

## **Project Part 4: Progress Report 2**

### **Project**

**Team Name:** GBAR

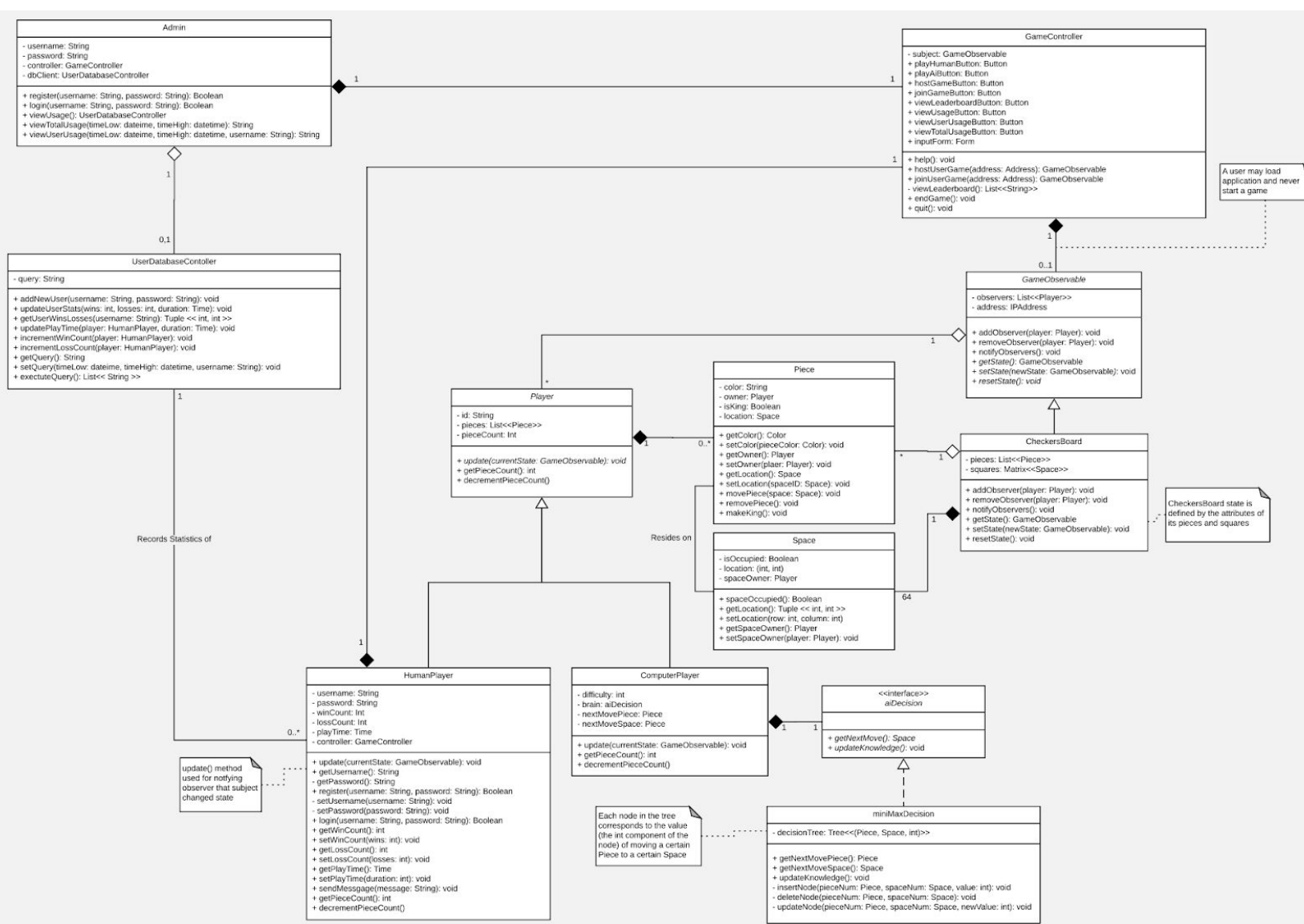
**Team Number:** 14

**Team Members:** Blake Galbavy, Austin Metz, Galen Pogoncheff, Richard Poulson

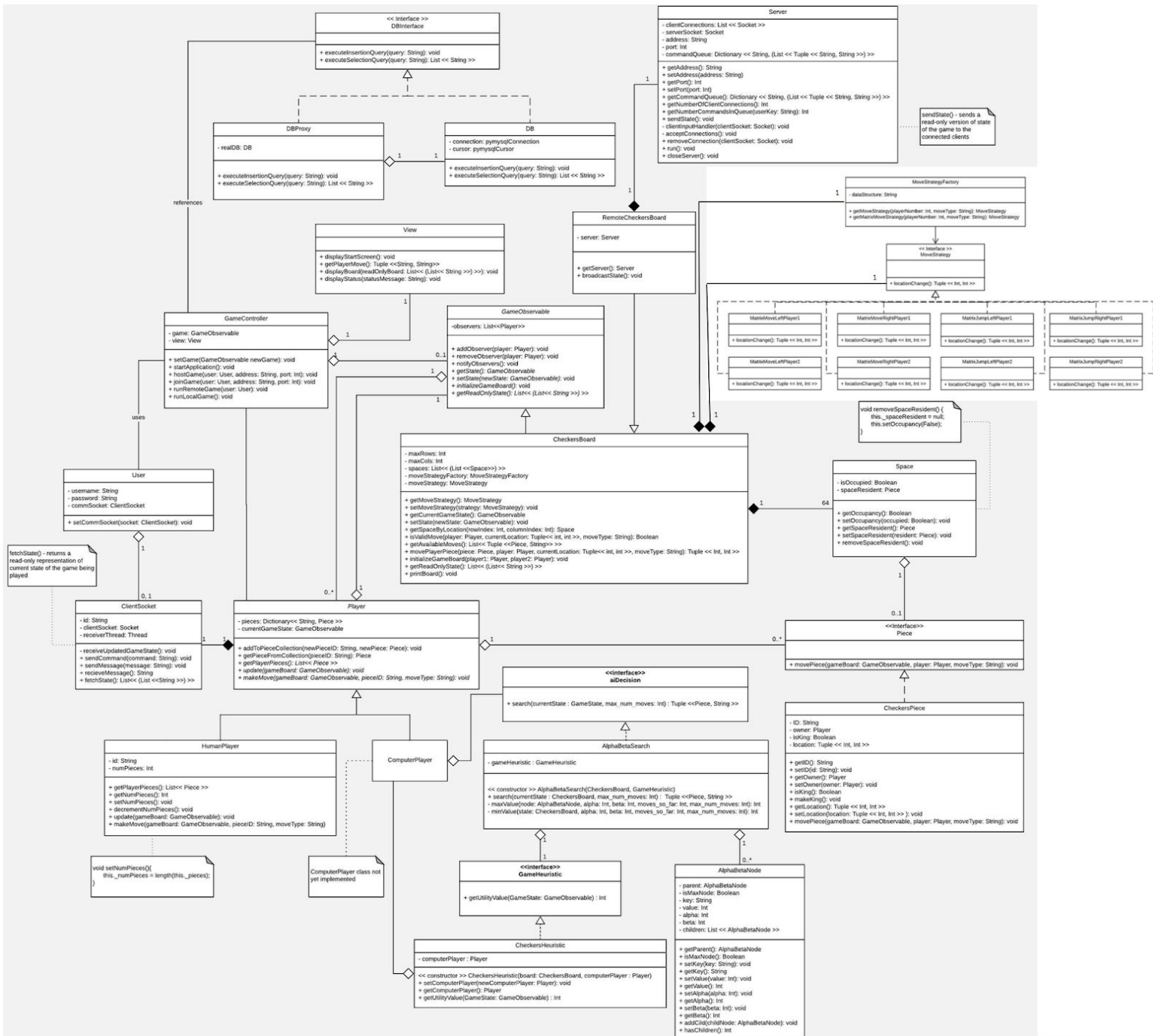
**Vision:** To create a Checkers game system that can be played both locally or remotely with a friend.

**Project Description:** 'Checkers' is an application that allows users to play the classic Checkers board game. Users of the application will have the ability to play checkers against other users on the network, as well as against an automated computer player (driven by artificial intelligence algorithms). Within the game, users can view and interact with a game board whose state reflects the moves of the players in real time. Users also have access to an in-game chat room with their opponent. User statistics, such as number of wins, number of losses, and total play time, will be recorded in a database. Standard users can access win/loss statistics through a leaderboard, and administrators can view user play time as they wish.

## Previous Class Diagram



## Completed Class Diagram



(See completed\_class\_diagram\_part4.png in our github home directory for a larger view of our completed class diagram.)

## Summary

During this iteration of the project, we have been working on implementing code to satisfy more of our project use cases, refactoring existing code, and integrating our system with an MVC architecture.

The work that has been done to satisfy more use cases includes the integration of a game server that allows a user to play a game of Checkers against another user over a network connection, continuing implementation of the classes associated with an artificial intelligence 'computer player' which will allow a user to play a game of Checkers against an artificial intelligence 'computer player', and work on a User class and Database Proxy which will allow for the recording of user gameplay statistic such as number of wins, number of losses, and total play time.

The refactoring of code that was done in this iteration included both the removal of 'code smells' and refactoring to patterns. An example 'code smell' refactoring that was performed in this iteration included replacing nested conditionals with guard clauses. Replacing nested conditionals with guard clauses was used in the `isValidMove()` method of the `CheckersBoard` class, which returns a boolean representing whether or not the movement of a game piece on the board was valid. We also identified code in our system that can utilize the Extract method. The `Server` and `ClientSocket` classes contain series of commands that frequently occur together for preparing data to be sent via sockets and for loading and preparing data that is received in socket communication. We will be replacing the series of commands for preparing data to be sent via sockets with a method, `prepareData()` that will wrap this series of commands. A similar approach will be used in creating a `unpackData()` method, which will load and prepare data that is received in socket communication. With respect to refactoring to patterns, we adopted the Strategy pattern by creating the `MoveStrategy` interface (implemented by `MatrixMoveLeftPlayer1`, `MatrixMoveLeftPlayer2`, `MatrixMoveRightPlayer1`, `MatrixMoveRightPlayer2`, `MatrixJumpLeftPlayer1`, `MatrixJumpLeftPlayer2`, `MatrixJumpRightPlayer1`, and `MatrixJumpRightPlayer2`) which includes the algorithm for determining how a game piece is moved on the game board. Prior to this refactoring, the piece movements were hardcoded in a dictionary. The dictionary mapped movement types (i.e., `moveLeft`, `moveRight`, `jumpLeft`, and `jumpRight`) to array indices that indicate the new indices of the piece on the game board after making this move. The piece movement in this implementation was therefore tightly coupled to the fact that the game board is represented as a 2D array. By using the Strategy pattern in `MoveStrategy`, if the data structure used for the game board were to change (for example, to a graph data structure), the only code that would need to change to enable the movement of pieces within this new data structure would be the addition of new movement strategies.

Regarding the implementation of the decision making algorithm (DMA), we now have a functional (albeit basic) heuristic function (`CheckersHeuristic`) that assigns a utility value to the current state of a given `CheckersBoard` object. This function was critical for the functionality of the DMA. Now that it's working, we can proceed to getting the DMA functional as well. We have been talking about possibly encapsulating the attributes that make up a game's state inside a separate class called `GameState`. However, after weighing the pros and cons, we decided to omit the `GameState` class and simply leave the attributes of a game's state inside the `CheckersBoard` class.

Lastly, another focus of this iteration was further implementation of the MVC architecture. The work completed in this area during this iteration involved separating responsibilities of each of the components of the MVC architecture and starting work on the View component of the MVC architecture.

Breakdown:

**Blake Galbavy:**

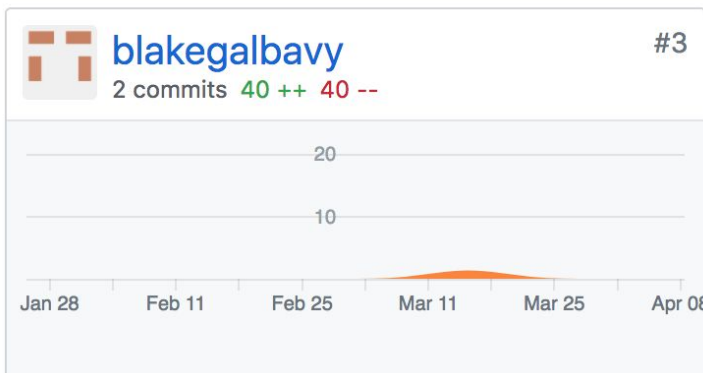
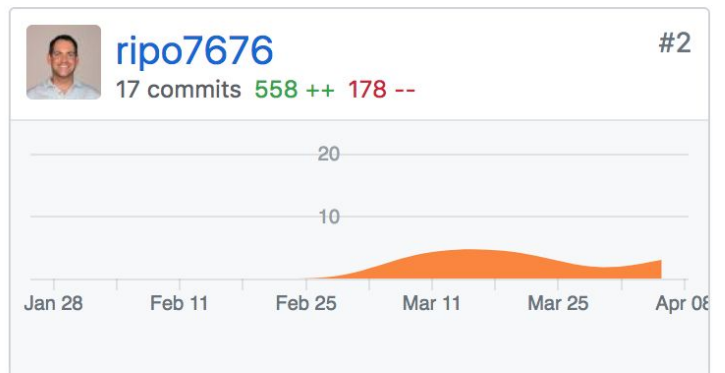
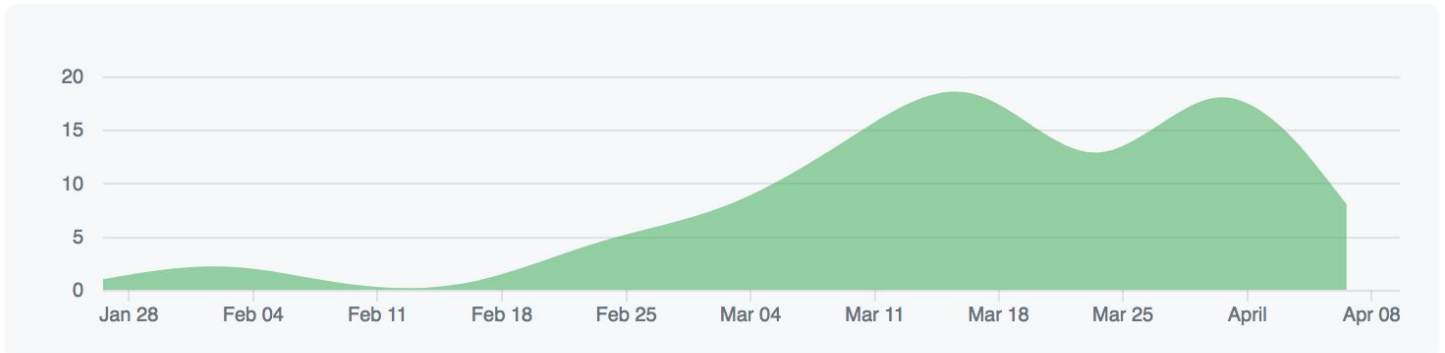
**Austin Metz:**

**Galen Pogoncheff:** Galen's work during this iteration consisted of implementing the components necessary for gameplay of a Checkers game over a network, implementing the classes required for database access from our system, integrating our system with a MVC architecture, and refactoring existing code. Integration of the Checkers game system with a game server involved integrating the classes associated with the game server (Server and ClientSocket) with the Checkers game itself, slight restructuring of the HumanPlayer class, and overall testing of this functionality. As a result, a user now has the ability to play a game of Checkers against another user over a network connection. Implementing database access from our system involved configuring a MySQL database and implementing the necessary classes to access and manage this database from our game system. These classes include DBInterface, DBProxy, and DB. Together they implement the Proxy Pattern. Work on the integration of a MVC architecture with our system involved separating responsibilities of each of the components of the MVC architecture within our system and starting work on the View component of the MVC architecture. Lastly, the refactoring that Galen did during this iteration involved the removal of a few 'code smells' that existed in our code and introducing of the Strategy Pattern for movement of a game piece on the game board and the Factory Pattern for instantiation of one of these many strategies. A full explanation of these 'code smell' and design pattern refactorings is provided in the Summary section above.

**Richard Poulson:** Some of the classes/interfaces that Richard has defined for the project this iteration include AlphaBetaNode, GameHeuristic and CheckersHeuristic. He has also continued to develop the AlphaBetaSearch class (decision-making algorithm), and has gotten the CheckersHeuristic getUtilityValue() method functional. Richard also defined some potential classes (PlayerTurn, PieceMovement, Event) that could be used in the project, time allowing. Richard is currently working on getting the methods in AlphaBetaSearch functional, and improving the heuristic function CheckersHeuristic.

## GitHub Graph:

Contributions to master, excluding merge commits



### Estimate Remaining Effort:

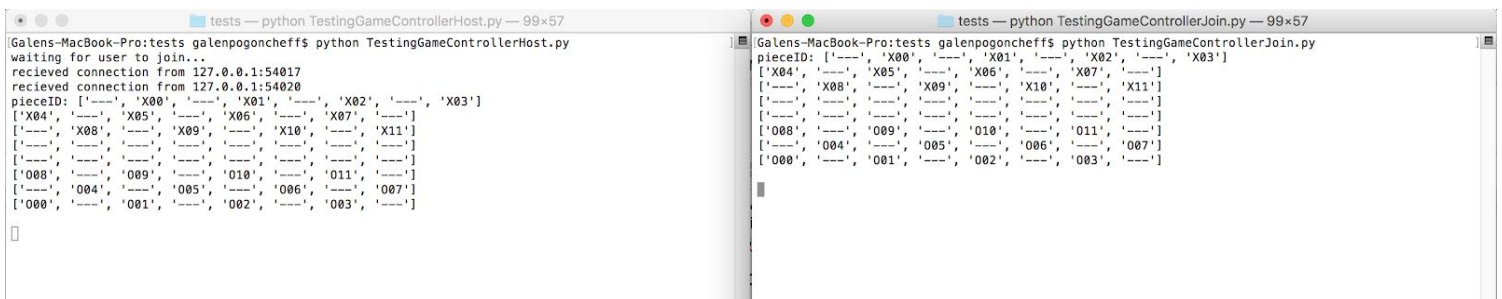
The majority of the design established in part 2 is complete at this point. Though, in order to have implemented all of the components of the design established in part 2, we must finish implementing the following. The ComputerPlayer class and its associated classes required to implement its artificial intelligence decision algorithms are currently in progress but need to be finished, we must add methods to the GameController to access and manage the system database, we must finish implementing the User and Admin classes, and lastly, we must finish implementing the view component of the MVC architecture.

To finish the AI decision algorithm, the existing AlphaBetaSearch (ABS) class will be adapted to use CheckersBoard objects instead of GameState objects which we predicted would be used in previous iterations. ABS will also be re-defined to use AlphaBetaNode as the nodes in a potential game state tree. The current heuristic function will be improved in order to assign a more accurate value of the game state.

Since altering the design of our system in the first iteration of implementation, we have only had to make slight changes to the design of our system. The most prominent reasons for these changes include better support of the Open-Closed Principle and maintaining high cohesion within our classes by creating a few new classes and leveraging composition.

Provided below are screenshots of our system in action:

Below shows the process of the user (in the terminal on the left) hosting a game, the user (in the terminal on the right) joining a game, and the initial remote game board being sent to, and displayed by, both users:



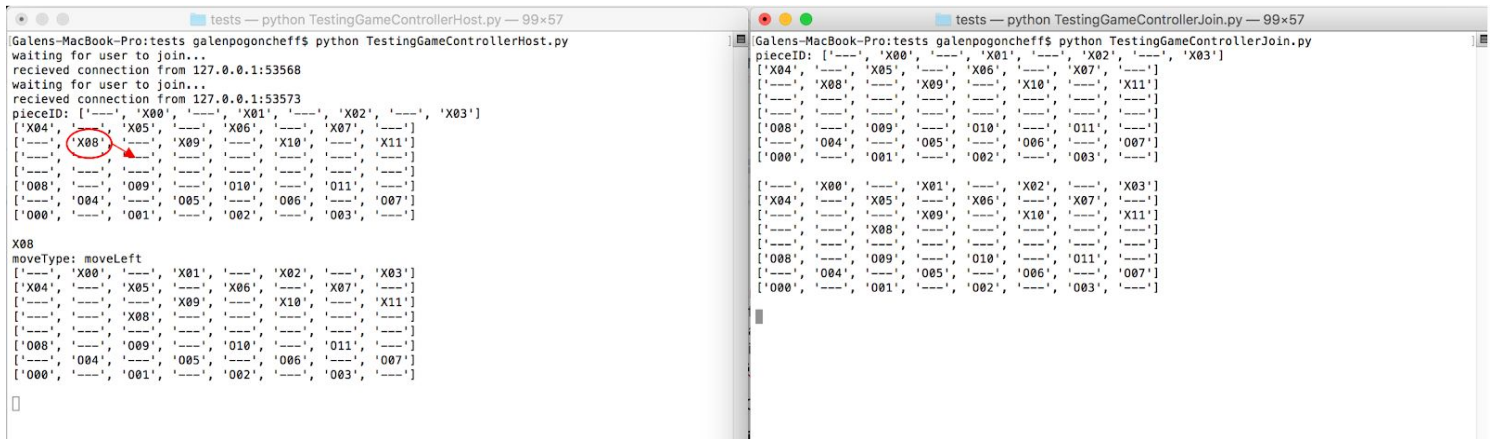
The image displays two side-by-side terminal windows. The left window, titled 'tests — python TestingGameControllerHost.py — 99x57', shows the execution of a Python script that waits for a user to join. It receives two connections from 127.0.0.1:54017 and 127.0.0.1:54020. It then sends a 'pieceID' list to the first connection and a full 11x11 game board state to the second. The board state is a list of lists, where each inner list represents a row of the board, containing piece IDs or empty strings. The right window, titled 'tests — python TestingGameControllerJoin.py — 99x57', shows the execution of a script that receives the 'pieceID' list and the full game board state from the host. It then displays the received data in the terminal.

```
Galens-MacBook-Pro:tests galenpogoncheff$ python TestingGameControllerHost.py
waiting for user to join...
received connection from 127.0.0.1:54017
received connection from 127.0.0.1:54020
pieceID: ['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

Galens-MacBook-Pro:tests galenpogoncheff$ python TestingGameControllerJoin.py
pieceID: ['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']
```



Hosting user moving piece 'X08' left (from the perspective of the piece) and the new game state being sent to, and displayed by, both users:



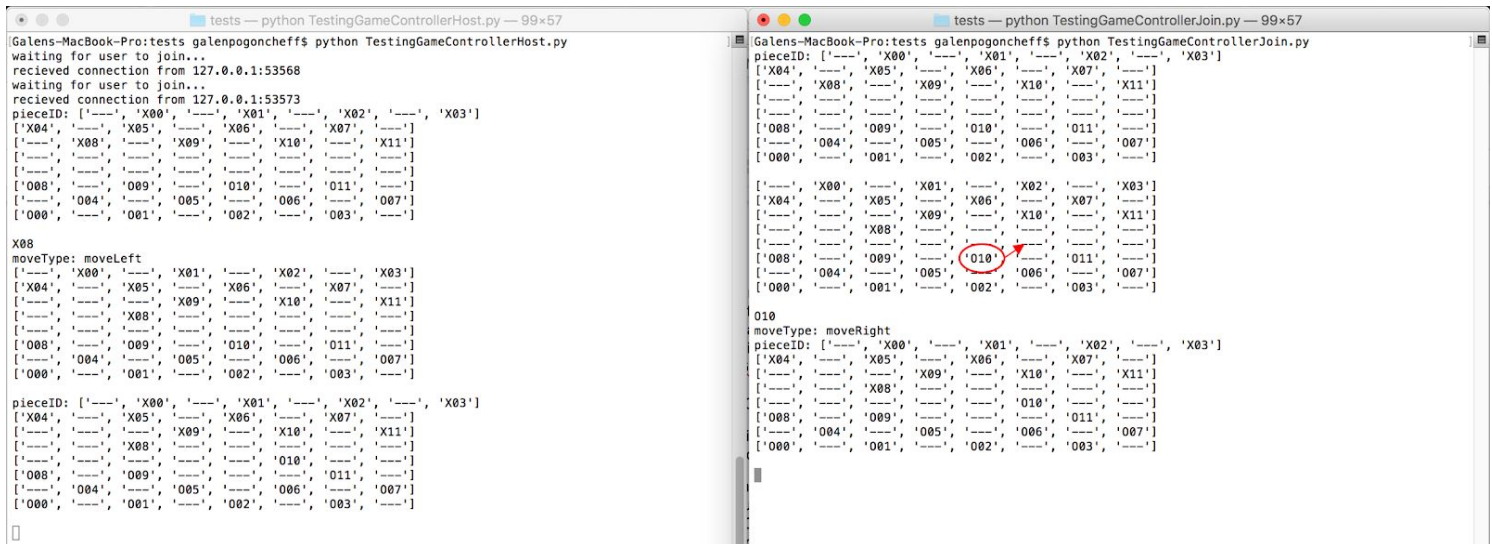
```
tests — python TestingGameControllerHost.py — 99x57
Galens-MacBook-Pro:tests galenpogoncheff$ python TestingGameControllerHost.py
waiting for user to join...
received connection from 127.0.0.1:53568
waiting for user to join...
received connection from 127.0.0.1:53573
pieceID: ['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

X08
moveType: moveLeft
['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', '---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

tests — python TestingGameControllerJoin.py — 99x57
Galens-MacBook-Pro:tests galenpogoncheff$ python TestingGameControllerJoin.py
pieceID: ['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', '---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']
```

Joining user moving piece 'O10' right (from the perspective of the piece) and then new game state being sent to, and displayed by, both users:



```
tests — python TestingGameControllerHost.py — 99x57
Galens-MacBook-Pro:tests galenpogoncheff$ python TestingGameControllerHost.py
waiting for user to join...
received connection from 127.0.0.1:53568
waiting for user to join...
received connection from 127.0.0.1:53573
pieceID: ['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

X08
moveType: moveLeft
['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', '---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

pieceID: ['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', '---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

tests — python TestingGameControllerJoin.py — 99x57
Galens-MacBook-Pro:tests galenpogoncheff$ python TestingGameControllerJoin.py
pieceID: ['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', '---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']

O10
moveType: moveRight
pieceID: ['---', 'X00', '---', 'X01', '---', 'X02', '---', 'X03']
['X04', '---', 'X05', '---', 'X06', '---', 'X07', '---']
['---', '---', 'X08', '---', 'X09', '---', 'X10', '---', 'X11']
['---', '---', '---', '---', '---', '---', '---']
['---', '---', '---', '---', '---', '---', '---']
['008', '---', '009', '---', '010', '---', '011', '---']
['---', '004', '---', '005', '---', '006', '---', '007']
['000', '---', '001', '---', '002', '---', '003', '---']
```

We have developed test cases for our system. The majority of these test cases revolve around testing the gameplay functionality of the system (testing that piece movements and the movement logic are properly functioning), testing the ability for two users to play against each other over a network connection, and testing access to our database from the game system. The local and remote gameplay tests involved both checking the outputs of the individual methods and testing the system as a whole through the user interface. The database access tests involve running a series of insertion and selection statements through the database proxy of our system and confirming that these statements had the intended effect on the database.

## Design Patterns:

We have incorporated four design patterns in our project thus far, the Strategy Pattern, the Factory Pattern, the Observer Pattern, and the Proxy Pattern.

We adopted the Strategy pattern by creating MoveStrategy, an interface (implemented by MatrixMoveLeftPlayer1, MatrixMoveLeftPlayer2, MatrixMoveRightPlayer1, MatrixMoveRightPlayer2, MatrixJumpLeftPlayer1, MatrixJumpLeftPlayer2, MatrixJumpRightPlayer1, and MatrixJumpRightPlayer2) which includes the algorithm for determining how a game piece is moved on the game board. A MoveStrategy object is incorporated into the GameObservable class via composition. When a player of the Checkers game moves a piece, the player does so with a specific movement type. For our Checkers system, these movement types include moveLeft, moveRight, jumpLeft, and jumpRight. Each movement type specifies the direction of the piece movement and how far the piece is moved (moveLeft moves a piece forward one space and to the left one space, moveRight moves a piece forward one space and to the right one space, jumpLeft moves a piece forward two spaces and to the left two spaces, and jumpRight moves a piece forward two spaces and to the right two spaces). The MoveStrategy interface is implemented by different algorithms for determining the direction and distance of the piece movement for a specific game board and specific player. Currently, the Checkers game board in our system is implemented as a 2D array. The strategy algorithms therefore calculate the change in array indices that occur as a result of the movement. However, by using polymorphism with the Strategy pattern, if the data structure used for the game board were to ever change (for example, to a graph data structure), the only code that would need to change to enable the movement of pieces within this new data structure would be the addition of new movement strategies. In order to instantiate a movement strategy object without having to code any conditional statements on the client side, these strategy objects are instantiated by a factory class, MoveStrategyFactory.

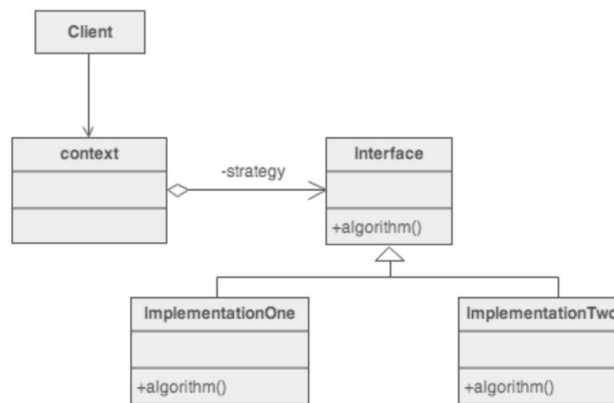
The Observer pattern is implemented in our system as follows. The subject of in this pattern is a class that implements the GameObservable interface. The observers of this subject are Player objects. When a user changes the state of a game board by moving a game piece, the subject notifies its observers that the state of the game board has changed. The observers then update their personal currentGameState attribute to reflect that of the current state of the game board. Keeping track of the current state of the game board is especially critical for ComputerPlayer objects, which automatically analyze the current state of the game board to decide the next move.

We are using the Proxy Pattern for access to the databases used by our system. To implement this pattern, we created a database interface, named DBInterface, which is implemented by both a database proxy class, DBProxy, and a class which contains the real database connection, DB. The methods executeInsertionQuery(query: String) and executeSelectionQuery(query: String): List <<String>> are implemented by DBProxy and DB. Composition is used in the DBProxy class, as it has an attribute of type DB. Implementing this structure as specified enables client objects to code to the DBInterface interface. The calls to executeInsertionQuery(query: String) and executeSelectionQuery(query: String): List <<String>> are made on the DBProxy object, which delegates these same operations to DB, the real database object.

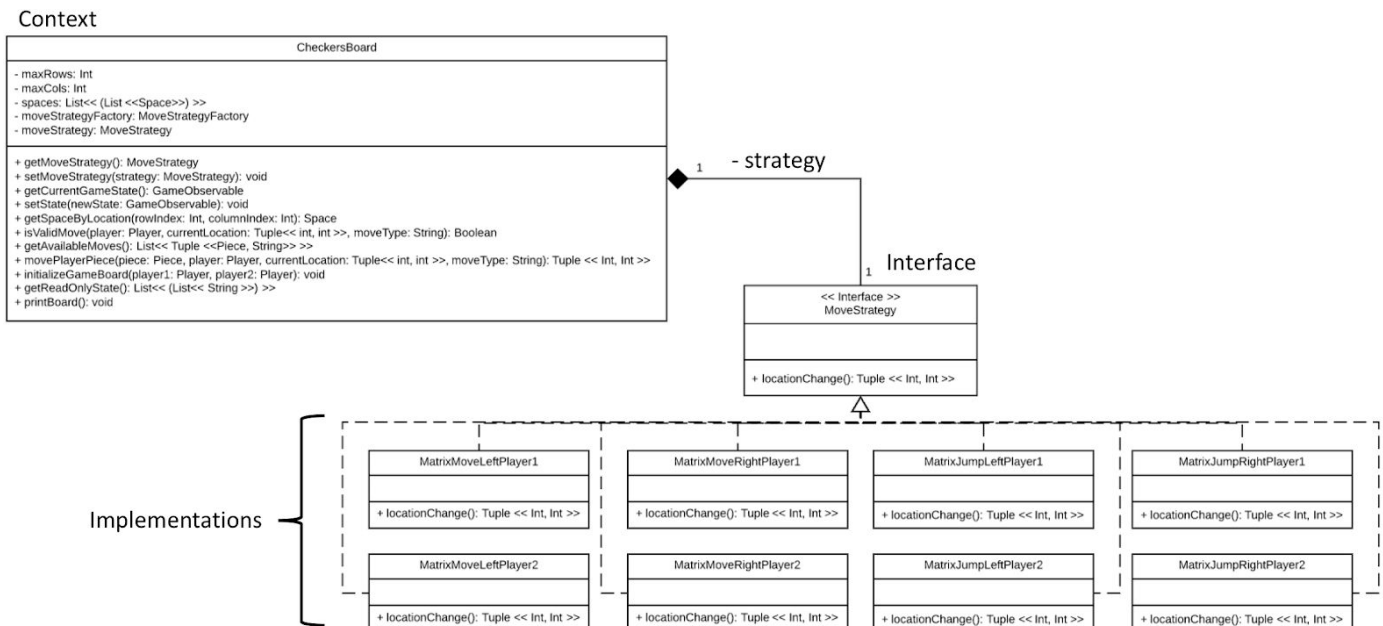
Provided below are images of the design patterns within our class diagram that have been implemented. For readability, some of the class containers have been moved from their position shown in our completed class diagram:

## Strategy Pattern:

Class diagram for actual design pattern:

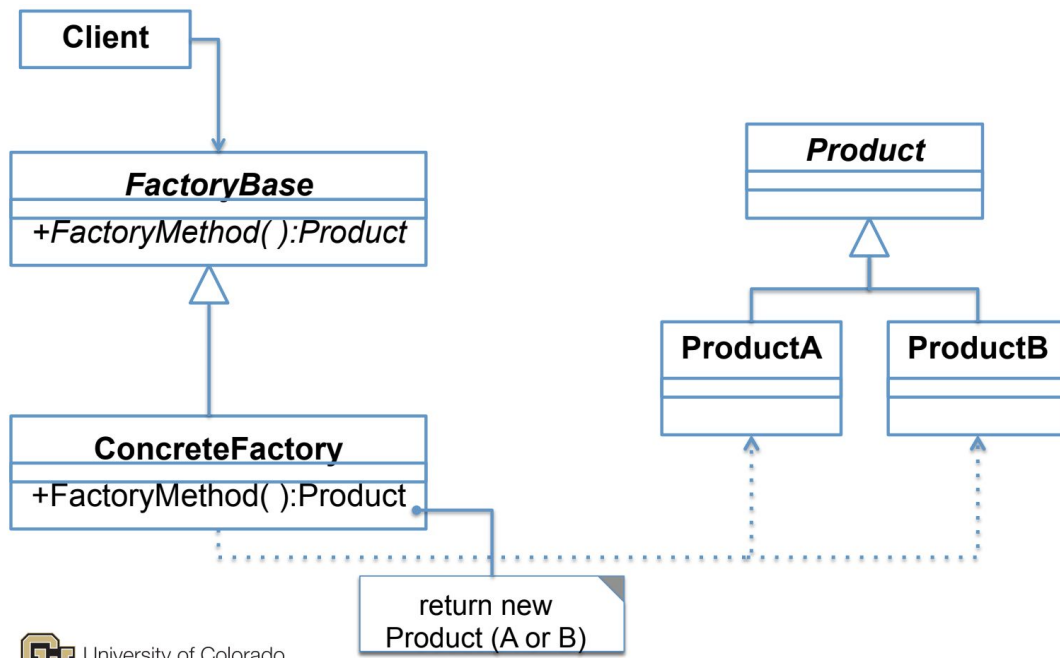


Portion of our class diagram that implements this pattern:

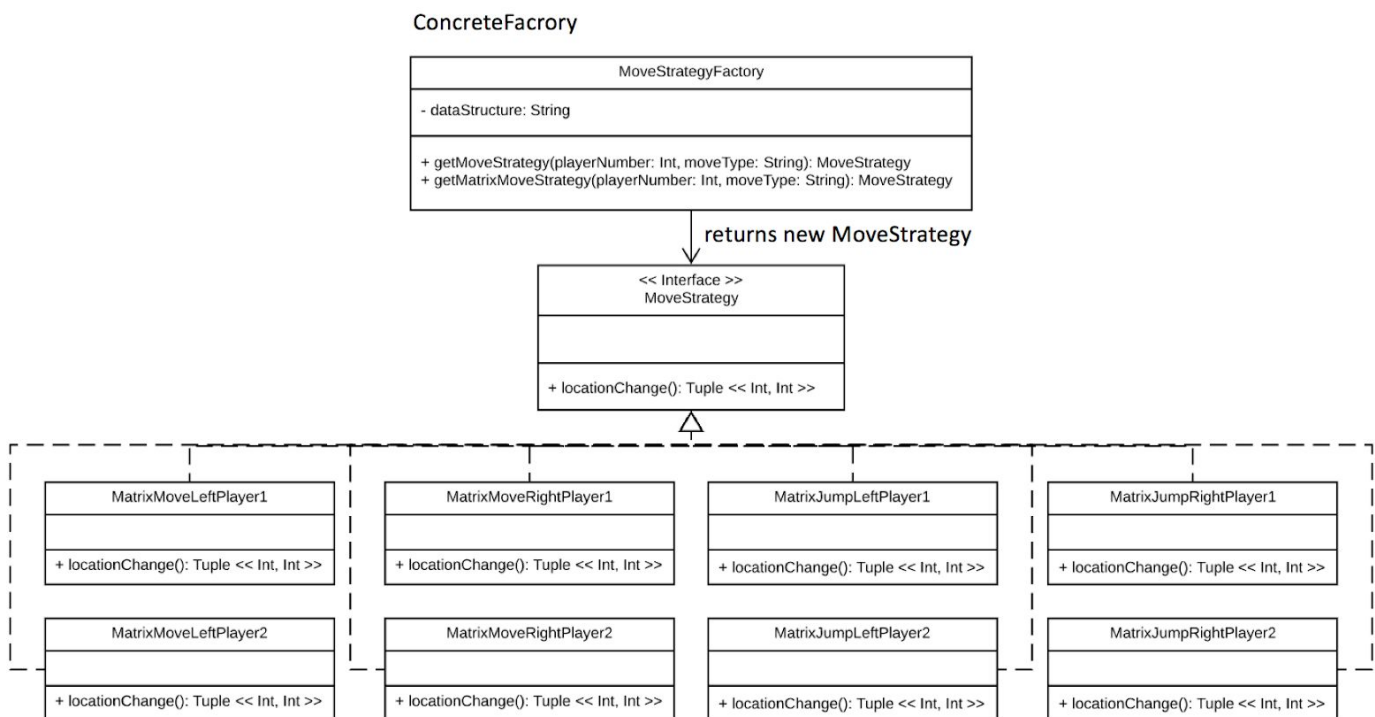


## Factory Pattern:

Class diagram for actual design pattern:

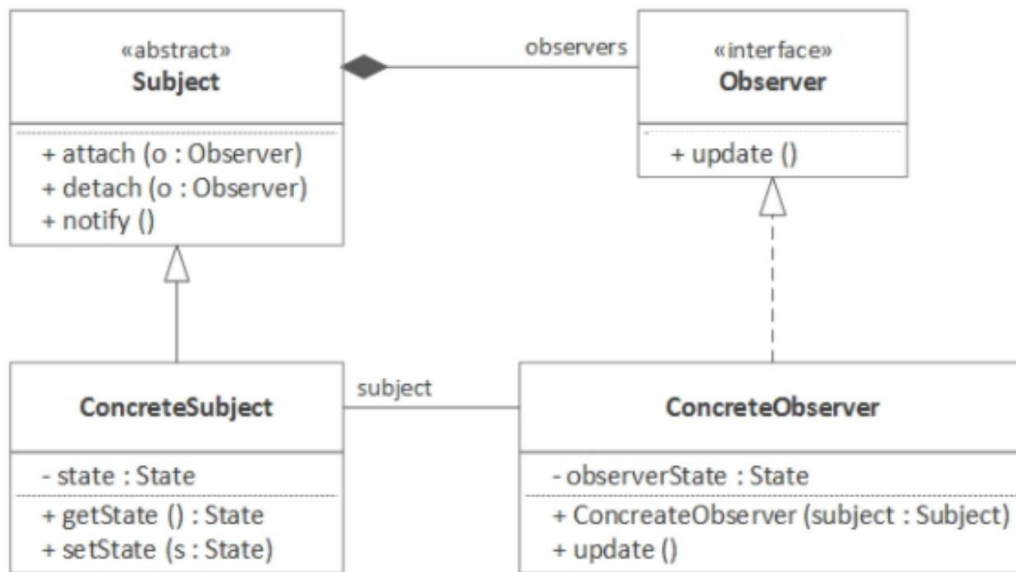


Portion of our class diagram that implements this pattern:

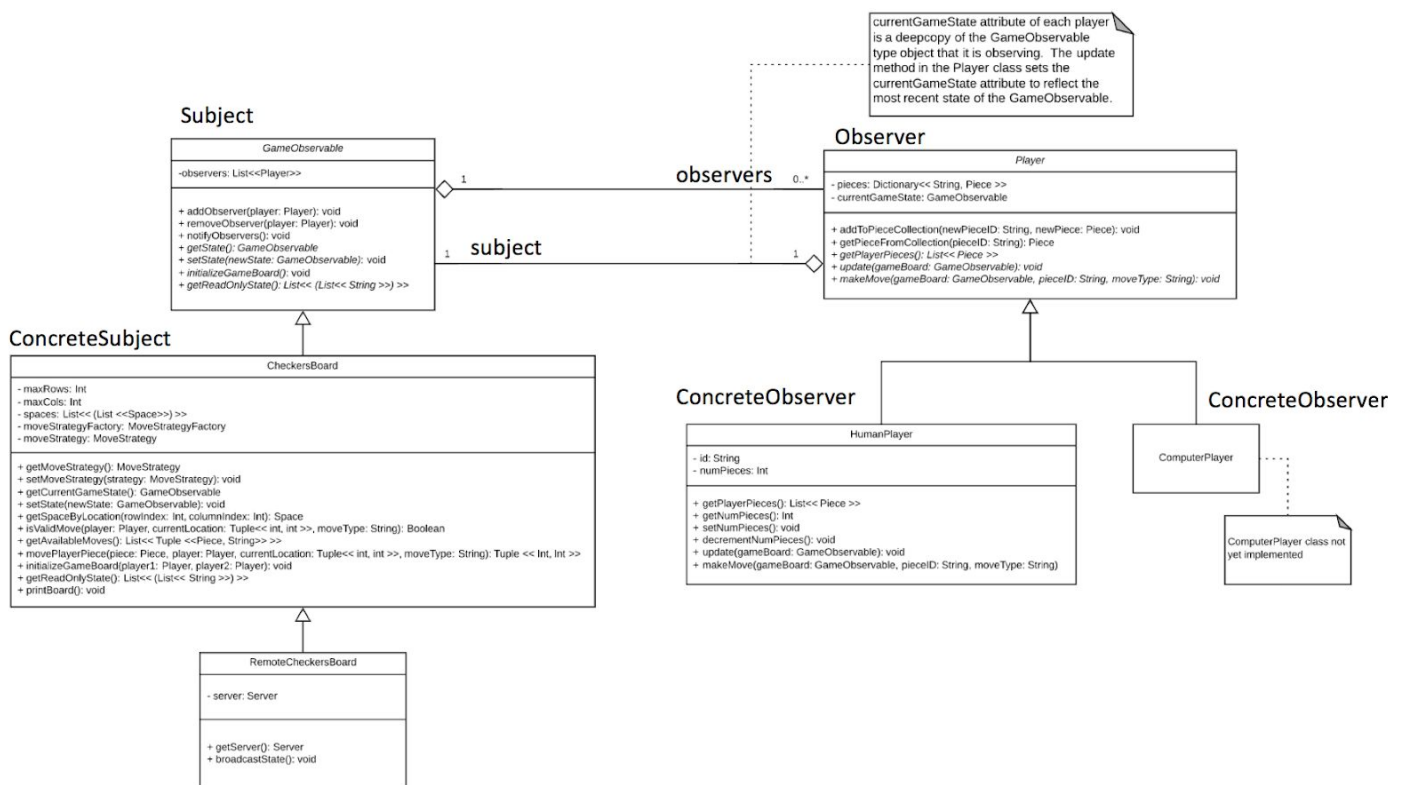


## Observer Pattern:

Class diagram for actual design pattern:

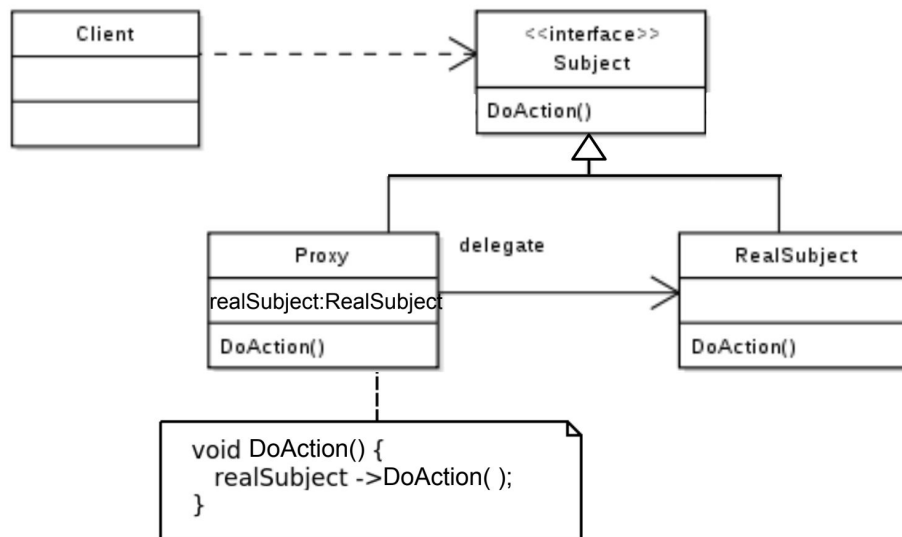


Portion of our class diagram that implements this pattern:

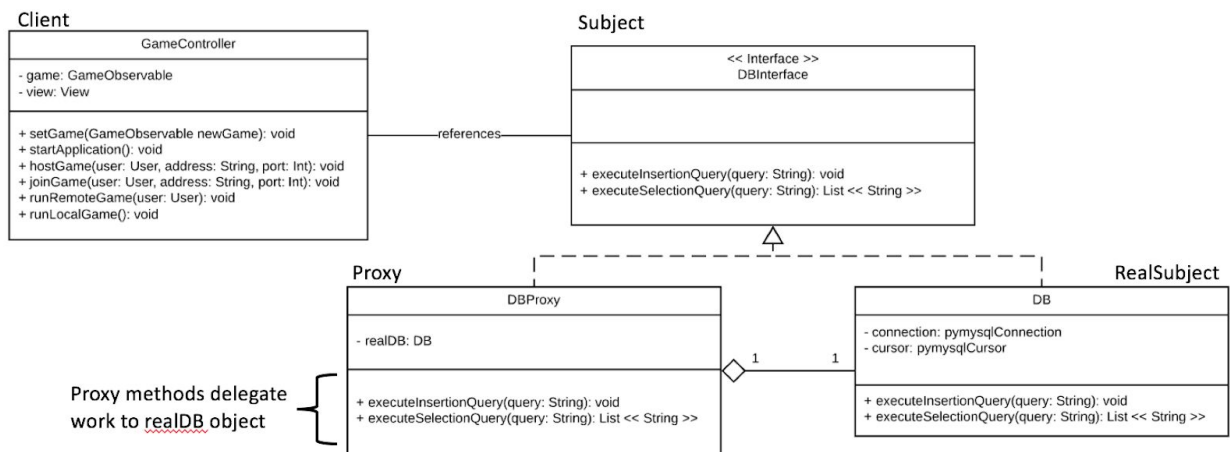


## Proxy Pattern:

Class diagram for actual design pattern:



Portion of our class diagram that implements this pattern:



## Final Iteration

For the final iteration of our project, our focus will be on implementing the use cases we established in part 2 that we have not yet completed, continuing to refactor our existing code, and finishing development of the view component of our MVC architecture. The highest priority use cases from part 2 that we hope to finish implementing are US-5, US-07, and US-10 (see below for use case descriptions associated with these use case IDs). The refactoring that we plan on doing during this final iteration will involve combing through our code base to identify and resolve 'code smells'. Our last area of focus during this final iteration is finishing a rough user interface for the view component of our MVC architecture. If time permits, we would like to add basic graphics to this interface as well.

### Use Case References:

US-05: As a user, I want the ability to play against an automated computer player so that if I cannot find another user to play against, I can still play a game of checkers.

US-07: As an administrator, I want to be able to view the play time of users so that I can monitor the system's use.

US-10: As a user, I want to be able to view my ranking in the system so that I can compare my proficiency with that of other users.