Zulu Team

CS619

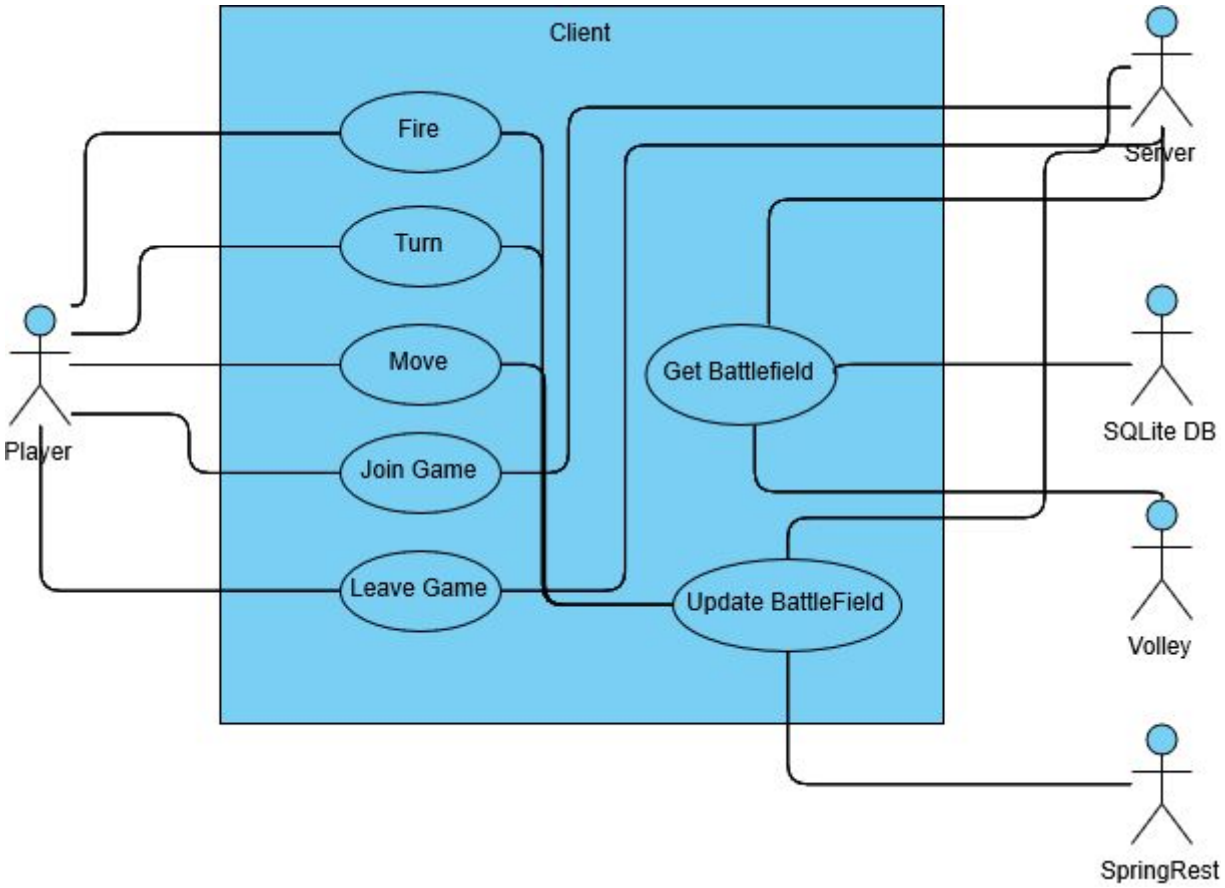
10/12/18

BulletZone Design

1. **[3 pts]** 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.

- The server must be able to receive events from the clients
- The client must save grid states to SQLite database
- There must be a 16x16 grid representing the battlefield.
- The server is responsible for managing the battlefield.
- The server is responsible for game time.
- The server will enforce the rules of the game.
- The client is responsible for showing the game state to the player.
- The client will provide a control interface.
- Only one server can be present in a single game.
- There can be multiple clients connected to a single game.
- Each player will have their own tank.
- Player can turn the tank every 0.5 seconds.
- Player can move the tank every 0.5 seconds.
- Player can fire bullets from the tank.
- Tank can only make one turn per step.
- Tank can only move forward and backward relative to current direction (no sideways movement).
- A maximum of two bullets fired from a single tank can be in the game at the same time.
- The poller thread will poll the server every 100ms for the current status of the game.

2. **[3 pts]** Use-case diagram (showing actors and potential use-cases you have identified so far).



3. **[4 pts]** (At least) one main success scenario for (at least) one non-trivial use-case for the client app.

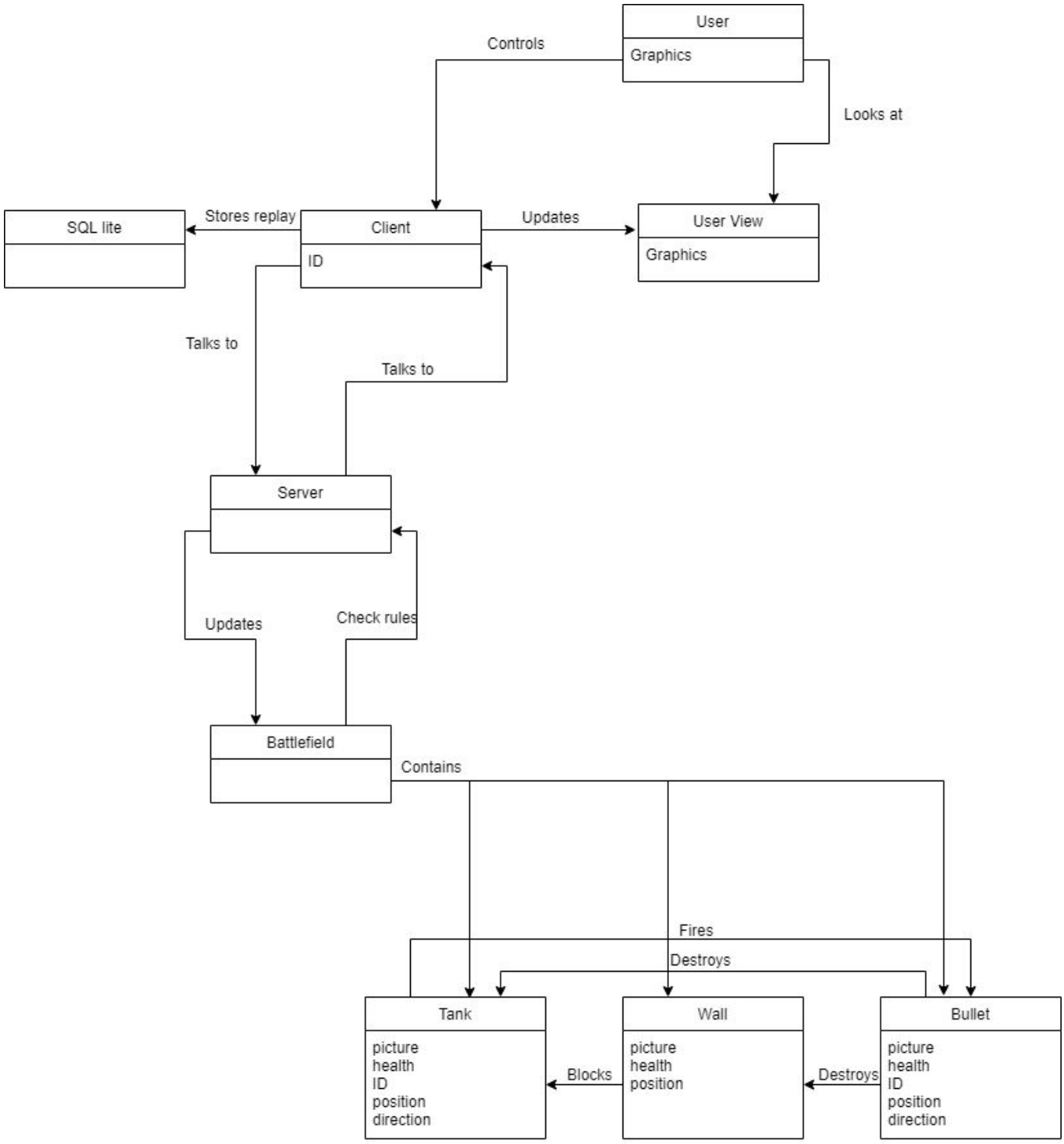
Player fires bullet

- Player shakes device
- Client sends a request to server to create a bullet of the player's tank ID
- Bullet is created
- Timer is created and updates the bullets position every _ seconds
- Bullet moves until it collides with an object or hits edge of grid
- Upon collision damage is calculated from bullet
- Bullet position is sent back to client
- UI is updated to show player where the bullet's position is

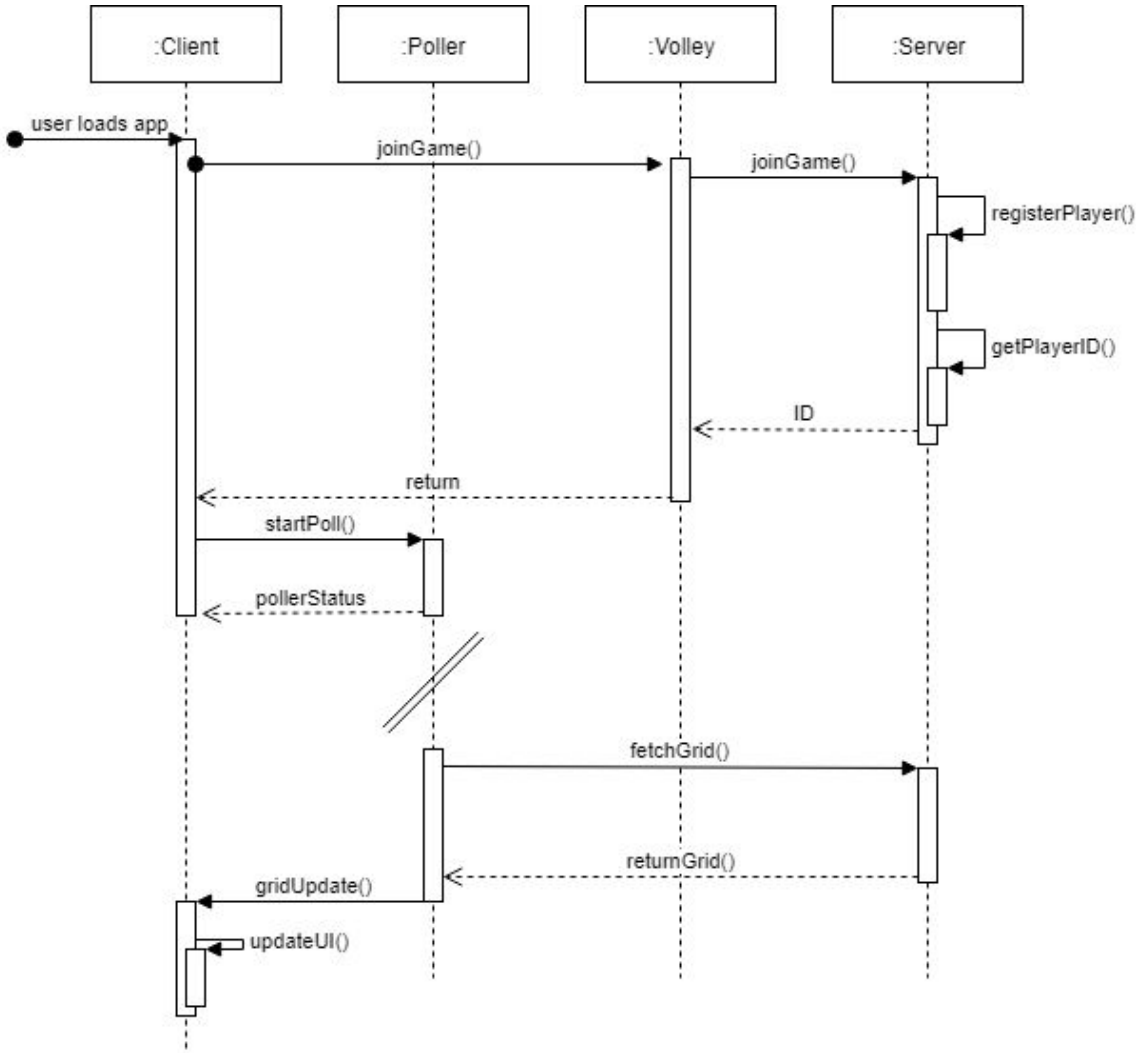
Precondition: Player can fire bullet

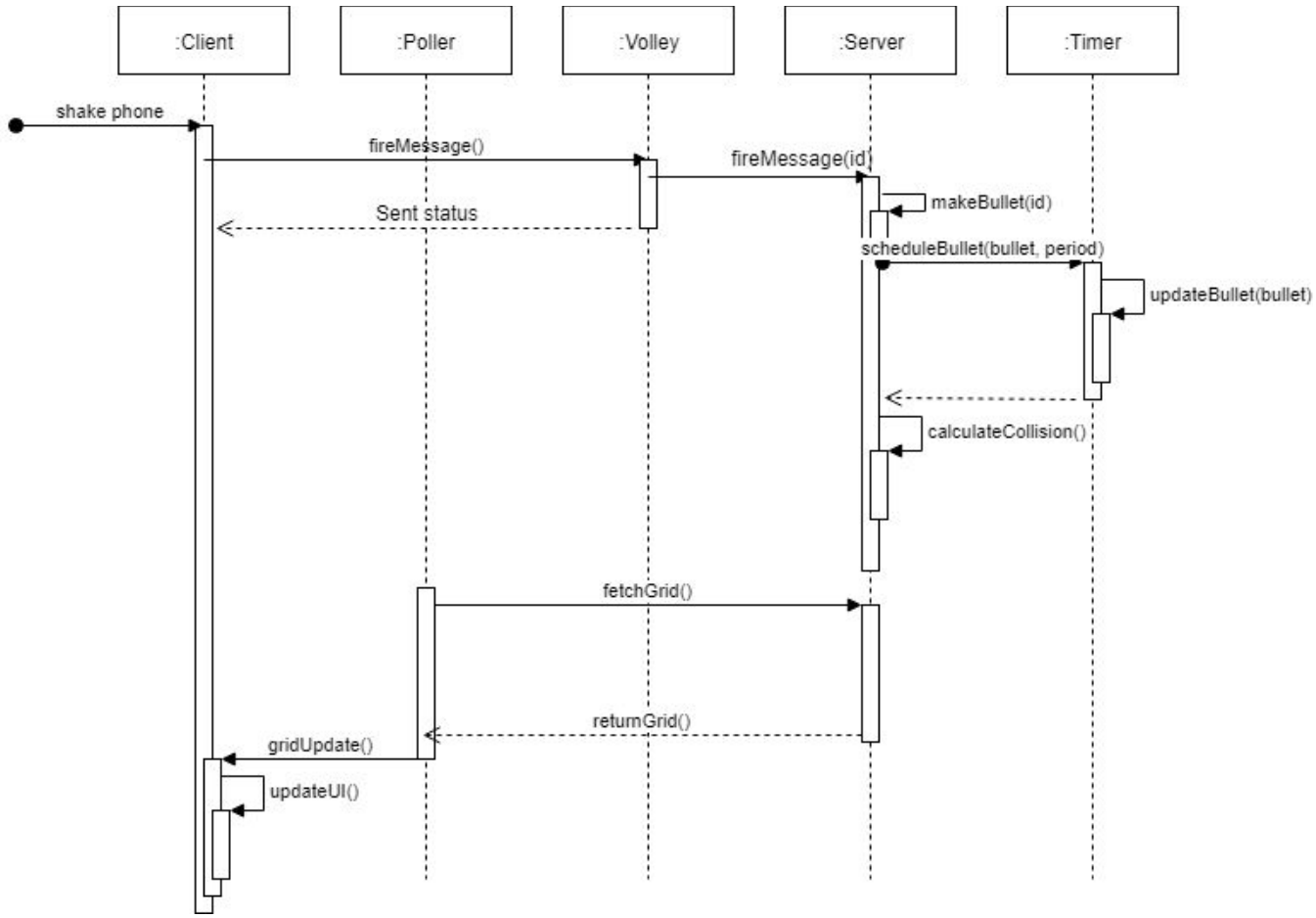
Postcondition: Player fires bullet

4. **[3 pts]** Domain model (diagram of domain concepts you have identified so far, and their relationships).

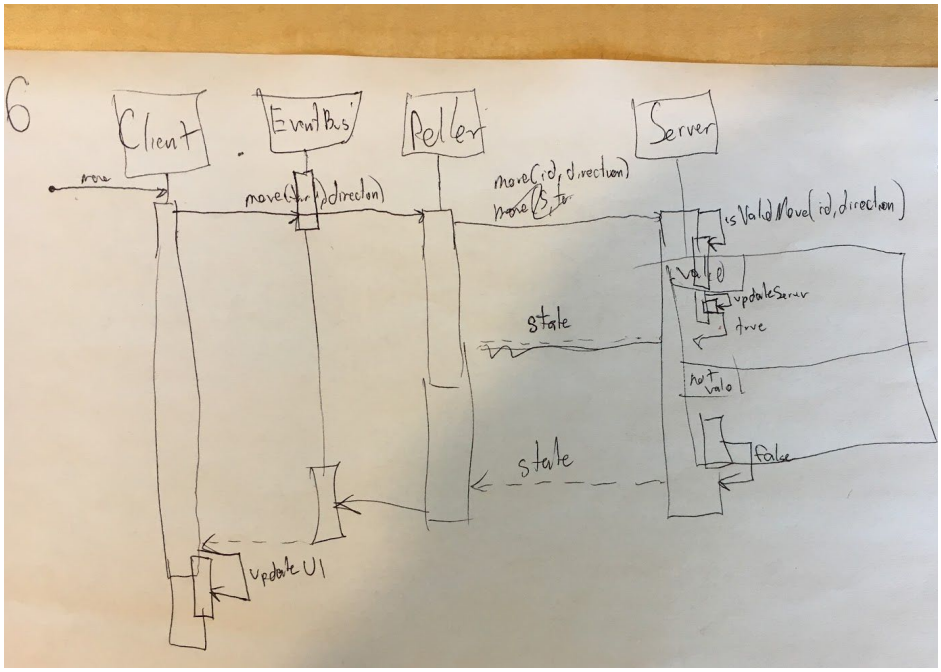


5. **[4 pts]** (At least) two UML sequence diagrams starting with a user action in the client, improved or different from what's in the assignment description. One sequence diagram should correspond to the success scenario you gave.

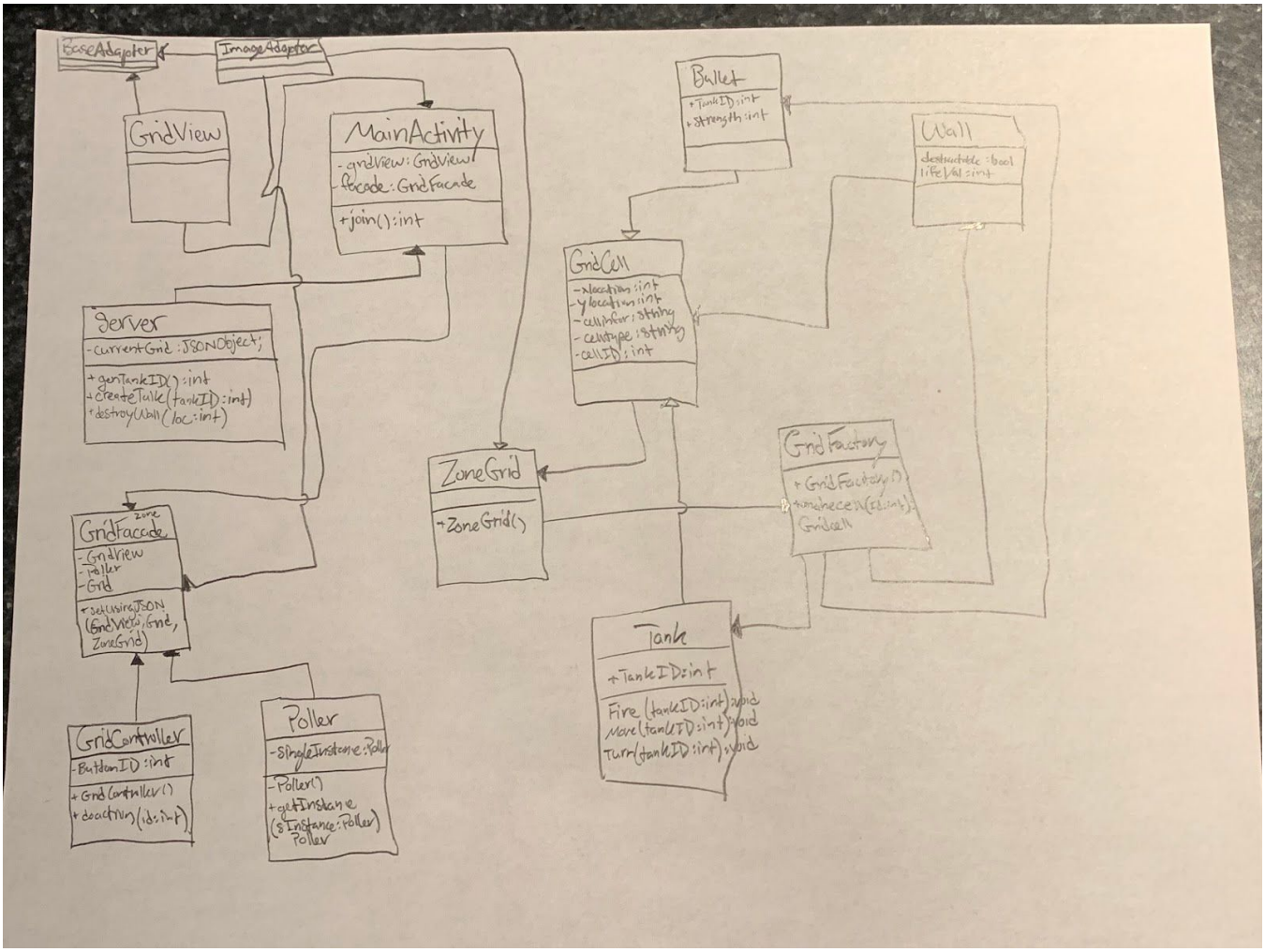



:Client
:Poller
:Volley
:Server
:Timer
shake phone
fireMessage()
fireMessage(id)
makeBullet(id)
Sent status
scheduleBullet(bullet, period)
updateBullet(bullet)
calculateCollision()
fetchGrid()
returnGrid()
gridUpdate()
updateUI()

6. **[2 pts]** (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.



7. **[8 pts]** 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



8. **[3 pts]** 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.

We've identified a few patterns in the design so far. The code will be designed based on the GridSim application, and will inherit some patterns from that application.

The factory pattern will be used to create all the objects of the grid based on a number received from the raw server value. The factory will be called GridCellFactory and will be responsible for correctly creating and displaying different GridCell objects. It will also be responsible for keeping a collection of TankItems and updating their locations on every server poll.

The facade pattern will be used to simplify and mask some of the subsystems such as displaying the graphics with Android, and interacting with the storage database. The facade

will be implemented in the MainActivity class to hide most of the creation and handling of logic for the program.

The observer/event bus will be used to communicate between subsystems in a simple manner. The Poller class will send data received from the JSONObjects acquired from the server and send the appropriate information over the EventBus. The GridFacade class will subscribe or some other class will subscribe to the EventBus and use the information to update and redraw the grid.

The singleton pattern will be used to make sure there is only one factory, and one controller for some of the complex systems. The singleton implementation is important for the factory pattern because there will be a collection of TankItems kept and upon creating/rotation of the MainActivity that information must be saved. The singleton will ensure the single collection is kept.

The MVC pattern will also be implemented as a part of our application design. The Model-View Controller is a few patterns in one. So this will be implemented by using the Observer pattern as described above; in addition to, adding a Strategy pattern and a Composite pattern. The strategy pattern will be implemented by using polymorphism and creating hierarchical classes to handle multiple algorithms getting called for different GridCell objects. The composite pattern will handle the UI components as well as the elements that make-up the UI components.