Tank Battle Game – Design Overview:

To implement the tank battle game, our initial approach was to store everything directly on a 2D grid and access the board directly. However, this created challenges when handling collisions between different object types (e.g., tanks and shells). We overcome those challenges by Introducing the following:

Object-Oriented Design

We restructured the design to use four separate vectors, each holding a distinct type of object:

- walls: barriers (destroyed after being hit by a shell two times)
- mines: Passive obstacles that destroy tanks on contact (they don't destroy shells)
- shells: Projectiles fired by tanks
- tanks: Player-controlled entities

Each of these object types inherits from an abstract GameObject base class, which provides shared functionalities such as:

- getPosition()
- Type checks like isWall(), isMine(), etc.

GameBoard

When the game starts, we parse the input and place all objects onto the GameBoard instance.
This class serves as the central data structure, managing object locations and providing access to the game state.

GameManager

The GameManager controls the overall flow:

1. Initializes the GameBoard and loads the grid.
2. Manages the simulation by requesting the next action from each tank algorithm.
3. Processes the tank action returned by the algorithm, including:
    - Movement (forward\backward)
    - Rotations
    - Shooting shells (and shell movement)
    - Collisions between tanks, walls, mines, and shells

AI Algorithms

We implemented two algorithms by inheriting from an abstract TankAlgorithm class:

- OffensiveAlgorithm: Aggressively chases and attacks the enemy.
- DefensiveAlgorithm: Prioritizes survival, evasion, and cautious engagement.

Each algorithm implements getNextAction(), allowing the game engine to ask each tank what it wants to do next.

All actions—valid or bad—are processed by the game loop, which uses the logic in the object classes to resolve outcomes and enforce the game rules.

Design Considerations & Alternatives:

Offensive Algorithm

Our offensive algorithm prioritizes aggression and pursuit. The core idea is:

- Always attempt to shoot when possible.
- Actively follow the direction of the opponent tank.
- Use Breadth-First Search (BFS) to plan up to 5 moves ahead, and execute the action if the path is considered safe.

This combination allows the offensive tank to intelligently chase and engage the opponent while avoiding recklessness.

Defensive Algorithm

The defensive strategy follows a priority-based decision hierarchy:

1. Evade immediate danger (e.g., incoming shells).
2. If a shell is approaching in the tank's direction, defend (e.g., by shooting or moving).
3. If not in danger but the enemy is in sight, attempt to shoot.
4. If no threats are detected and no targets are visible, find a safe or strategic position to hide or prepare.

This logic helps the tank survive longer and only engage when it's safe to do so.

Testing Approach:

To validate our logic, we followed a structured testing approach:

1. **Initial Simple Tests**:
    - We began by creating simple tests that concluded in a small number of steps. These tests were easy to calculate and verify, allowing us to ensure that our project behaved as expected.
    - Any issues identified during these tests were promptly fixed.
2. **Edge-Case Scenarios**:
    - Next, we developed test cases involving unusual or edge-case scenarios.
    - We utilized print commands and a display board helper function to monitor the game's steps and verify their alignment with our logic.
    - Detailed records of the game steps were maintained manually for each test.
3. **Deterministic Verification**:
    - Finally, we re-ran the tests multiple times throughout our work, to confirm that our algorithm behaved deterministically, as intended.

This approach helped us refine both the evasion logic and predictive movement handling, thereby improving the overall reliability of our project.

UML sequence diagram of the main flow of our program:

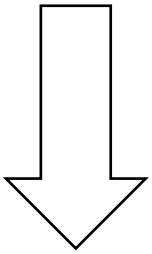
The textual description

```
title Tank Battle - Main Flow

actor main

participant GameManager

participant GameBoard

participant OffensiveAlgorithm

participant DefensiveAlgorithm

participant DirectionUtil

main->GameManager: initialize()

GameManager->GameBoard: loadFromFile()

GameManager->OffensiveAlgorithm: constructor

GameManager->DefensiveAlgorithm: constructor

return true

loop Game loop
```

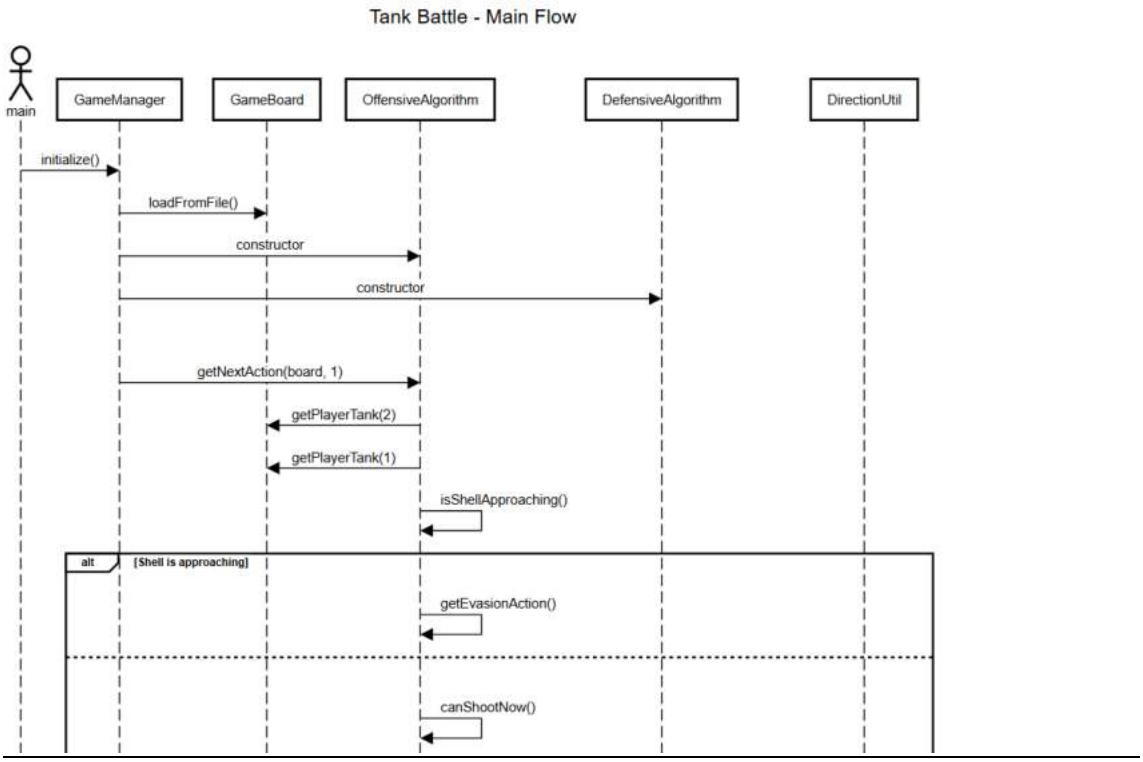
```
GameManager->OffensiveAlgorithm: getNextAction(board, 1)
OffensiveAlgorithm->GameBoard: getPlayerTank(2)
OffensiveAlgorithm->GameBoard: getPlayerTank(1)
OffensiveAlgorithm->OffensiveAlgorithm: isShellApproaching()
alt Shell is approaching
    OffensiveAlgorithm->OffensiveAlgorithm: getEvasionAction()
    return evasionAction
else
    OffensiveAlgorithm->OffensiveAlgorithm: canShootNow()
    OffensiveAlgorithm->OffensiveAlgorithm: enemyInSight()
    alt Can shoot and enemy in sight
        return "SHOOT"
    else
        OffensiveAlgorithm->DirectionUtil: getMovement(direction)
        OffensiveAlgorithm->OffensiveAlgorithm: isInDirection()
        alt isInDirection is true
            OffensiveAlgorithm->GameBoard: wrapCoordinates()
        else
            OffensiveAlgorithm->OffensiveAlgorithm: getBestRotationToEnemy()
            alt Needs rotation
                OffensiveAlgorithm->OffensiveAlgorithm: getRotationAction()
                return rotationAction
            else
                OffensiveAlgorithm->OffensiveAlgorithm: getChaseAction()
                alt Chasing:Calculate shortest path
                  OffensiveAlgorithm -> OffensiveAlgorithm:bfsPath: findind Partial BFS
                return chaseAction
            end
        end
    end
end

GameManager->OffensiveAlgorithm: processTankAction(action)

GameManager->DefensiveAlgorithm: getNextAction(board, 2)
DefensiveAlgorithm->GameBoard: getPlayerTank(2)
```

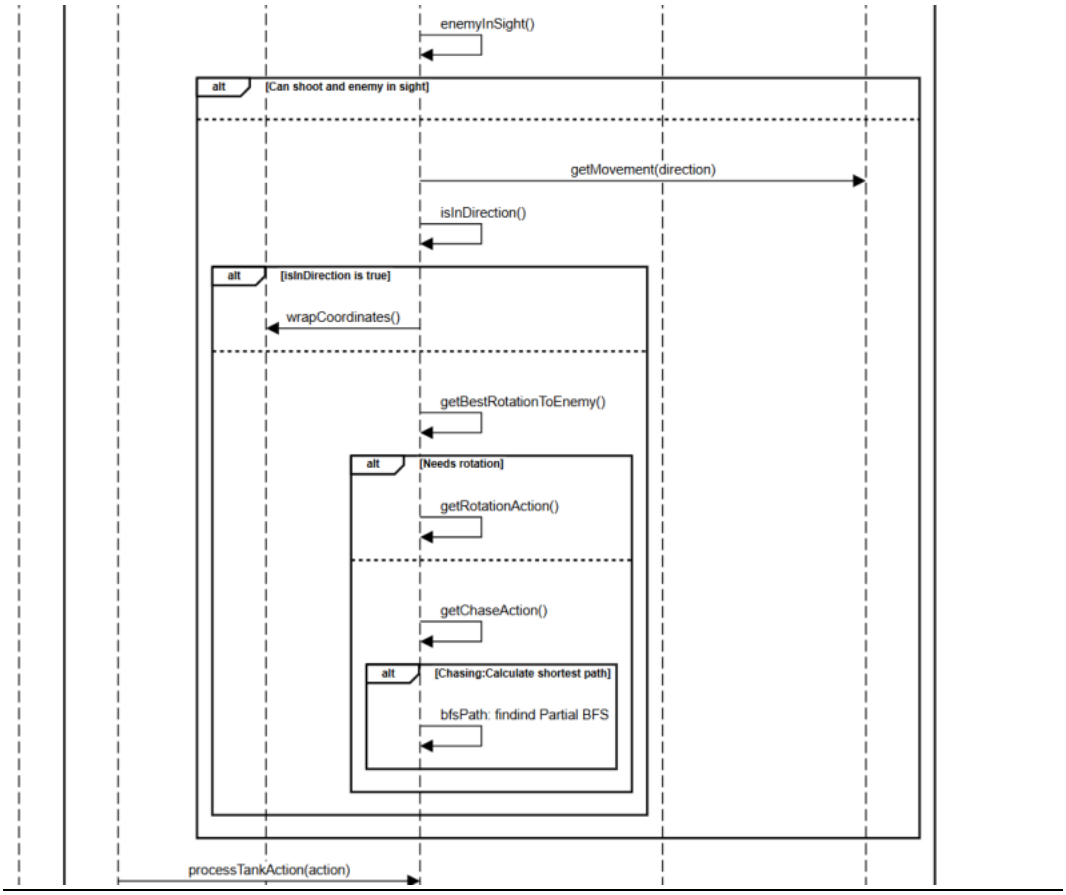
```
    DefensiveAlgorithm->DefensiveAlgorithm: isInImmediateDanger()
    alt In immediate danger
        DefensiveAlgorithm->DefensiveAlgorithm: handleImmediateDanger()

    end
    DefensiveAlgorithm->DefensiveAlgorithm: canSafelyShootOpponent()
    alt Can safely shoot
        return "SHOOT"
    else
        DefensiveAlgorithm->DefensiveAlgorithm: getDefaultSafeMove()
        DefensiveAlgorithm->DefensiveAlgorithm: isInDirection()
        DefensiveAlgorithm->DefensiveAlgorithm: findSafeNeighbor()
        return defaultSafeMove
    end

    GameManager->DefensiveAlgorithm: processTankAction(action)

    GameManager->GameBoard: processShells()
    GameManager->GameBoard: checkCollisions()
    GameManager->GameBoard: checkGameEnd()
end
```

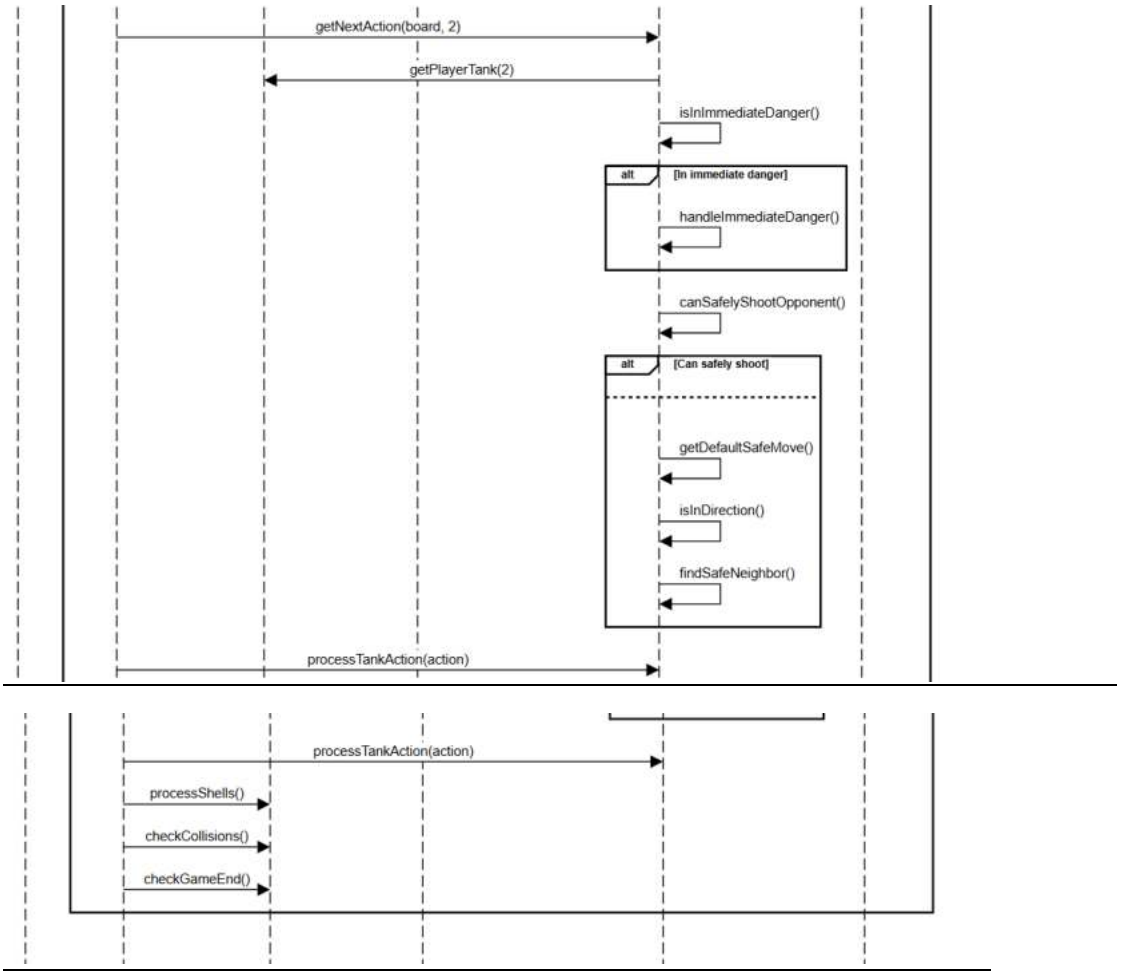
the resulting sequence diagram



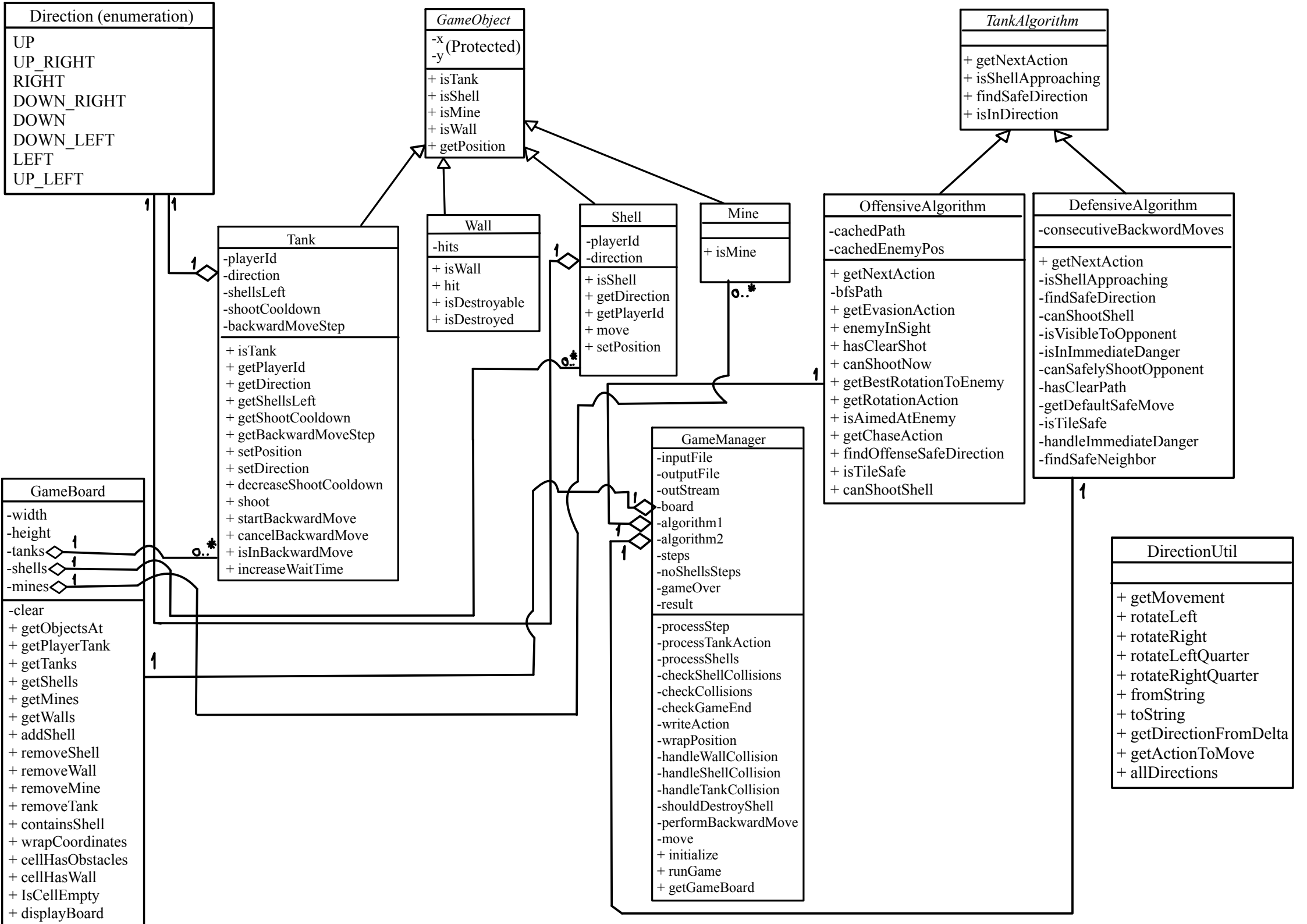
(distributed over multiple images because of its size)


Tank Battle - Main Flow
main
GameManager
GameBoard
OffensiveAlgorithm
DefensiveAlgorithm
DirectionUtil
initialize()
loadFromFile()
constructor
constructor
getNextAction(board, 1)
getPlayerTank(2)
getPlayerTank(1)
isShellApproaching()
alt
[Shell is approaching]
getEvasionAction()
canShootNow()


enemyInSight()
alt
[Can shoot and enemy in sight]
getMovement(direction)
isInDirection()
alt
[isInDirection is true]
wrapCoordinates()
getBestRotationToEnemy()
alt
[Needs rotation]
getRotationAction()
getChaseAction()
alt
[Chasing:Calculate shortest path]
bfsPath: findind Partial BFS
processTankAction(action)



UML class diagram of our design:


Direction (enumeration)
UP
UP_RIGHT
RIGHT
DOWN_RIGHT
DOWN
DOWN_LEFT
LEFT
UP_LEFT
GameObject
-x
-y (Protected)
+ isTank
+ isShell
+ isMine
+ isWall
+ getPosition
TankAlgorithm
+ getNextAction
+ isShellApproaching
+ findSafeDirection
+ isInDirection
Tank
-playerId
-direction
-shellsLeft
-shootCooldown
-backwardMoveStep
+ isTank
+ getPlayerId
+ getDirection
+ getShellsLeft
+ getShootCooldown
+ getBackwardMoveStep
+ setPosition
+ setDirection
+ decreaseShootCooldown
+ shoot
+ startBackwardMove
+ cancelBackwardMove
+ isInBackwardMove
+ increaseWaitTime
Wall
-hits
+ isWall
+ hit
+ isDestroyable
+ isDestroyed
Shell
-playerId
-direction
+ isShell
+ getDirection
+ getPlayerId
+ move
+ setPosition
Mine
+ isMine
OffensiveAlgorithm
-cachedPath
-cachedEnemyPos
+ getNextAction
-bfsPath
+ getEvasionAction
+ enemyInSight
+ hasClearShot
+ canShootNow
+ getBestRotationToEnemy
+ getRotationAction
+ isAimedAtEnemy
+ getChaseAction
+ findOffenseSafeDirection
+ isTileSafe
+ canShootShell
DefensiveAlgorithm
-consecutiveBackwordMoves
+ getNextAction
-isShellApproaching
-findSafeDirection
-canShootShell
-isVisibleToOpponent
-isInImmediateDanger
-canSafelyShootOpponent
-hasClearPath
-getDefaultSafeMove
-isTileSafe
-handleImmediateDanger
-findSafeNeighbor
GameBoard
-width
-height
-tanks
-shells
-mines
-clear
+ getObjectsAt
+ getPlayerTank
+ getTanks
+ getShells
+ getMines
+ getWalls
+ addShell
+ removeShell
+ removeWall
+ removeMine
+ removeTank
+ containsShell
+ wrapCoordinates
+ cellHasObstacles
+ cellHasWall
+ IsCellEmpty
+ displayBoard
GameManager
-inputFile
-outputFile
-outStream
-board
-algorithm1
-algorithm2
-steps
-noShellsSteps
-gameOver
-result
-processStep
-processTankAction
-processShells
-checkShellCollisions
-checkCollisions
-checkGameEnd
-writeAction
-wrapPosition
-handleWallCollision
-handleShellCollision
-handleTankCollision
-shouldDestroyShell
-performBackwardMove
-move
+ initialize
+ runGame
+ getGameBoard
DirectionUtil
+ getMovement
+ rotateLeft
+ rotateRight
+ rotateLeftQuarter
+ rotateRightQuarter
+ fromString
+ toString
+ getDirectionFromDelta
+ getActionToMove
+ allDirections