Term-Project Report *Team: R2-D2:*

Due: Dec. 10, 2014 Anthony Morla

Thomas McNulty

Weslee Galloway

Ryan Ahern

**Five-In-A-Row Term Project Report**

**Introduction**

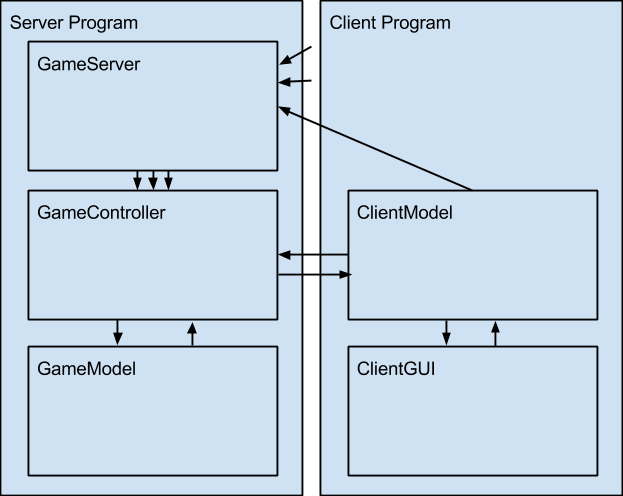
Our goal for this project was to design a two-player game, where two players (or clients) connect via a multi-threaded connection to a server. This server then allows for both players to manipulate the same 15x15 game board via a graphical user interface. Also, we want to allow a way for these two players to communicate throughout the duration of the game.

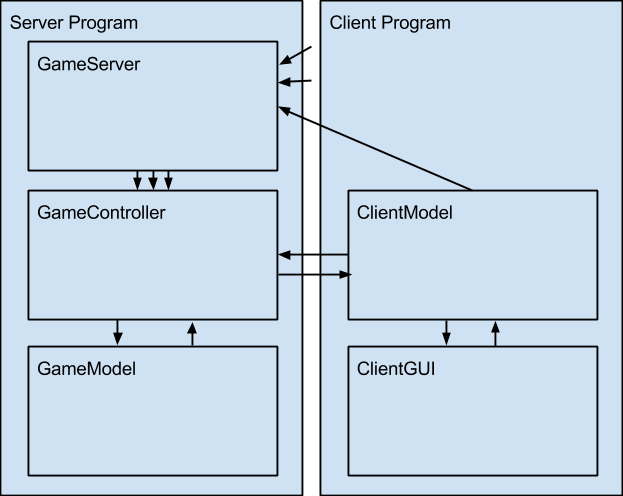
**Strategy of the Game**

Our Five-In-A-Row game is played similarly to the popular connect four game. The objective of the game is to be the first player to align your pieces in 5 consecutive spots on the game board. The orientation of these pieces can be horizontal, vertical, or any diagonal row. The size of the game board allows the player many potential combinations of winning marker placement. It is important to not only align your pieces into 5 consecutive markers but also to prevent your opponent from doing so using your markers to break any consecutive patterns.

**Overall Development & Flow Charts**

Our game is broken down as follows:





We used a client-server architecture that implements MVC (Model View Controller). 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. Our game follows 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)
      * 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
    - 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.

**Conclusion**

We divided work out well, the MVC model of our design allowed us all to work on a specific aspect of the game while at the same time being able to all contribute to the same project. We also used a repository (Github) to allow us to have version control on our project files, so no two people over-wrote anyone else’s work. Looking back, we believe the biggest issue was making all of our files work together. Making the GUI and the supporting files work together. Also ensuring that the game works properly in all situations given the user inputs.

**Future Works**

TBD