Faculty of Computing Informatics

CCP -

Trimester 2, 2023/2024

Assignment:

Lecture Section:
Tutorial Section:
Prepared for:

Group Members:

Student ID	Name
1221103634	Wan Abdul Rahman
1221104047	Eleonore Tan
1221103412	Chan Jun Tien
1221103818	Jason Lam Jia Hao

1. explain where did you implement INHERITANCE, POLYMORPHISM, OPERATOR OVERLOADING and any number of C++ object oriented features used.
2. Screen-shots and explanation of your program running compiled into a document in PDF format
3. (modify last used class diagrams), flowcharts, and how OOP features are used in the program in PDF format to explain your work.

introduction
class hierachy
Implementation details
Simulation process
Flowchart

Introduction

The Robot War Simulation program simulates a battlefield where different types of robots engage in combat based on specifications read from a file. This report details the entire simulation process, including initialization, robot deployment, action simulation, and outcome evaluation.

Class Hierarchy and OOP Features Documentation

This hierarchy outlines the relationships between the main classes and their roles in simulating the battlefield and robot interactions.The program consists of several classes, each representing different components of the simulation:

1. **Battlefield**: Represents the battlefield grid where robots are placed and perform actions.
2. **Circular_linked_list**: Manages a circular list of robots for sequential action simulation.
3. **FileReader**: Reads configuration details from a file and initializes simulation parameters.
4. **Node**: Store information related to robots that are read from an input file and processed within the simulation.
5. **Queue**: Implements a queue data structure to manage robots awaiting deployment.
6. **Robot (Base Class)**: Represents a generic robot with basic properties and actions like position, name, and battlefield reference.
7. **Derived Robot Classes (ExplodingRobot, MovingRobot, SeeingRobot, ShootingRobot, SteppingRobtot, Navigatebot, SharafBot, RoboCop, SniperBot,Terminator,TerminatorRoboCop, BlueThunder, Madbot, RoboTank, UltimateRobot)**: Represent specific types of robots with unique behaviors and attributes.
8. **Rocks:** Represents an obstacle on the battlefield that robots need to navigate around.

Implementation Details

1. Battlefield Class

Composition in Battlefield Class:

- The battlefield attribute is a 2D vector representing the grid, which is composed of various elements, including robots and rocks.
- spawnrocks method populates the battlefield with rock objects, demonstrating how the Battlefield class manages the presence of rocks.

Encapsulation in Battlefield Class

- The private members (battlefield_length, battlefield_width, etc.) are not directly accessible from outside the class. Public methods like setbot(), display(), and spawnrocks() provide controlled access to modify and view the battlefield's state.

```
class Battlefield
{
private:
    int battlefield_length;
    int battlefield_width;
    char** battlefield;
    bool** is_rock;
    bool** is_rock_spawn;
    bool** is_rock_cannot_spawn;
    bool rock_setting;

public:
    Battlefield(int x, int y)
    {
        battlefield_length = x;
        battlefield_width = y;
        // initialization and memory allocation
        // ...
    }
    void setbot(int x, int y, char bot_type)
    {
        battlefield[x][y] = bot_type;
    }
    void display()
    {
        for (int i = 0; i < battlefield_length; i++)
        {
            for (int j = 0; j < battlefield_width; j++)
            {
                cout << battlefield[i][j] << " ";
            }
            cout << endl;
        }
    }
    void spawnrocks()
    {
        // method to spawn rocks on the battlefield
        // ...
    }
```

Abstraction in Battlefield Class

In the Battlefield class, abstraction is achieved by providing simple methods to interact with the battlefield, without exposing the underlying implementation details:

```
void setbot(int x, int y, char bot_type);
void display();
void spawnrocks();
void clearfire();
```

Users of the Battlefield class can call these methods to perform actions without needing to know how these actions are implemented internally.

2. Circular_Linked_List Class

```
class circular_linked_list {
private:
    node* head;
    int size;
    int steps;
    Queue q_done;

public:
    circular_linked_list(int turn) : head(nullptr), size(0), steps(turn) {}
    void push(Robot* turn);
    void printList(ofstream &File);
    void cleanbattlefields(Queue& q);
    void bots_fired(Battlefield& A, Robot* turn, ofstream& File);
    void bots_stepped(Robot* turn, ofstream& File);
    bool respawn(Battlefield& A, Queue& q);
    void print_result(ofstream &File);
    void checkdead();
    void upgrade(Robot*& robot, Battlefield& A);
    void spawnsharaf(Battlefield& A);
    void simulate(Battlefield& A, Queue& q);
    void deleteNode(Robot* key);
};
```

Encapsulation in Circular_Linked_List Class

- **Private Node Structure**: The internal structure of the linked list nodes (node) is typically declared as a private nested structure or class within circular_linked_list. This encapsulation hides the node details from external access, maintaining them as internal implementation details.
- **Private Member head**: The head pointer (head) is likely a private member of the class, encapsulating access to the start of the circular linked list.
- **Public Methods**: Methods such as push(), printlist(), cleanbattlefields(), and others encapsulate operations that can be performed on the linked list. These methods control how robots are added, removed, and simulated within the list, ensuring consistency and integrity of the list structure.

3. Node Class

```
class node {
public:
    int r_x;        // X-coordinate of the robot
    int r_y;        // Y-coordinate of the robot
    std::string r_type;   // Type of the robot (e.g., RoboCop, Terminator)
    std::string r_name;   // Name of the robot

    // Constructor to initialize node with robot data
    node(int x, int y, const std::string& type, const std::string& name)
        : r_x(x), r_y(y), r_type(type), r_name(name) {}
};
```

Encapsulation in Node Class

- The node class encapsulates data members such as r_x, r_y, r_type, r_name, next, and robot.
- These members represent the position, type, name of the robot, and pointers to the next node and the robot itself.

Data Abstraction in Node Class

- The node class abstracts the details of how nodes are linked together in the circular linked list. Users of this class do not need to know how the linking is implemented; they only need to understand how to interact with node objects.

Constructors in Node Class

- The node class provides a default constructor to initialize a node with default values.
- It also provides a parameterized constructor to initialize a node with specific values.

4. Queue Class

```
class Queue {
private:
    node* front;
    node* rear;
public:
    Queue() : front(nullptr), rear(nullptr) {}
    void enqueue(node* n);
    void dequeue();
    // Other queue-related methods...
};
```

Encapsulation and Abstraction in Queue Class

The class encapsulates the data (robot details) and the operations (enqueue, dequeue, etc.) that can be performed on the queue. The internal implementation details (like how nodes are linked) are abstracted away from the user of the class.

5. FileReader Class

```
class FileReader {
private:
    // Attributes to store the dimensions of the battlefield,
    // number of steps in the simulation, and the number of robots.
    int height, width, steps, robotCount;
```

Encapsulation in FileReader Class

- The FileReader class encapsulates the behavior of reading a file and initializing the battlefield and robots based on the file contents.
- The attributes height, width, steps, and robotCount are encapsulated within the class.

6. Rock Class

```
class Rock : public Robot {
public:
    Rock(int x, int y, string robot_name, Battlefield& A)
        : Robot(x, y, robot_name, A) {
        this->name = robot_name;
        this->symbol = '#'; // The symbol representing a rock on the battlefield
    }

    void action(Battlefield& A) override {
        // Rocks don't perform any action
    }
};
```

Composition in the Rock Class:

- The Rock class inherits from Robot and thus has a Battlefield& battlefield reference, reinforcing the composition relationship.
- The Rock objects are part of the battlefield grid and managed by the Battlefield class.
- Inheritance: The Rock class inherits from the Robot base class.
- Constructor: The constructor initializes the Rock object and places it on the battlefield.
- action: Overrides the action method from Robot but does nothing because rocks are immobile.

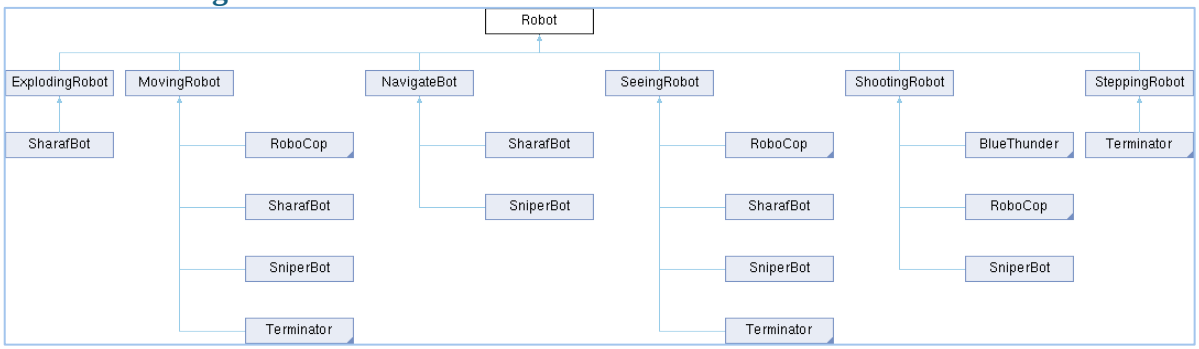
7. Robot Class

Inheritance in the Robot Class

- The Robot class acts as a base class.
- Derived classes like ExplodingRobot, MovingRobot, SeeingRobot, ShootingRobot, NavigateBot and SteppingRobot inherit from the Robot base class. These derived classes reuse the attributes and methods of the Robot class and provide their specific implementations of the action method.

```
class ExplodingRobot : public Robot { /* ... */ };
class MovingRobot: public Robot { /* ... */ };
class SeeingRobot: public Robot { /* ... */ };
class ShootingRobot : public Robot { /* ... */ };
class NavigateBot : public Robot { /* ... */ };
class SteppingRobot : public Robot { /* ... */ };
```

Inheritance diagram for Robot:



Composition in the Robot Class

```
class Robot {
protected:
    int x, y;
    string name;
    char symbol;
    Battlefield& battlefield;

public:
    Robot(int x, int y, string name, Battlefield& A)
        : x(x), y(y), name(name), battlefield(A), symbol('R') {}

    virtual void action(Battlefield& A) = 0;  // Pure virtual function

    // Other methods...
};
```

- Battlefield& battlefield is a reference to the Battlefield object, showing that each Robot object is composed within a specific battlefield context.
- The action method is a pure virtual function, indicating that derived classes must implement their own specific actions.

Encapsulation in the Robot Class

Private Attributes: Attributes such as positionX, positionY, live_count, total_kill_count, and others are typically declared as private members of the class.

```
class Robot {
private:
    int positionX;
    int positionY;
    int live_count;
    int total_kill_count;
    // Other private attributes...
public:
    // Public methods to access and modify these attributes
};
```

- Private attributes are encapsulated within the class, meaning they cannot be directly accessed or modified from outside the class. Instead, access to these attributes is controlled through public methods (getters and setters).

Public Methods (getters and setters): These methods allow controlled access to the private attributes.

```
class Robot {
public:
    int get_positionX() const {
        return positionX;
    }

    void set_positionX(int x) {
        positionX = x;
    }
    // Similar methods for other attributes...
};
```

- Getter methods (get_positionX()) allow retrieving the value of positionX, while setter methods (set_positionX(int x)) allow modifying it. This encapsulation ensures that the internal state of the Robot object is controlled and maintained according to the class's logic.

Abstraction and Polymorphism in the Robot Class and its subclasses (RoboCop, Terminator, etc.)

- Each robot type (RoboCop, Terminator, etc.) inherits from the Robot class, which serves as an abstract base class defining common properties and behaviors for all robots.
- Abstraction is achieved through **polymorphism**, where each subclass implements its specific behavior (action method) while inheriting common functionalities from the base class.

```
class Robot {
public:
    virtual void action(Battlefield& A) = 0; // Abstract method for action
    // Other common methods and attributes
};

class RoboCop : public Robot {
public:
    void action(Battlefield& A) override {
        // Implementation specific to RoboCop's action
    }
    // Specific attributes and methods for RoboCop
};
// Other subclasses like Terminator, TerminatorRoboCop, etc.
```

- In the simulate function of circular_linked_list, each robot's action method is called polymorphically during each turn of the simulation:

```
void simulate(Battlefield& A, Queue& q) {
    // Iterating through each robot in the linked list
    node* botcursor = head;
    do {
        // Calling polymorphic action function
        botcursor->robot->action(A);

        // Other operations specific to simulation
        // ...

        botcursor = botcursor->next;
    } while (botcursor != head);
}
```

Dynamic Binding (Virtual Functions) in the Robot Class

Class Hierarchy

The Robot class is the base class, and Rock is a derived class. The key element here is the action method, which is declared as a virtual function in the Robot class and overridden in the Rock class.

Robot Class Definition

```
class Robot {
public:
    int x;
    int y;
    char symbol;
    string name;

    Robot(int x, int y, string robot_name, Battlefield& A) {
        this->x = x;
        this->y = y;
        this->name = robot_name;
        this->symbol = 'R';
        A.setPosition(x, y, symbol);
    }
    virtual void action(Battlefield& A) {
        // Default action for a robot
    }
    // Other members...
};
```

- The action method is declared as virtual, allowing it to be overridden in derived classes and enabling dynamic binding.

Rock Class Definition

```
class Rock : public Robot {
public:
    Rock(int x, int y, string robot_name, Battlefield& A)
        : Robot(x, y, robot_name, A) {
        this->name = robot_name;
        this->symbol = '#';
    }

    void action(Battlefield& A) override {
        // Rocks don't perform any action
    }
};
```

- In the Rock class, the action method is overridden. Although the method in Rock does nothing, it still demonstrates the concept of overriding a virtual function.

Dynamic Binding in Action

Dynamic binding is employed when the action method is called on a Robot pointer or reference that can point to an object of either Robot or Rock type. For example:

```
void someFunction() {
    Battlefield A;
    Robot* r = new Rock(5, 5, "Static Rock", A); // Base class pointer to derived class
object
    r->action(A); // Dynamic binding: calls Rock::action()
}
```

In this snippet:

- r is a pointer of type Robot* but points to an instance of Rock.
- When r->action(A) is called, the Rock class's action method is executed because of dynamic binding.

Dynamic Binding in Simulation

The action method might be called during the simulation:

```
void simulate(Battlefield& A, queue<Robot*>& robot_q) {
    while (!robot_q.empty()) {
        Robot* current_robot = robot_q.front();
        robot_q.pop();

        current_robot->action(A); // Dynamic binding: correct action method called
based on object type
        robot_q.push(current_robot);
    }
}
```

In the simulate method:

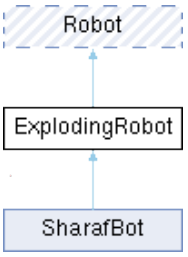
- current_robot is a pointer to a Robot object, but it could point to any derived class object, such as Rock.
- When current_robot->action(A) is invoked, the appropriate method for the actual object type (determined at runtime) is executed, demonstrating dynamic binding.

8. Derived Robot Classes Inherited from Robot

These derived classes override the action method from the Robot base class to implement specific behaviors for each type of robot. The move_a_step method is also overridden where specific movement logic is required. Each derived class constructor initializes the robot with specific attributes like its symbol, health, and attack power.

1. ExplodingRobot

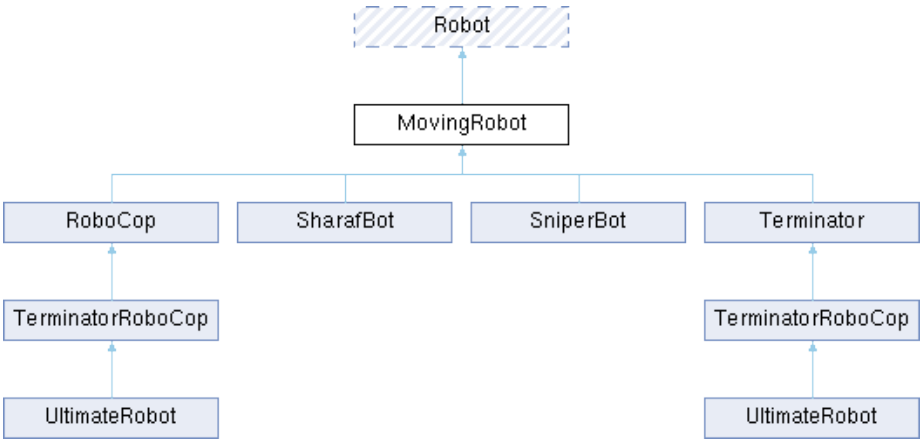
```
class ExplodingRobot : virtual public Robot{
    public:
    virtual void explode(Battlefield& A)=0;
    int radius_e = 2;
};
```



2. MovingRobot

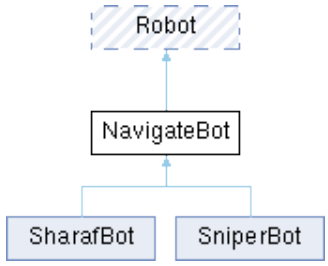
```
class MovingRobot : virtual public Robot{
```

Inheritance diagram for MovingRobot:



3. NavigateBot

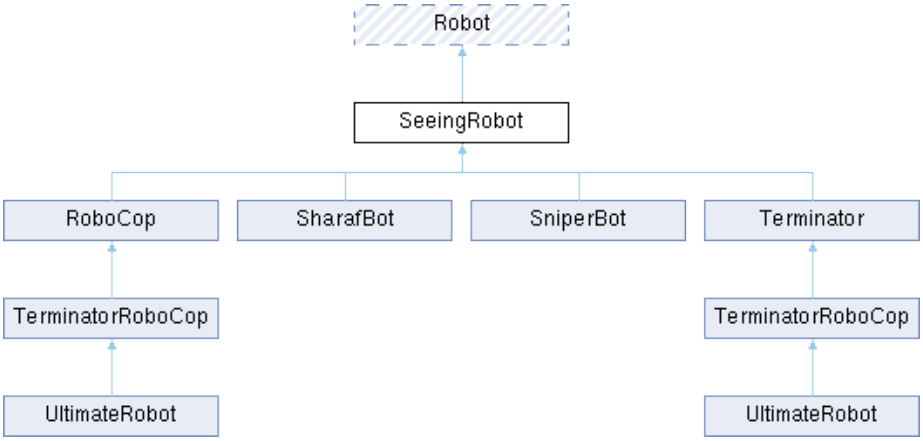
```
class NavigateBot : virtual public Robot{
```



4. SeeingRobot

```
class SeeingRobot : virtual public Robot{
```

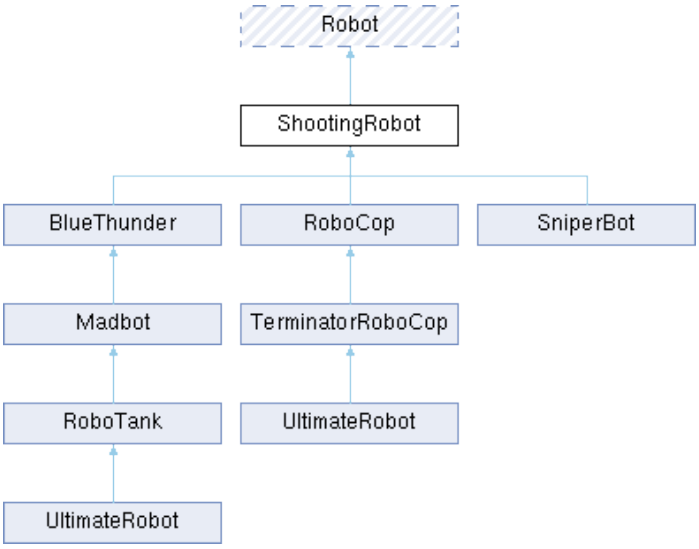
Inheritance diagram for SeeingRobot:



5. ShootingRobot

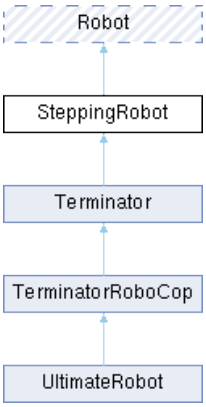
```
class ShootingRobot : virtual public Robot{
```

Inheritance diagram for SeeingRobot:



6. SteppingRobot

```
class SteppingRobot : virtual public Robot{
```

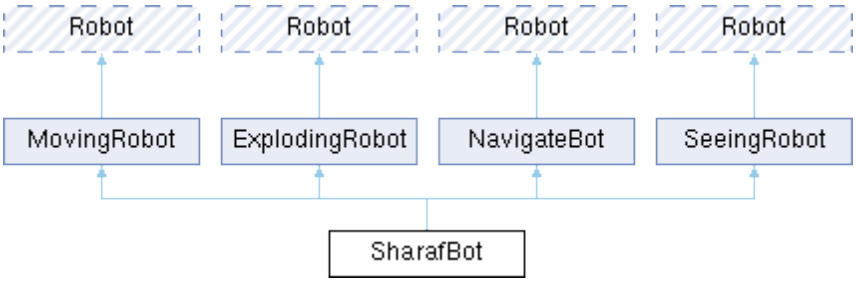


9. Derived Robot Classes from ActionRobots

1. SharafBot

```
class SharafBot: public MovingRobot,public ExplodingRobot,public
NavigateBot,public SeeingRobot{
```

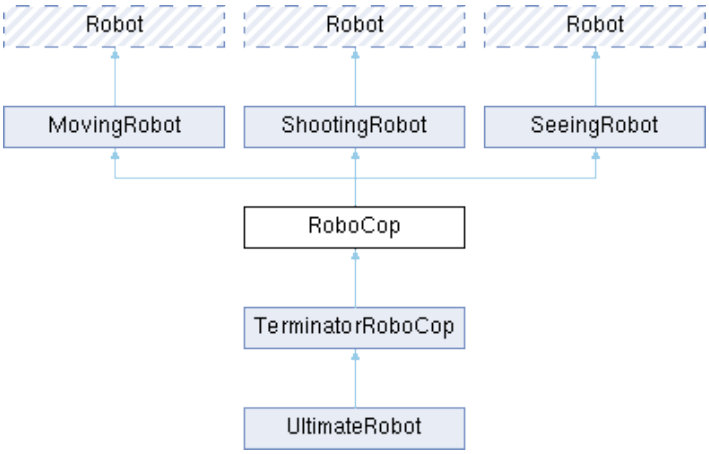
Inheritance diagram for SeeingRobot:



2. RoboCop

```
class RoboCop :public MovingRobot, public ShootingRobot, public SeeingRobot{
```

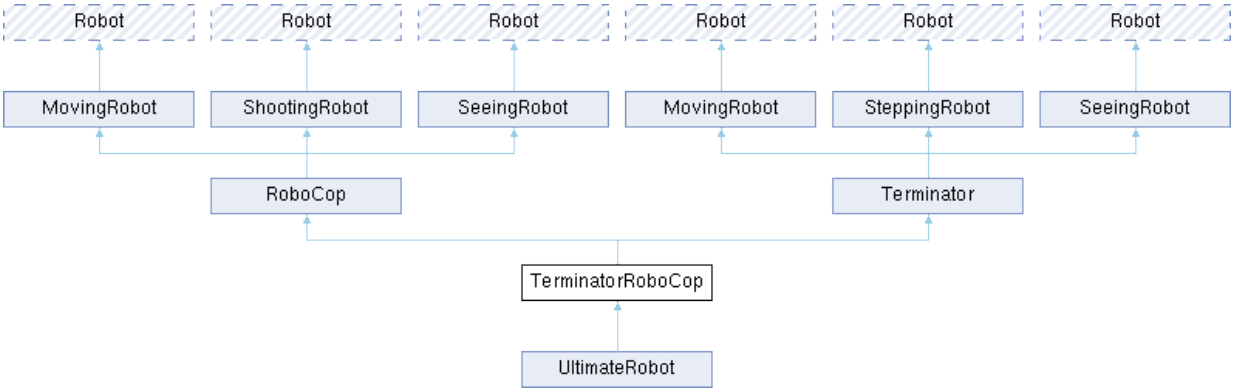
Inheritance diagram for Robocop:



3. TerminatorRoboCop

```
class TerminatorRoboCop : public RoboCop, public Terminator{
```

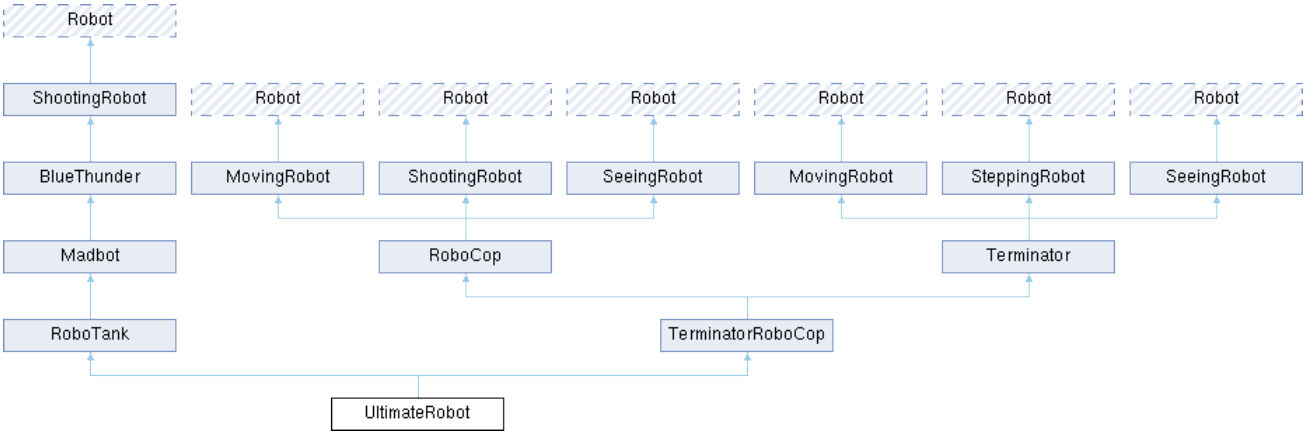
Inheritance diagram for TerminatorRoboCop:



4. UltimateRobot

```
class UltimateRobot : public RoboTank, public TerminatorRoboCop{//The Strongest
bot there is (sniper bot is stronger)
```

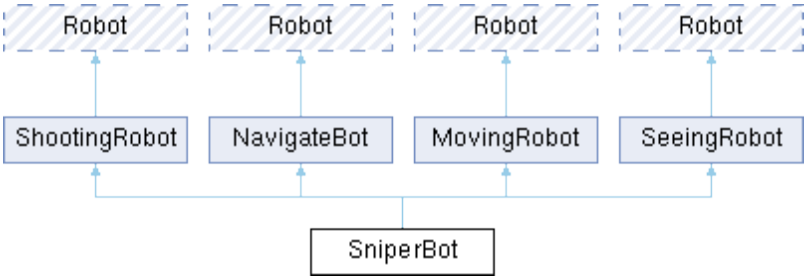
Inheritance diagram for UltimateRobot:



5. SniperBot

```
class SniperBot : public ShootingRobot, public NavigateBot, public MovingRobot,
public SeeingRobot
```

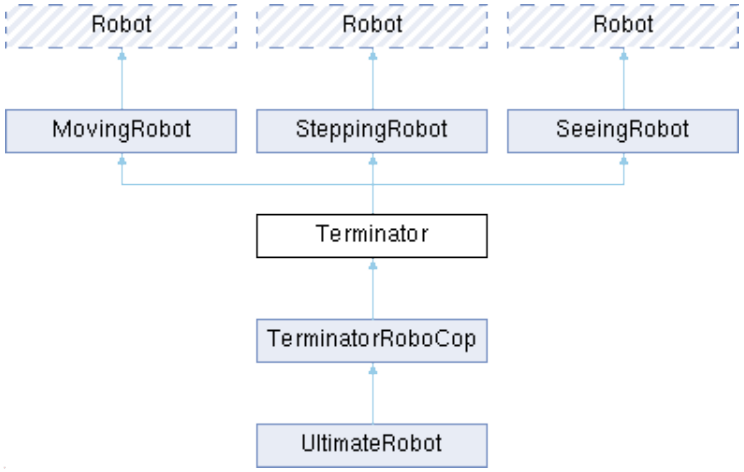
Inheritance diagram for Sniperbot:



6. Terminator

```
class Terminator : public MovingRobot, public SteppingRobot, public SeeingRobot{
```

Inheritance diagram for Termnator:



7. BlueThunder

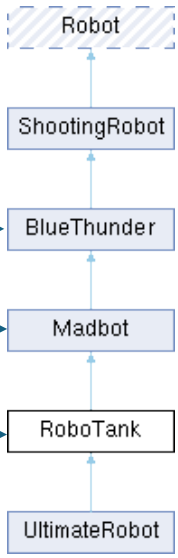
```
class BlueThunder : public ShootingRobot{//Blue thunder can't move
```

8. Matbot

```
class Madbot : public BlueThunder{
```

9. RoboTank

```
class RoboTank : public Madbot{
```



Main Function

```
int main() {
    srand(time(0));
    Queue robot_q;
    Queue robot_list;

    FileReader F;
    F.InitializeFile("robot.txt", robot_list);
    Battlefield A(F.width, F.height);
    circular_linked_list cll(F.steps);

    for (int i = 0; i < F.robotCount; i++) {
        Robot *spawn_robot;
        if (robot_list.front->r_type == "RoboCop") {
            spawn_robot = new RoboCop(robot_list.front->r_x, robot_list.front->r_y,
robot_list.front->r_name, A);
        } else if (robot_list.front->r_type == "Terminator") {
            spawn_robot = new Terminator(robot_list.front->r_x, robot_list.front->r_y,
robot_list.front->r_name, A);
        } else if (robot_list.front->r_type == "TerminatorRobocop") {
            spawn_robot = new TerminatorRoboCop(robot_list.front->r_x,
robot_list.front->r_y, robot_list.front->r_name, A);
        } else if (robot_list.front->r_type == "BlueThunder") {
            spawn_robot = new BlueThunder(robot_list.front->r_x, robot_list.front->r_y,
robot_list.front->r_name, A);
        } else if (robot_list.front->r_type == "Madbot") {
            spawn_robot = new Madbot(robot_list.front->r_x, robot_list.front->r_y,
robot_list.front->r_name, A);
        } else if (robot_list.front->r_type == "RoboTank") {
            spawn_robot = new RoboTank(robot_list.front->r_x, robot_list.front->r_y,
robot_list.front->r_name, A);
        } else if (robot_list.front->r_type == "UltimateRobot") {
            spawn_robot = new UltimateRobot(robot_list.front->r_x, robot_list.front-
>r_y, robot_list.front->r_name, A);
        } else {
            cout << robot_list.front->r_type << " is not a valid robot type" << endl;
        }
        cll.push(spawn_robot);
        robot_list.dequeue();
    }

    cout << "Press To Start Simulation" << endl;
    cin.get();

    cll.simulate(A, robot_q);

    return 0;
}
```

Simulation Process

1. Initialization

FileReader Initialization

The simulation begins with the `FileReader` class reading the specifications from a file (`robot.txt`). The `InitializeFile`method parses the file to extract information about the battlefield dimensions, number of simulation steps, number of robots, and details of each robot to be deployed.

```
FileReader F;
F.InitializeFile("robot.txt", robot_list);
```

- **File Structure**: The file format includes lines specifying the battlefield dimensions (M by), the number of simulation steps (steps:), the number of robots (robots:), followed by lines for each robot detailing its type, name, and initial position.

Battlefield Initialization

After reading the file, the Battlefield object `A` is initialized using the dimensions (width and height) obtained from FileReader. The battlefield serves as the grid where robots will move and interact during the simulation.

```
Battlefield A(F.width, F.height);
```

2. Robot Deployment

Queue and Circular Linked List

Robots read from the file are enqueued into robot_list, a Queue data structure. Each robot type (RoboCop, Terminator, etc.) is instantiated based on its specifications and added to a circular_linked_list for sequential action simulation.

```
for (int i = 0; i < F.robotCount; i++) {
    Robot *spawn_robot;
    // Determine robot type and create corresponding object
    if (robot_list.front->r_type == "RoboCop") {
        spawn_robot = new RoboCop(robot_list.front->r_x, robot_list.front-
>r_y, robot_list.front->r_name, A);
    } else if (robot_list.front->r_type == "Terminator") {
        spawn_robot = new Terminator(robot_list.front->r_x, robot_list.front-
>r_y, robot_list.front->r_name, A);
    } else if (robot_list.front->r_type == "TerminatorRobocop") {
        spawn_robot = new TerminatorRoboCop(robot_list.front->r_x,
robot_list.front->r_y, robot_list.front->r_name, A);
    } // Repeat for other robot types
    cll.push(spawn_robot);  // Add robot to circular linked list for
simulation
    robot_list.dequeue();   // Remove robot from queue after processing
}
```

3. Simulation Execution

Circular Linked List Simulation

The circular_linked_list manages the sequence of robot actions. Each robot in the list is processed in a circular manner for a specified number of simulation steps (F.steps). The simulate method invokes each robot's action method, simulating their behavior on the battlefield.

```
cll.simulate(A, robot_q);
```

- **Robot Actions**: During each simulation step, robots execute their specific actions based on their type. For example, RoboCop may patrol or defend, while Terminator may seek and destroy.

```
void circular_linked_list::simulate(Battlefield &field, Queue &queue) {
    ListNode *current = head;
    for (int i = 0; i < stepCount; ++i) {
        current->robot->action(field);
        current = current->next;
    }
}
```

Battlefield Display

After the simulation completes, the state of the battlefield can be displayed to visualize the outcome of robot actions. This helps evaluate the effectiveness and interactions of different robot types during the simulation.

```
A.display();
```

- **Visualization**: The display method of Battlefield can show the current positions and statuses of all robots on the grid, reflecting any changes resulting from their actions.

Flowchart

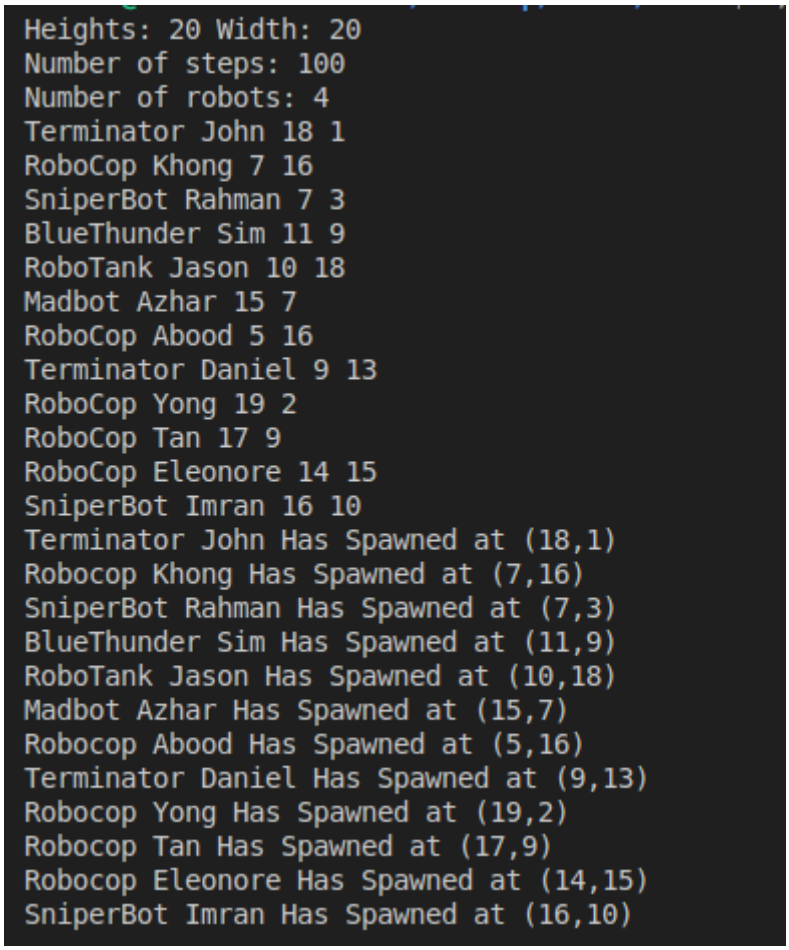
Conclusion

The Robot War Simulation program effectively demonstrates the principles of Object-Oriented Programming and efficient data structure usage for simulating dynamic scenarios. By initializing, deploying, simulating actions, and evaluating outcomes, the program provides a comprehensive example of how OOP can be applied to model complex systems like robot warfare. This report highlights the structured approach to simulation design and execution, emphasizing modularity, extensibility, and real-time interaction within a controlled environment.

- Attached source code and class diagram

Screenshot and explanation of the Main.cpp

The program will print out the information in a txt file. And to start simulation, you need to click a button on your keyboard to start.

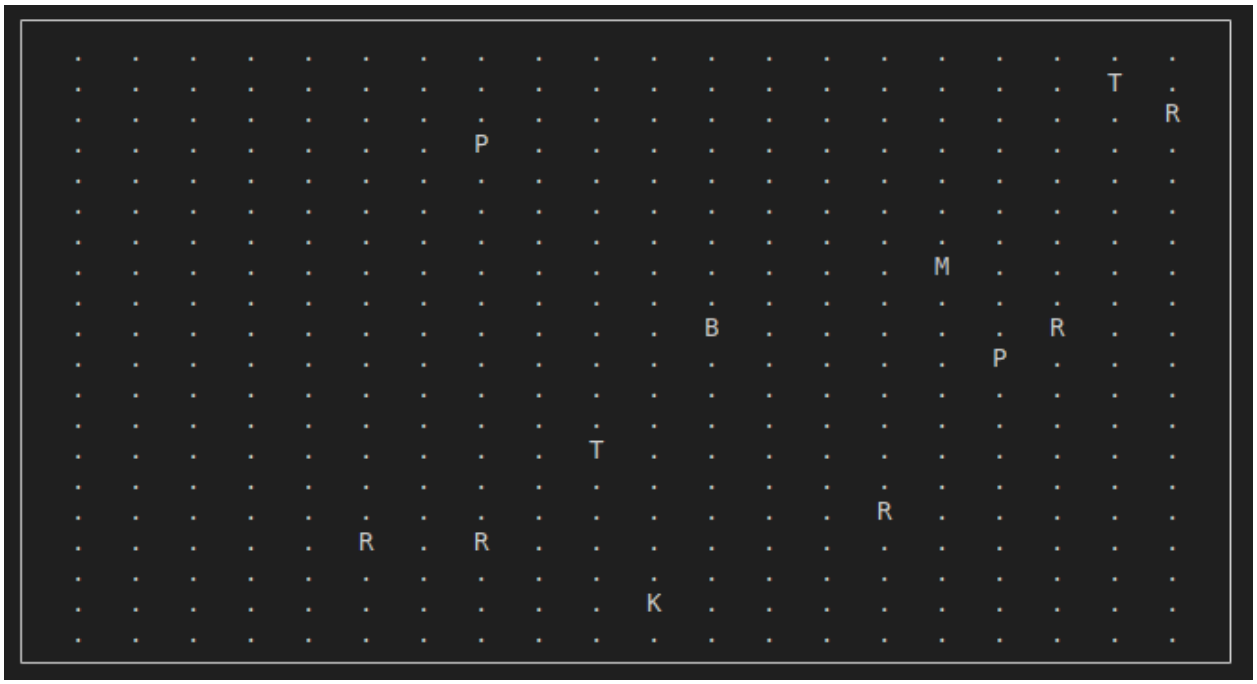


The program will also have 3 different modes to play. Which is Normal, Play with Rocks and Play with Rocks and Rock Spawning. It will have rocks among the battlefield if you choose 2 & 3, and in the Play with Rocks and Rock Spawning, the rocks will spawn randomly.

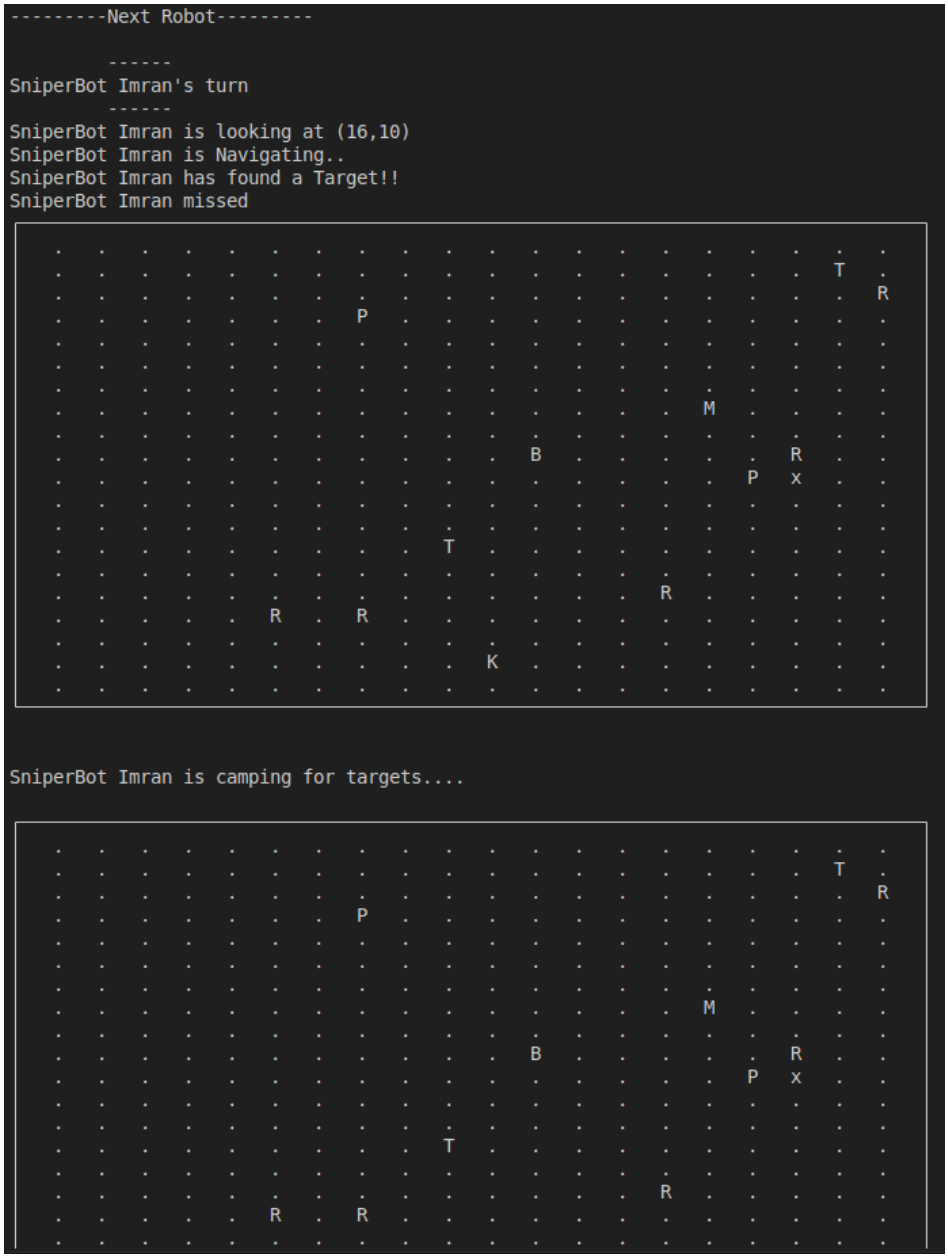
Then after spawn robots, the program will print all robots currently in the battlefield.

```
=========TURN 1=========

Current Robot in Battlefield:
  1: SniperBot Imran
   (16,10) life: 3 Total Kills: 0
  2: Robocop Eleonore
   (14,15) life: 3 Total Kills: 0
  3: Robocop Tan
   (17,9) life: 3 Total Kills: 0
  4: Robocop Yong
   (19,2) life: 3 Total Kills: 0
  5: Terminator Daniel
   (9,13) life: 3 Total Kills: 0
  6: Robocop Abood
   (5,16) life: 3 Total Kills: 0
  7: Madbot Azhar
   (15,7) life: 3 Total Kills: 0
  8: RoboTank Jason
   (10,18) life: 3 Total Kills: 0
  9: BlueThunder Sim
   (11,9) life: 3 Total Kills: 0
 10: SniperBot Rahman
   (7,3) life: 3 Total Kills: 0
 11: Robocop Khong
   (7,16) life: 3 Total Kills: 0
 12: Terminator John
   (18,1) life: 3 Total Kills: 0
```

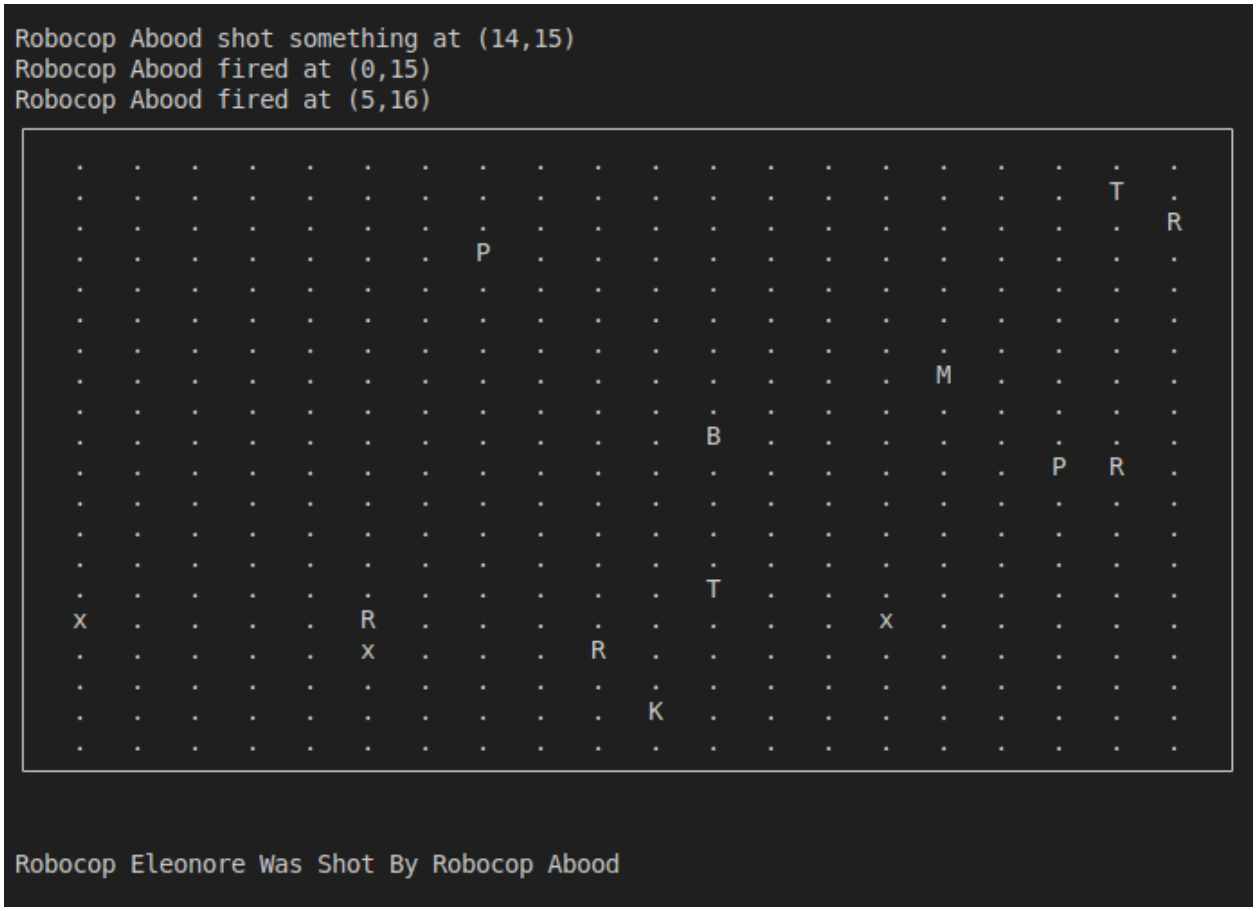
Then after spawn robots, the program will create a view of battlefield.



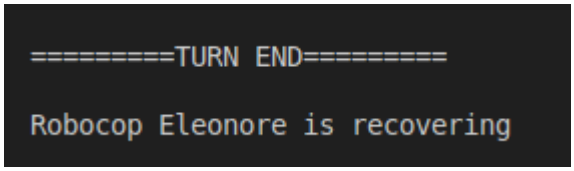
Then the robot will start working automatically, it will look randomly and move randomly.



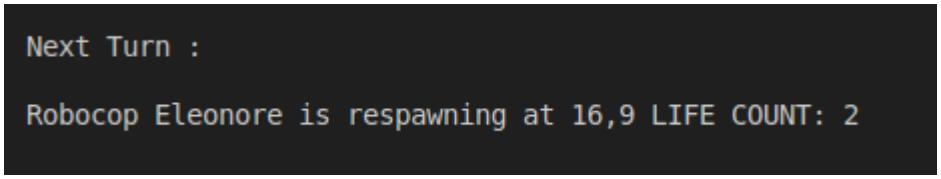
The Robot can detect if it shoots a target.
The Battlefield will highlight if the bot is killed, and the bot will be set to dead.



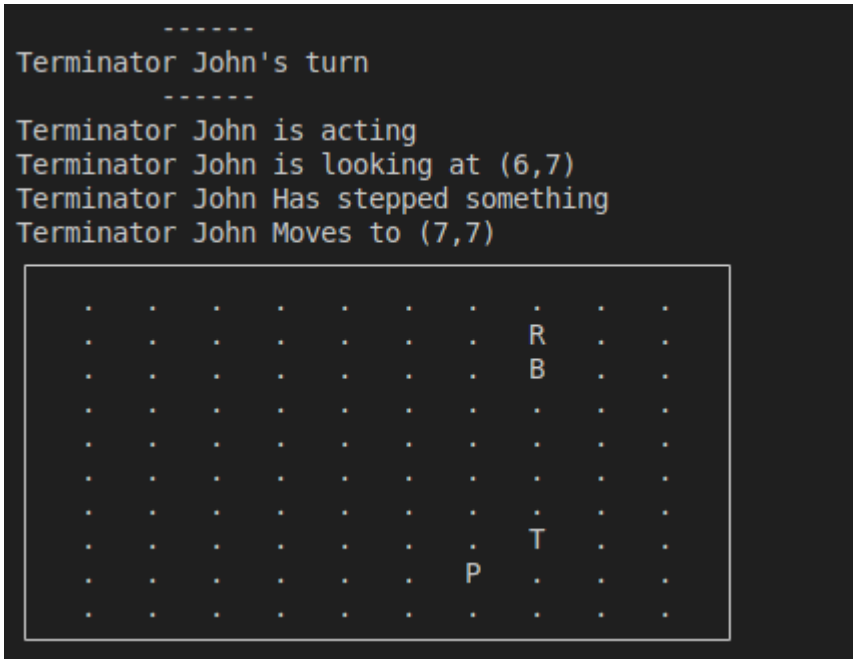
Recovering means put died robot into queue, and it will respawn back to battlefield next turn.



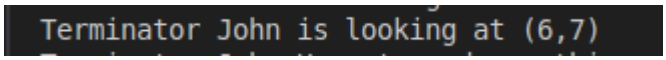
Respawns and dequeue, when it respawns, it will show the life count and where it is at.



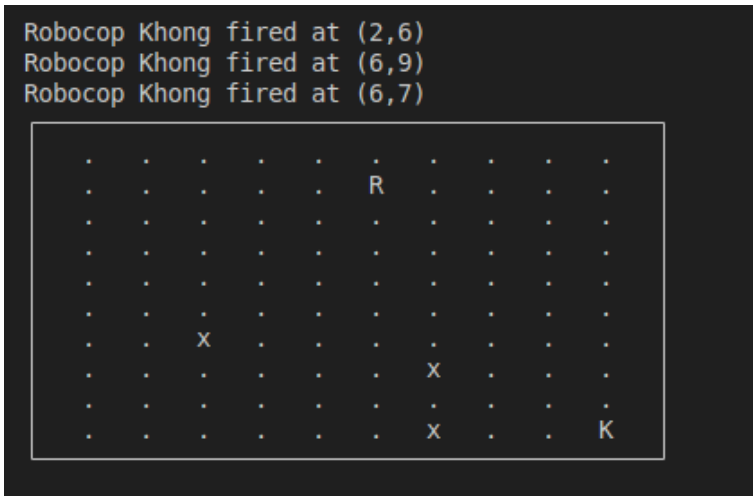
Checking Step



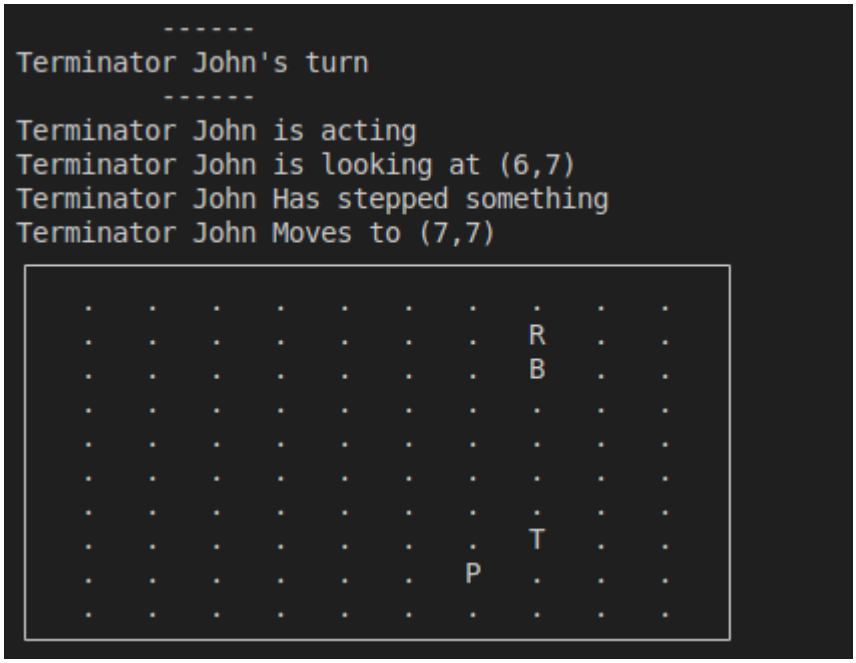
LOOK input coordinate into 2-dimensional array



Bots can fire and will highlight the battlefield with x



Stepping



Terminator John's turn

Terminator John is acting
Terminator John is looking at (6,7)
Terminator John Has stepped something
Terminator John Moves to (7,7)
R
B
T
P

Bots will upgrade after 3 kills (Robocop->TerminatorRoboCop)

```
---------Next Robot---------

         ------
Robocop Khong's turn
         ------
Robocop Khong is acting
Robocop Khong is looking at (0,2)
Robocop Khong Moves to (0,1)
```

```
   .   .   .   .
   R   .   .   .
   .   P   .   .
   .   .   .   .
```

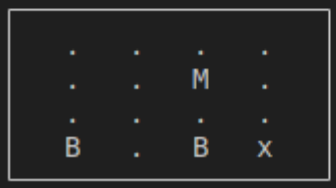
```
Robocop Khong fired at (2,2)
Robocop Khong shot something at (1,2)
Robocop Khong fired at (3,1)
```

```
   .   .   .   .
   R   .   .   x
   .   x   x   .
   .   .   .   .
```

```
SniperBot Jason Was Shot By Robocop Khong

WHAT?
Robocop 0 3
Robocop Khong is evolving!

Robocop Khong is evolving....
1
TerminatorRobocop Khong Has Spawned at (0,1)
```

(BlueThunder->Madbot)

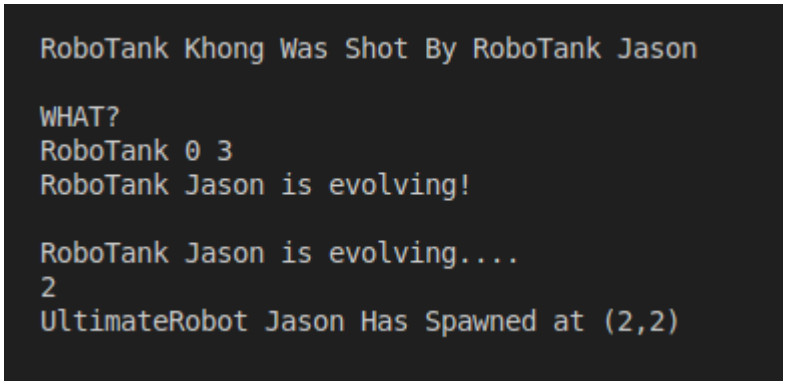
```
BlueThunder Khong Was Shot By BlueThunder Jason

WHAT?
BlueThunder 0 3
BlueThunder Jason is evolving!

BlueThunder Jason is evolving....
3
Madbot Jason Has Spawned at (2,1)


---------Next Robot---------

         ------
BlueThunder Jason's turn
         ------
BlueThunder Jason is acting
BlueThunder Jason fired at (3,3)
```



(Madbot->RoboTank)

```
Madbot Jason Was Shot By Madbot Jason

WHAT?
Madbot 0 3
Madbot Jason is evolving!

Madbot Jason is evolving....
4
RoboTank Jason Has Spawned at (3,2)
```

RoboTank->UltimateRobot

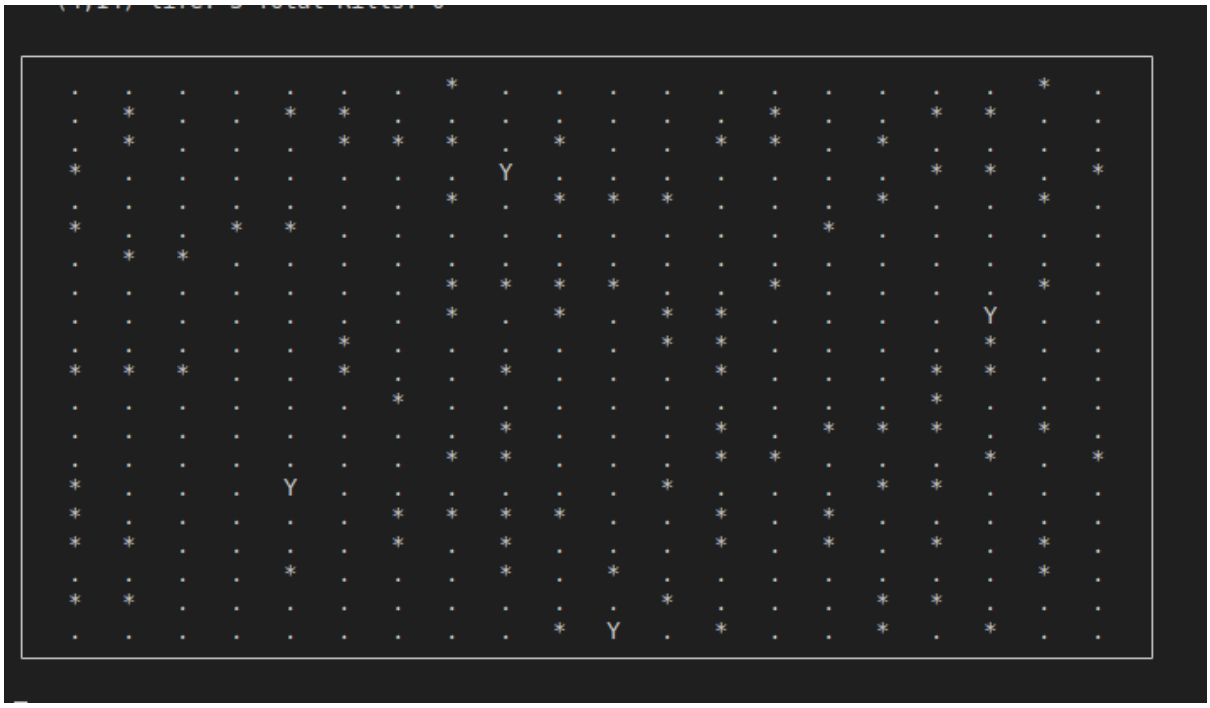


TerminatorRobocop->UltimateRobot

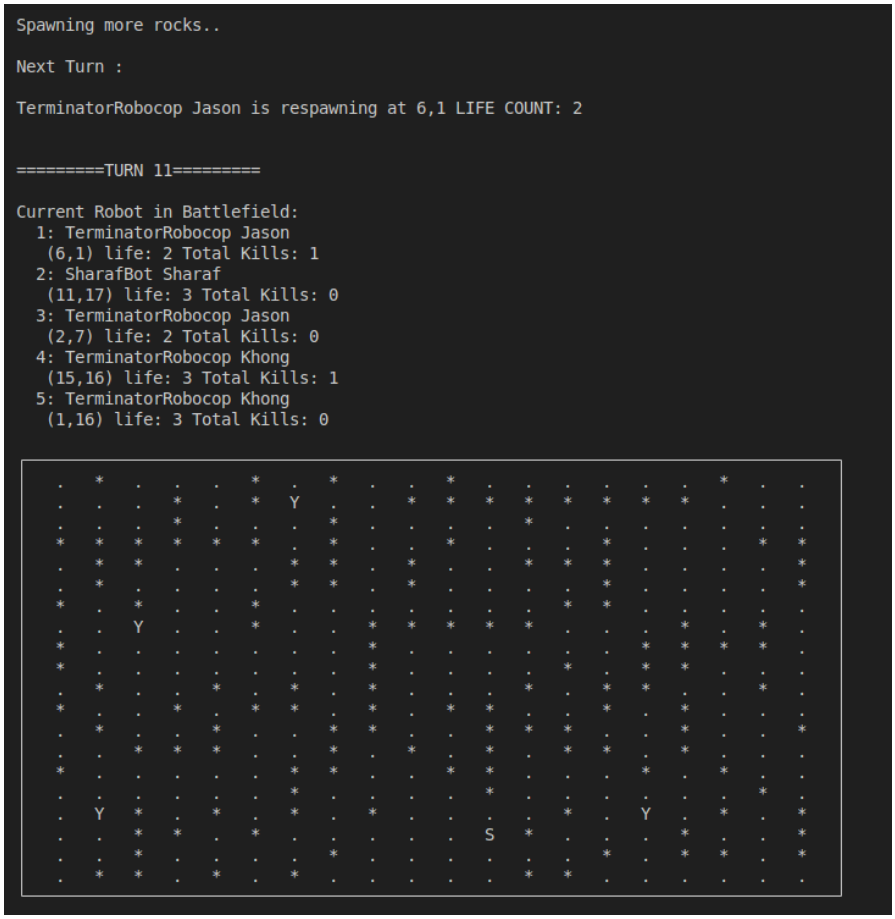
Robot will win if no more bots in battlefield and respawn queue

```
Next Turn :

no bots respawning ..

TerminatorRobocop Khong HAS WON THE GAME


==========ROBOT WAR SIMULATION HAS ENDED=========

==========TURN 17 =========


=========RESULT OF SIMULATION========



=========ROBOT IN BATTLEFIELD========

  0: TerminatorRobocop Khong
   (0,1) life: 3 Total Kills: 3
    Total Step Travelled 16
```

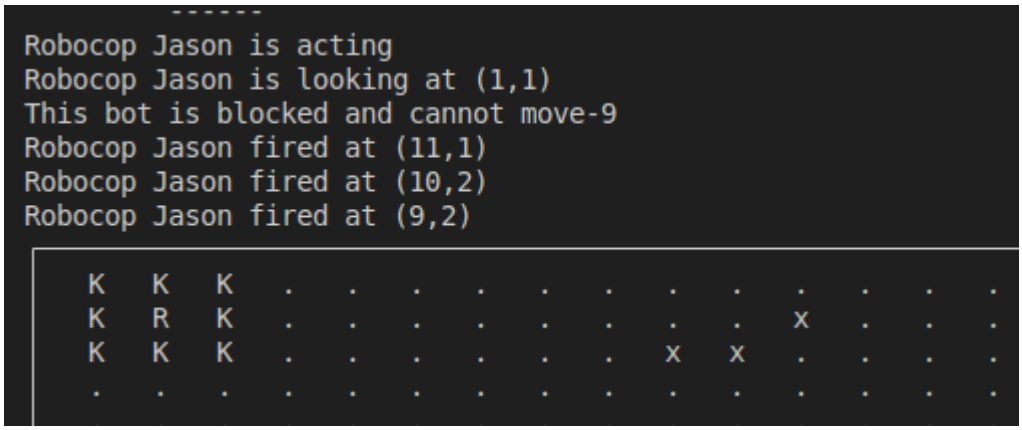
Battlefield filled with rocks(2 and 3 mode(rock mode))



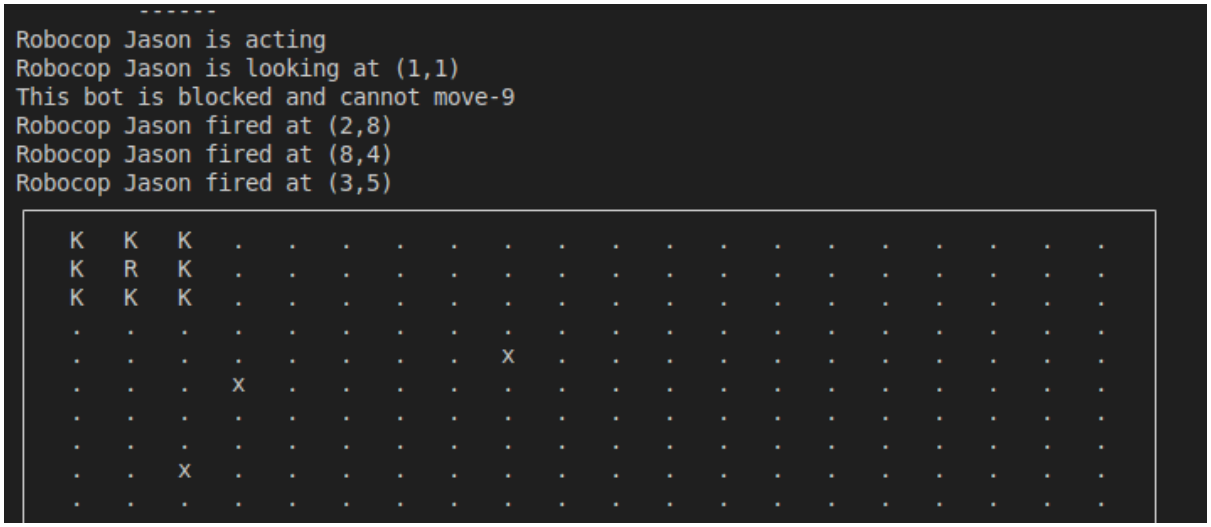
Every 10 turn spawn more rock with limit



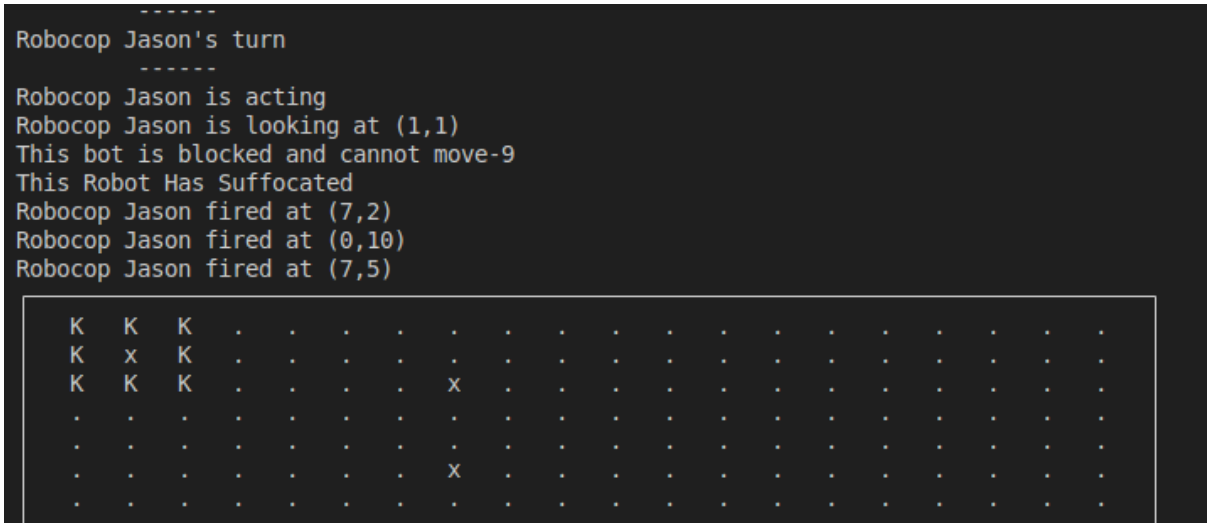
With the addition of rocks robot are more likely to get blocked



Units can be blocked

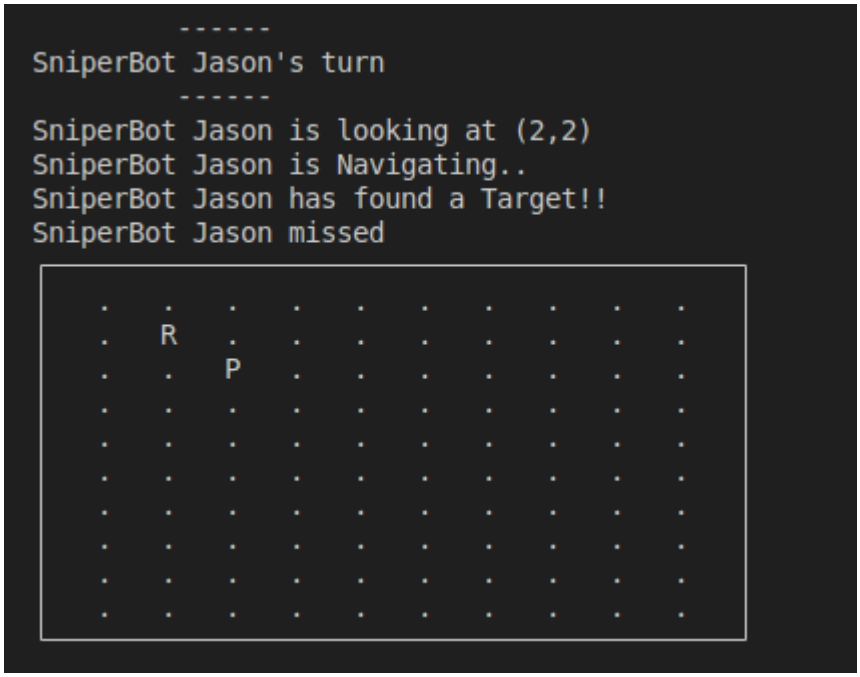


If a robot is blocked for more than 3 turns it will die and speed up game

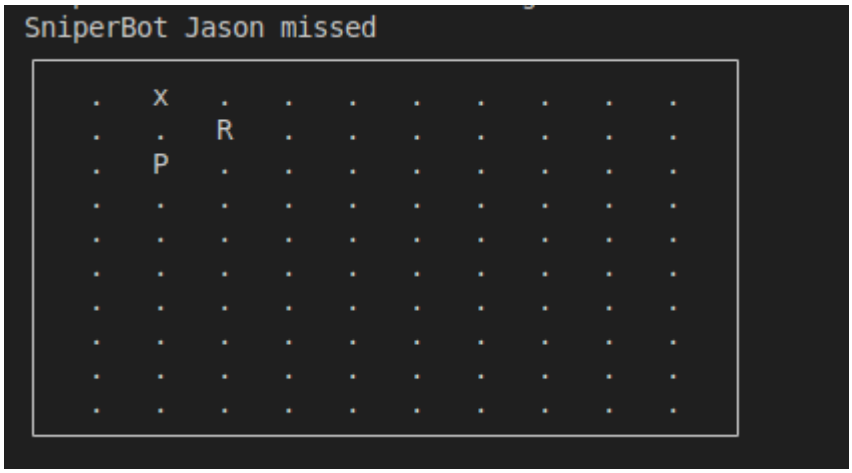


Introducing Sniper Bot

Can navigate a target

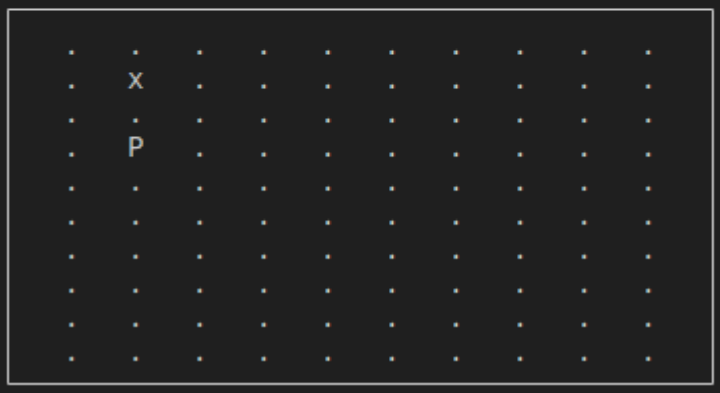


The robot can miss and shoot neighboring squares



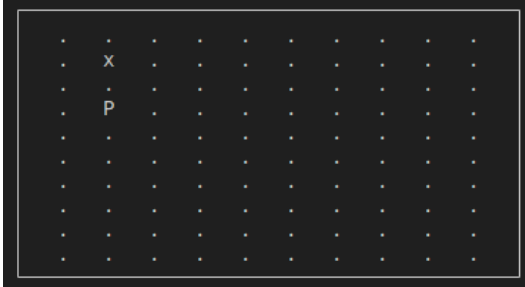
Sniper can snipe bots

```
---------Next Robot---------

         ------
SniperBot Jason's turn
         ------
SniperBot Jason is looking at (1,3)
SniperBot Jason is Navigating..
SniperBot Jason has found a Target!!
SniperBot Jason sniped a target with accuracy!!
```



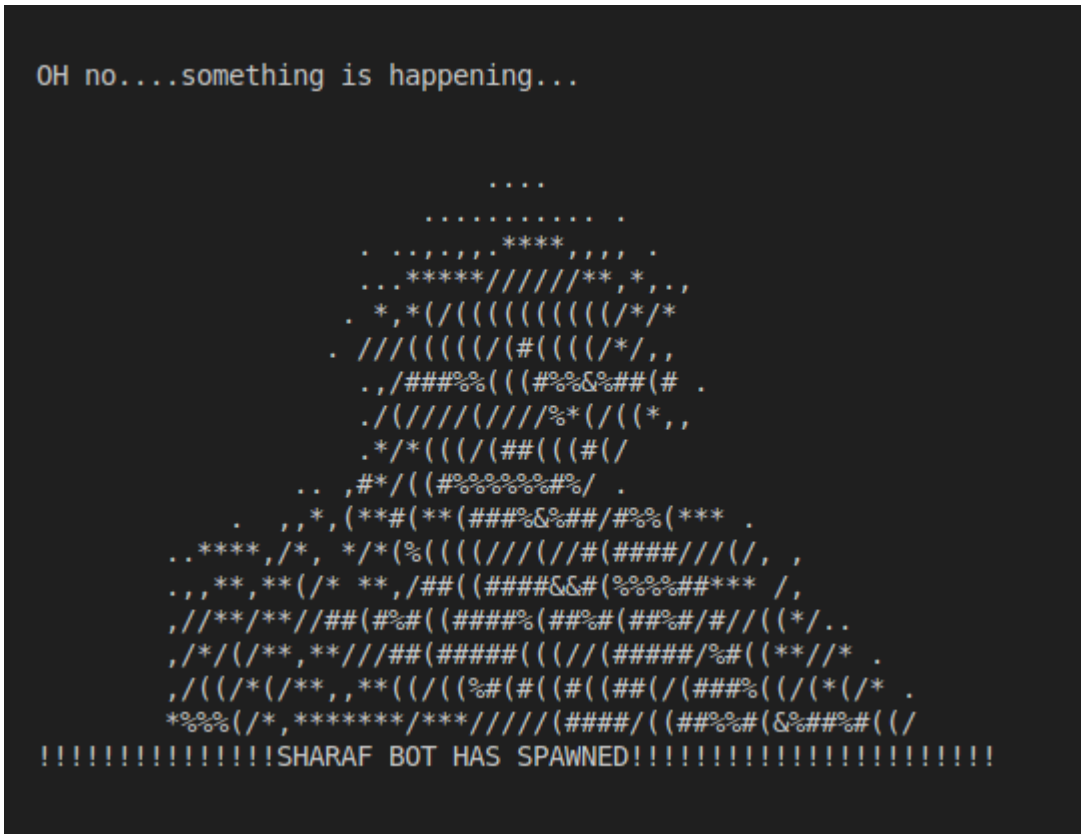
After a successful snipe he doesn’t want to move

```
SniperBot Jason feels happy and doesn't want to move
```

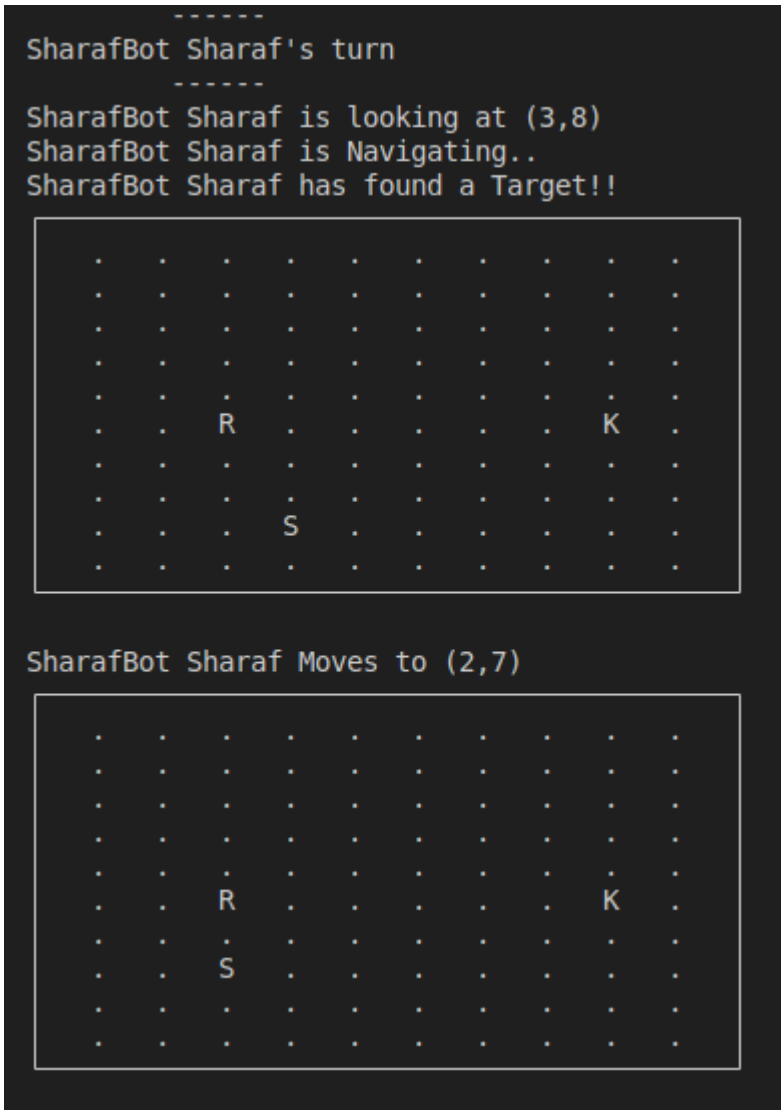


INTRODUCING SHARAFBOT

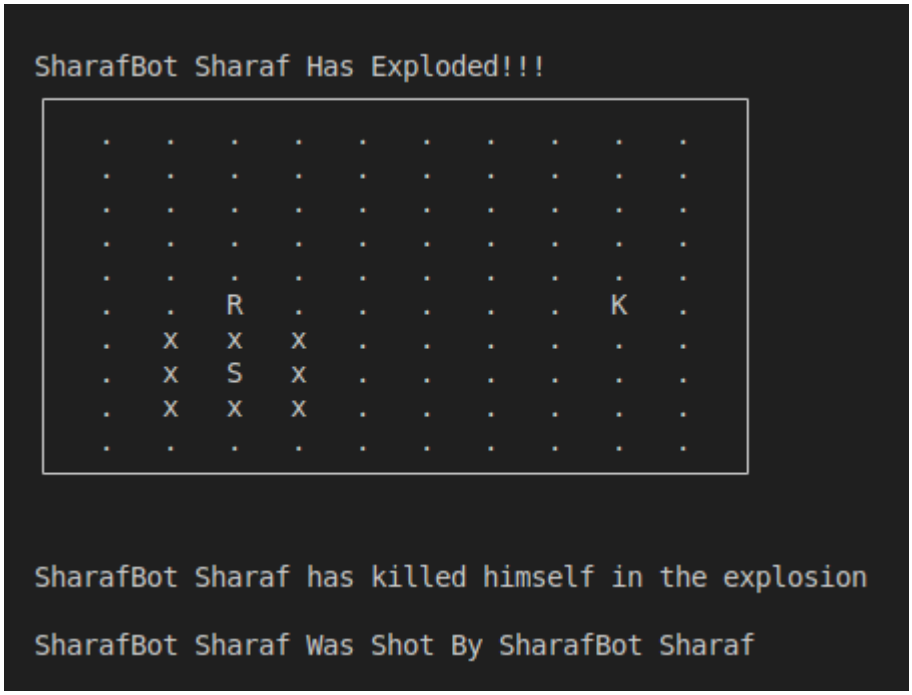
The program will spawn a special robot called The SharafBot. The SharafBot cannot spawn by itself but has a 1/100 chance to spawn in a simulation. And it's made to speed up the unimaginably long and dragged-out nature of the game.



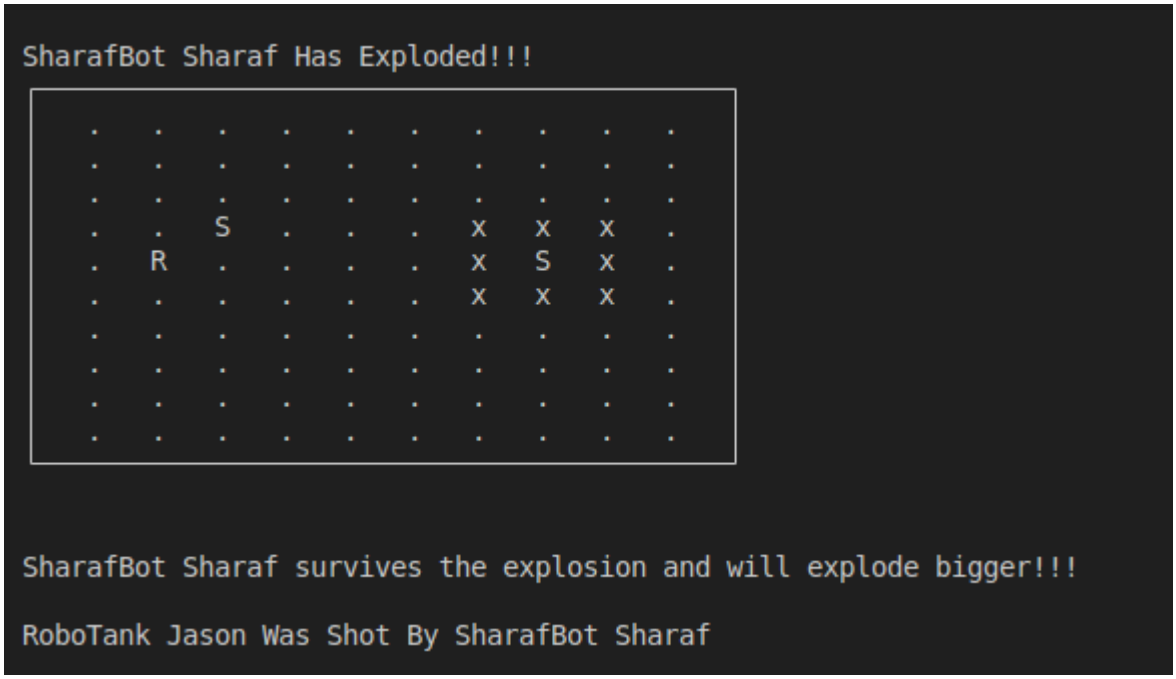
SharafBot can navigate a target then move to the square nearest to target



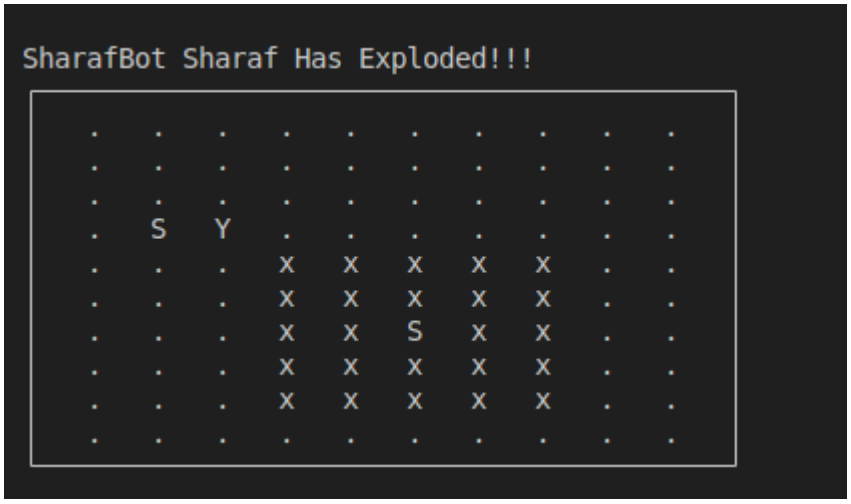
SharafBot Explodes and has a chance to die



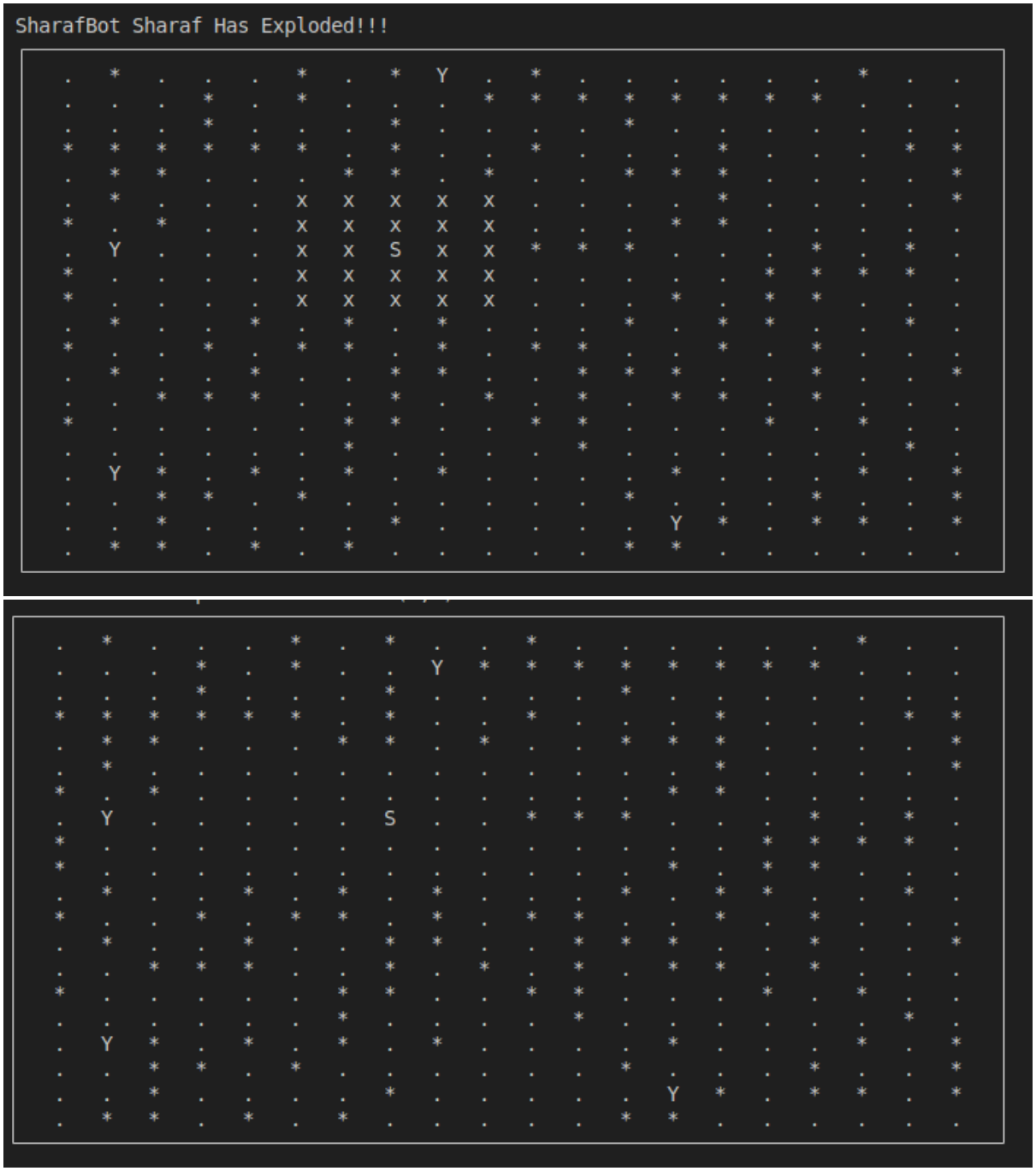
If it doesn't die it will explode in a larger radius



The explosion increases

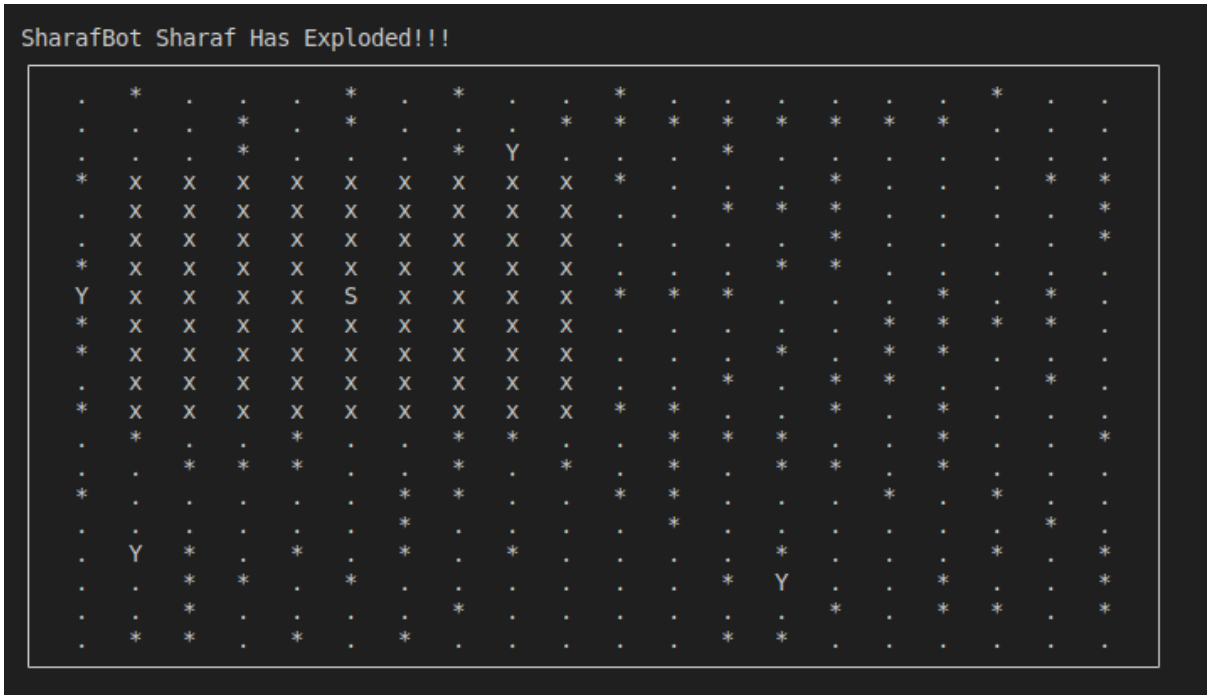


Sharaf bot can explode rocks

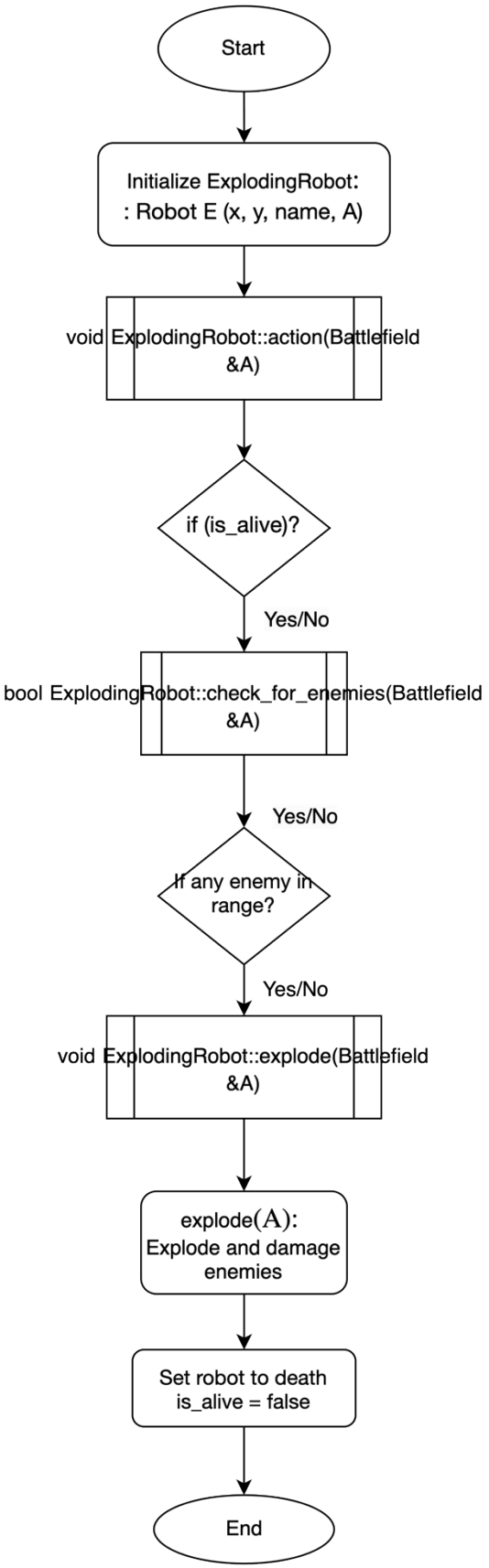


Sharaf bot is very dangerous

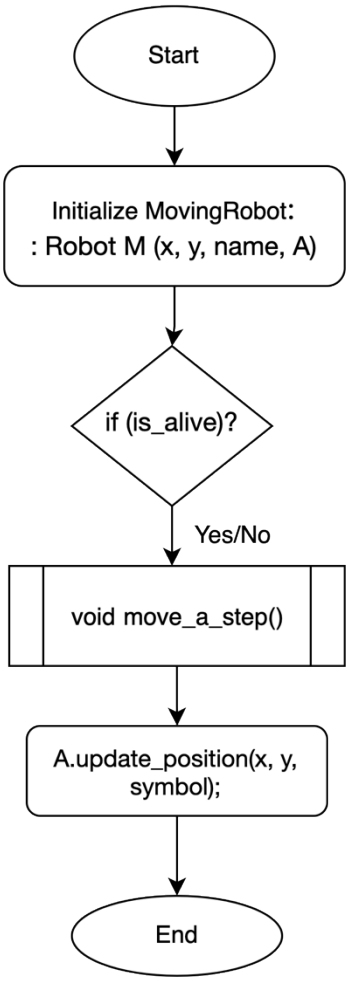
Explode radius gets bigger if the SharafBot survive multiple time when it’s almost exploded



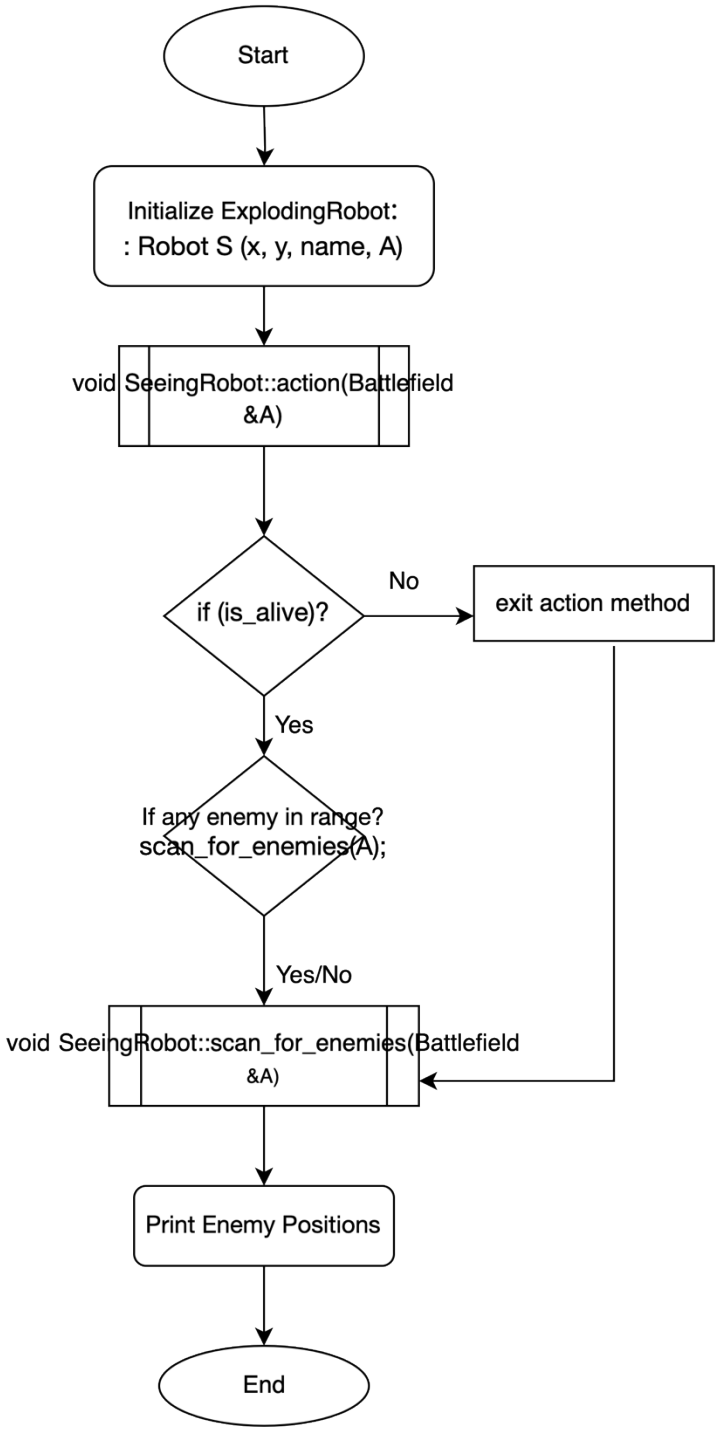
`ExplodingRobot`



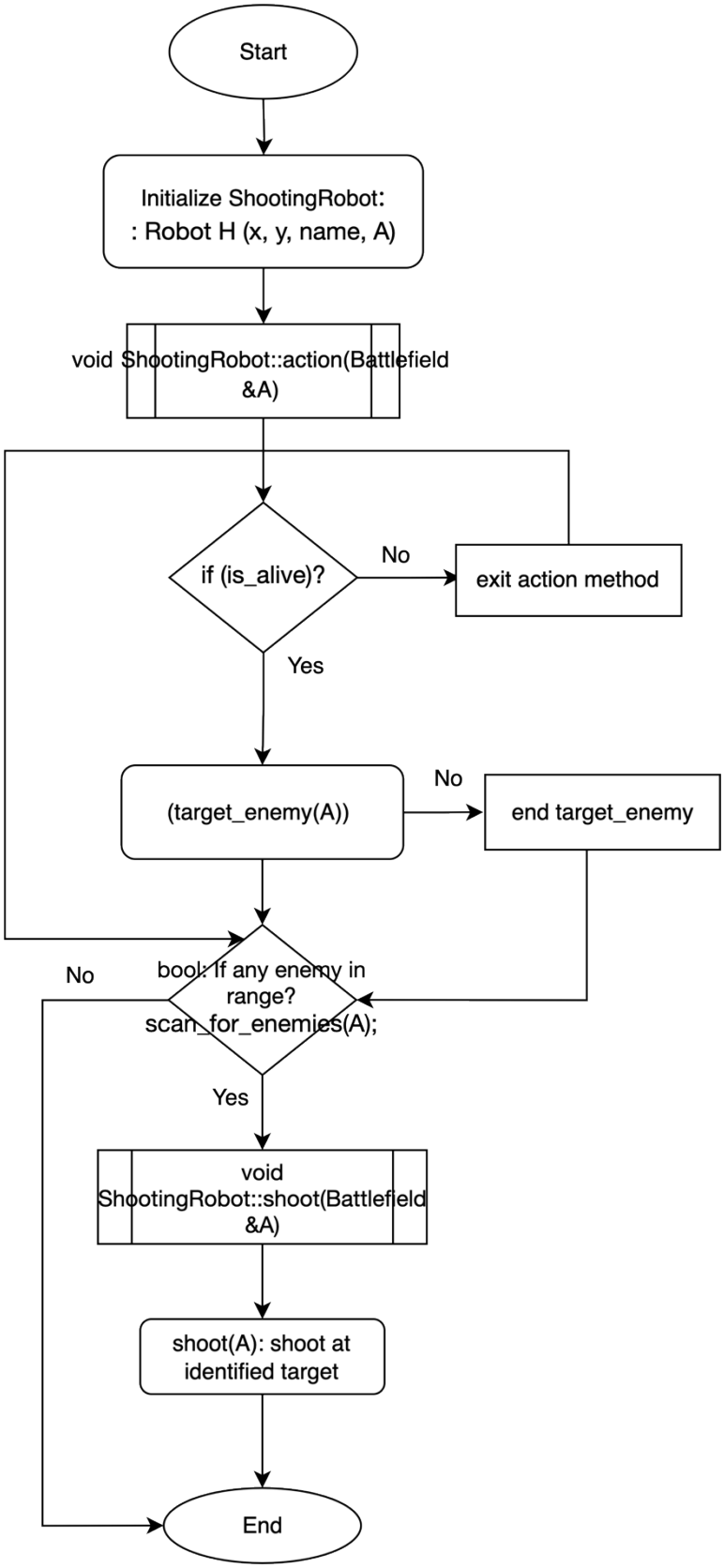
`MovingRobot`



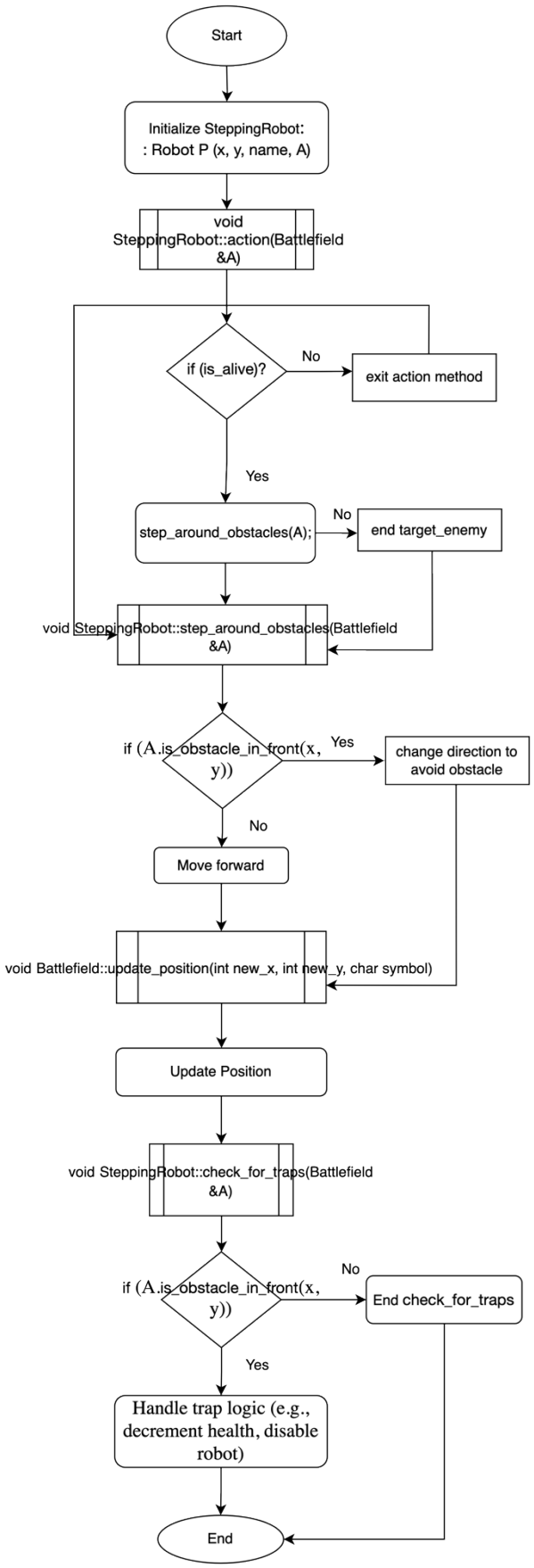
SeeingRobot



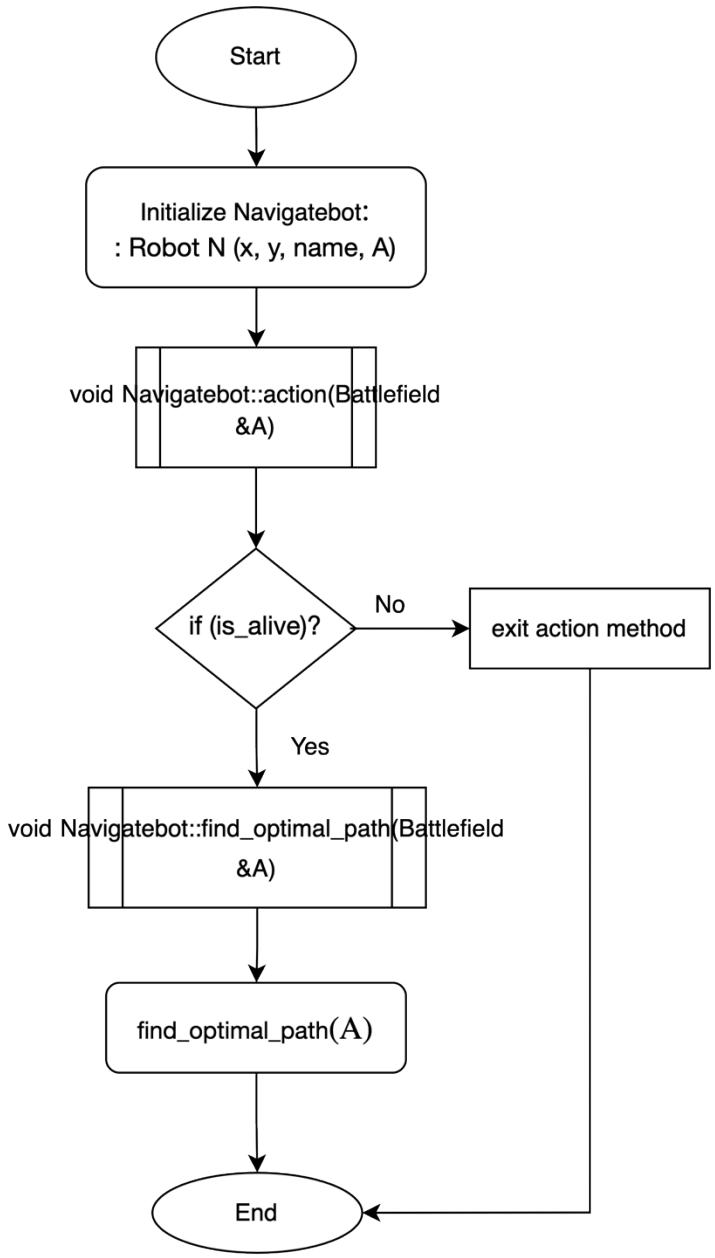
ShootingRobot

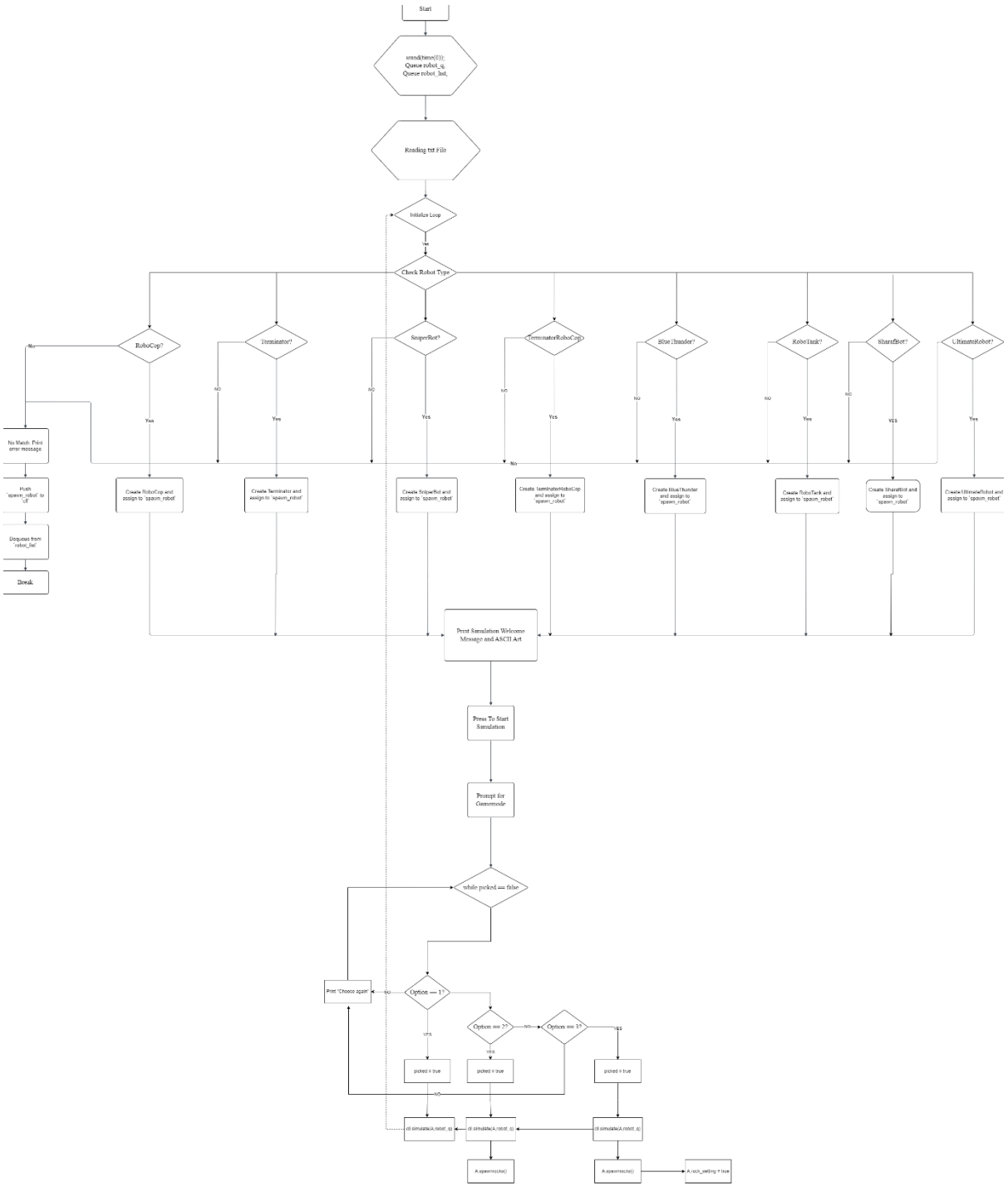


SteppingRobot



Navigatebot




Start
srand(time(0));
Queue robot_q;
Queue robot_list;
Reading txt File
Initalize Loop
Yes
Check Robot Type
RoboCop?
Terminator?
SniperBot?
TerminatorRoboCop?
BlueThunder?
RoboTank?
SharatBot?
UltimateRobot?
No
NO
Yes
No Match. Print error message
Push 'spawn_robot' to q
Dequeue from robot_list
Break
Create RoboCop and assign to 'spawn_robot'
Create Terminator and assign to 'spawn_robot'
Create SniperBot and assign to 'spawn_robot'
Create TerminatorRoboCop and assign to 'spawn_robot'
Create BlueThunder and assign to 'spawn_robot'
Create RoboTank and assign to 'spawn_robot'
Create SharatBot and assign to 'spawn_robot'
Create UltimateRobot and assign to 'spawn_robot'
Print Simulation Welcome Message and ASCII Art
Press To Start Simulation
Prompt for Gamemode
while picked == false
Option == 1?
Option == 2?
Option == 3?
Print 'Choose again'
YES
picked = true
cll.simulate(A.robot_q)
A.spawnrocks()
A.rock_setting = true