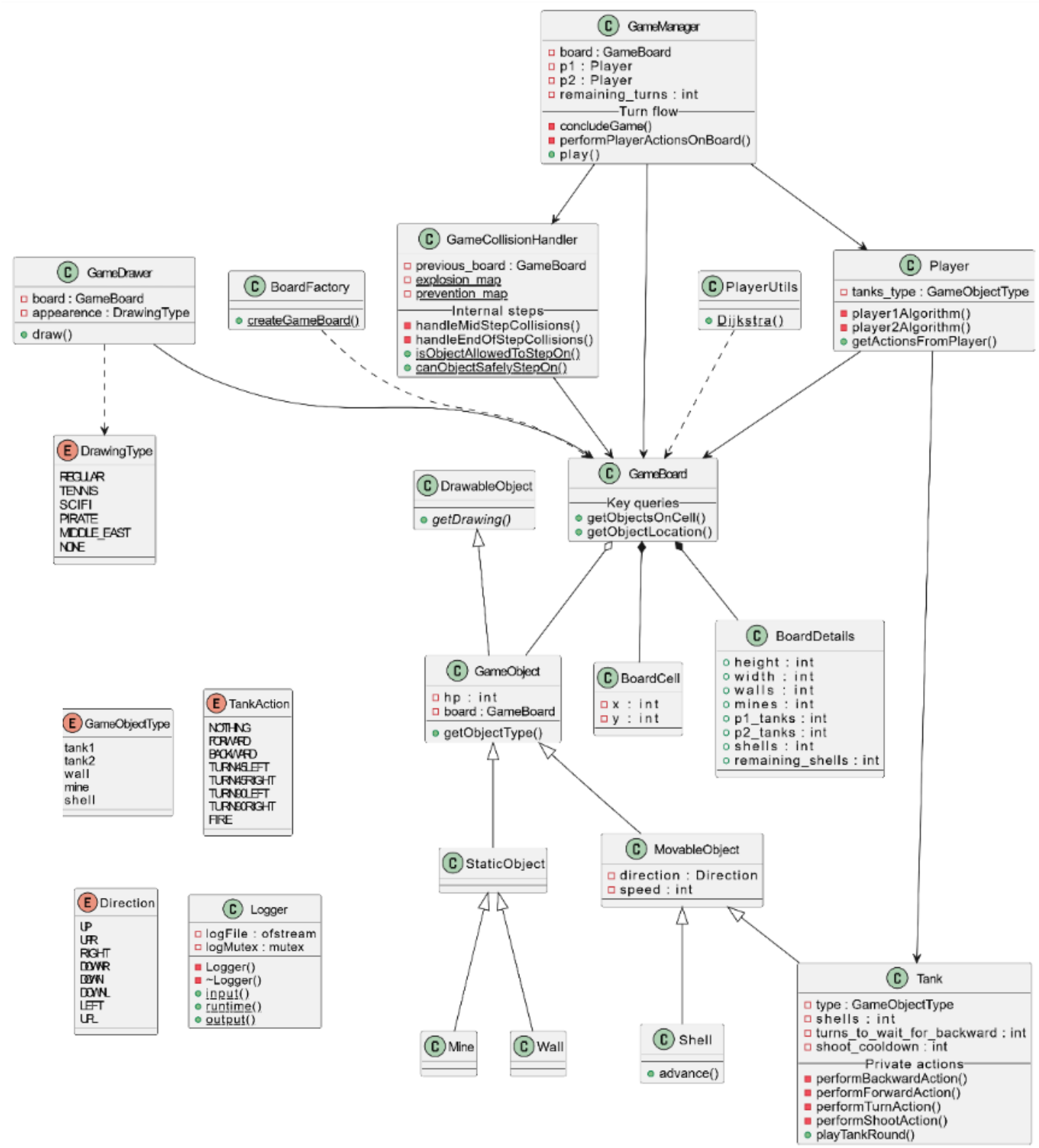
Tanks Game – High-Level Design

Class Design UML

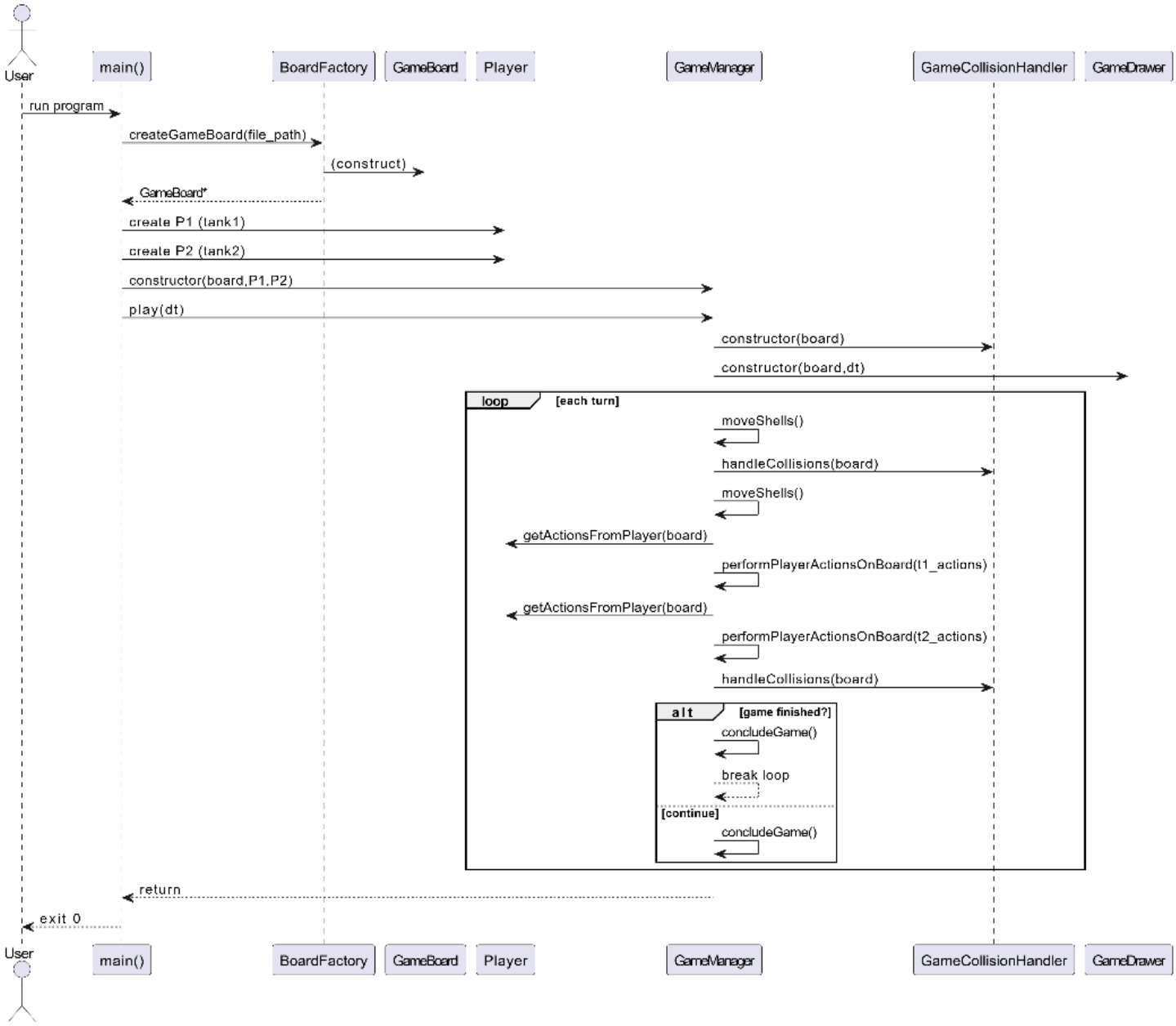
Our design follows Object-Oriented Programming (OOP) principles, ensuring clear separation of responsibilities and promoting modularity and scalability. It makes use of enums to represent fixed categories, such as object types and directions, improving readability and type safety. The architecture organizes tanks, shells and other game objects into distinct, reusable classes, supporting easy maintenance and future feature expansion.



Main Flow UML

The sequence diagram shows the linear flow of calls coming out from GameManager, making three main design ideas clear:

- **Central control** – One coordinator handles all state changes.
- **Deterministic loop** – Fixed call order per turn ensures identical input yields identical output, making tests repeatable.
- **Clear separation** – Collision checks, and (optional) drawing sit in separate lanes.



Design Explanations

- Separation of responsibilities:
  - GameManager handles only game logic each turn, using CollisionHandler for handling collisions.
  - Players choose their next move.
  - Tanks action logic is internal.
  - BoardFactory reads a text file and builds a board.
- Simple class tree:
  - All pieces of the game inherit from GameObject.
  - Things that don't move use StaticObject.
  - things that can move use MovableObject.

  This logic avoids repeating code for health, direction, 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:
  - One side loses all its tanks → the other side wins.
  - Both sides lose their last tank in the same turn* → tie.
  - All remaining tanks run out of shells; if no tank is destroyed within the next 40 turns, the game is declared a tie.

  * 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 gam 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.
- Smart pointers (std::shared_ptr) vs. manual deletes:
  Let C++ manage object lifetimes automatically, which removes the risk of leaks, but it also adds reference-count overhead and would change many signatures. For this assignment we kept raw pointers plus clear ownership rules to stay lightweight.

Testing Approach Explanation

- Random tests:
  Created a random board generator, to create and test several different scenarios.
- Scenario coverage:
  fixtures cover corner cases; simultaneous shell collisions, invalid tank moves, mine detonation chains, and game-end conditions.
- Logging validation:
  output log files are parsed to ensure illegal moves are flagged correctly and winner/tie messages are produced.

Itai Missri, 322330820
Moshe Zeitouny, 211388913