OOPDS ASSIGNMENT 40%

TRIMESTER 2430

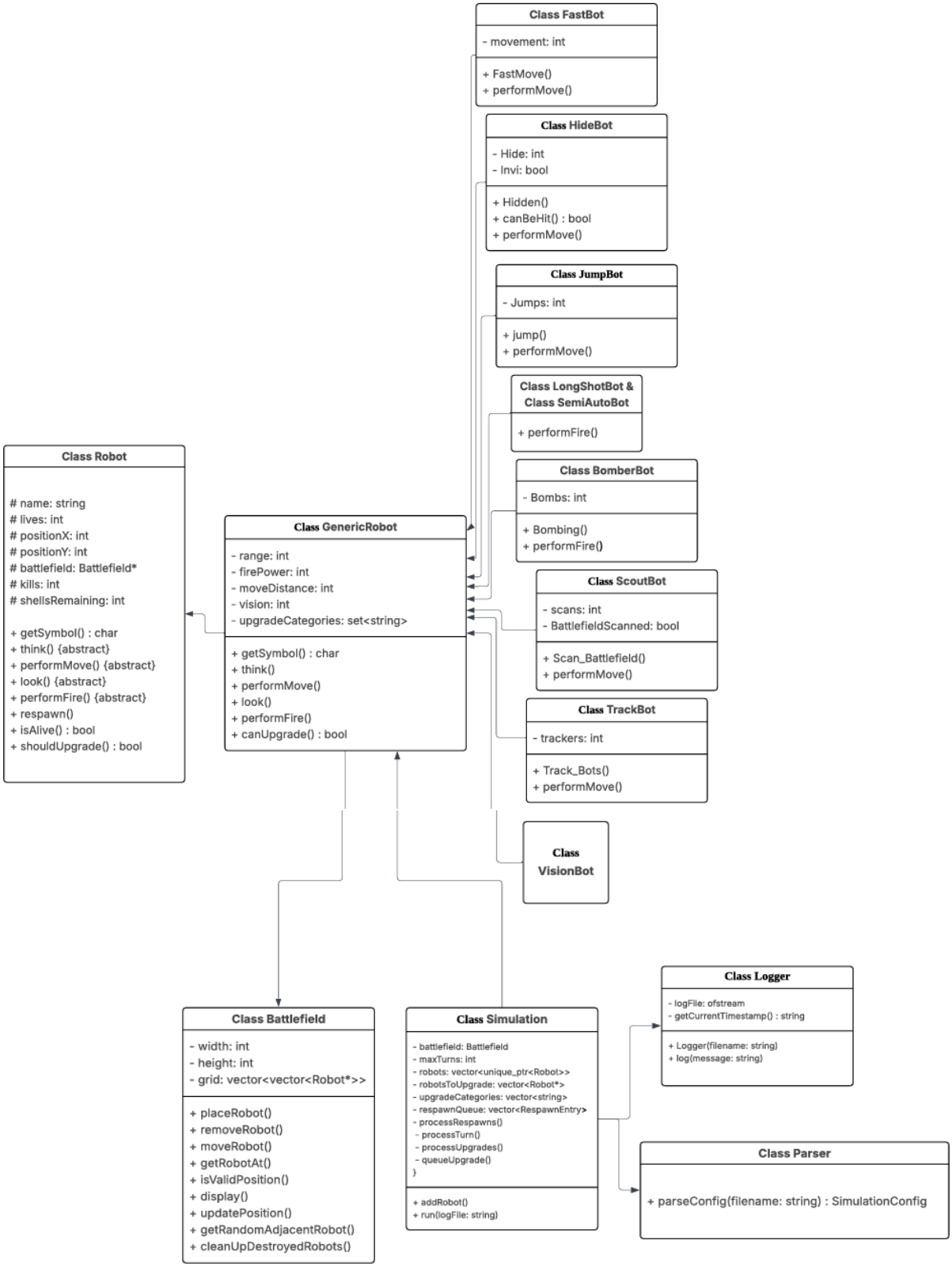
LECTURE TC01

TUTORIAL TT01

GROUP 16

NAME	ID	EMAIL
ESVAN RAO A/L PERASATH RAW	243UC245HJ	esvan.rao.perasath@student.mmu.edu.my
BRADLEY BEN LEE	243UC245SA	bradley.ben.lee@student.mmu.edu.my
DAVID TING ZI XIANG	242UC244PE	david.ting.zi@student.mmu.edu.my

UML CLASS DIAGRAM



PSEUDOCODE

Main Program

```
FUNCTION main():
  SET random seed using current time
  TRY:
    SET configFile = first command-line argument or "config.txt"
    SET logFile = second command-line argument or "robot_war.log"

    // Parse configuration
    config = Parser.parseConfig(configFile)

    // Validate battlefield size
    IF config.width <= 0: SET width = 10
    IF config.height <= 0: SET height = 10

    // Initialize simulation
    sim = new Simulation(config.width, config.height, config.maxTurns)

    // Add robots from config
    FOR EACH robotCfg IN config.robots:
      robot = new GenericRobot(robotCfg.name, sim.getBattlefield())
      IF robotCfg.randomPosition:
        sim.addRobot(robot)
      ELSE:
        sim.addRobot(robot, robotCfg.x, robotCfg.y)

    // Run simulation
    sim.run(logFile)
    PRINT "Simulation complete. Results logged to " + logFile

  CATCH exceptions:
    PRINT error message
    RETURN 1
  RETURN 0
```

Battlefield Class

```
CLASS Battlefield:
  PROPERTIES:
    width, height: int
    grid: 2D array of Robot pointers

  METHODS:
```

```
    CONSTRUCTOR(width, height)
    placeRobot(robot, x, y) -> bool
    removeRobot(x, y)
    moveRobot(robot, newX, newY) -> bool
    getRobotAt(x, y) -> Robot*
    isValidPosition(x, y) -> bool
    getRandomEmptyCell() -> (x, y)
    display()
    updatePosition(oldX, oldY, newX, newY)
    getRandomAdjacentRobot(x, y) -> Robot*
    cleanUpDestroyedRobots()
    hasRobotAt(x, y) -> bool
```

Robot Base Class

```
ABSTRACT CLASS Robot:
  PROPERTIES:
    name: string
    lives: int
    positionX, positionY: int
    battlefield: Battlefield*
    kills: int
    shellsRemaining: int
    destroyedThisRound: bool
    upgrades: list<string>
    needsUpgrade: bool
    upgradeCategory: string

  METHODS:
    CONSTRUCTOR(name, battlefield)
    PURE VIRTUAL getSymbol() -> char
    PURE VIRTUAL think()
    PURE VIRTUAL performMove()
    PURE VIRTUAL look(x, y)
    PURE VIRTUAL performFire(x, y)
    respawn()
    isAlive() -> bool
    shouldUpgrade() -> bool
    addUpgrade(category)
    hasUpgrade(category) -> bool
    displayUpgrades()
```

Simulation Flow

CLASS Simulation: PROPERTIES: battlefield: Battlefield maxTurns: int currentTurn: int robots: list<unique_ptr> respawnQueue: list

```
METHODS:
    addRobot(robot)
    addRobot(robot, x, y)
    run(logFile):
        PRINT start message
        FOR turn = 1 TO maxTurns:
            PRINT "=== TURN [turn] ==="
            logger.log(turn header)

            // Process game state
            battlefield.cleanUpDestroyedRobots()
            battlefield.display()
            processRespawns(currentTurn)
            processTurn(logger)

            IF robots.count <= 1: BREAK

        // Game over handling
        IF robots.empty: PRINT "All robots destroyed!"
        ELSE IF robots.count == 1: PRINT winner
        ELSE: PRINT remaining robots count

    processTurn(logger):
        // Remove destroyed robots
        FOR EACH robot IN robots:
            IF !robot.isAlive():
                IF robot.lives > 0:
                    queue respawn
                REMOVE from battlefield
                REMOVE from robots list

        // Check and queue upgrades
        FOR EACH robot IN robots:
            IF robot.shouldUpgrade():
                category = determine_category_based_on_kills()
                queueUpgrade(robot, category)

        // Process robot turns
        FOR EACH robot IN robots:
            IF robot.isAlive():
                LOG robot status
                robot.think()

    processUpgrades():
        FOR EACH queued upgrade:
            newRobot = Upgrade(oldRobot, category)
            REPLACE oldRobot with newRobot in robots list
```

```
            UPDATE battlefield position

    processRespawns(currentTurn):
        FOR EACH respawnEntry IN queue:
            IF currentTurn >= respawnTurn:
                FIND empty position
                robot.respawn()
                PLACE on battlefield
```

Generic Robot Behavior

```
CLASS GenericRobot EXTENDS Robot:
  PROPERTIES:
    range = 1, firePower = 1, moveDistance = 1, vision = 2

  METHODS:
    think():
      FOR EACH adjacent cell (including diagonals):
        look(x, y)
        IF enemy found: SET enemy coordinates

      IF enemy in range: performFire(enemyX, enemyY)
      ELSE: performMove()

    performMove():
      TRY 20 times:
        GENERATE random dx, dy in [-1,0,1]
        CALCULATE new position
        IF position valid and empty:
          UPDATE battlefield position
          RETURN
      PRINT "Couldn't move"

    look(x, y):
      IF enemy present at (x,y):
        SET enemyFound = true
        RECORD enemy position/name

    performFire(x, y):
      IF out of shells: SELF-DESTRUCT
      IF target exists:
        REDUCE target lives
        IF target destroyed:
          INCREMENT kills
          QUEUE upgrade if needed
      DECREMENT ammo
```

Upgrade Robot

```
FUNCTION Upgrade(GenericRobot* Bot, string category) -> GenericRobot*: IF Bot already has category upgrade: RETURN Bot

SWITCH category:
    CASE "Moving":
        OPTIONS = [HideBot, JumpBot, FastBot]
        SELECT random available option
        RETURN upgraded robot

    CASE "Shooting":
        OPTIONS = [LongShotBot, SemiAutoBot, BomberBot]
        SELECT random available option
        RETURN upgraded robot

    CASE "Seeing":
        OPTIONS = [ScoutBot, TrackBot, VisionBot]
        SELECT random available option
        RETURN upgraded robot

RETURN Bot  // Fallback


// Example Upgrade Class CLASS HideBot EXTENDS GenericRobot: PROPERTIES: Hide = 3, Invi = false

METHODS:
    Hidden():
        IF Hide > 0:
            SET Invi = true
            DECREMENT Hide
        ELSE: SET Invi = false

    performMove():
        CALL Hidden()
        CALL GenericRobot::performMove()

    canBeHit() -> bool: RETURN !Invi
```

Parser

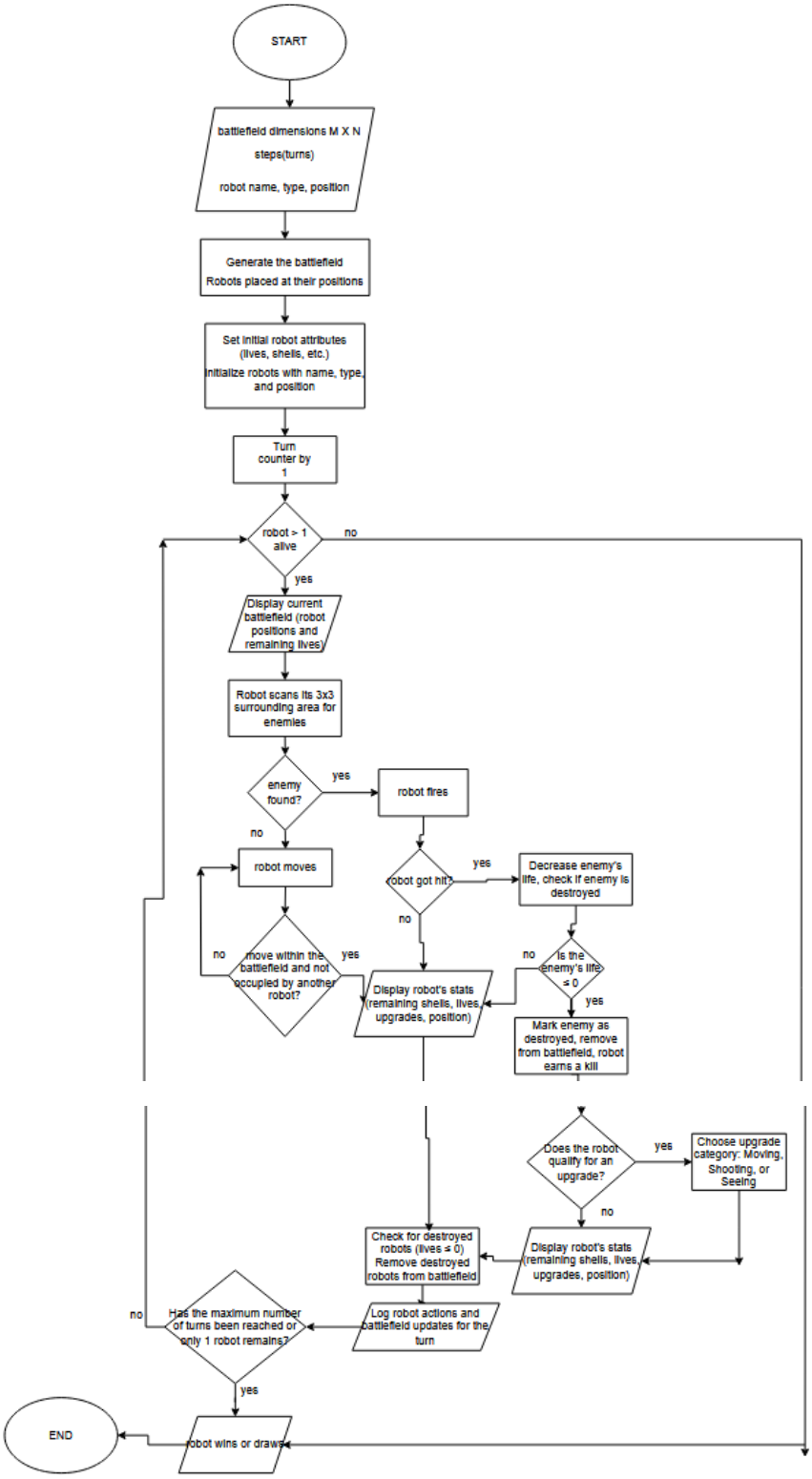
```
CLASS Parser:
  STATIC METHOD parseConfig(filename) -> SimulationConfig:
    OPEN file
```

```
INIT config object

FOR EACH line in file:
  SKIP empty/comments
  IF line contains colon:
    EXTRACT key/value
    HANDLE "MbyN" -> set width/height
    HANDLE "steps" -> set maxTurns
  ELSE:  // Robot line
    EXTRACT robot type
    EXTRACT name until "random" or coordinates
    ADD to config.robots

RETURN config
```

Turn Processing

```
FOR EACH turn:
 1. Clean destroyed robots
 2. Process respawns
 3. Display battlefield
 4. For each alive robot:
   a. Check upgrade eligibility
   b. Think (detect enemies -> move or attack)
   c. Log actions
 5. Process upgrades
```

FLOWCHART


START
battlefield dimensions M X N
steps(turns)
robot name, type, position
Generate the battlefield
Robots placed at their positions
Set initial robot attributes (lives, shells, etc.)
Initialize robots with name, type, and position
Turn counter by 1
robot > 1 alive
no
yes
Display current battlefield (robot positions and remaining lives)
Robot scans its 3x3 surrounding area for enemies
enemy found?
yes
robot fires
no
robot moves
robot got hit?
yes
Decrease enemy's life, check if enemy is destroyed
no
no
move within the battlefield and not occupied by another robot?
yes
Display robot's stats (remaining shells, lives, upgrades, position)
no
Is the enemy's life ≤ 0
yes
Mark enemy as destroyed, remove from battlefield, robot earns a kill
Does the robot qualify for an upgrade?
yes
Choose upgrade category: Moving, Shooting, or Seeing
no
Display robot's stats (remaining shells, lives, upgrades, position)
Check for destroyed robots (lives ≤ 0) Remove destroyed robots from battlefield
Log robot actions and battlefield updates for the turn
Has the maximum number of turns been reached or only 1 robot remains?
no
yes
END
robot wins or draw

Screen-shots and explanation

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>
#include <ctime>
#include <set>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <memory>
#include <algorithm>
```

Explanation:
These are standard C++ library headers providing essential functionality:

- `<iostream>`: Input/output operations (e.g., `cout`, `cin`).
- `<vector>`: Dynamic array container (`std::vector`).
- `<string>`: String manipulation (`std::string`).
- `<cstdlib>`: General utilities (e.g., `rand()`, `exit()`).
- `<ctime>`: Time functions (`std::time` for seeding random numbers).
- `<set>`: Sorted unique-element container (`std::set`).
- `<fstream>`: File stream operations (reading/writing files).
- `<sstream>`: String stream processing (`std::stringstream`).
- `<iomanip>`: Input/output formatting (e.g., `setw()`, `fixed`).
- `<memory>`: Smart pointers (`std::unique_ptr`, memory management).
- `<algorithm>`: Algorithms (e.g., `sort()`, `find()`).

```cpp
using namespace std;
using std::unique_ptr;
using std::make_unique;

// Forward declarations
class Battlefield;
class Robot;
class GenericRobot;
class Logger;
class Simulation;
class Parser;
GenericRobot* Upgrade(GenericRobot* Bot, const string& category);

// Constants
const int MAP_WIDTH = 10;
const int MAP_HEIGHT = 10;
```

Explanation:

1. `using` **Directives**:
    a. `using namespace std`: Imports all symbols from the `std` namespace (risky due to name collisions).
    b. Explicit imports for `unique_ptr` and `make_unique` (safe for smart pointers).
2. **Forward Declarations**:
    a. Declares classes/functions without defining them. Allows referencing these types before their full definitions (e.g., `Battlefield`, `Robot`).
3. **Constants**:
    a. `MAP_WIDTH` and `MAP_HEIGHT` define the battlefield grid size (10x10).

```cpp
// Battlefield class
class Battlefield {
private:
    int width, height;
    std::vector<std::vector<Robot*>> grid;

public:
    Battlefield(int width, int height);
    bool placeRobot(Robot* robot, int x, int y);
    void removeRobot(int x, int y);
    bool moveRobot(Robot* robot, int newX, int newY);
    Robot* getRobotAt(int x, int y) const;
    bool isValidPosition(int x, int y) const;
    std::pair<int, int> getRandomEmptyCell() const;
    void display() const;
    int getWidth() const { return width; }
    int getHeight() const { return height; }
    bool hasRobotAt(int x, int y) const { return getRobotAt(x, y) != nullptr; }
    void updatePosition(int oldX, int oldY, int newX, int newY);
    Robot* getRandomAdjacentRobot(int x, int y) const;
    void cleanUpDestroyedRobots();
};
```

Key Components:

1. **Grid Representation**:
    a. 2D vector `grid` storing pointers to `Robot` objects. Each cell can hold a robot or be `nullptr`.
2. **Core Functionality**:
    a. **Placement/Removal**: `placeRobot()`, `removeRobot()`, and `moveRobot()` manage robot positions.
    b. **Queries**: `getRobotAt()`, `isValidPosition()`, `hasRobotAt()` check grid state.
    c. **Utilities**: `getRandomEmptyCell()` finds empty spots; `display()` visualizes the grid.
    d. **Movement Handling**: `updatePosition()` internally updates coordinates during moves.
    e. **Combat Utilities**: `getRandomAdjacentRobot()` likely for targeting neighbors in battles.
    f. **Cleanup**: `cleanUpDestroyedRobots()` removes destroyed robots (prevents dangling pointers).
3. **Design Notes**:
    a. Manages a **10x10 grid** (per `MAP_WIDTH/MAP_HEIGHT`).
    b. Uses **raw pointers** (`Robot*`), implying manual memory management (caution needed to avoid leaks).
    c. `const` member functions ensure state isn't modified during queries.

```cpp
// Robot base class
class Robot {
protected:
    string name;
    int lives;
    bool destroyedThisRound = false;
    int positionX, positionY;
    Battlefield* battlefield;
    vector<string> AddedUpgrades;
    bool enemyFound = false;
    int enemyX, enemyY;
    string enemyName;
    int kills = 0;
    int shellsRemaining = 10;
    bool needsUpgrade = false;
    string upgradeCategory;

public:
    Robot(const string& name, Battlefield* battlefield)
        : name(name), lives(3), battlefield(battlefield) {
        positionX = rand() % battlefield->getWidth();
        positionY = rand() % battlefield->getHeight();
    }

    virtual ~Robot() {}

    virtual char getSymbol() const = 0;
    virtual void think() = 0;
    virtual void performMove() = 0;
    virtual void look(int x, int y) = 0;
    virtual void performFire(int x, int y) = 0;

    virtual void respawn() {
        lives = 3;
        destroyedThisRound = false;
        // Reset any temporary state flags
        enemyFound = false;
        shellsRemaining = 10;  // Reset ammo for new life
    }

    string getName() const { return name; }
    bool isAlive() const { return lives > 0; }
    void setLives(int l) { lives = l; }
    int getLives() const { return lives; }
```

```
(); ++i) {
```

Key Member Variables (Protected):

1. `name`: Robot's identifier.
2. `lives`: Health points (default: 3).
3. `destroyedThisRound`: Flag if destroyed in current round.
4. `positionX/Y`: Grid coordinates.
5. `battlefield`: Pointer to the game's battlefield.
6. `AddedUpgrades`: List of acquired upgrades (e.g., "Armor", "Weapon").
7. `enemyFound`, `enemyX/Y`, `enemyName`: Tracked enemy info during scanning.
8. `kills`: Number of defeated enemies.
9. `shellsRemaining`: Ammunition count (default: 10).
10. `needsUpgrade`, `upgradeCategory`: Flags for upgrade requests.

1. **Core Robots**:

- `respawn()`:
  Resets robot on death: refills lives (3), ammo (10), clears destruction flag.
- `isWithinShootingRange(int enemyX, int enemyY)`:
  Checks if an enemy is adjacent (Chebyshev distance ≤ 1, excluding self).

2. **Upgrade System**:

- `addUpgrade("Category")`:
  Adds an upgrade (e.g., `addUpgrade("Armor")`).
- `hasUpgrade("Category")`:
  Checks if upgrade exists.
- `shouldUpgrade()`:
  Base logic: robot can upgrade after ≥1 kill and <3 upgrades.

3. **Pure Virtual Functions (Must Be Implemented by Derived Classes)**:

- `char getSymbol() const`: Returns ASCII char for battlefield display.
- `void think()`: AI logic (e.g., scan for enemies, set `enemyFound`).
- `void performMove()`: Movement logic.
- `void look(int x, int y)`: Scan a grid cell for enemies.
- `void performFire(int x, int y)`: Attack logic.

4. **Utility Functions**:

- `log("message")`: Outputs messages for debugging.
- `displayUpgrades()`: Prints current upgrades.
- **Position Accessors:** `getX()`, `getY()`, `setPosition()`.

```cpp
// GenericRobot implementation
class GenericRobot : public Robot {
protected:
    int range, firePower, moveDistance, vision;

public:
    GenericRobot(string name, Battlefield* battlefield)
        : Robot(name, battlefield), range(1), firePower(1), moveDistance(1), vision(2) {}

    char getSymbol() const override {
        return '@';
    }

    int getRange() const { return range; }
    int getFirePower() const { return firePower; }
    int getMoveDistance() const { return moveDistance; }
    int getVision() const { return vision; }
    set<string> upgradeCategories;

    void think() override {
        enemyFound = false; // Reset enemy detection at start of turn

        log(getName() + " is looking around...\n");

        // Look in all adjacent positions (including diagonals)
        for (int dx = -1; dx <= 1; ++dx) {
            for (int dy = -1; dy <= 1; ++dy) {
                if (dx == 0 && dy == 0) continue; // Skip self

                int lookX = getX() + dx;
                int lookY = getY() + dy;

                if (battlefield->isValidPosition(lookX, lookY)) {
                    look(lookX, lookY);
                }
            }
        }

        if (enemyFound && isWithinShootingRange(enemyX, enemyY)) {
            performFire(enemyX, enemyY);
        } else {
            performMove();
        }
    }
```

```
    void performMove() override {
        int attempts = 0;
        const int maxAttempts = 20;

        while (attempts < maxAttempts) {
            int x = rand() % 3 - 1; // -1, 0, 1
            int y = rand() % 3 - 1; // -1, 0, 1

            // Skip if both x and y are 0 (no movement)
            if (x == 0 && y == 0) {
                attempts++;
                continue;
            }

            int newX = getX() + x;
            int newY = getY() + y;

            if (newX >= 0 && newX < battlefield->getWidth() &&
                newY >= 0 && newY < battlefield->getHeight() &&
                !battlefield->hasRobotAt(newX, newY)) {

                battlefield->updatePosition(getX(), getY(), newX, newY);
                return;
            }

            attempts++;
        }

        // If all attempts fail, stay in place
        log(getName() + " could not find a safe place to move!\n");
    }

    void look(int x, int y) override {
        Robot* target = battlefield->getRobotAt(x, y);
        if (target && target != this && target->isAlive()) {
            enemyFound = true;
            enemyX = x;
            enemyY = y;
            enemyName = target->getName();

            log(getName() + " found " + enemyName + " at (" + to_string(enemyX) + ", " + to_string(enemyY) + ")!\n");
        }
    }
    void performFire(int x, int y) override {
    if (shellsRemaining <= 0) {
        log("\n" + getName() + " is out of shells and self-destructs!\n");
        setDestroyedThisRound(true);
        battlefield->removeRobot(getX(), getY());
        setLives(0);
        return;
    }

    Robot* target = battlefield->getRobotAt(x, y);
    if (target && target != this && target->isAlive()) {
        log("\n" + getName() + " FIRES at " + target->getName() + " at (" +
            to_string(x) + ", " + to_string(y) + ")!\n");

        if (!target->canBeHit()) {
            log(target->getName() + " avoided the attack (cannot be hit)!\n");
        } else {
            int newLives = target->getLives() - 1;  // Decrement lives
            target->setLives(newLives);

            if (newLives <= 0) {
                log("DIRECT HIT! " + target->getName() + " is destroyed!\n");
                target->setDestroyedThisRound(true);
                battlefield->removeRobot(x, y);
                kills++;

                // Set upgrade flag based on kill count
                if (kills == 1 && !hasUpgradeCategory("Moving")) {
                    needsUpgrade = true;
                    upgradeCategory = "Moving";
                } else if (kills == 2 && !hasUpgradeCategory("Shooting")) {
                    needsUpgrade = true;
                    upgradeCategory = "Shooting";
                } else if (kills >= 3 && !hasUpgradeCategory("Seeing")) {
                    needsUpgrade = true;
                    upgradeCategory = "Seeing";
                }
            } else {
                log("HIT! " + target->getName() + " has " + to_string(newLives) + " lives remaining!\n");
            }
        }

        shellsRemaining--;
    } else {
        log("\n" + getName() + " fired at empty space at (" + to_string(x) + ", " + to_string(y) + ").\n");
        shellsRemaining--;
    }
}
```

```
    bool canUpgrade(const string& category) const {
        return upgradeCategories.find(category) == upgradeCategories.end();
    }

    bool hasUpgradeCategory(const string& category) const {
    return upgradeCategories.count(category) > 0;
    }

    void markUpgraded(const string& category) {
    upgradeCategories.insert(category);  // Track the category only here
    // Don't add the category to AddedUpgrades here
    }

    bool hasUpgrade(const string& upgradeName) const {
    for (const auto& upgrade : AddedUpgrades) {
        if (upgrade == upgradeName) return true;
    }
    return false;
}
bool shouldUpgrade() const override {
    if (kills > 0 && upgradeCategories.size() < 3) {
        if (kills == 1 && !hasUpgradeCategory("Moving")) return true;
        if (kills == 2 && !hasUpgradeCategory("Shooting")) return true;
        if (kills >= 3 && !hasUpgradeCategory("Seeing")) return true;
    }
    return false;
}
};

// Moving type Upgrades (HideBot, jumpBot, FastBot)
namespace Moving {

// Shooting type Upgrades (LongShot, SemiAutoBot, BomberBot)
namespace Shooting {

// Seeing type Upgrades (ScoutBot, TrackBot, VisionBot)
namespace Seeing {

// GenericRobot Upgrades
GenericRobot* Upgrade(GenericRobot* Bot, const string& category) {
```

Key Attributes:

1. **Combat Stats** (Initial Defaults):
    a. `range = 1`: Shooting range (adjacent cells only)
    b. `firePower = 1`: Damage per shot
    c. `moveDistance = 1`: Cells moved per turn
    d. `vision = 2`: Line-of-sight distance
2. **Upgrade Tracking**:
    a. `upgradeCategories`: `std::set` to track applied upgrade types ("Moving", "Shooting", "Seeing")

Core Functionality:

1. **Symbol & Getters**:

- `getSymbol()`: Returns `'@'` for battlefield visualization.
- Getters for stats: `getRange()`, `getFirePower()`, etc.

2. **AI Logic (`think()`)**:

The `think()` method scans adjacent cells for enemies and either fires if one is in range or moves randomly otherwise.

3. **Movement (`performMove()`)**:

- Attempts random direction moves (up to 20 tries).
- Moves to first valid empty cell found.
- Failsafe: Stays in place if no valid move.

4. **Enemy Detection (`look()`)**:

- Checks a cell for alive enemy robots (non-self).
- Logs detection and caches enemy position/name.

5. **Combat (`performFire()`)**:

Key Combat Flow:

1. Ammo check → self-destruct if empty.
2. Valid target? Apply damage.
3. On kill:
    a. Track kill count
    b. Request specific upgrade type based on kills (1→Moving, 2→Shooting, 3+→Seeing)

6. **Upgrade Management**:

- `hasUpgradeCategory()`: Checks if upgrade type applied.
- `markUpgraded()`: Adds type to `upgradeCategories` set.
- `shouldUpgrade()`: Override checks kill-based upgrade conditions.

```cpp
// Logger class
class Logger {
private:
    std::ofstream logFile;
    std::string getCurrentTimestamp();

public:
    Logger(const std::string& filename);
    ~Logger();
    void log(const std::string& message);
};

Logger::Logger(const std::string& filename) {
    logFile.open(filename);
}

Logger::~Logger() {
    if (logFile.is_open()) {
        logFile.close();
    }
}

std::string Logger::getCurrentTimestamp() {
    auto now = std::time(nullptr);
    auto tm = *std::localtime(&now);
    std::ostringstream oss;
    oss << std::put_time(&tm, "[%Y-%m-%d %H:%M:%S]");
    return oss.str();
}

void Logger::log(const std::string& message) {
    if (logFile.is_open()) {
        logFile << getCurrentTimestamp() << " " << message << "\n";
    }
}
```

```cpp
Logger::Logger(const std::string& filename) {
    logFile.open(filename);
}

Logger::~Logger() {
    if (logFile.is_open()) {
        logFile.close();
    }
}

std::string Logger::getCurrentTimestamp() {
    auto now = std::time(nullptr);
    auto tm = *std::localtime(&now);
    std::ostringstream oss;
    oss << std::put_time(&tm, "[%Y-%m-%d %H:%M:%S]");
    return oss.str();
}

void Logger::log(const std::string& message) {
    if (logFile.is_open()) {
        logFile << getCurrentTimestamp() << " " << message << "\n";
    }
}
```

Logger Class

Purpose: Handles timestamped logging to a file.

Key Features:

1. **File Management**:
   a. Constructor opens log file: `Logger("filename.log")`
   b. Destructor closes file automatically

Timestamp Generation:

   a. Formats timestamp as `[2023-12-01 14:30:45]`

Logging:

   a. Writes timestamp + message to file
   b. Example output: [2023-12-01 14:30:45] Simulation started

```
// Battlefield implementation
Battlefield::Battlefield(int width, int height)
    : width(width), height(height) {
    grid.resize(width, std::vector<Robot*>(height, nullptr));
}

bool Battlefield::placeRobot(Robot* robot, int x, int y) {

void Battlefield::removeRobot(int x, int y) {

bool Battlefield::moveRobot(Robot* robot, int newX, int newY) {

Robot* Battlefield::getRobotAt(int x, int y) const {

bool Battlefield::isValidPosition(int x, int y) const {

std::pair<int, int> Battlefield::getRandomEmptyCell() const {
    std::vector<std::pair<int, int>> emptyCells;
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            if (grid[x][y] == nullptr) {
                emptyCells.emplace_back(x, y);
            }
        }
    }
    if (emptyCells.empty()) return {-1, -1};
    return emptyCells[rand() % emptyCells.size()];
}

void Battlefield::display() const {

void Battlefield::updatePosition(int oldX, int oldY, int newX, int newY) {
```

```
Robot* Battlefield::getRandomAdjacentRobot(int x, int y) const {
    vector<Robot*> adjacentRobots;
    for (int dx = -1; dx <= 1; dx++) {
        for (int dy = -1; dy <= 1; dy++) {
            if (dx == 0 && dy == 0) continue; // Skip self
            int nx = x + dx;
            int ny = y + dy;
            if (isValidPosition(nx, ny)) {
                Robot* robot = grid[nx][ny];
                if (robot && robot->isAlive()) {
                    adjacentRobots.push_back(robot);
                }
            }
        }
    }
    if (adjacentRobots.empty()) return nullptr;
    return adjacentRobots[rand() % adjacentRobots.size()];
}

void Battlefield::cleanUpDestroyedRobots() {
```

Battlefield Implementation

1. Core Grid Management:

- **Constructor**: Initializes grid with `nullptr`
  `grid.resize(width, std::vector<Robot*>(height, nullptr));`
- **Placement/Removal**:
  - `placeRobot()`: Validates position, sets robot's coordinates
  - `removeRobot()`: Sets grid cell to `nullptr`

2. Movement System:

```
bool moveRobot(Robot* robot, int newX, int newY) {
    auto [currX, currY] = robot->getPosition();
    if (!isValidPosition(newX, newY) return false;
    if (grid[newX][newY]) return false; // Cell occupied
    grid[currX][currY] = nullptr;
    grid[newX][newY] = robot;
    robot->setPosition(newX, newY);
    return true;
}
```

- **updatePosition()**: Direct position swap (no checks)

3. Queries & Utilities:

- **Random Empty Cell**:

```
std::pair<int, int> getRandomEmptyCell() const {
    std::vector<std::pair<int, int>> emptyCells;
    // Collect all empty cells
   return emptyCells[rand() % emptyCells.size()];
}
```

- **Adjacent Robot Search**:

```
Robot* getRandomAdjacentRobot(int x, int y) const {
  vector<Robot*> adjacentRobots;
  // Check all 8 directions
  return adjacentRobots[rand() % adjacentRobots.size()];
}
```

- **4. Display System**:

```
void Battlefield::display() const {
  // Print column headers (0-9)
  for (int x=0; x<width; x++)
    cout << setw(2) << x << " ";

  // Print grid rows
  for (int y=0; y<height; y++) {
    cout << setw(2) << y << " ";  // Row header
    for (int x=0; x<width; x++) {
      if (grid[x][y])
        cout << " " << grid[x][y]->getSymbol() << " ";
      else
        cout << " . ";
    }
    cout << "\n";
  }
}
```

- **5. Cleanup**:

```
void cleanUpDestroyedRobots() {
  for (int x=0; x<width; x++) {
    for (int y=0; y<height; y++) {
      if (grid[x][y] && !grid[x][y]->isAlive()) {
        grid[x][y] = nullptr;
```

```
            }
        }
    }
}
```

- Removes dead robots from grid (prevents dangling pointers)

```
// Simulation class
class Simulation {
private:
    Battlefield battlefield;
    int maxTurns;
    int currentTurn = 0;
    vector<unique_ptr<Robot>> robots;
    vector<Robot*> robotsToUpgrade;
    vector<string> upgradeCategories;

     struct RespawnEntry {
        unique_ptr<Robot> robot;
        int respawnTurn;
    };
    vector<RespawnEntry> respawnQueue;
    const int respawnDelay = 3;

public:
    Simulation(int width, int height, int maxTurns)
        : battlefield(width, height), maxTurns(maxTurns) {}

    void addRobot(unique_ptr<Robot> robot) {

    void addRobot(unique_ptr<Robot> robot, int x, int y) {

    Battlefield* getBattlefield() { return &battlefield; }

    void run(const string& logFile) {

    void processRespawns(int currentTurn) {

    void processTurn(Logger& logger) {

void processUpgrades() {


    void queueUpgrade(Robot* robot, const string& category) {

};
```

`Simulation` Class Explanation

Purpose:

Manages the entire robot battle simulation lifecycle, including turn progression, robot management, upgrades, respawning, and logging.

Key Components:

1. **Member Variables**:
    a. `battlefield`: Game grid
    b. `maxTurns`: Simulation length limit
    c. `robots`: Active robots (owned via `unique_ptr`)
    d. `respawnQueue`: Destroyed robots waiting to respawn
    e. `robotsToUpgrade/upgradeCategories`: Upgrade processing queues
    f. `respawnDelay = 3`: Turns between destruction and respawn
2. **RespawnEntry Struct**:

```
struct RespawnEntry {
  unique_ptr<Robot> robot;  // Robot to respawn
  int respawnTurn;          // Turn when respawn occurs
};
```

Core Functionality:

1. **Robot Management**:

- `addRobot()`: Places robots on battlefield
    - Random position: `battlefield.getRandomEmptyCell()`
    - Fixed position: Validates placement
- `getBattlefield()`: Provides battlefield access

2. **Main Loop (`run()`)**:

```
void run(const string& logFile) {
  Logger logger(logFile);
  for (currentTurn = 1; currentTurn <= maxTurns; currentTurn++) {
    processUpgrades();
    processTurn(logger);       // Handle robot actions
    processRespawns(currentTurn); // Respawn queued robots
    if (robots.size() <= 1) break;// Early exit if winner
  }
  // Display final results
}
```

3. **Respawn System (`processRespawns()`)**:

- Checks `respawnQueue` for robots due to respawn
- Attempts random placement (100 tries max)
- Successful respawn:
    - Robot moved back to `robots` vector
    - Lives reset via `robot->respawn()`
- Permanent death if lives = 0

4. **Turn Processing (`processTurn()`)**:

1. **Cleanup**: Remove dead robots via `battlefield.cleanUpDestroyedRobots()`
2. **Display**: Show battlefield state
3. **Respawn Queueing**:

```
if (robot->isAlive()) continue;
if (robot->getLives() > 0) {
  // Queue for respawn
  respawnQueue.push_back({move(robot), currentTurn + respawnDelay});
}
```

4. **Upgrade Detection**:
    a. Checks `shouldUpgrade()` for eligible robots
    b. Queues upgrades via `queueUpgrade()`
5. **Robot Actions**:
    c. Display stats (position, lives, upgrades)
    d. Execute `robot->think()` (AI decision)

5. **Upgrade System**:

- **Queueing**: `queueUpgrade()` stores robot + category
- **Processing**:

```
void processUpgrades() { for (size_t i=0; i<robotsToUpgrade.size(); i++) { GenericRobot* base =
dynamic_cast<GenericRobot*>(robotsToUpgrade[i]); GenericRobot* upgraded = Upgrade(base,
upgradeCategories[i]);
   // Replace robot in vector and battlefield
    auto it = find_if(robots.begin(), robots.end(),
        [base](auto& ptr){ return ptr.get() == base; });

    if (it != robots.end()) {
        battlefield.removeRobot(base->getX(), base->getY());
        it->reset(upgraded); // Ownership transfer
        battlefield.placeRobot(upgraded, ...);
    }
}

}
```

```cpp
// Parser and config structures
struct RobotInit {
    std::string type;
    std::string name;
    int x, y;
    bool randomPosition;
};

struct SimulationConfig {
    int width, height;
    int maxTurns;
    std::vector<RobotInit> robots;
};
```

Purpose:
Used by a `Parser` class (not shown) to load simulation settings from configuration files.

Key Workflows:

1. **Robot Destruction**:
    a. Lose life → Queue respawn if lives remain
    b. Permanent death at 0 lives
2. **Respawn Process**:
- Destroyed (turn 1) → Wait 3 turns → Respawn (turn 4)
3. **Upgrade Progression**:
    a. Kill-based eligibility (1 kill → Moving, etc.)
    b. `Upgrade()` function transforms robot (concrete implementation not shown)
4. **Victory Conditions**:

a. Single survivor: Declared winner
b. Multiple survivors: Draw after `maxTurns`
c. No survivors: All destroyed

```cpp
class Parser {
public:
    static SimulationConfig parseConfig(const std::string& filename);
};

SimulationConfig Parser::parseConfig(const std::string& filename) {
    std::ifstream file(filename);
    SimulationConfig config;
    std::string line;

    while (std::getline(file, line)) {
        // Remove leading/trailing whitespace
        line.erase(0, line.find_first_not_of(" \t"));
        line.erase(line.find_last_not_of(" \t") + 1);

        if (line.empty() || line[0] == '#') continue;

        size_t colonPos = line.find(':');
        if (colonPos != std::string::npos) {
            std::string key = line.substr(0, colonPos);
            // Remove whitespace from key
            key.erase(std::remove_if(key.begin(), key.end(), ::isspace), key.end());

            std::string value = line.substr(colonPos + 1);
            // Remove leading whitespace from value
            value.erase(0, value.find_first_not_of(" \t"));

            if (key == "MbyN") {
                std::istringstream iss(value);
                iss >> config.width >> config.height;
            } else if (key == "steps") {
                config.maxTurns = std::stoi(value);
            }
```

```
        } else {
            // Parse robot line
            std::istringstream iss(line);
            RobotInit robot;

            // First word is always the type
            iss >> robot.type;

            // The rest of the line until "random" or coordinates is the name
            std::string namePart;
            robot.name = "";
            while (iss >> namePart) {
                if (namePart == "random") {
                    robot.randomPosition = true;
                    robot.x = robot.y = -1;
                    break;
                } else if (isdigit(namePart[0])) {
                    // This is the x coordinate
                    robot.randomPosition = false;
                    robot.x = std::stoi(namePart);
                    iss >> robot.y;
                    break;
                } else {
                    if (!robot.name.empty()) {
                        robot.name += " ";
                    }
                    robot.name += namePart;
                }
            }
            config.robots.push_back(robot);
        }
    }
    return config;
}
```

File Parsing Logic
1. File Handling
Opens the file using std::ifstream.

Processes each line sequentially using std::getline.

2. Line Preprocessing
Trimming whitespace: Removes leading and trailing spaces/tabs.

Skipping comments/empty lines: Lines starting with # or empty lines are ignored.

3. Key-Value Parsing (for Simulation Settings)
If a line contains a colon (:), it is treated as a key-value pair:

Key extraction:

Left side of : is taken as the key.

Whitespace in the key is removed (e.g., "MbyN " → "MbyN").

Value extraction:

Right side of : is taken as the value.

Leading whitespace is trimmed.

Supported keys:

MbyN: Expects two integers (width and height).

Example: MbyN: 800 600 → config.width = 800, config.height = 600.

steps: Expects an integer (max simulation turns).

Example: steps: 1000 → config.maxTurns = 1000.

4. Robot Definition Parsing
If a line does not contain a colon, it is treated as a robot definition:

First word: Robot type (e.g., Explorer, Miner).

Subsequent words:

If a word is "random", the robot is assigned a random position (x = y = -1).

If a word is numeric, it is treated as the x-coordinate, and the next word is the y-coordinate.

Otherwise, words are concatenated into the robot's name.

Example:

Explorer R2D2 100 200 → Type: Explorer, Name: R2D2, Position: (100, 200).

Miner Bob random → Type: Miner, Name: Bob, Position: Random.

```
int main(int argc, char* argv[]) {
    srand(time(nullptr));

    try {
        std::string configFile = (argc > 1) ? argv[1] : "config.txt";
        std::string logFile = (argc > 2) ? argv[2] : "robot_war.log";

        SimulationConfig config = Parser::parseConfig(configFile);

        // Ensure minimum battlefield size
        if (config.width <= 0) config.width = MAP_WIDTH;
        if (config.height <= 0) config.height = MAP_HEIGHT;

        Simulation sim(config.width, config.height, config.maxTurns);

        for (const auto& robotCfg : config.robots) {
            auto robot = make_unique<GenericRobot>(robotCfg.name, sim.getBattlefield());
            if (robotCfg.randomPosition) {
                sim.addRobot(move(robot));
            } else {
                sim.addRobot(move(robot), robotCfg.x, robotCfg.y);
            }
        }

        sim.run(logFile);
        std::cout << "Robot War End. Results logged to " << logFile << "\n";

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
        return 1;
    }
    return 0;
}
```

1. Initialization

Random seed setup: srand(time(nullptr)) ensures random robot placement (if specified).

Default file handling:

If no config file is provided, uses "config.txt".

If no log file is provided, uses "robot_war.log".

2. Configuration Parsing

Loads simulation settings via Parser::parseConfig(configFile).

Ensures minimum battlefield size:

If width or height is invalid (≤ 0), falls back to MAP_WIDTH and MAP_HEIGHT.

3. Simulation Setup

Creates a Simulation object with parsed dimensions and max turns.

Adds robots based on configuration:

If randomPosition = true, places robot randomly.

Otherwise, places robot at specified (x, y) coordinates.

Robots are stored as unique_ptr for memory safety.

4. Execution & Logging

Runs the simulation (sim.run(logfile)).

Outputs results to the specified log file.

Prints confirmation ("Robot War End. Results logged to ...").

5. Error Handling

Catches exceptions (e.g., file errors, parsing failures) and prints an error message.

Returns 1 on failure, 0 on success.