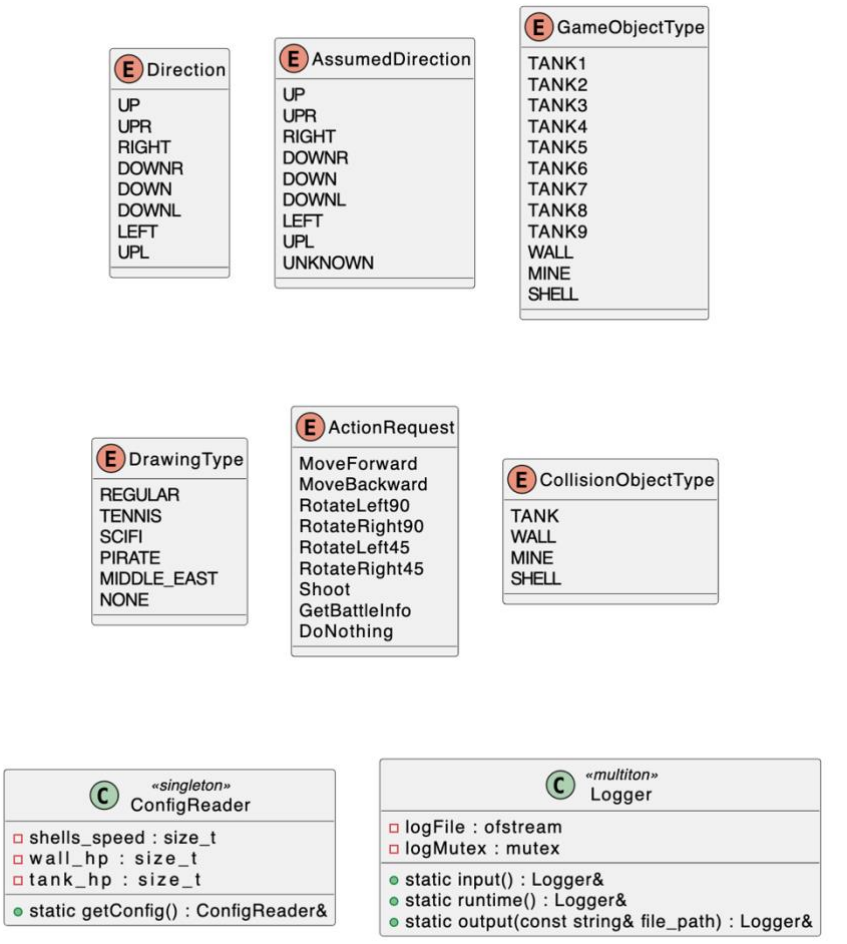
Tanks Game – High-Level Design

Class Design UML

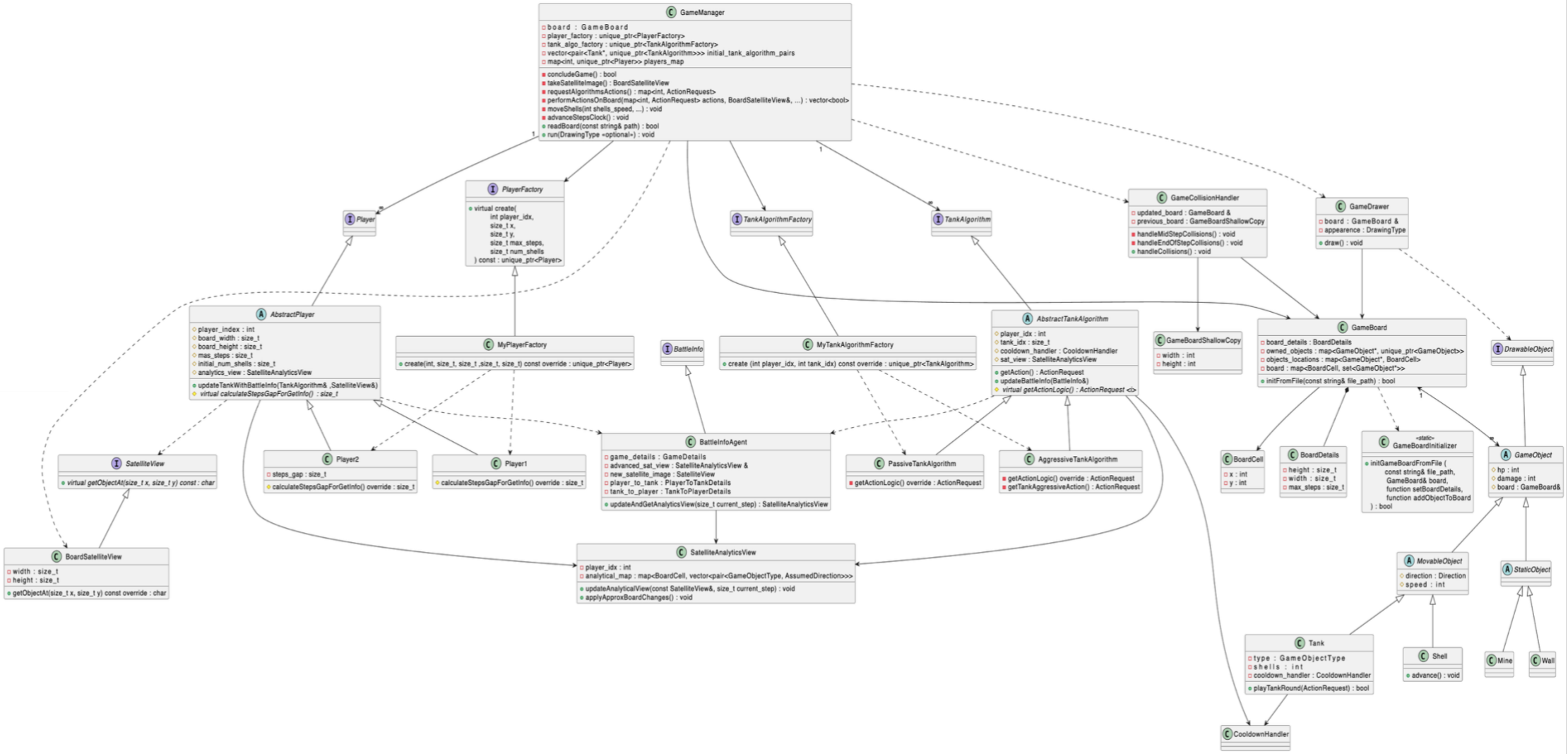
Our design follows Object-Oriented Programing (OOP) principles, ensuring clear separation of responsibilities and promoting modularity and scalability. It makes use of Enums to represent fixed categories, such as object types and directions, improving readability and type safety. The architecture organizes tanks, shells and other game objects into distinct reusable classes, supporting easy maintenance and future feature expansions. The design stays faithful to the assignment's provided skeleton, fully compatible with the given interfaces and easy to extend through them.

For clarity, we split the design diagram to 2 parts:

1. Enums and centralized-access classes:



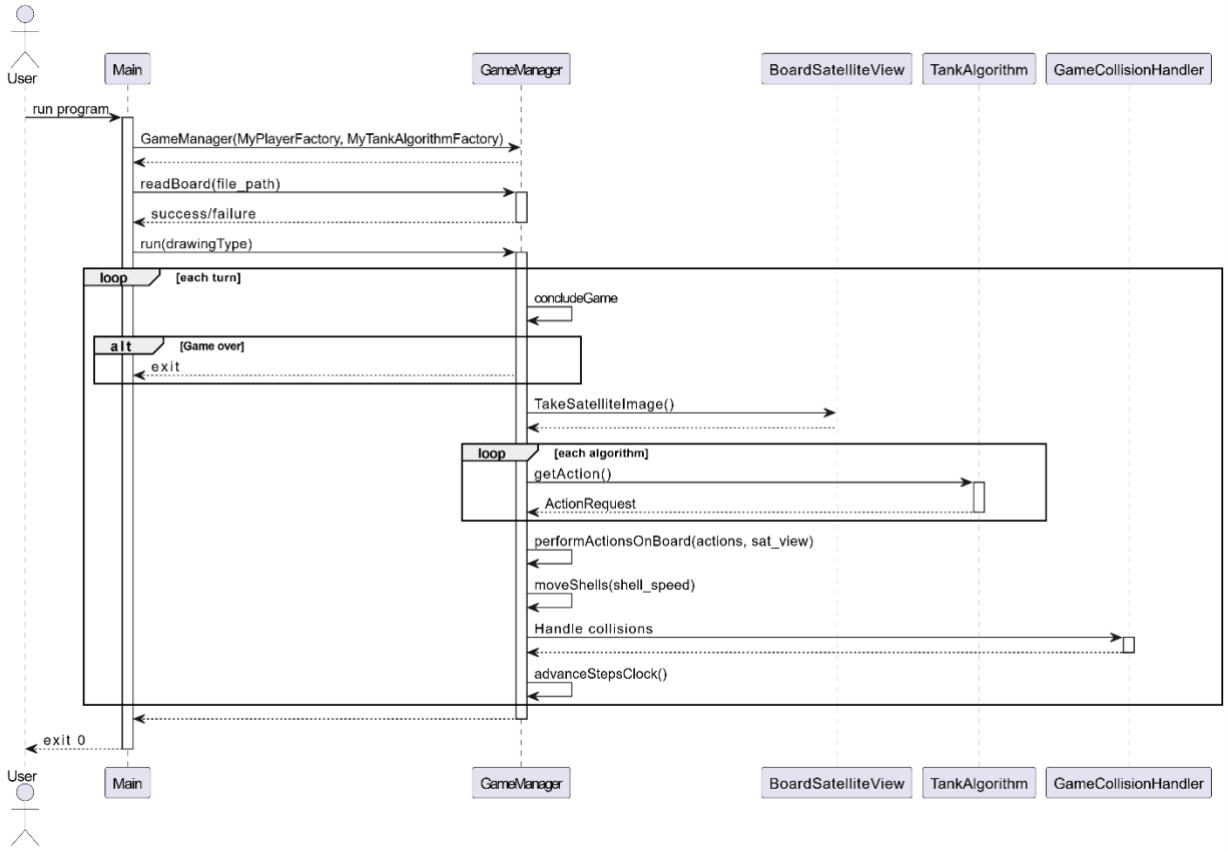
2. Structured Game Design:
   These classes use the Enums and the Logger and ConfigReader classes.



Work-Flow UML

The sequence diagram captures the high-level execution flow starting from main and driven entirely by the GameManager, emphasizing three key design principles:

- **Single entry point** – GameManager is the only component initialized directly in main, and it orchestrates all gameplay behaviour from start to finish.
- **Structured step logic** – Each game step follows a fixed sequence: checking game status, collecting actions from tank algorithms, then applying movement and collision handling. This predictability makes the game loop easy to debug and test.
- **Encapsulated behaviour** – Tanks determine their next move independently via TankAlgorithm, while collision resolution and board updates are managed centrally, enforcing a clear separation of roles.



Design Explanations

- Separation of responsibilities:
  - GameManager handles only game logic each step, using CollisionHandler for handling collisions.
  - GameBoard is the sole owner of all GameObjects and delegates the input file parsing logic to a designated static class - GameBoardInitializer.
  - Each TankAlgorithm determines the next ActionRequest for its corresponding Tank.
  - Tanks ActionRequest handling logic is internal.
- Simple GameObjects class tree:
  - All pieces of the game inherit from GameObject.
  - Objects that don't move use StaticObject.
  - Objects that can move use MovableObject.

  This logic avoids repeating code for health, direction, damage and speed.
- Data-driven collision handling:
  GameCollisionHandler uses two static maps (`explosion_map`, `prevention_map`) instead of hardcoded `switch` blocks, simplifying addition of new object types.
- Logger singleton:
  A single, thread-safe Logger object writes all log messages, so we don't have scattered `cout` calls and we avoid race conditions.
- Clear win/tie conditions:
  GameManager ends the match in any of these cases:
  - All sides lose their tanks → the only remaining player that has tanks wins.
  - All remaining tanks run out of shells, if no tank is destroyed within the next 40 turns* ->tie
  - Game reached the maximum number of steps without a clear winner* -> tie.
  - All sides lose their last tank in the same turn** → tie.

  * Once the game reaches a state where no tank has any remaining shells, it will end based on the minimum steps between point 2 and 3.

  ** A turn is the tank-movement step, divided into two half-turns (shell turn). If Player 1's final tank is destroyed in the first half-turn and Player 2's in the second half-turn of that same turn, it counts as a simultaneous knockout, resulting in a tie.

  Alternatives considered:
- Entity Component System (ECS):
  Build objects out of small "components" instead of an inheritance tree. This gives lots of flexibility, but for our tiny game it would add a lot of extra code and setup we don't really need.
- After-Act Drawing:
  Instead of GameDrawer checking the board every turn, it could wait for the game to end, read both input and output files and draw the game according to the files.
- A* search instead of Dijkstra:

A* can reach a target faster because it guesses the right direction, but plain Dijkstra is already quick enough here and is simpler to understand and debug.

- Reusing GameBoard in Player and TankAlgorithm classes:
  To provide a view of the board, the Player and TankAlgorithm use a new class called SatelliteAnalyticsView, which offers a simplified version of the board without detailed information and without GameObjects ownership. Alternatively, we considered adapting the existing GameBoard class to meet these new requirements.

Testing Approach Explanation

- Random Tests:
  Created a random board generator, to create and test several different scenarios.
- Automated Tests:
  We developed a script that executes extensive and thorough tests, covering a wide range of scenarios including edge cases such as simultaneous shell collisions, invalid tank movements, mine detonation chains, and game-ending conditions. The script also handles input verification and output validation to ensure correctness.
- Logging validation:
  output log files are parsed to ensure illegal moves are flagged correctly and winner/tie messages are produced.