HIGH LEVEL DESIGN: Roni Segal & Yuval Namir Barr

1. UML Class Diagram & UML Sequence Diagram

Added as an extra files.

2. Class Responsibilities

GameManager

The GameManager class orchestrates the entire gameplay.
It manages the board, players, shells, and tank algorithms.
Firstly, It loads game setups from input files.
Then, it processes each game turn, in which it applies actions, updates positions, handles collisions, and checks win conditions.
For debugging, it also print the current state of the board.

Board

The Board class represents the game arena as a two-dimensional grid of cells.
It provides access to cells by coordinates.
Most importantly, it supports wrapping positions around the board edges.

Cell

The Cell class represents a single tile on the board.
Each cell stores its coordinates and its terrain type (empty, wall, or mine).
It also manages terrain-specific states like if the cell is a wall, what it hit count, and ypdated it when needed.

Player

The Player class represents a player in the game.
It holds a set of tanks under the player's control and provides operations to add, remove, and access tanks.
A player is considered alive as long as at least one tank is active.

Tank

The Tank class models a single tank unit.
It maintains its position, direction, ammunition still in stock, and whether he is in shooting cooldown, or waiting for a backward movement.
It provides performing movement (like forward/backward movement, rotation and shooting).
for backward movement and shooting it store all the requirements check for those actions.

Shell

The Shell class models a single shell.
Due to requirements, the shell move in a straight line, in the direction of the tank at the time that it shot him.
It keeps track of its position, direction, and supports position updates after each turn.

Entity (abstract)

The Entity class is an abstract base class for all game entities placed on the board.
It defines a basic interface for getting type and position information.
Both Tank and Shell inherit from Entity.

Direction

The Direction module defines the possible directions tanks and shells can face.
It provides utilities to calculate next movement per direction, and support direction-based logic.
Directions are discretized into eight compass points: U, UR, R, DR, D, DL, L, UL.

TankAlgorithm (abstract)

The TankAlgorithm class defines an abstract interface for tank decision-making.
It declares the getNextAction() function that must be implemented by concrete algorithms.
It also provides utility functions for the tanks algorithms like functions that get the danger zones and safe zones around me.

BasicTankAlgorithm

The BasicTankAlgorithm implements a simple decision-making logic for a tank.
It focuses on escaping immediate danger zones, basic movement, and shooting when an enemy is nearby.
It does not use any advanced pathfinding techniques.

ChasingTankAlgorithm

The ChasingTankAlgorithm implements a more advanced decision-making strategy.
firstly, it start by checking if there is a need to escape immediate danger zones, or shoot when an enemy is nearby(using the basic algorithm).
If it doesn’t find any danger, it aims to chase and corner the opponent by computing paths toward the enemy tank.

Action

The Action enum lists all possible tank actions during a game turn.
These include movement (like forward/backward movement, rotation and shooting) and waiting.

3. Design Considerations

In designing the system, we focused on clean structure, clear division of responsibilities, and the use of efficient standard data structures (STL).
Each class is responsible for its own internal logic.
For example, the Tank class knows how to move, rotate, and shoot, but it does not update the overall game state — not even its own position in it!
The GameManager is the only class responsible for updating the global positionMap, tracking the locations of all tanks and shells.
Moreover, the GameManager serves as the central controller, coordinating between the board, players, tanks, shells, and algorithms.
This separation helps maintain clean architecture, where each component handles only what it truly owns.

We chose to use pointers for tanks and shells because these entities are referenced from multiple places.
This allows flexible management of tanks and shells, enabling runtime creation, deletion, and ownership transfer.
For example, a tank belongs to a player but also appears in the board's position map and is involved in collision detection.
By using pointers, we avoid duplicating entities, ensure consistent state, and control memory carefully by deleting each object exactly once, when necessary.

We introduced an abstract base class called Entity, from which both Tank and Shell inherit.
This enables unified management of movable game entities, particularly in data structures like positionMap.
Without Entity, code duplication would have been inevitable.

We introduced a Player class even though, for this assignment, each player controls only one tank.
This decision was made in anticipation of future expansions, where players may control multiple tanks simultaneously.
By structuring the system around Player objects from the beginning, we ensure the design remains scalable, clean, and ready for more complex gameplay scenarios.

To handle different behaviors, we created the TankAlgorithm abstract class with a pure virtual function getNextAction().
Concrete classes like BasicTankAlgorithm and ChasingTankAlgorithm implement the getNextAction method differently.
This design allows assigning different behaviors to tanks easily without modifying the main game logic.
Moreover, inside the ChasingTankAlgorithm, we reuse logic from BasicTankAlgorithm — before chasing an enemy, the tank must first handle threats and avoid dangers.
This approach helps avoid duplication of code for the same purpose.

In terms of data structures, we deliberately used STL containers:

- std::set<Shell*> stores active shells uniquely without duplicates.
- std::vector<Action> and std::vector<std::pair<int, int>> are used in algorithms to store movement plans and history.
- std::map<std::pair<int, int>, std::vector<Entity*>> efficiently maps board positions to the entities occupying them, greatly simplifying collision handling and allowing fast lookup of entities per cell.
  All containers manage their own memory, reducing the risk of manual memory errors.

Finally, the game loop in GameManager::run() is designed to be straightforward and extendable.
It processes turns, moves shells, rebuilds entity maps, checks for collisions, and evaluates win conditions.
This structure supports adding additional game rules in the future without restructuring the core logic.

Overall, the system is designed to be clean, modular, flexible, and easy to expand in future assignments.

alternative designs

We considered several alternative designs but chose the current approach for better modularity and flexibility:

- One alternative was to give each Tank a pointer to the Board. However, this would tightly couple the Tank to the Board, making future changes harder. Instead, the board is passed as a parameter when necessary, preserving loose coupling and clear responsibility.
- Another alternative was embedding the tank decision-making logic directly inside the Tank or GameManager. This was rejected because it would make the logic rigid and harder to extend. By defining a polymorphic TankAlgorithm interface, we allow adding new behaviors in the future easily without touching the rest of the game's code.
- It was also considered to have Player classes manage the shells fired by their tanks. However, this would create redundant complexity since shells naturally exist independently on the board after firing and could even be destroyed by other shells. Therefore, shells are managed globally by GameManager, simplifying collision handling and movement logic.

Additional Edge Case Considerations

While designing the system, we also identified specific edge cases that could potentially arise:

Shell and Tank One Tile Apart with Opposite Directions
In our current implementation, the GameManager does not detect a collision when a shell and a tank are located one tile apart and moving in opposite directions.
However, we are fully aware that this could be a theoretical edge case that the GameManager would need to handle in a more general solution.
That said, due to the way the tank algorithms are designed (to avoid moving into direct shell paths), such a situation does not actually occur during gameplay.

Given the fact that this edge case is practically impossible in the current system, we decided not to address it explicitly in this exercise.

Handling Mutual Shooting at Close Range

During the design of the collision handling system, we specifically tested the scenario where two tanks are positioned adjacent to each other and shoot at the same time.
In such a situation, both shells would meet between the two cells.
We decided that in this case, both shells should explode, and both tanks should be destroyed along with them.
This decision reflects the idea that the explosion occurs effectively between the two tiles, impacting and damaging both occupied cells.
By handling it this way, we ensure consistent and logical behavior during simultaneous close-range engagements.

4. Testing Approach

To validate our implementation, we designed a variety of tests covering both standard gameplay and edge cases.
Our goal was to ensure that the game logic behaves correctly under normal conditions, unusual scenarios, and invalid inputs.

We tested:

- Standard gameplay flow including tank movement, shooting, and game ending conditions.
- Shell collisions and complex interactions between tanks and shells.
- Board features like wall destruction, mine behavior, and board wrapping (position wrapping at edges).
- Handling of invalid input files and ensuring that the program exits cleanly on unrecoverable errors.
- Correct reporting of invalid actions ("Bad Actions") during gameplay.

By covering these areas, we made sure the system is robust, resilient, and behaves as expected under a wide range of conditions.

Test cases:

To validate our implementation, we designed a variety of test cases that cover both standard gameplay scenarios and critical edge cases:

Standard game flow: Two tanks positioned correctly, moving and shooting until a clear win or tie is determined.

Shell collision: Testing two shells fired toward each other and ensuring they explode upon meeting.

Shell pass: Testing two shells passing each other cells and ensuring they explode upon meeting.

Mutual tank destruction: Verifying that two adjacent tanks shooting simultaneously are both destroyed along with their shells, based on explosion logic.

Wrap-around movement: Ensuring tanks and shells correctly wrap around the board edges.

Shell exhaustion and countdown: Verifying that when both tanks don't have ammunition, the 40-turn countdown is activated and ends the game in a tie if no resolution occurs.

Invalid input files: Testing board loading errors, such as missing tanks or invalid characters, and confirming appropriate error handling and logging.

Wall destruction: Testing that walls break after two hits.

Bad Action: Testing that if there is invalid action, it is reported to the screen.

Unrecoverable: testing that if there is an unrecoverable input file, the program ends cleanly and an error message prints out to the screen. We did the unrecoverable test that there isn’t a tank for one of the players.

Shell above mine: Testing that if there’s a shell above a mine the mine doesn’t explode.

Almost Locked Player: Testing that if a player is almost locked between mines (hence his move opportunities is limited) he is able to get out of it.