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**

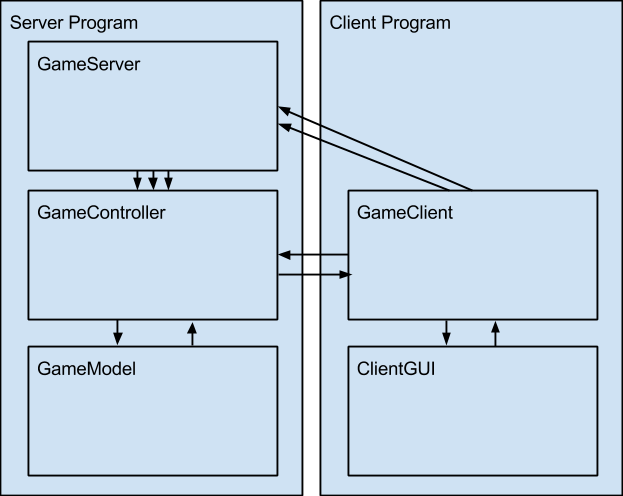
Our goal for this project was to design a two-player game, where two players (or clients) connect via a multi-threaded connection to a server. This server then allows for both players to manipulate the same 15x15 game board via a graphical user interface. Also, we want to allow a way for these two players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

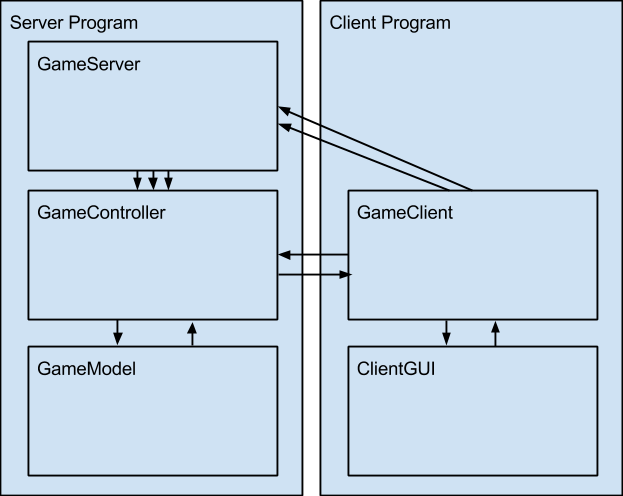
**Strategy of the Game**

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. Being able to place pieces so that they contribute to multiple potential avenues of 5 in a row is important.

**Flow Charts**

Our game is broken down as follows:





**Overall Development:**

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 creates its own ClientGUI. Whenever the server 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. The game controller controls the flow of turns, but it does not advance turns until the model accepts a move request generated by a client clicking on a spot in the GUI.

The client and server send communications back and forth. They use a helper class called R2D2Connection, which contains an internal queue of messages and a separate thread pulling data off the stream, to allow for simultaneous messaging. The GameClient and GameController each have an R2D2Connection object which they use to send Message objects to each other.

**Design Aspects:**

*Class Documentation:*

* Server package:
  + 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.
  + GameController (implements Runnable):
    - Maintains connections to two clients.
    - Receives move requests and chat messages from clients
    - Updates model with move requests
    - Sends clients confirmed moves, chat messages, and game info / setup messages.
  + GameModel:
    - Keeps track of board state: Past moves, winner
    - Handles game logic (is it a legal move, is the game over, who won, etc)
    - Informs the controller of board state information (when a move is recorded, when someone wins the game, etc).
* Client package:
  + GameClient (Client’s model):
    - Connects to server using an R2D2Connection
    - Receives click data from GUI and sends move requests to server
    - Receives strings from GUI and sends chat messages to server
    - Receives recorded moves and chat messages from server and updates GUI with move and chat data.
    - Receives info and hello messages from server and updates internal model + GUI as necessary.
  + 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 and notifications
      * Takes text from a chatbox in the GUI and sends to client model
* Messaging package:
  + Message: Generic interface for Message objects
  + HelloMessage: Message sent to clients on creation to give them an ID number
  + InfoMessage: Message sent to clients to tell them whose turn it is and who’s won
  + MoveMessage: Message sent between clients and server to request a move and to display an accepted move.
  + ChatMessage: Message sent between clients and server when a user inputs a chat message.
  + R2D2Connection: Uses internal Socket, ObjectInputStream, ObjectOutputStream, and queue to send and receive messages.

*Inter-Class Communication:*

The three classes within the Server program communicate with each other normally, using method calls. The GameServer creates the GameController and the GameController will create the GameModel. The two classes within the Client program also communicate with each other normally, using method calls. The GameClient creates a new ClientGUI when it starts. There is a networking connection made between the GameClient and the GameServer. The GameServer then passes the two client connections to be managed by the GameController when it creates the GameController.

**Conclusion**

The MVC model of our design allowed us to divide the work and all to work on a specific aspect / class of the game, while at the same time being able to effectively contribute to the same project. Anthony worked on the GameModel, Thomas worked on the GameController, Ryan worked on the GameClient, and Wes worked on the ClientGUI. We also used a Github repository to allow us to have version control on our project files, so no two people over-wrote each other’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. Fortunately, the interfaces and the Messaging package that we created before starting work on the main body of code gave us a structure for interclass communication. Also ensuring that the game works properly in all situations given the user inputs.

**Future Works**

* Maintain connection after the game ends so that you can continue to chat post-game
* Post-game menu which allows users to exit, play again, or play again vs. same opponent
* Keep track of a user’s win/loss record
* Display a message if the players tie (place tiles in all positions with no winner)