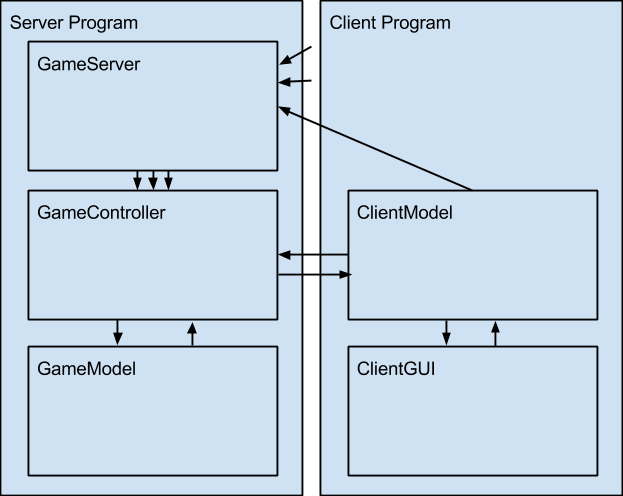
I propose that we use a client-server architecture that implements MVC (Model View Controller).

Class Diagram:



There is a single GameServer. It accepts connections from many ClientModels, each of which is connected internally to a ClientGUI. Each time it has two free ClientModels connected, it creates a new thread with a GameController connected to those two clients. Each GameController uses a GameModel to keep track of the board state.

I also suggest a **Client driven control flow**.

The client sends some sort of query to the server, and the server responds. The server does not initiate any communications, so the client only needs to listen immediately after sending a message to the server.

**Class Functionality:**

Server Side:

* + GameServer (for Multithreading):
    - Uses a loop to accept pairs of client connections.
    - Creates a separate thread for each game with 2 clients.
    - Can create multiple threads to handle multiple games running simultaneously.
    - OPTIONAL: Keep track of connected clients and allow clients who finish their game to rejoin the queue for a new game.
  + GameModel:
    - Keeps track of board state
      * Pieces on board
      * Win state (playing, tied, p1 won, p2 won)
      * OPTIONAL: Chat logs
    - Handles game logic (is it a legal move, is the game over, who won, etc)
    - Serves board state updates to the controller so it can send to clients.
  + GameController (implements Runnable):
    - Maintains connections to two clients.
    - Receives queries from clients
    - Parses queries as move requests, general board state update requests, or OPTIONAL: chat messages.
    - Updates model with moves and OPTIONAL: chat messages
    - Responds to client queries with a board state update taken from the game model.
* Client side:
  + ClientModel:
    - Connects to server (game controller) and maintains one connection to it.
    - Receives clicks from controller and sends move requests to server
    - OPTIONAL: Sends chat messages to server
    - Queries server for board state update at regular intervals (1 second?)
      * Updates view with board state
    - Listens after each query of any type for a board state update
    - To consider: Should the client model hold a board state and only receive updates from the game’s master model, or should the client receive a full copy of the board state every time it queries the game and not store the board state locally? (Affects GameModel, GameController)
  + ClientGUI (Client’s View+Controller)
    - Displays pieces on board as the board state depicts them
    - Parses mouse clicks on the board and sends to the client model
    - OPTIONAL: Chat
      * Displays chat logs as the board state remembers them
      * Takes text from a chatbox in the GUI and sends to client model

**Inter-Class Communication:**

The three classes within the Server program will communicate with each other normally, using method calls. The GameServer will create the GameController and a connected GameModel (pass GameModel into GameController constructor).

The two classes within the Client program will communicate with each other normally, using method calls. The ClientModel will create a new ClientGUI when it starts.

There will be a networking connection made between the ClientModel and the GameServer. The GameServer will then pass two connections to be managed by the GameController when it creates the GameController.

The most difficult part of this will come when working on communication between two classes being created by different people. We should discuss messaging standards + list public method signatures before beginning to code, so that we know how to communicate with each other’s classes. On that note…

**Division of Labor:**

The base functionality of GameServer is relatively simple. As such it makes sense for us to split the work between the four remaining classes:

1. GameController - Talk to the clients and the Game model, handle interactions and message bundling / parsing between them.
2. GameModel - Store data, make simple logical decisions about that data, and bundle board state to pass to the GameController.
3. ClientModel - Interface between GUI and Game. Send and parse messages to/from the GameController. Update the GUI when you receive board state updates.
4. ClientGUI - Primary goal is to make the GUI work and look good. Very little logical work. Your class is the face of the game! Create public methods which allow the ClientModel to perform specific actions in your GUI (like placing pieces, displaying winner, etc.)