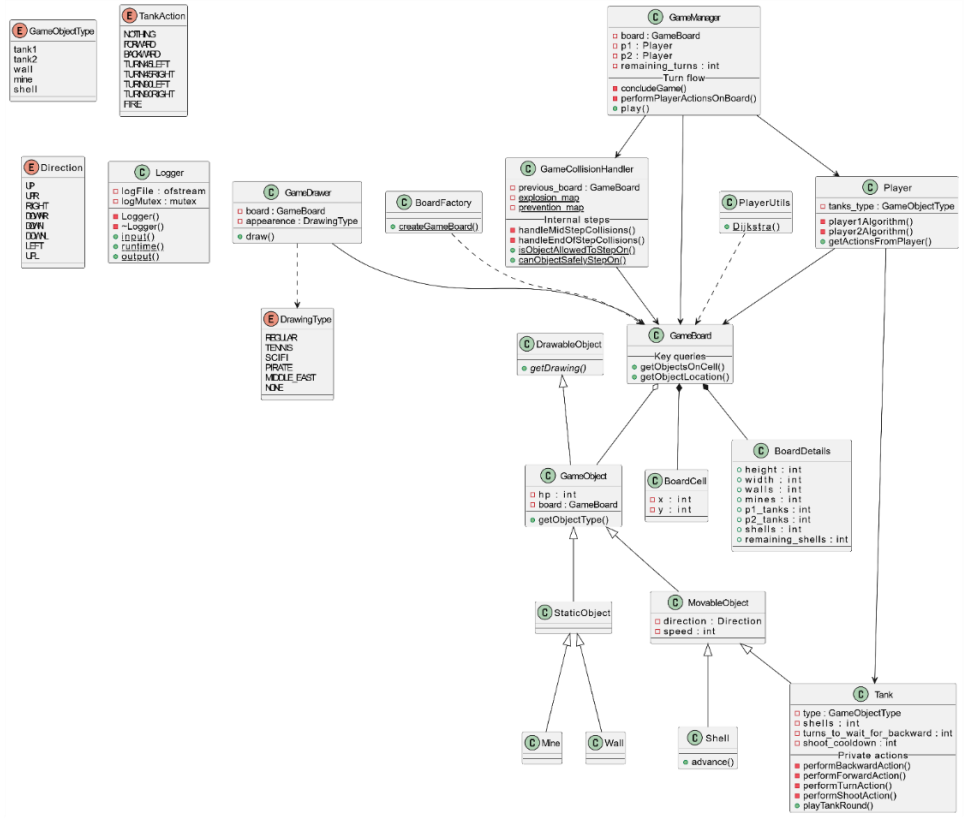
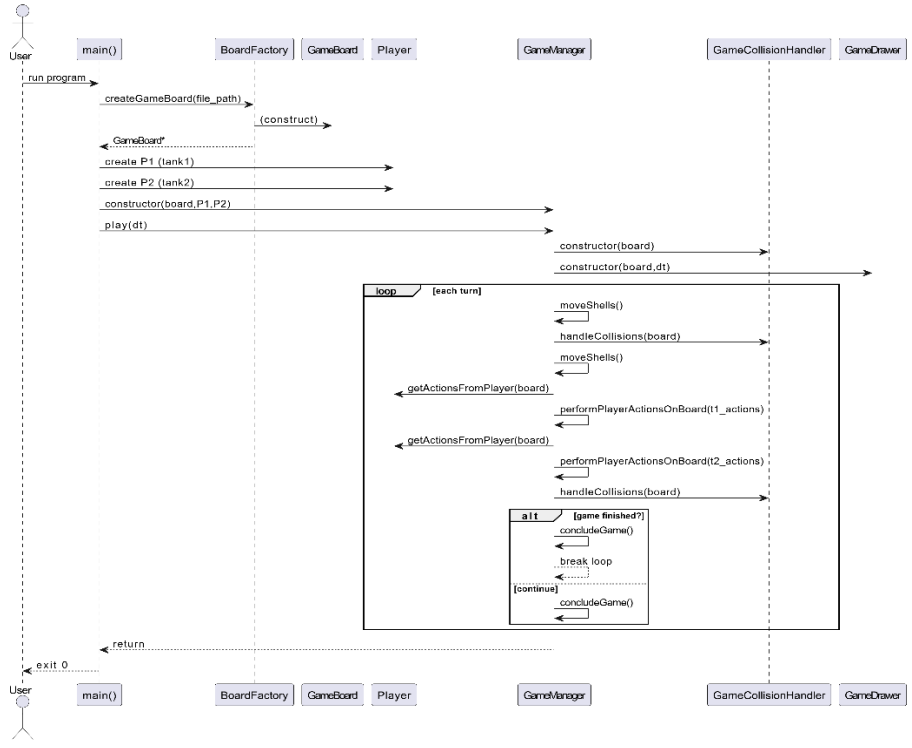
Tanks Game – High-Level Design

1. Class Design UML



2. Main Flow UML



3. **Design Explanations**

- Separation of responsibilities- GameManager runs each turn; each Tank just chooses its next move. Tanks never change the board by themselves—only the manager does.
- 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 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.
- Factory for reproducible boards- BoardFactory reads a text file and builds a board. Because the same file always makes the same board, we can run repeatable unit and integration tests.
- 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:
  1. One side loses all its tanks → the other side wins.
  2. Both sides lose their last tank in the same turn* → tie.
  3. 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 shell-movement step, divided into two half-turns. 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.
- Observer pattern- Instead of GameDrawer checking the board every turn, it could "listen" for board-updated events and redraw only when something changes. That would loosen the link between logic and rendering, but we'd first have to build an event system.
- 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.

4. **Testing Approach Explanation**

- Unit tests- focus on isolated logic such as BoardCell operators, `Tank::playTankRound` validation rules, and collision map lookups.
- Deterministic integration tests- each text fixture processed by BoardFactory creates a board with a known layout; test cases run multiple full turns and compare the resulting board snapshot to an expected baseline.

- 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.