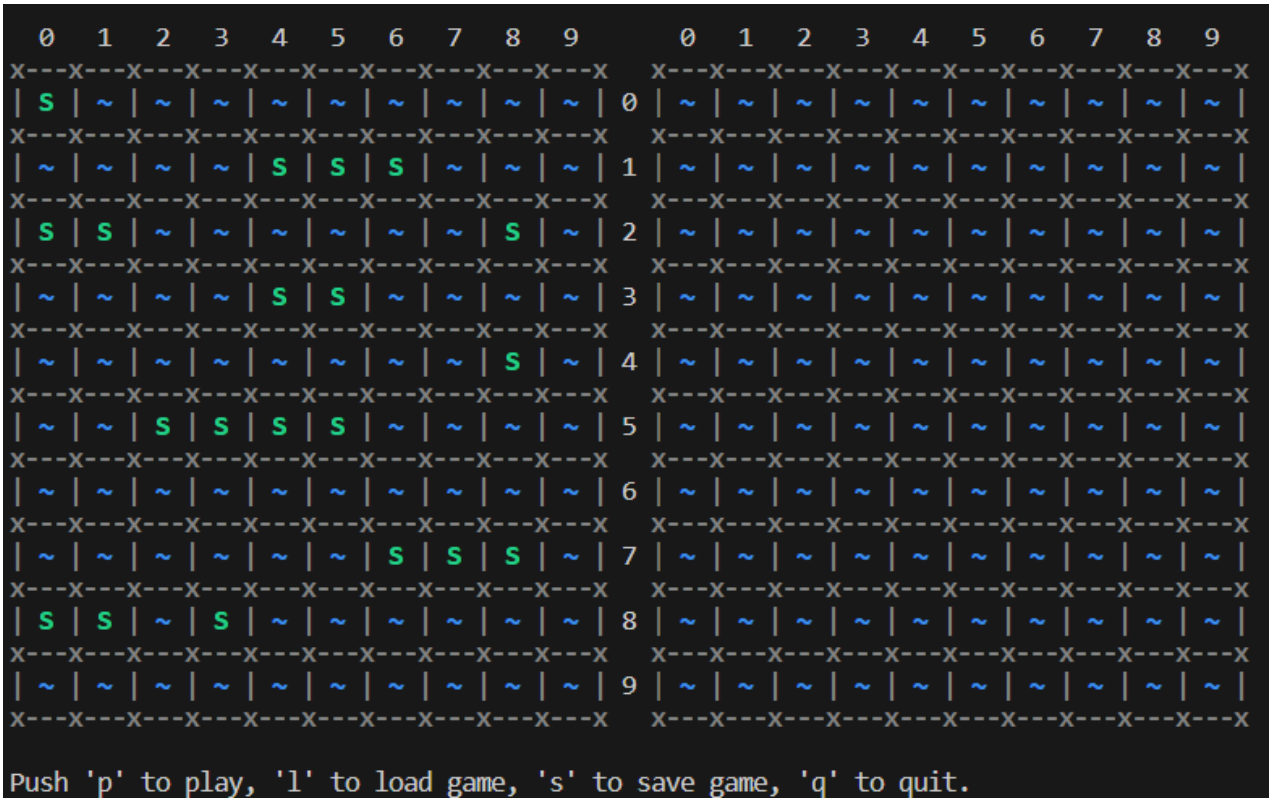

Рисунок 2 – Начало игры

Рисунок 3 – Игра сохранена и закрыта.

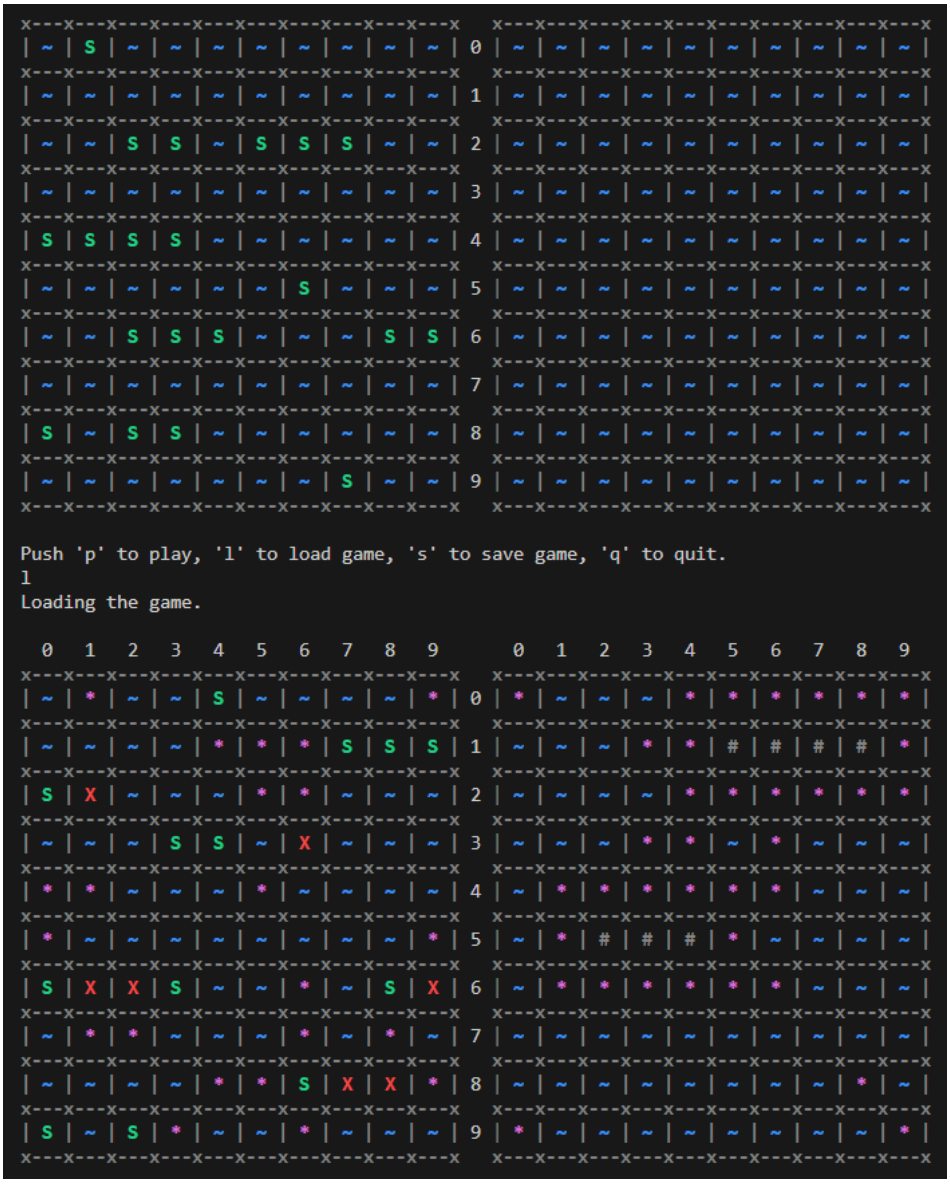

Рисунок 4 – Игра загружена

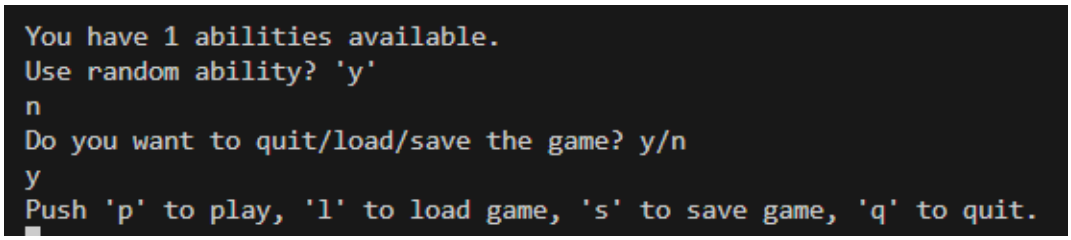

Рисунок 5 – Пример выхода в меню выбора после работы со способностью (до атаки)

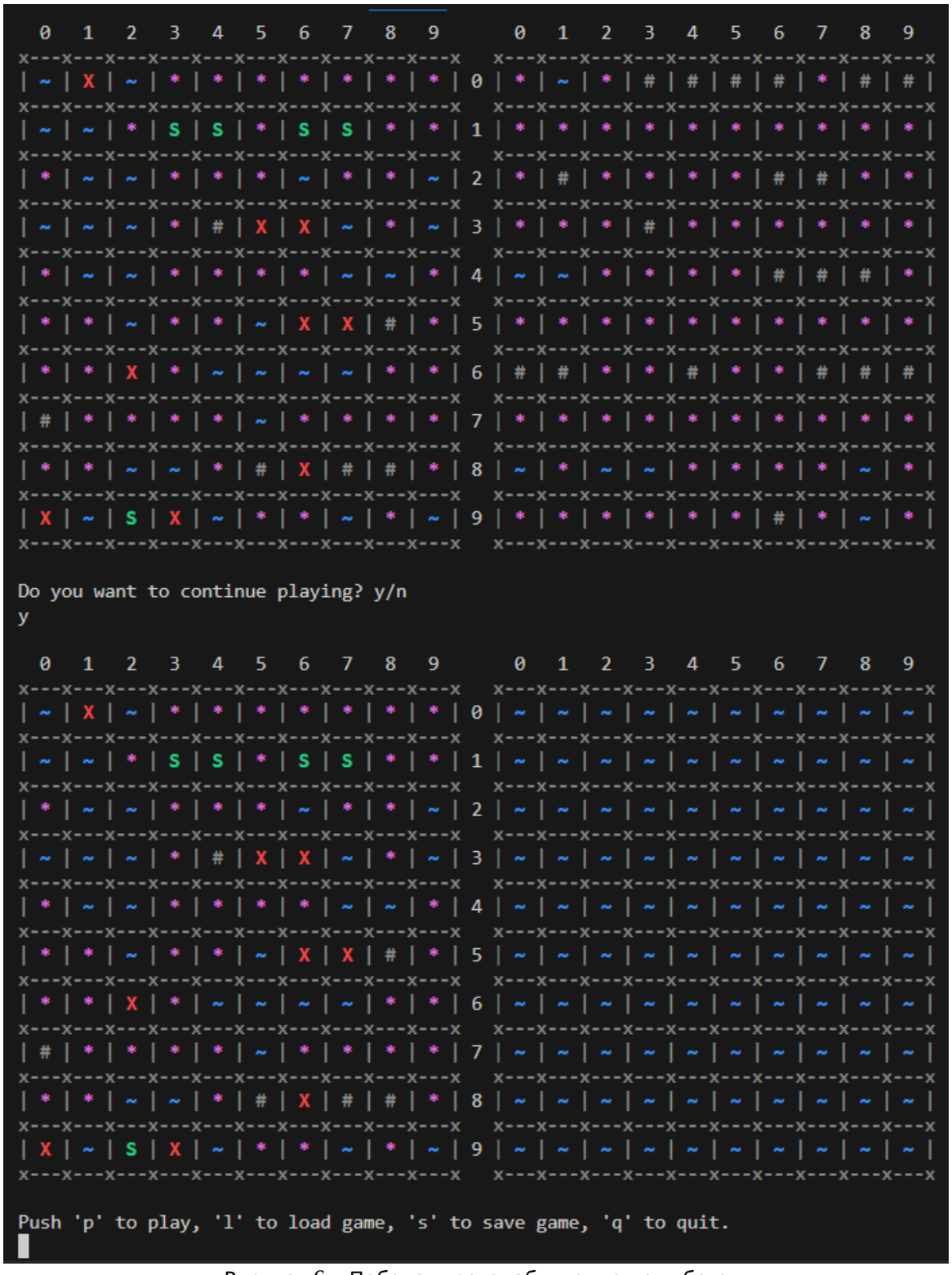

Рисунок 6 – Победа игрока, обнуление поля бота

Выводы

Во время выполнения лабораторной работы, было изучено связывание классов и созданные соответствующие заданию классы.