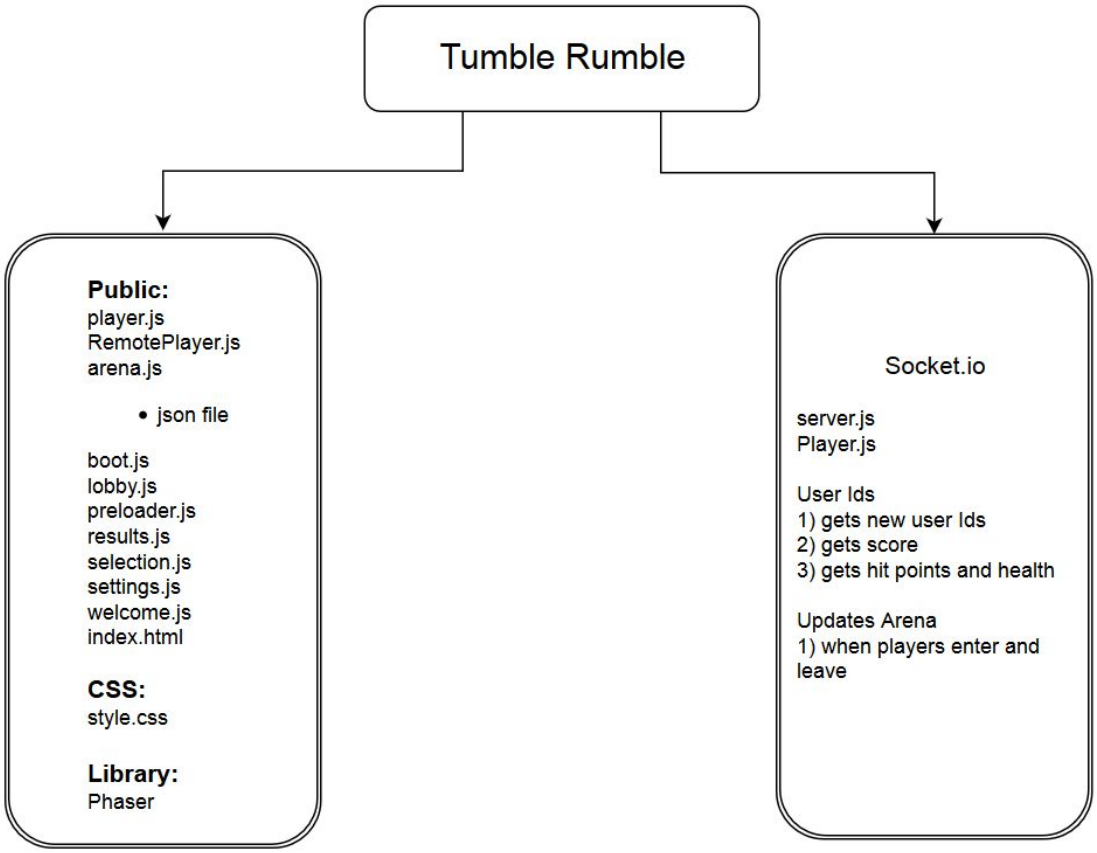
Functional Specification

High Level Design:

- Major Components and Modules:
    - Game States: We will have 10 different game states: Boot, Preloader, Main-Menu, Game Settings, Lobby-Create, Lobby-Join, Character Selection, Arena Selection, Arena, and Results. This allows us to handle each page of our game individually and efficiently.
    - Arenas: Each Arena will be structured using a program called Tiled. This will let us graphically create each fighting stage, with several layers of collisions and aesthetics, and then store it as a json file.
- Relationships between Modules:
    - Our Arena game state will read in an Arena json file when it is being loaded. Each Arena json file will consist of several layers, some that will collide with the characters and some that will not. These layers will be named consistently with each different arena json file, so the game state will be able to properly read it.
- Interface Elements and Media Assets:
    - Main Menu:
        - Buttons:
            - Create Game: Loads the Lobby-Create game state
            - Join Game: Loads the Lobby-Join game state
            - Settings: Loads the Game Settings Game state
        - Sound:
            - A western-style soundtrack taken from a public-domain website and put on loop, fading in and out at the beginning and end. This soundtrack will continue for the next several game states, up until the users reach the character selection.
    - Lobby-Create
        - Text:
            - Party ID: Displays the party's randomly generated ID
        - Buttons:
            - Start: Once at least 2 players are in a party, all players must press this button, and then the game will proceed to the character selection game state.
    - Lobby-Join
        - Text Fields:
            - Party ID: The user enters the party ID here
        - Buttons:
            - Join Party: The user can press this button after entering an ID. This will call a method that will search for an active party, and if it exists, join the user to it. This will then cause the game to load the Lobby-Create game state, where they will wait in their party or start the game.

- Game Settings
  - Checkboxes:
    - Number of Rounds: The number of rounds a party wants to play. Winner just has to win the majority, like best 2 out of 3, or 3 out of 5, etc. Defaults to 1.
    - Round Time: The maximum amount of time for each round. Defaults to 120 seconds.
    - Lives: The amount of lives for each player, if any. Defaults to 1.
- Character Selection
  - Each user will have their column, which will contain two checkbox areas and a “Ready” button.
    - The first checkbox area will allow the user to choose their color. Options are Red/Green/Blue/Orange
    - The second checkbox area allows the user to choose their weapon. Options are Fists/Bat/Golf Club/Sword.
    - When the “Ready” button is pressed for all characters, the players game will load the Arena Selection game state.
- Arena Selection
  - Here each player will anonymously vote for an arena. Each Arena will be represented with a thumbnail image that acts as a button, and when the user clicks on any one of these, their vote is cast towards that Arena. After all players have voted on an Arena, the Arena with the highest votes, or if there’s a tie, the arena randomly chosen from the highest rated, will be selected and stored as a string to be passed to the next game state, the Arena game state.
- Arena
  - This section of the game will display the 2-dimensional tiled map for players to fight on. The map is built from the specified json file, which the users specified in the previous game state.
  - In addition to the Arena, some other UI elements include:
    - A timer at the top of the page
    - Health bars for each player (these are visual bars with numbers inside them representing their health)
- Results
  - After the fight ends, this game state is loaded and the winner for the round is shown.

System Architecture:



Classes (Pseudo code):

- class Player

```
Player(nameID){
        Set name ID
        Set player color
        Set player weapon
        Set initial score to zero
        Set weapons and moving objects to NULL until user joins
        Set position // this will be constantly updated, so the server will always
                know where each player is
}

Functions:
        getNameID() - user's unique ID
        getCurrentScore() - current score
        getCurrentWeapon() - weapon user is holding
        getPlayerColor() - user's color
        Make sure the score resets to zero after every round
```

- class Weapon

    Weapon(weaponType){

        Initialize weapon state to NULL
        Initialize weapon holder to NULL - weapons will be selected prior to match
        Create weapons - options are fists, sword, bat, or golf club
        Create attacks and movements with weapons and players
        isBeingUsed // Allows the server to know if the other clients should show animations for the weapon being used
        }

    Functions:

        setState() - the position of the weapon in player's hand
        setOwner(Player) - specifies what weapon a player is using
        setAttack() - how the player and weapon will move when attacking an opponent
        setDeffense() - how the player and weapon will move when defending against an attack by another opponent

- class Arena

    Arena(loadMap){

        Initialize timer
        Initialize players and their weapons
        Randomize or initialize how players spawn
        Initialize level size and objects in level
        }

    Functions:

        movePlayerRight(Player, direction) - will move player in the right facing direction
        movePlayerLeft(Player, direction) - will move player in the left-facing direction
        attack(Player, direction) - Player will attack in a specific direction
        defend(Player, direction) - Player will defend in a specific direction
        jump(Player, direction) - allows Player to jump a certain height in the air
        die(Player, direction) - If health reaches zero, player will fall to ground and then disappear from the screen
        gloat(Player, direction) - Victory dance for winner of match
        endGame() - Win timer reaches zero or Player defeats all other enemy Players; will bring Players back to main lobby

Programming Language: HTML 5 (HTML, CSS, and JavaScript)

External Libraries and Frameworks: We plan on only importing one library, Phaser, as our gaming library. We chose Phaser because it is up-to-date, works well, and one of our team members has experience using it. Phaser will supply support for sounds, collisions, and physics.

Client-Server Interaction: We are using socket.io as our middle-man for client-server interaction. Our Server will be constantly listening for events from all sockets, such as "move player" or "damage player" and "remove player." These events are triggered when a client announces them. Once the server is aware of the event, it sends the new data to all the clients, which are also listening for events such as these. This allows each client to move their local player, and have that movement reflected across all other clients. In addition to just moving the players, we can also activate their weapons, allow them to be damaged and die, and respawn.

Database Management: Firstly, we won't be storing things like player-profiles or achievements. Any achievements in the game will be local to a single session, and players will be unique through a username and a custom tumbleweed avatar. This will change every time the player refreshes the page.

Secondly, we will not use any SQL or any pseudo-SQL, to manage our player and class databases, as it's simply unnecessary. We will instead have a library of json files that contain all relevant information, such as the different image files that make up a player's avatar, since each avatar will be constructed from a template tumbleweed and overlays, such as clothing or weapons. Reading and writing these files will be done in real-time as needed.

Hosting: We will write our web server in Node.js and host it on an actual public domain, given to one of our team members at TAMUhack. This way, anyone can play our game without having to VPN into the TAMU network.

Communication between Clients and Server:
All of the communication that is done between Client and Server is done through socket.io. Socket connection happens immediately upon connection between the client and the server.