Anthony Papetti, Aiden Landman, Cameron Sullivan, Jackson Haubert, Nicolas Karpf

CS 619

Prof. Plumlee

10/9/2022

<u>Requirements</u>:

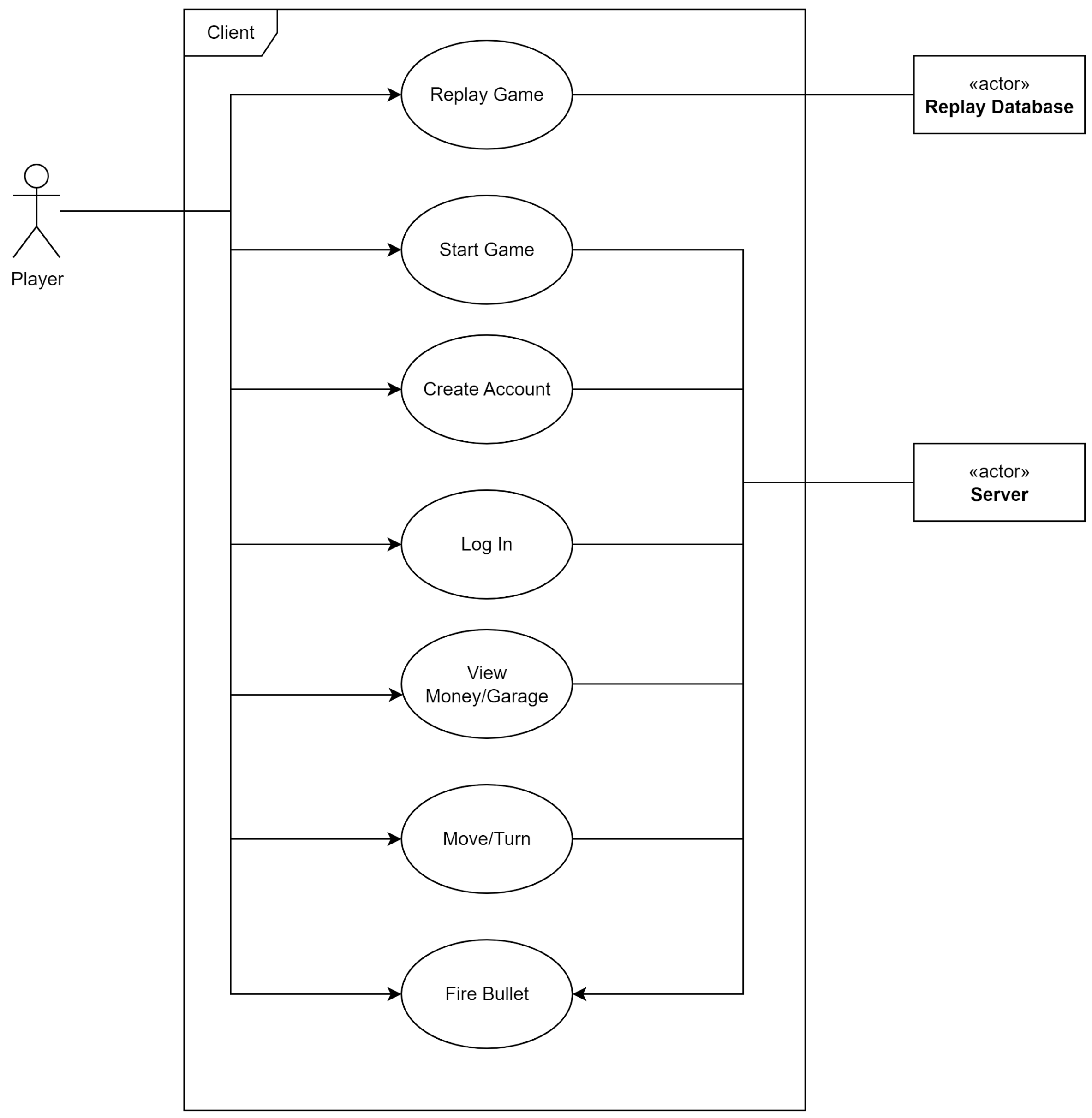
Client:

1. The client displays a 16x16 grid of cells.
2. Each cell in the grid should be capable of containing one entity at a time.
    a. An entity must be a Tank, Wall, or Bullet.
3. Each cell must support the properties of a terrain type, an improvement, and an item.
4. The player must be able to create an account.
5. The player must be able to log in to an account.
6. The player must be able to configure and save tank tokens on their turn.
7. The player must be able to turn and move their token, as well as fire bullets.
    a. A bullet must be fired when the player shakes the device on their turn.
8. The player must be able to view their account balance.
9. The player must be able to see their currently owned tanks and components in a "garage" view.
10. The client must store game-state information in a file or SQLite database
    a. The client must be able to replay the recorded game
    b. Replay must take place in real time or n times faster

Server:

1. The server must maintain game state and time.
2. The server must enforce the rules of the game. Rules the server must enforce go as follows:
    a. Tank can only move every X seconds (X=0.5).
    b. Tank can only fire every Y seconds (Y=0.5).
    c. Only Z fired bullets from a given tank can be in the game at the same time (Z=2).
    d. A tank can only make only one turn per step
    e. Tanks can only move forward or back relative to its current direction
3. The server must keep persistent player accounts that the player can log in to
4. The account must keep track of what tanks and components the player owns, as well as their current balance of in-game money
5. Whenever a new game is played, the server must initialize the game board
6. The server must store at least 30 seconds but no more than a minute of game event history.
7. The server must support a get request called “events” that accepts a timestamp and returns a list of event objects that represent the changes that have taken place since the timestamp.

Use Case Diagram:



(OLD) Success Scenario for the "Replay Game" Use Case:

1. The player chooses a "Replay Game" option from the main menu
2. The client queries the replay database to find the list of replays
3. The replay database returns a list of replays

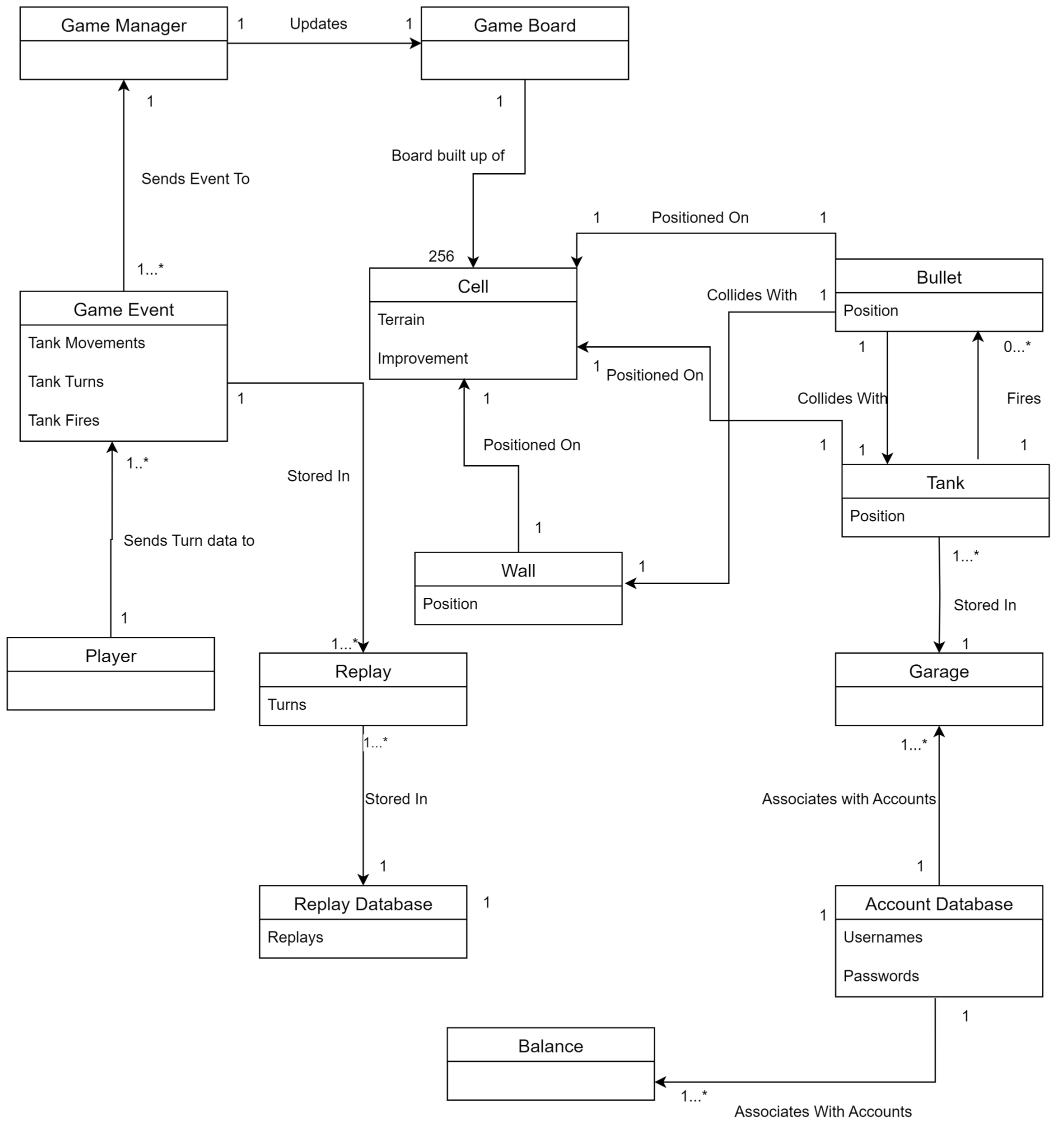
4. The client displays new view with a list of games with replay data and a option for playback speed
5. The user chooses a game to be replayed
6. The user enters a playback speed
7. The user clicks “Replay Game”
8. Client queries the replay database to find the game replay that the player has chosen
9. Replay database returns replay data
10. Client displays initial state of game board according to replay data
11. Client redraws board every step / n seconds with next move on the replay data
12. Repeat step 11 until all moves have been executed sequentially.
13. Client prompts user with a “Finish” and “Replay Again” Button
14. Player clicks “Finish button”
15. Client returns to main menu

Success Scenario for the “Fire Bullet” Use Case:

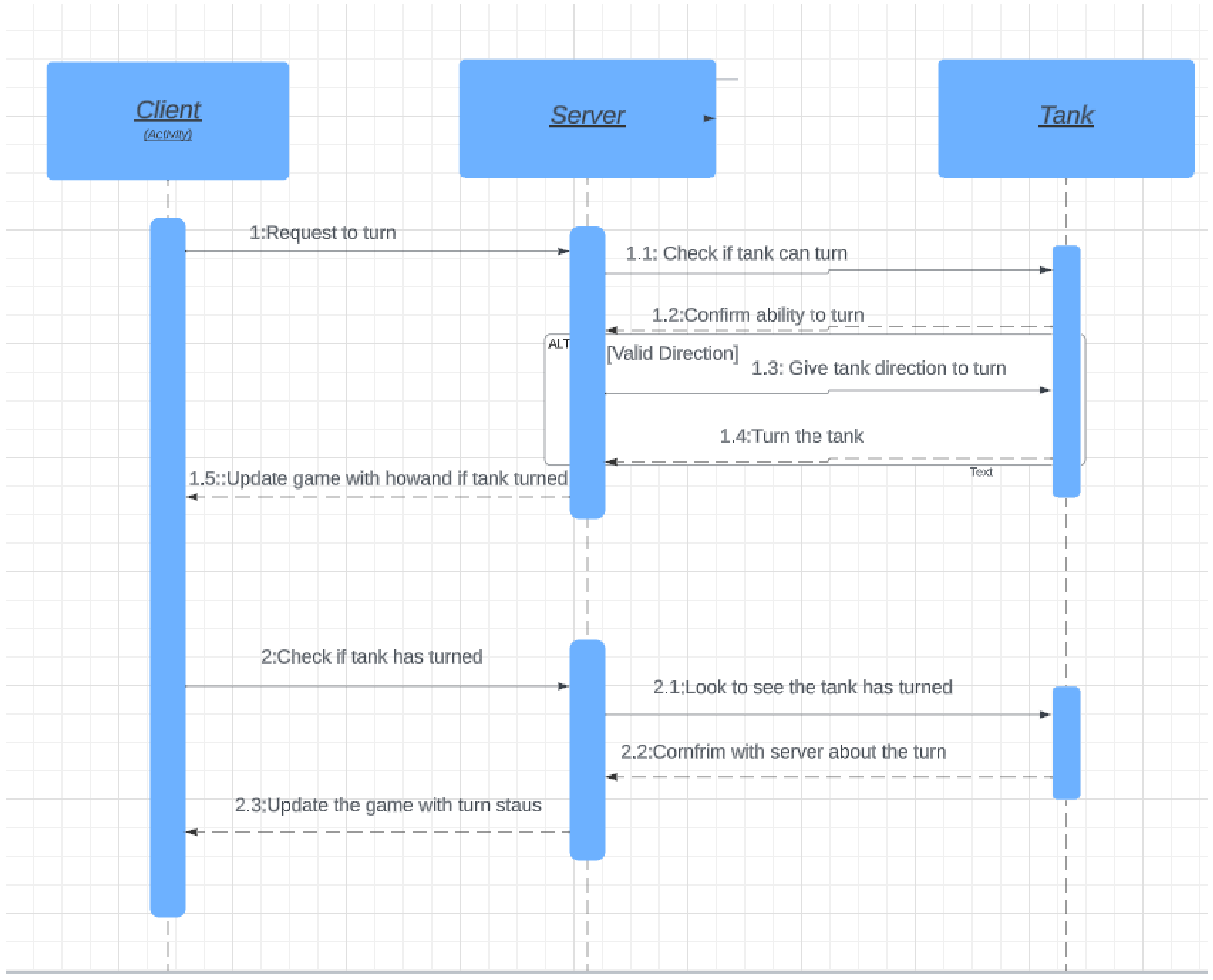
1. The player clicks the “Fire” Button from the game UI.
2. Client gets the id and direction of the tank being controlled.
3. Client sends a “fire” request, using the tank’s id (tankID) and direction (tankDir) as parameters.
4. Server checks that a tank with an id of tankID exists.
5. Server checks that the tank has not fired in the past Y seconds (Y defaults to 0.5).

6. Server checks that there are not 2 or more bullets from the tank on the game board.
7. Server updates the server event history with a new event that spawns a bullet in front of the tank according to tankDir.
8. Server sends a response of “true” to the client.
9. At a given interval (default is 500 ms), the Client sends an “event” request to the server.
10. The server sends a response containing the server event history.
11. The client updates the board according to the event history, including drawing the bullet that was fired.

Domain Model:

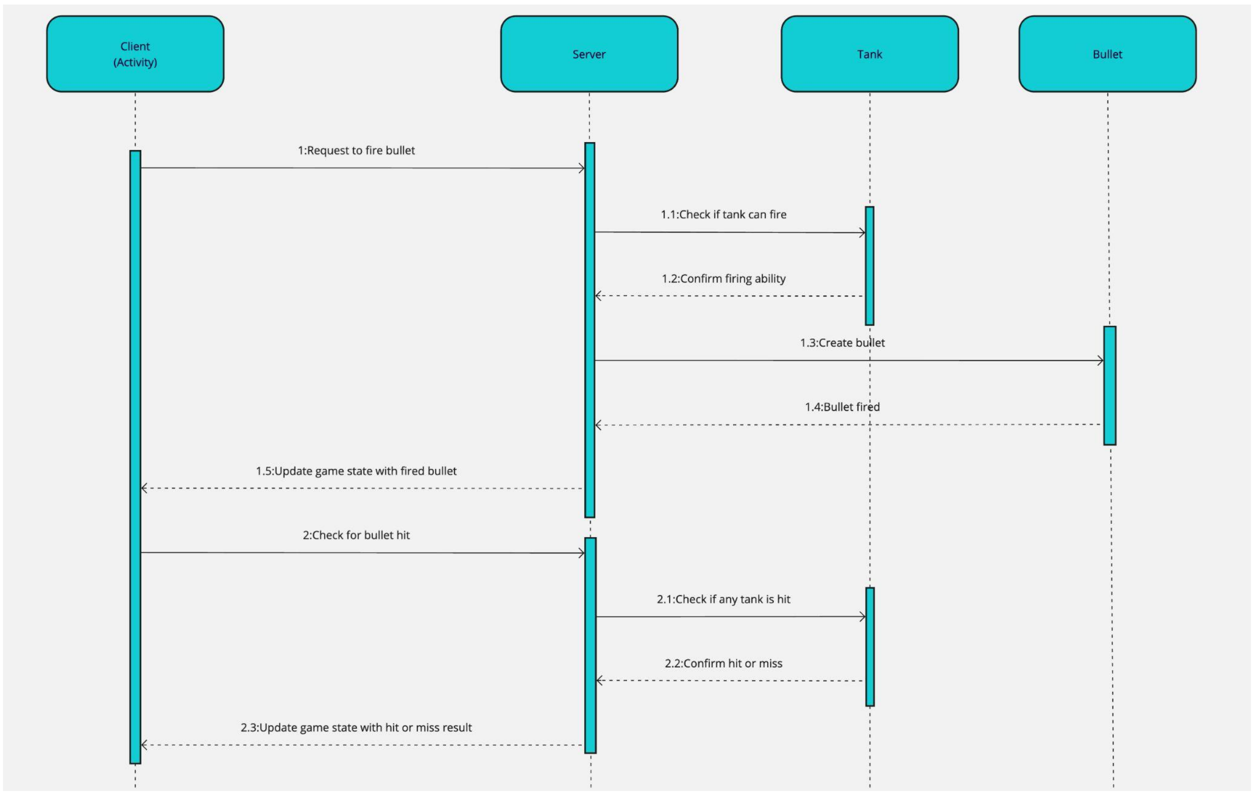


UML Sequence Diagram for turn:

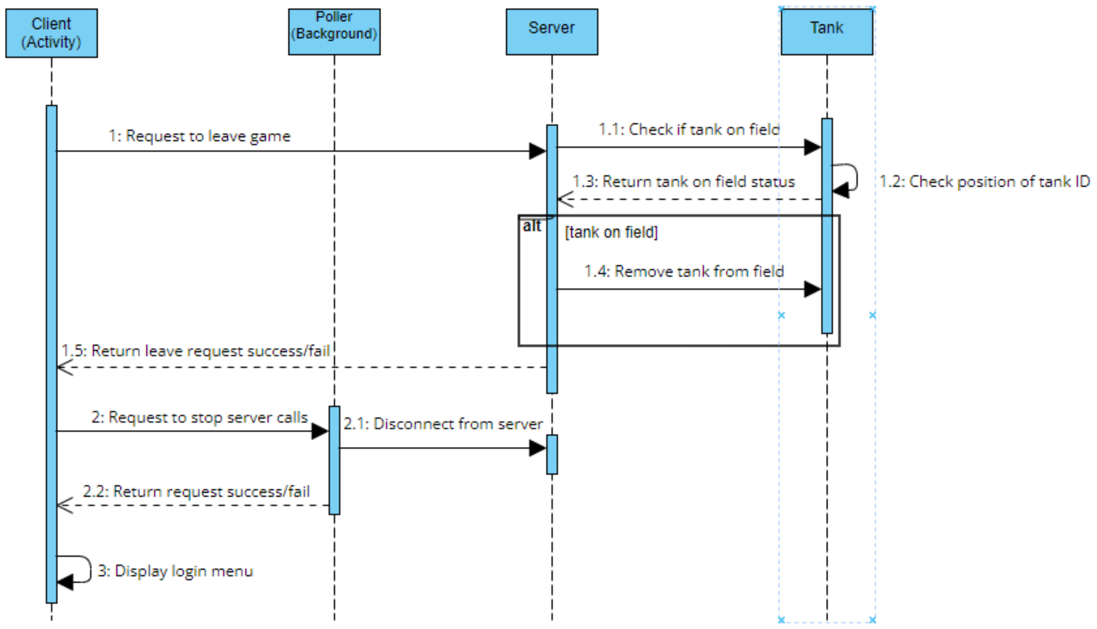


Client
(Activity)
Server
Tank
1:Request to turn
1.1: Check if tank can turn
1.2:Confirm ability to turn
ALT
[Valid Direction]
1.3: Give tank direction to turn
1.4:Turn the tank
Text
1.5::Update game with howand if tank turned
2:Check if tank has turned
2.1:Look to see the tank has turned
2.2:Confrim with server about the turn
2.3:Update the game with turn staus

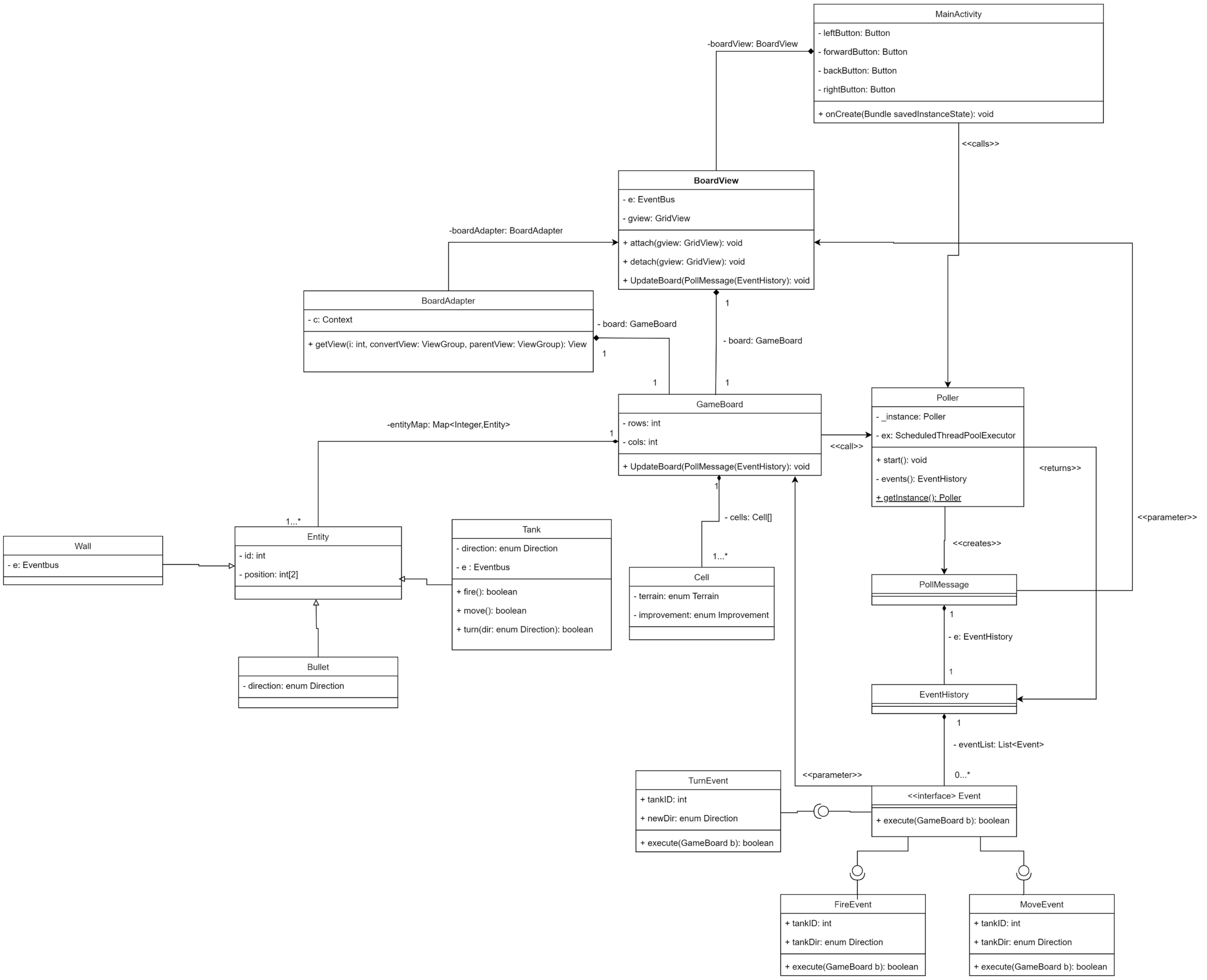
UML Sequence Diagram for “Hit Register”:


Client
(Activity)
Server
Tank
Bullet
1:Request to fire bullet
1.1:Check if tank can fire
1.2:Confirm firing ability
1.3:Create bullet
1.4:Bullet fired
1.5:Update game state with fired bullet
2:Check for bullet hit
2.1:Check if any tank is hit
2.2:Confirm hit or miss
2.3:Update game state with hit or miss result

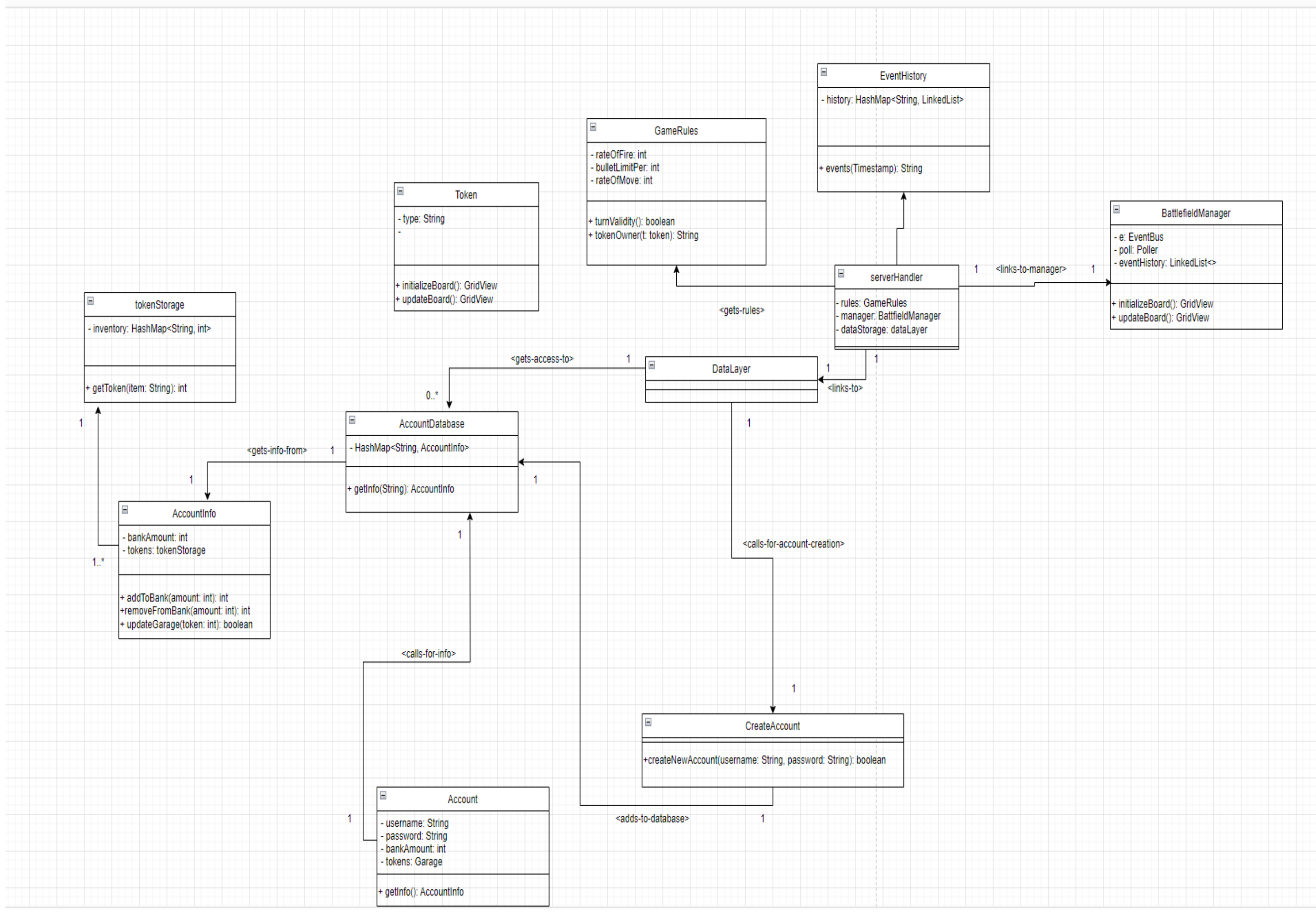
UML Sequence Diagram for “Leaving a Game”:


Client (Activity)
Poller (Background)
Server
Tank
1: Request to leave game
1.1: Check if tank on field
1.2: Check position of tank ID
1.3: Return tank on field status
alt
[tank on field]
1.4: Remove tank from field
1.5: Return leave request success/fail
2: Request to stop server calls
2.1: Disconnect from server
2.2: Return request success/fail
3: Display login menu

UML Class Diagram for Client:



UML Class Diagram for “Server”:


EventHistory
- history: HashMap<String, LinkedList>
+ events(Timestamp): String
GameRules
- rateOfFire: int
- bulletLimitPer: int
- rateOfMove: int
+ turnValidity(): boolean
+ tokenOwner(t: token): String
Token
- type: String
-
+ initializeBoard(): GridView
+ updateBoard(): GridView
BattlefieldManager
- e: EventBus
- poll: Poller
- eventHistory: LinkedList<>
+ initializeBoard(): GridView
+ updateBoard(): GridView
serverHandler
- rules: GameRules
- manager: BattfieldManager
- dataStorage: dataLayer
1
<links-to-manager>
1
<gets-rules>
tokenStorage
- inventory: HashMap<String, int>
+ getToken(item: String): int
<gets-access-to>
1
0..*
DataLayer
1
1
<links-to>
AccountDatabase
- HashMap<String, AccountInfo>
+ getInfo(String): AccountInfo
<gets-info-from>
1
1
1
1
AccountInfo
- bankAmount: int
- tokens: tokenStorage
1..*
+ addToBank(amount: int): int
+removeFromBank(amount: int): int
+ updateGarage(token: int): boolean
1
1
<calls-for-account-creation>
<calls-for-info>
1
CreateAccount
+createNewAccount(username: String, password: String): boolean
Account
- username: String
- password: String
- bankAmount: int
- tokens: Garage
+ getInfo(): AccountInfo
1
<adds-to-database>
1

Description of Patterns:

- Observer- Useful because other objects need to be notified of state changes from the server that Poller finds.
  - Classes involved: Poller, PollMessage, GameBoard, EventBus library.
  - Observer: The EventBus library provides an interface to allow objects to listen for updates.
  - ConcreteObserver: GameBoard stores state information given from updates and responds to updates accordingly.

  - Subject: The EventBus library provides the interface for registering and deregistering observers and keeping track of them.
  - ConcreteSubject: Poller and Pollmessage keep track of state, the EventBus library and Pollmessage notify observers of an update.
- Commander- Useful because there are many types of events from the server that need to be executed that affect lots of other objects.
  - Classes/Java Interfaces involved: Event, TurnEvent, FireEvent, MoveEvent, Poller, GameBoard
  - Command: Event provides the interface for executing events.
  - ConcreteCommand: TurnEvent, FireEvent, MoveEvent bind the receiver (GameBoard) and allow for the action to be invoked
  - Client: Poller creates the Event objects
  - Receiver: GameBoard actually does most of the operations needed for the event.
  - Invoker: EventHistory stores command objects. Does not invoke them as this model is slightly modified for our purposes.
- Singleton- Useful because we want to maintain a single object to poll the server for events every on a time interval. If there are multiple objects, there will be too many requests being sent to the server. It could slow down the app if too many requests/responses are being sent at once.
  - Classes Involved: Poller, MainActivity.
  - Singleton: Poller defines a method that allows clients access to its unique instance.

- Client: MainActivity uses the methods of Poller.
- Class Operations: start() is the method that requires there to be a single instance.