# Tank Battle Game - High-Level Design

## 1. Introduction

This document outlines the high-level design for the Tank Battle Game assignment. The game simulates two tanks battling on a 2D board, with features including movement, shooting, collision detection, and algorithm-driven tank behavior.

## 2. Class Diagram

[PLACEHOLDER FOR UML CLASS DIAGRAM]

## 3. Sequence Diagram for Main Game Flow

[PLACEHOLDER FOR UML SEQUENCE DIAGRAM]

## 4. Design Considerations and Alternatives

### 4.1 Game Management

* The Game Manager acts as a central coordinator, owning the game loop and managing the lifecycle of all game objects.
* Chosen over distributed control to enforce the requirement that algorithms can't modify game state directly
* Allows centralized validation of actions and tracking of invalid moves

### 4.2 Game State

* Separation between Board (static environment), Tank/Shell (dynamic objects) -
  + Tanks and shells perform action on each game step, requiring efficient handling (doesn’t require changing the game board representation on every step).
  + Only change board when a wall/mine state changes.
* Board handles position wrapping, objects not aware of board dimensions and therefore require the game manager mediation to determine the real positions on the board.
* Chosen over a unified state class as the game state is currently simple. This allows to avoid a level of indirection that needs to be updated on each step.

### 4.3 Algorithms

* A very extensive base class with basic utils for making algorithm decisions - prevents code duplications and allows the derived classes to focus on core mechanics and specialized priority management.

### 4.4 Collision Handler

* Dedicated class for detection and resolution of various collision types.
* Centralizes complex collision logic that would otherwise clutter GameManager.
* Storing last known position in object classes in order to be able to handle path collisions.

### 4.5 Utility Classes

* Contains reusable components like Point, Direction, and Action to avoid duplication.
* Provides consistent handling of wrapped coordinates and direction calculations
* The MidPoint class is derived from Point and provides a special handling for positions between adjacent cells, enabling the collision handler to avoid inaccurate float representation.

## 5. Testing Approach

### 5.1 Unit Testing

Each class is tested using Google Test framework with:

* Tests for constructors and basic properties
* Tests for core functionality under normal conditions
* Boundary and edge case testing

### 5.2 Integration Testing

Component interactions are tested by:

* Testing GameManager with mock algorithms
* Verifying the complete game cycle
* Validating file input/output operations

### 5.3 Manual Testing with Visualization

The visualization component serves as both a bonus feature and a valuable testing tool:

* **Game State Verification**: The visual representation allows direct inspection of game state after each step
* **Collision Detection**: Visualizing object movements and collisions helps verify complex interaction scenarios
* **Algorithm Behavior Analysis**: Visual patterns of tank movement help validate algorithm decision-making
* **Edge Case Identification**: Unusual game states become more apparent when visualized
* **Debugging Aid**: The step-by-step visual replay simplifies tracing the source of unexpected behaviors
* **Mock algorithms:** creating a Mock Class with the option to set a sequence of actions in order to test specific test scenarios.