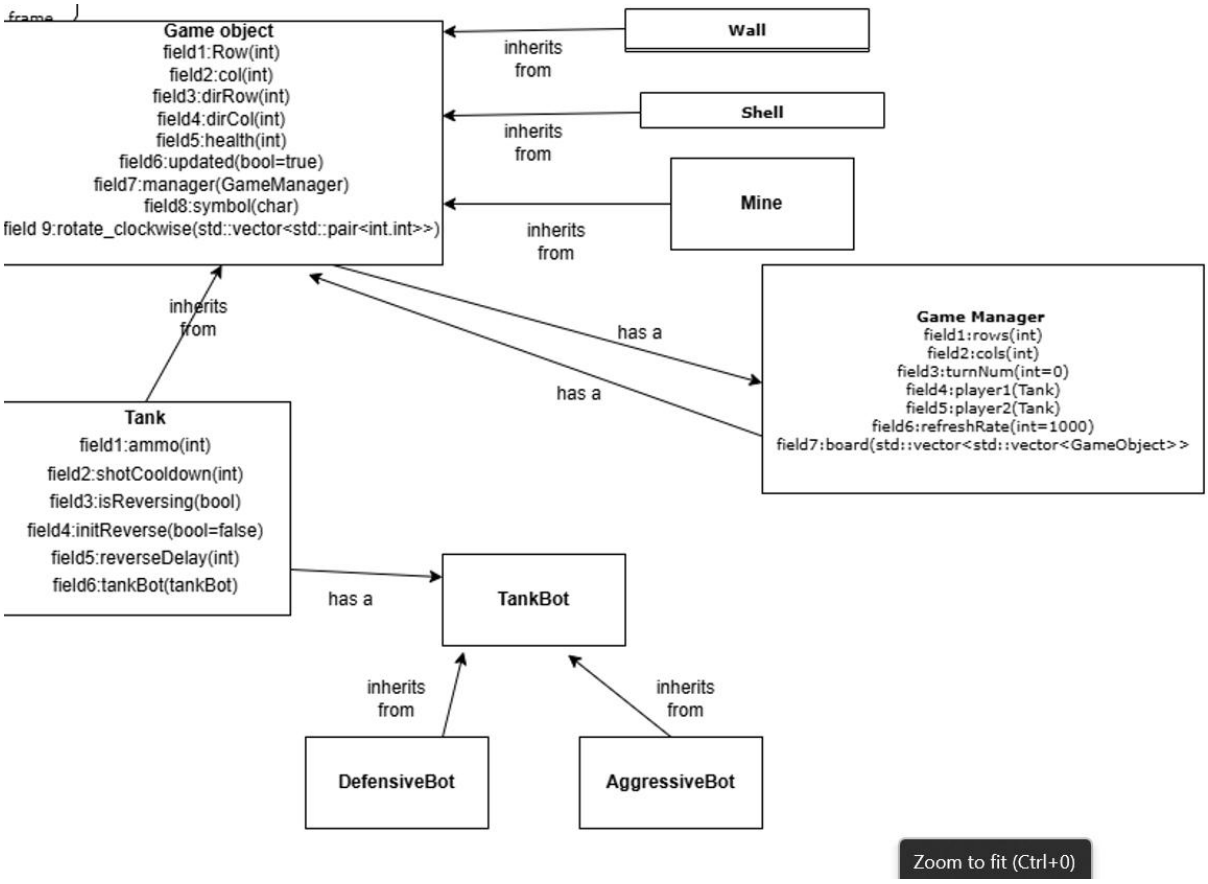
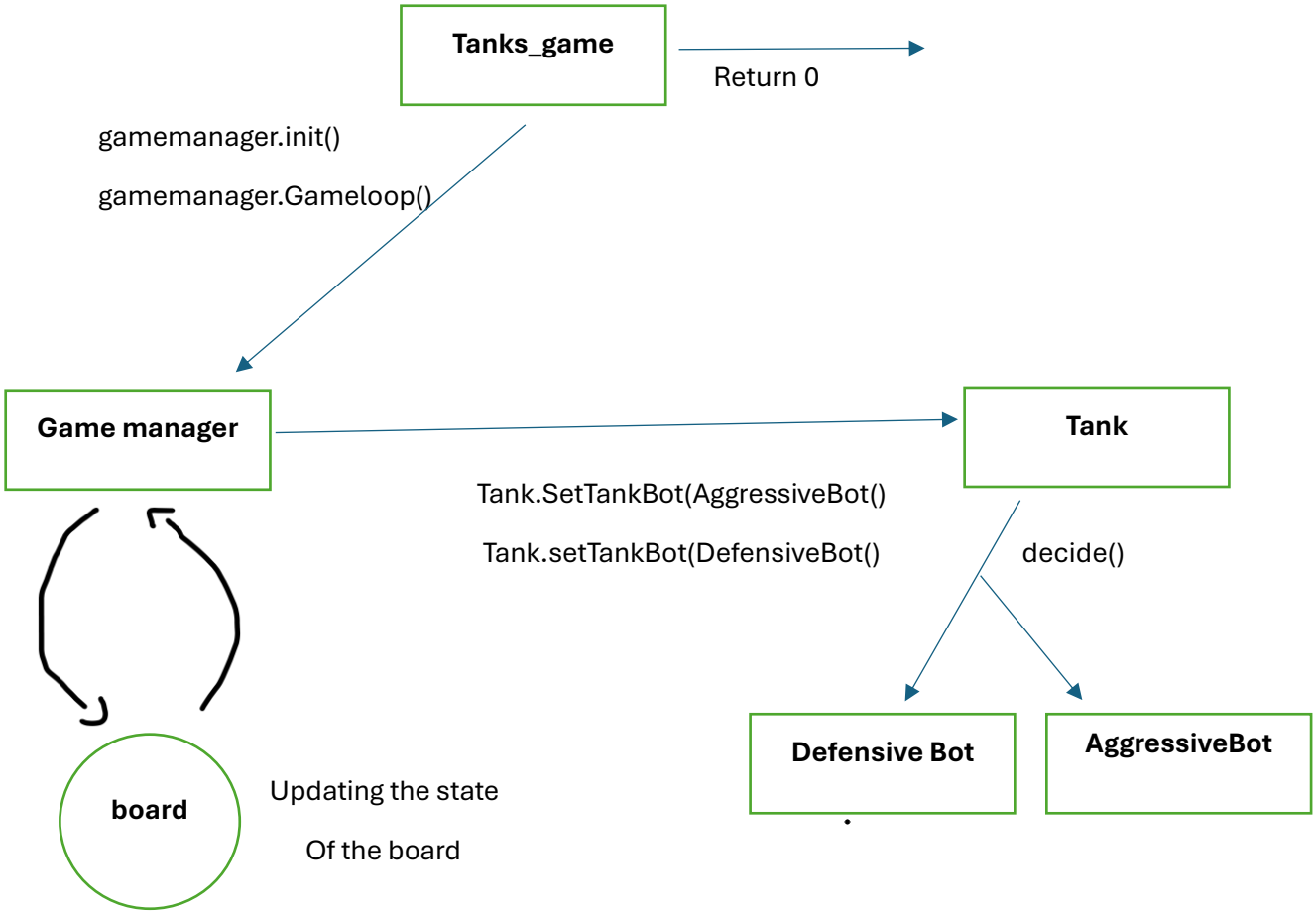
UML class design:



UML **sequence diagram** of the main flow of your program:



The **Tank Battle** game starts by receiving an input file that describes the game setup, including the board height and width, tank positions, and the representation of the board by the symbols given to us. The design follows an object-oriented approach, where the game functionality is split across several classes and interaction between them.

" Tank" class is responsible for managing the properties and actions of the tank, shooting, rotation, moving forward and reverse. "Tank Bot" class serves as an abstract base for different bot types" Aggressive Bot and Defensive Bot ". The bots determine the best way to behave and move based on the game state. The" Game Manager" class manages the game flow, handling turns, updating the board, and ensuring the interactions between the tanks and other game objects are processed correctly and game ending. Separating tank movement logic from the "Game Manager" make the design more modular and maintainable. "Aggressive Bot" uses the **BFS** algorithm to find the shortest path to the enemy tank, ensuring a balanced and fair gameplay experience. Having a single Game Manager class control the entire game flow, provides a simple structure that is easy to follow. However, this can make the class quite large and potentially harder to maintain as the game logic grows.

An alternative approach is Splitting the game logic into separate components, like having a Tank Manager and Turn Manager would distribute the responsibility, making each class focused on a specific concern. This follows the Single Responsibility Principle more strictly, but it might make the game flow harder to track as it requires coordination between multiple classes. In addition, the tank directly handles its movement and rotation based on player input or the bot's decision. This keeps the logic simpler and faster but might lead to more tightly coupled code. Having the tank's behaviour abstracted through the TankBot class allows for greater flexibility, as you can easily switch out different bot strategies (e.g., defensive, aggressive). However, this adds complexity to the design since now both the tank and the bot need to interact effectively.

The testing strategy includes unit tests to verify individual components like tank actions and bot decisions, integration tests for overall game flow, and edge case testing for scenarios like tanks running out of ammo or moving outside the board.