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.

Functional Requirements

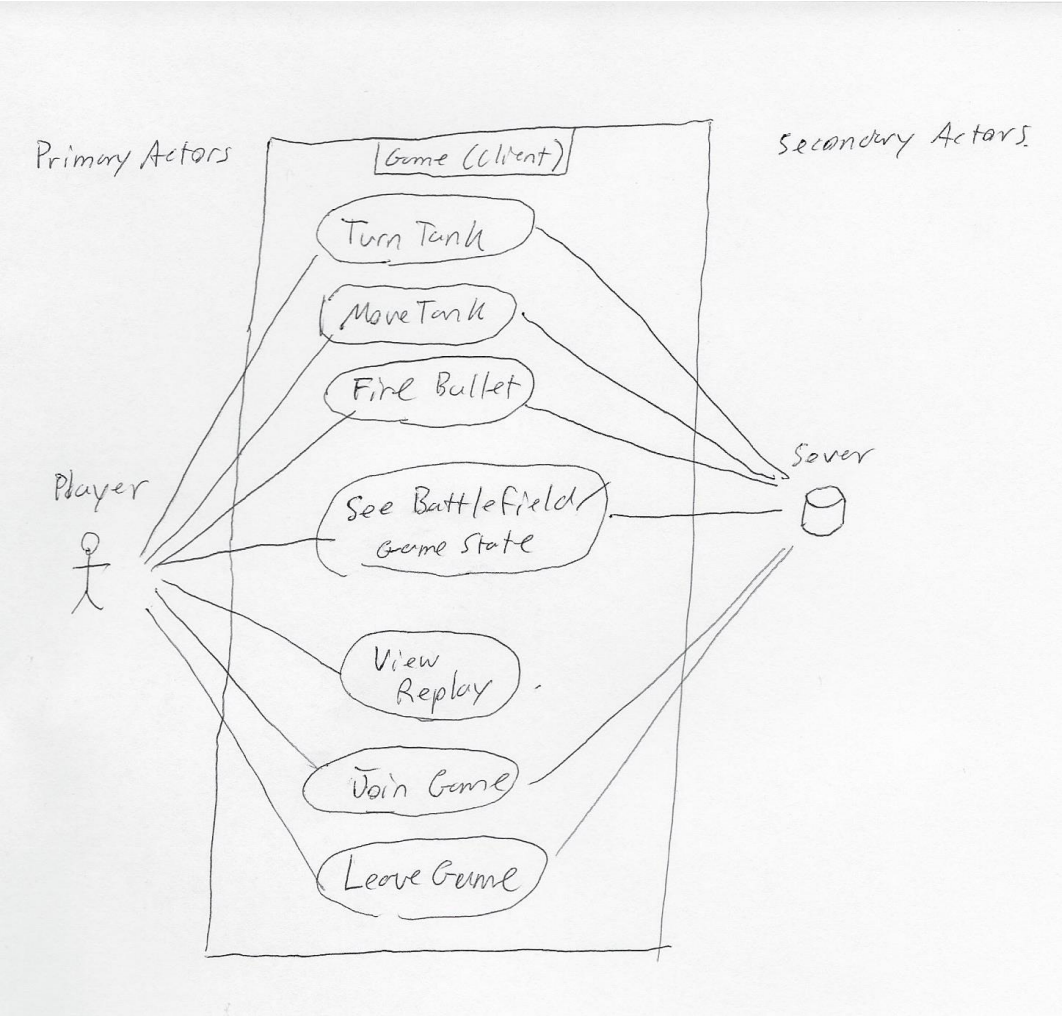
Client:

- Through UI, the player can:
  --True tank
  --Move tank
  --Fire bullets
- Player can see the battlefield
- Player can control tank
- Player can fire bullets of various strengths
- Bullet can be fired when player shakes device
- Replay feature

Server:

- One entity per cell
- Only one server per game
- Multiple clients can connect to the server at once
- Tank can move every x second; x= 0.5
- Tank can fire once every y seconds; y = 0.5
- Only z bullets from a given tank can exist at once; z = 2
- Tank can only make 90° turn per step
- Tank can only move forward / backward relative to current direction
- Poller thread can poll the server every 100ms for current status, return a grid and a timestamp

Use-case diagram (showing actors and potential use-cases you have identified so far).



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

Scenario: Turn Tank

Step

1. Player uses UI to press a button in the control panel to initiate a turn.
2. Turn function sends a turn message to the server including the tank ID and the desired direction.
3. Server calls its own turn tank method on the proper tank based on the passed ID and the passed direction
4. Server's turn tank method verifies that the desired turn is legal then sets the tank's

direction if so.
5. Server builds new double JSON array with latest data, return true.
6. Poller/rest client requests the newest data from server.
7. Poller/rest client receives JSON data from the server.
8. Poller/rest client extracts JSON arrays from the data and forwards this to the gridView adapter for visual implementation.
9. GridAdapter uses the data to rewrite the battlefield
10. Client displays the new battlefield through gridView.

4a. Illegal turn

4a.1 return false

4a.2 terminate process

7a. Poller fails to receive from server

7a.1 Abort attempted turn

9a. GridAdapter fails to get data from Poller.

9a.1 Abort attempted turn

Scenario: Join Game

Step

1. Play clicks “Join Game”
2. Send join request from server.
3. Server generate tank/tank ID
4. Server returns tank ID;
5. Client stores tank;
6. Clients creates Poller.
7. Poll server to get initial battlefield.
8. Server returns the battlefield.
9. Clients display the b.f.

3a. Server (for some reason) fails to generate tank.

3a.1 Retry step 2 X times

5a. Client receives bad data

5a.1 Restart join process (step 2) X times

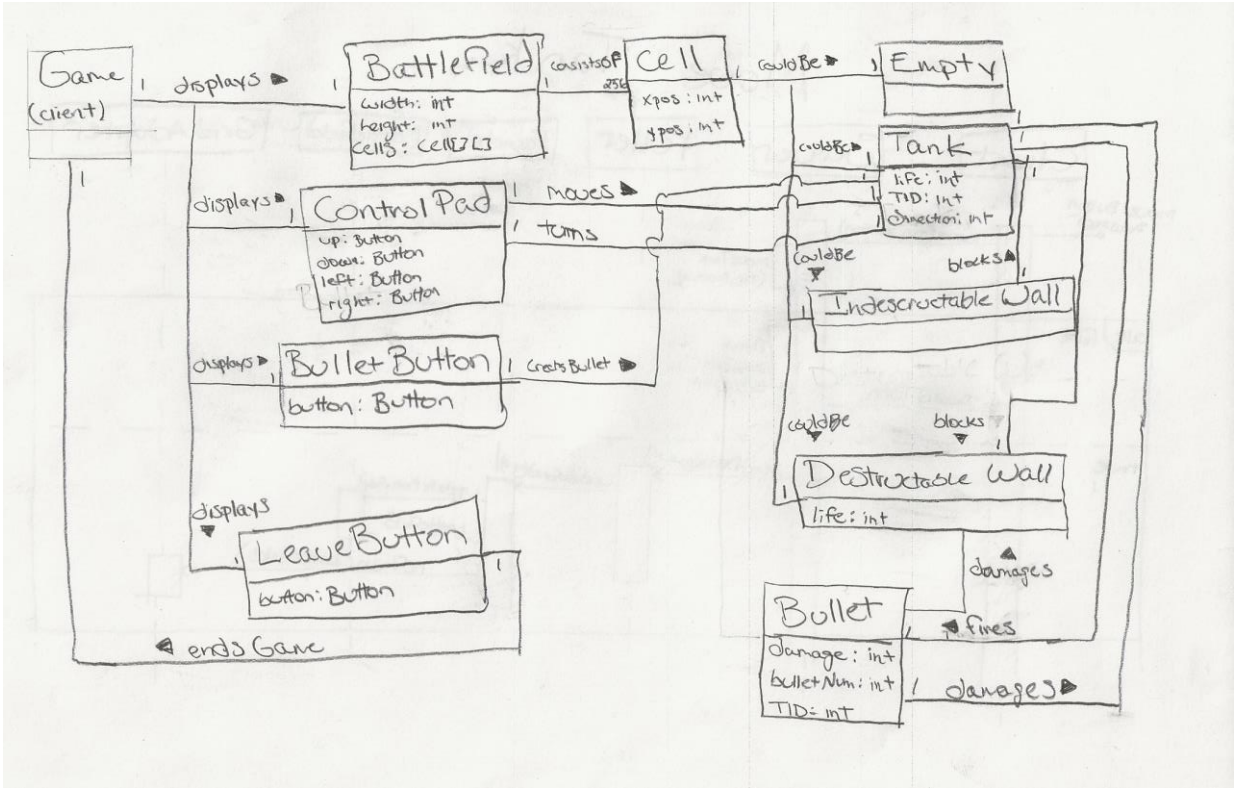
7a. Cannot reach server

7a.1 retry step 7 X times

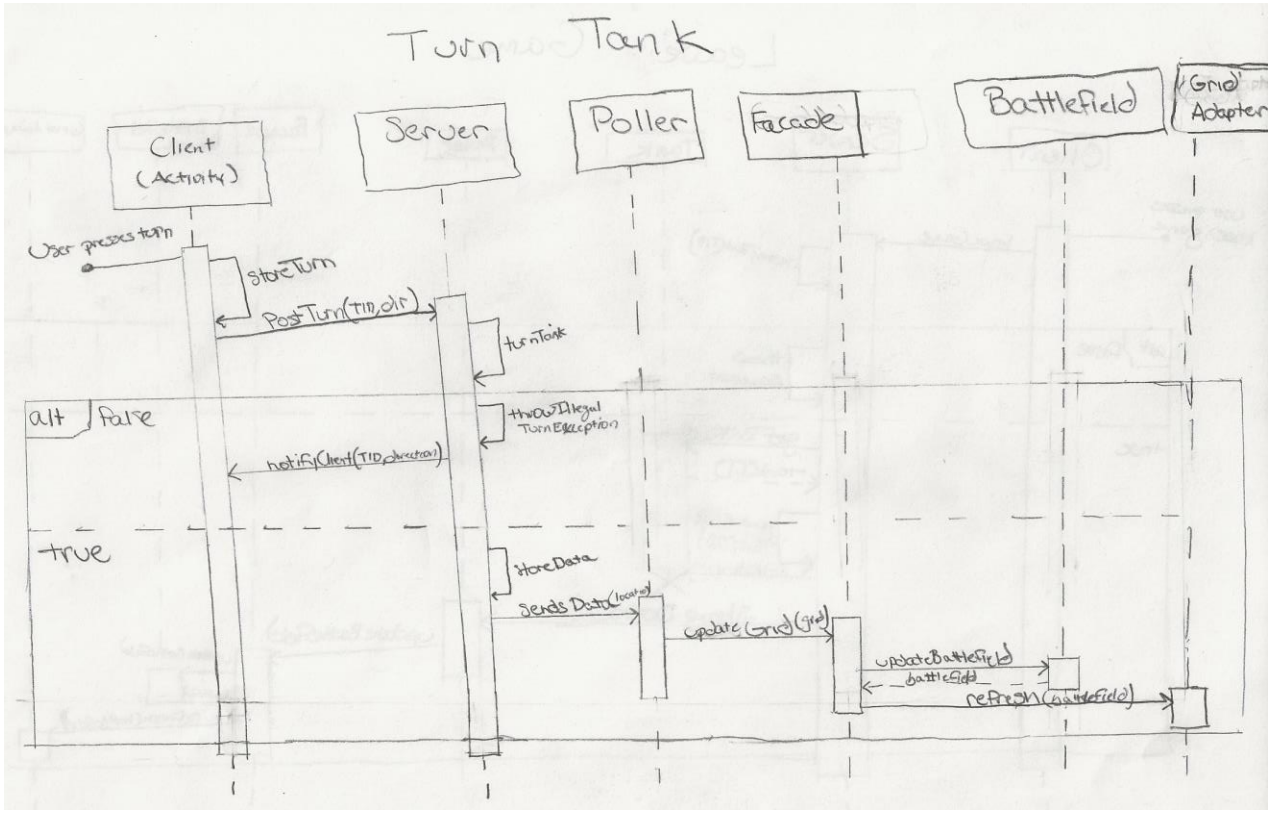
9a. No/bad battlefield

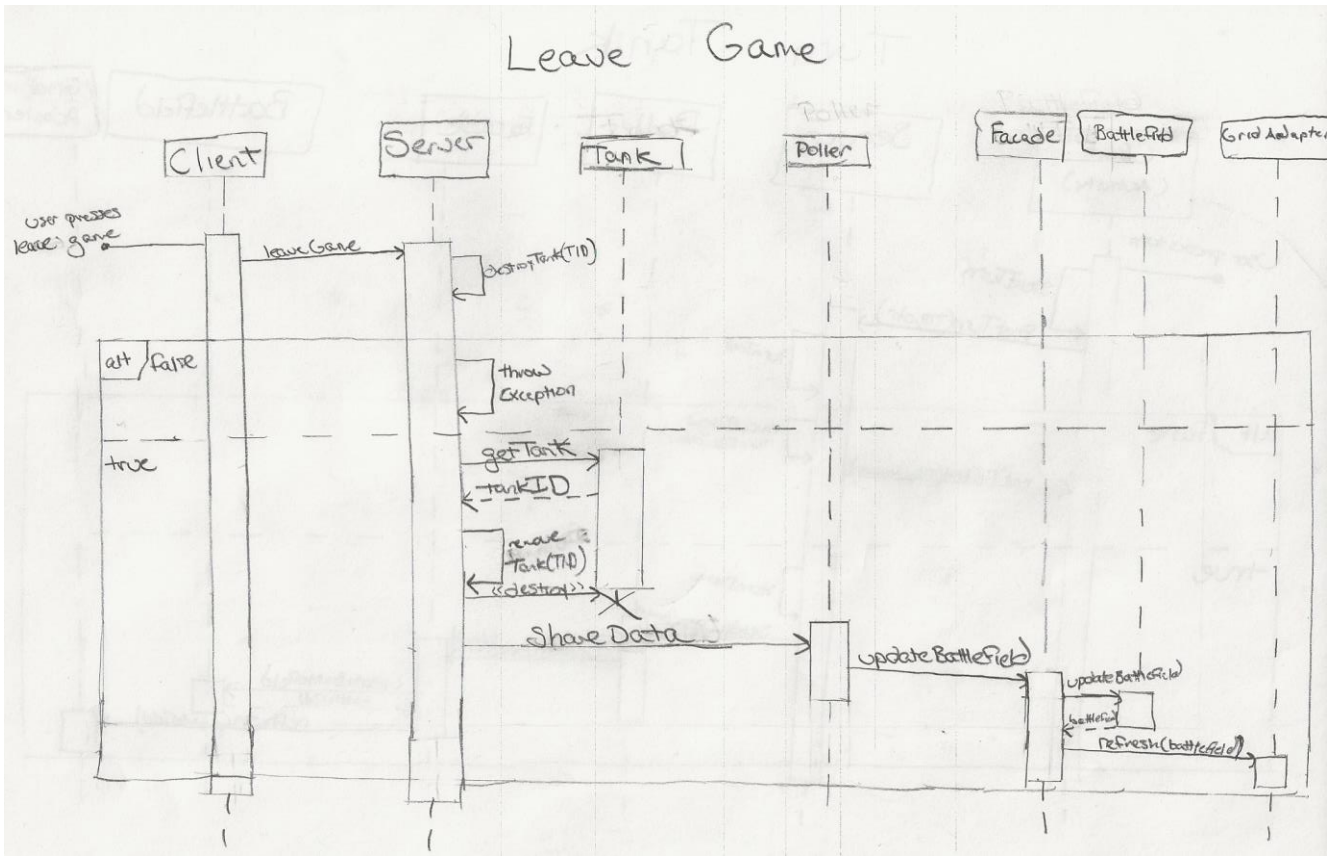
9a.1 retry step 7 X times.

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

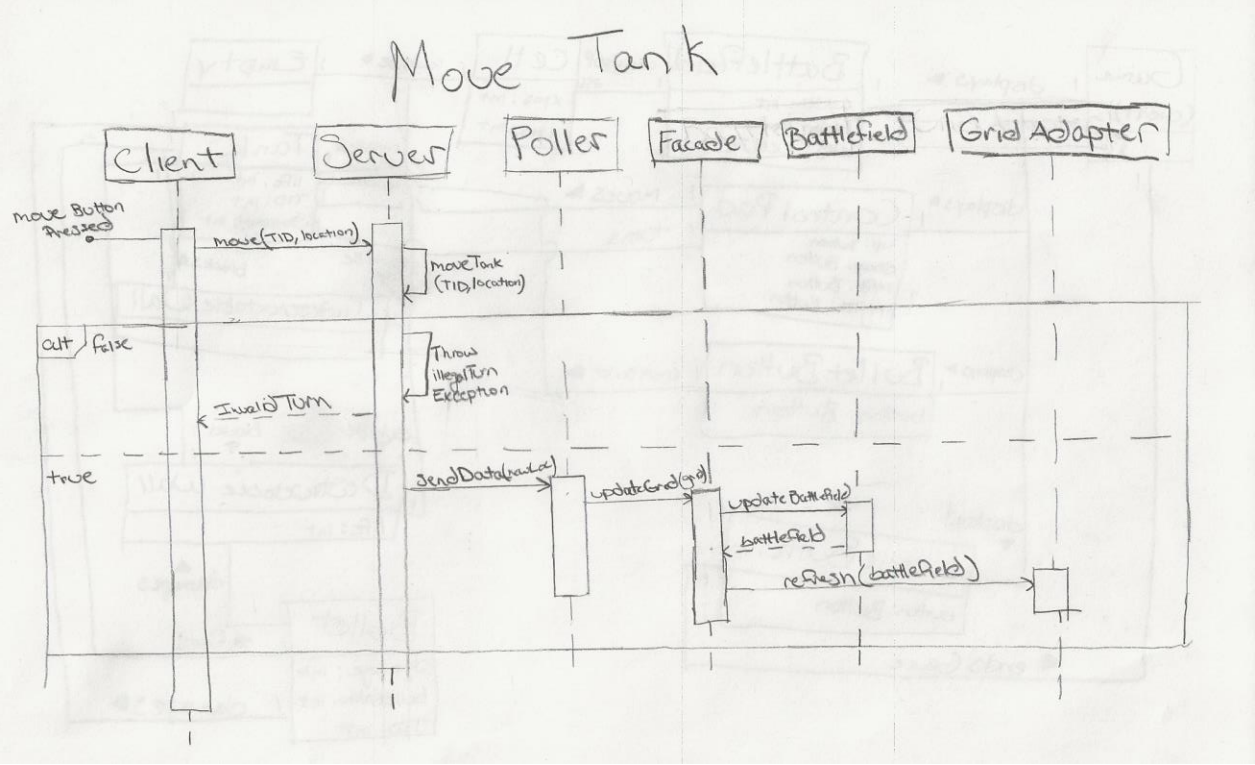


(At least) two UML sequence diagrams starting with a user action in the client, improved or different from what's in the assignment description. One sequence diagram should correspond to the success scenario you gave.

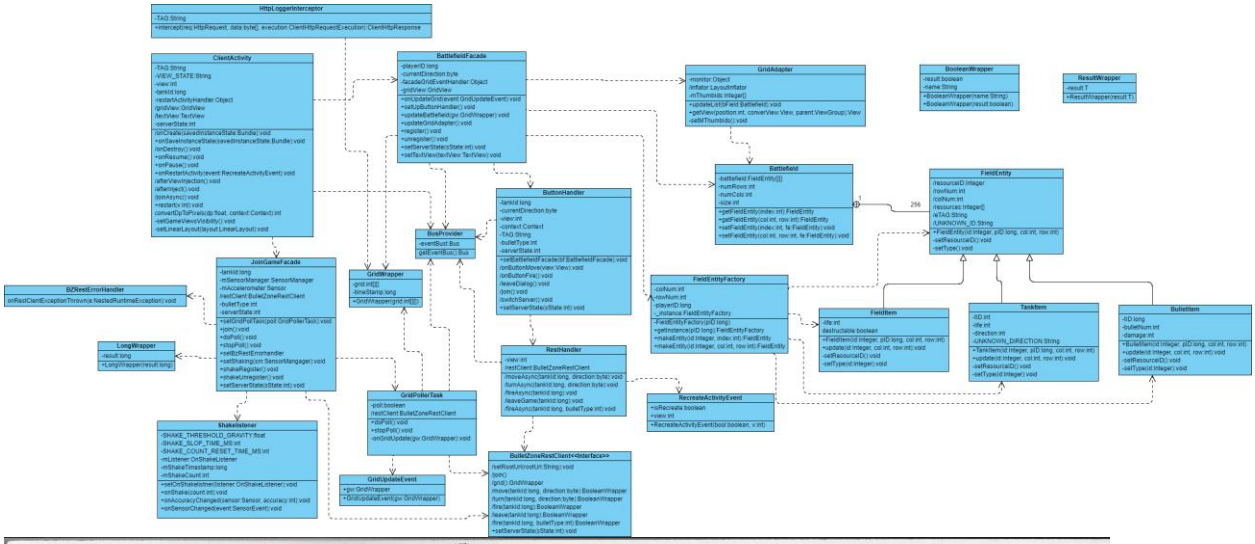


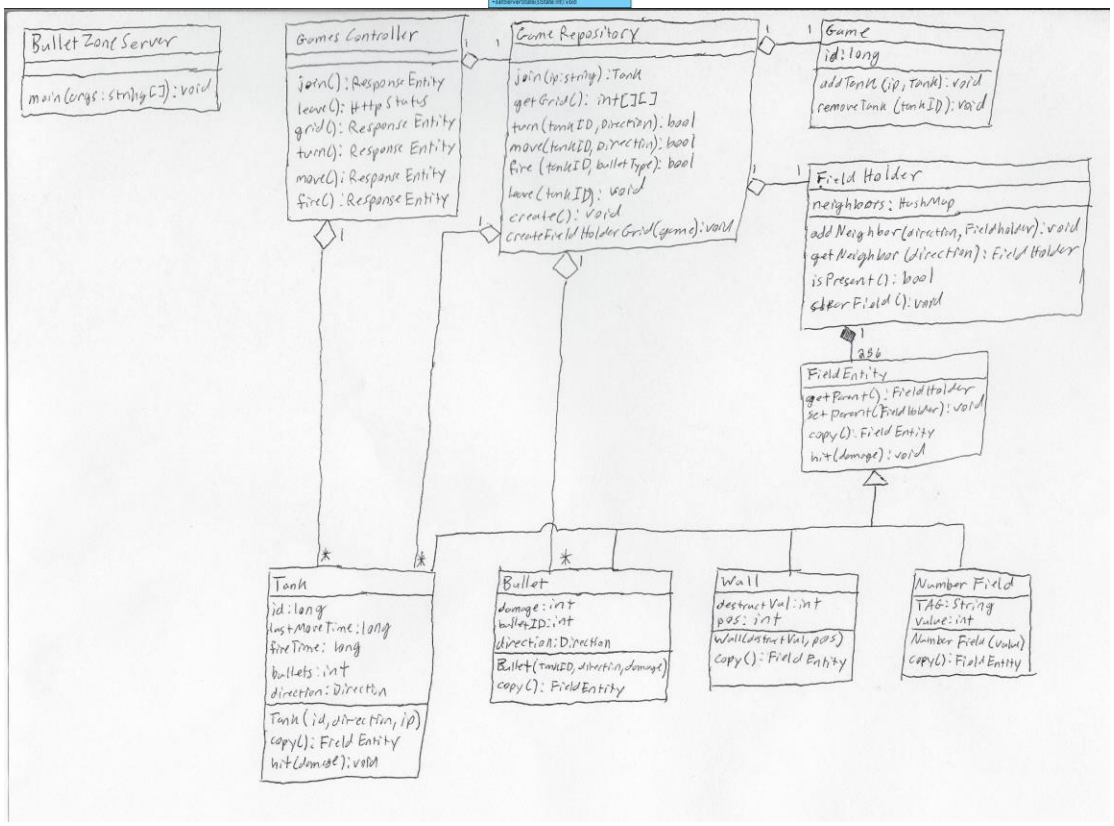

Leave Game
Client
Server
Tank
Poller
Facade
BattleField
User presses leave game
leaveGame
destroyTank(TID)
alt
false
throw Exception
true
getTank
tankID
remove Tank(TID)
«destroy»
ShareData
updateBattlefield
updateBattlefield
battlefield
Refresh(battlefield)

(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.



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





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.

Singleton Pattern: The server interface will be a singleton class as there can only ever be one at a time. This can be instantiated by the main activity of the first client to attempt to connect.

Façade Pattern: The ClientActivity will have a façade to separate it from the Battlefield, GridAdapter, FieldEntityFactory, and FieldEntity classes, about which it needs to know nothing. When the poller receives data from the server, it will send it directly to the façade class, bypassing the ClientActivity entirely. This way, the client activity is only responsible for reading button presses and sending the correct messages, not for the display that the user sees.

Observer Patter. The Poller will implement an eventbus over which to send messages. Main activity will subscribe to the eventbus to receive the messages from Poller. This is very similar to the Observer Pattern. Additionally, The Replay Class can use the eventbus to send messages as if they were coming from the server to show the replay of the last game, to which the main activity would again be the subscriber.

Simple Factory. There are a lot of different cells with complicated rules for construction, so instead of BattleField (or a façade) creating the cells, a factory class will be instantiated. The BattleField (or façade) will call the factory, which will then make and return the cells.

Strategy: There are a lot of different cell, but they all share the same base characteristics. So, a strategy pattern will be implemented so that a generic cell can be created, called, or manipulated and only that cell will know what type it is. The different types of cells will all extend and abstract cell class whose constructor and methods will be called by the the factory, battlefield, or façade.