1. **Consistent and unambiguous functional requirements for both client and server. Don't forget to include the constraints on tanks and bullets as requirements for the server.**

Client

1. Display the 16 by 16 dimension grid
2. Have each cell contain one entry
3. Show game state
4. Bullet has ID of tank that followed it
5. Provide control interface with buttons for each control
   1. Shoot bullet
      1. Shoot bullet when phone is shaken.
   2. Bullet strength
   3. Move tank
   4. Rotate tank
6. Client can connect to server
7. Client can leave server
8. Store game replay
   1. Store the time-stamped game states in a SQLitedatabase and it should be able to replay the recorded game.
   2. Replay must be time driven and in real time or n times faster.
9. Wall is destructible or indestructible

Server

1. SERVER RULES TO BE ENFORCED
   1. Tank can move once in every X seconds (X = 0.5)
   2. Tank can fire once in every Y seconds (Y = 0.5)
   3. Only Z fired bullets from a given tank can be in the game at the same time (Z = 2)
   4. Tank can make only one turn per step
   5. Tank can only move forward or back relative to its current direction. No sideways movements.
   6. Tank must have ID
2. One or more clients can connect to existing server
3. Server returns a grid and a timestamp
4. **Use-case diagram (showing actors and potential use-cases you have identified so far).**

Primary Actor

1. Player

Supporting Actors

1. Server
2. SQL Database

Goals

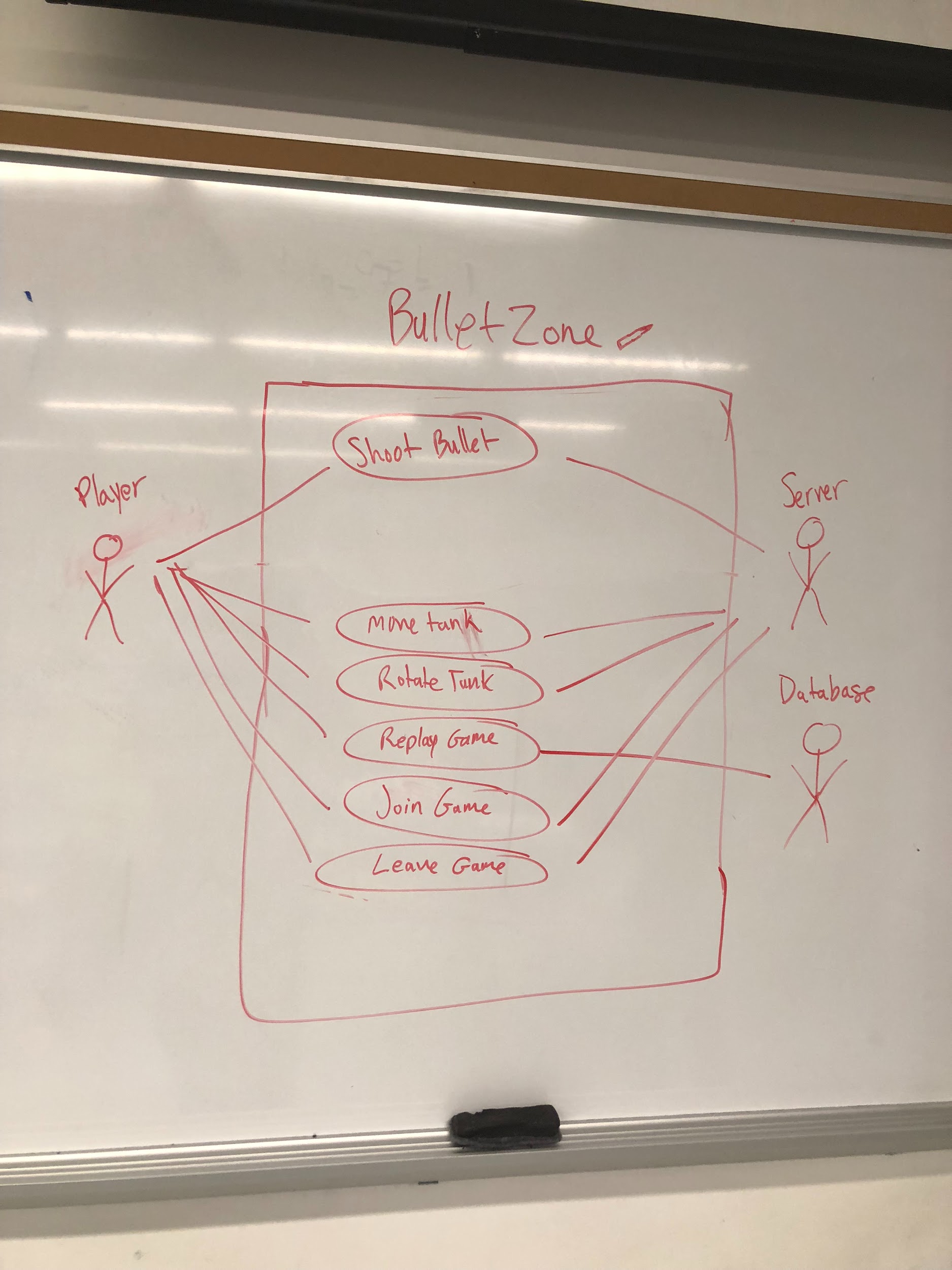
Player

1. Shoot bullet - high
   1. Precondition = Bullet is created and assigned to tank ID
   2. Postcondition = Hit tank and cause damage or miss, potentially damage wall

1. Change bullet strength - medium
   1. Precondition = Bullet exists and has set default damage value
   2. Postcondition = Bullet strength value has changed
2. Move tank - high
   1. Precondition = Tank exists and has ID and position in grid
   2. Postcondition = Position is updated
3. Rotate tank - high
   1. Precondition = Tank exists, has position, and has set direction
   2. Postcondition = Position is the same but direction has changed

1. Replay game - low
   1. Precondition = Game and all recorded values have been properly stored in database.
   2. Postcondition = Start new game available or watch replay again
2. Player can join game - high
   1. Precondition = Game exists and player exists
   2. Postcondition = Player has joined the game and has tank created.

1. Player can leave game - high
   1. Precondition = Player exists or has ended game
   2. Postcondition = Player has been removed from server.

****

1. **(At least) one main success scenario for (at least) one non-trivial use-case for the client app.**

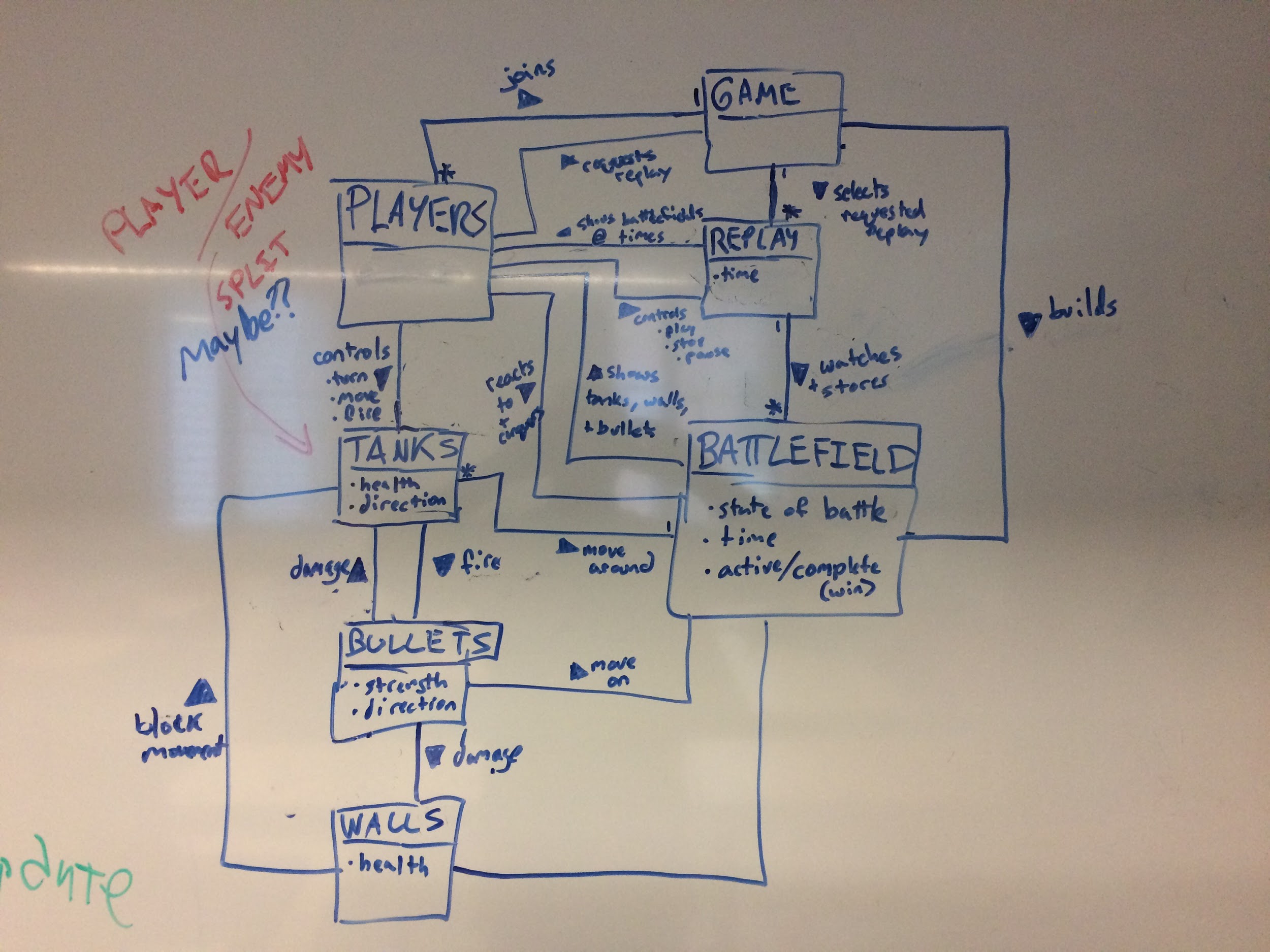
Join Game

1. User clicks UI button to start game or join server on client startup.
2. Client requests to join game from server.
3. Server creates tank with unique ID.
4. Server returns tank to client.
5. Client request grid
6. Server return grid
7. Client displays grid with tank to user.

1. **Domain model (diagram of domain concepts you have identified so far, and their relationships).**

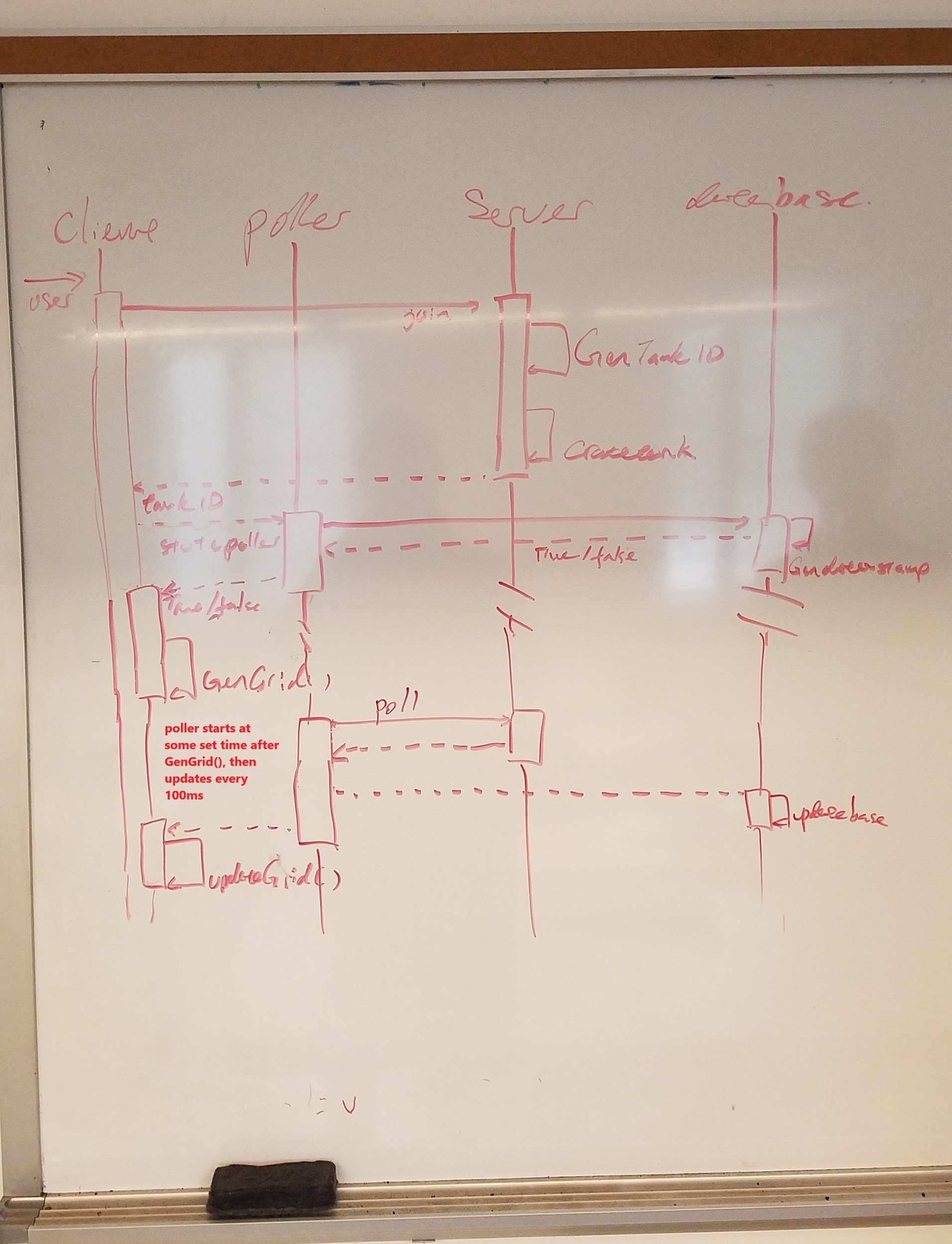
Good Candidates for Conceptual Classes

1. Player
2. Tank
3. Bullet
4. Wall
5. Grid

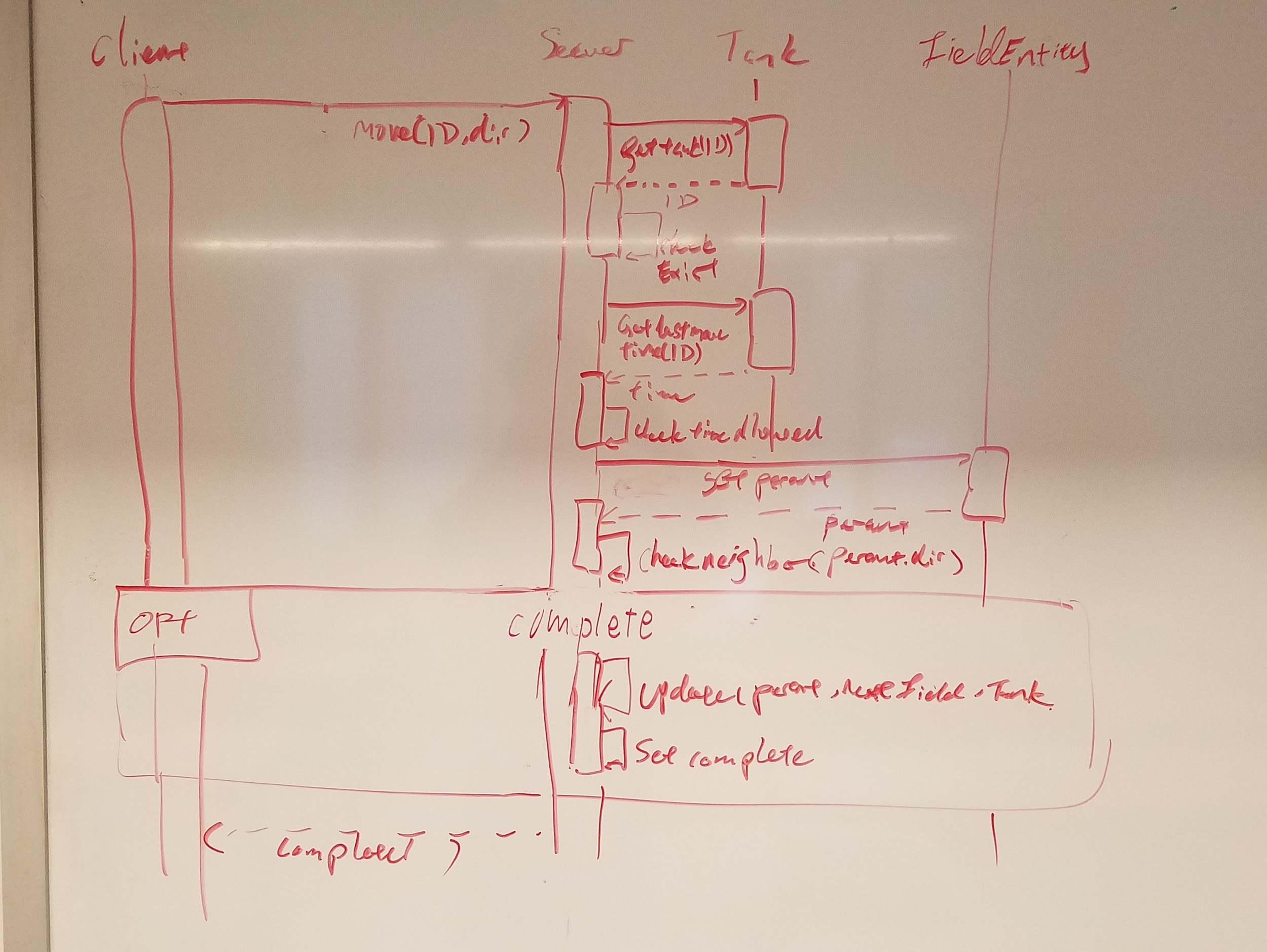
****

1. **(At least) two UML sequence diagrams starting with a user action in the client, improved or different from what's in the assignment description. One sequence diagram should correspond to the success scenario you gave.**

Joining the server

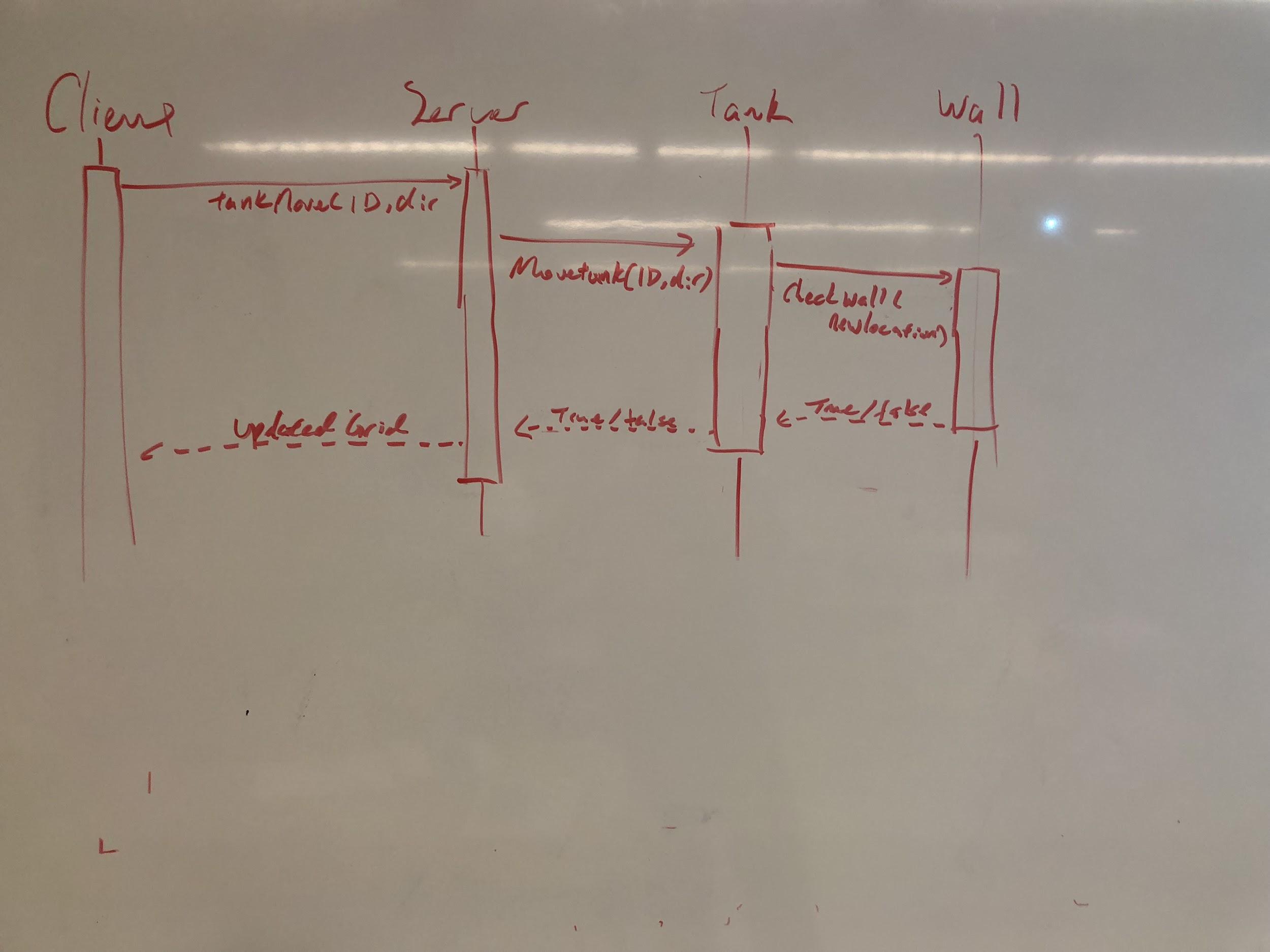
****

Move Tank



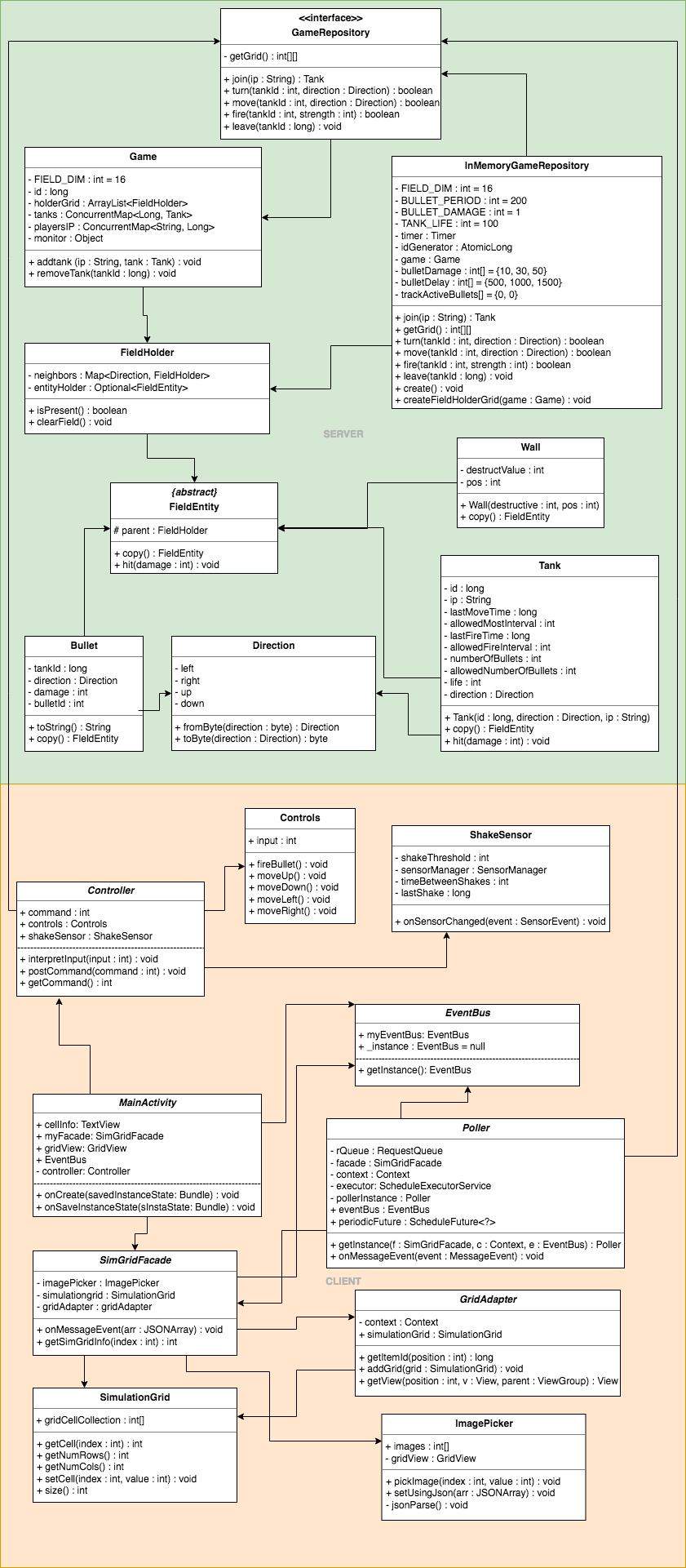
1. **(At least) one UML sequence diagram illustrating how constraints on turning or movement will be enforced on the server, resulting in a different return value in server responses to turning or movement requests from a client.**

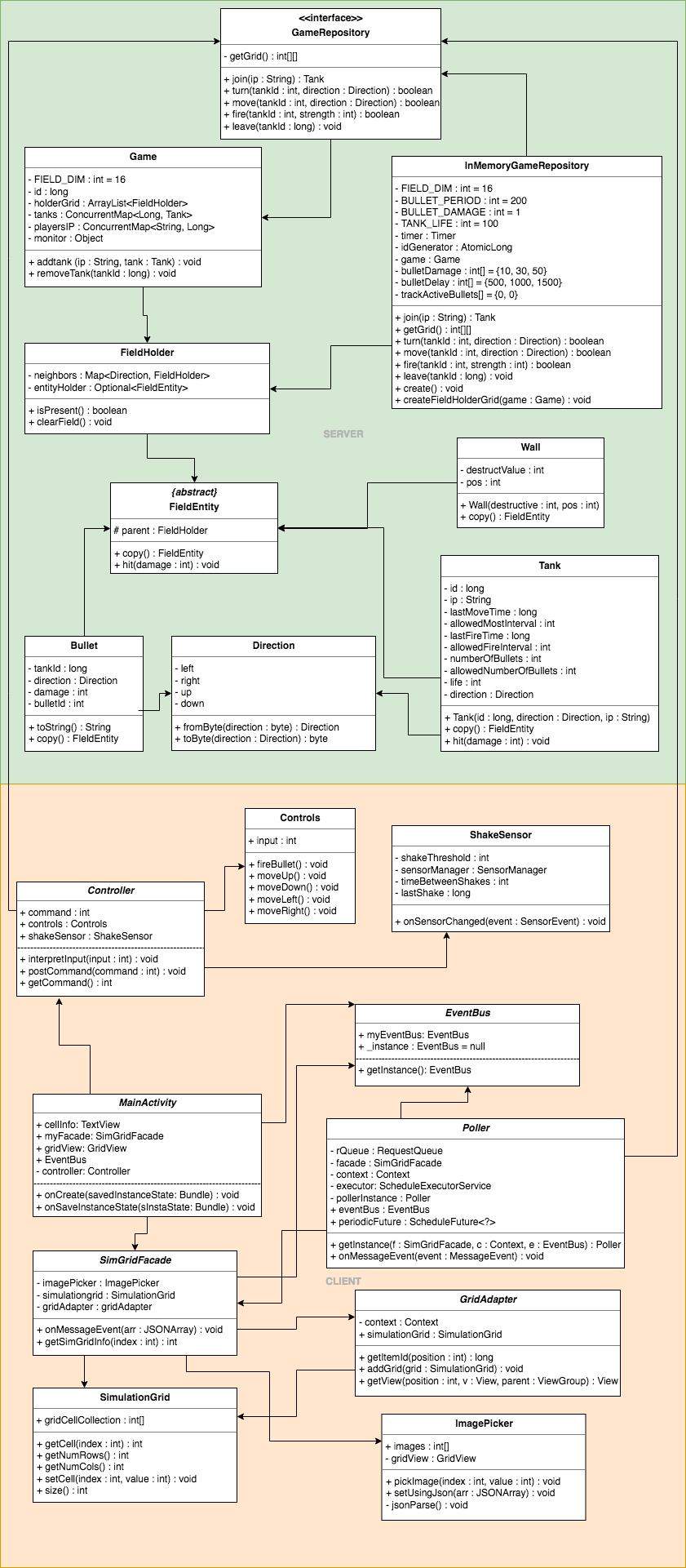
Moving tank

****

1. **UML design class diagram(s) that supports joining a game plus (at least) one gameplay use-case... preferably should address the needs of the entire system**

Next page...

****

****

1. **A brief description of what patterns you have identified so far as being useful for your project--indicate which classes will participate in each pattern, and what roles they will play in each pattern.**

Facade Pattern

The SimGridFacade class will forwards requests to the appropriate subsystem classes. This includes calling an ImagePicker, adding those cells to a SimGridFacade class to store the grid, and sending a SimulationGrid object to the GridAdapter class to be displayed in the UI. The Facade will also allow simplicity since there will be a current game that is occurring and a view for the replays which will both utilize the facade.

Observer Pattern

The Poller class will allow multiple subscribers to be notified of the server information through the EventBus that is created to allow for their corresponding actions, such as the SimulationGrid needing to be updated from our current server values.

Singleton Pattern

The Poller class and the SimGridFacade within the application will be made into singletons to allow for the same instances to remain after device rotation and to allow some of the SimGridFacade’s static functions to be accessible globally.

Model View Controller

We have a player (***USER***) that will control the tank via button input through a controller class (***CONTROLLER***) on the client side. It will parse those button inputs and send a command to the server (***MODEL***) which will handle those commands accordingly whether its a move, turn, or fire command. It will handle those commands by updating a GridView (***VIEW***) that is sent to the client and displayed on the screen.