Project Part 7: Final Report

1. Project Team Name and #. List of team members. Vision. Project description.

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. User statistics, such as number of wins, number of losses, and total play time, are recorded in a database. Standard users can access win/loss statistics through a leaderboard, and administrators can view user play time as they wish.

2. List the features that were implemented (table with ID and title).

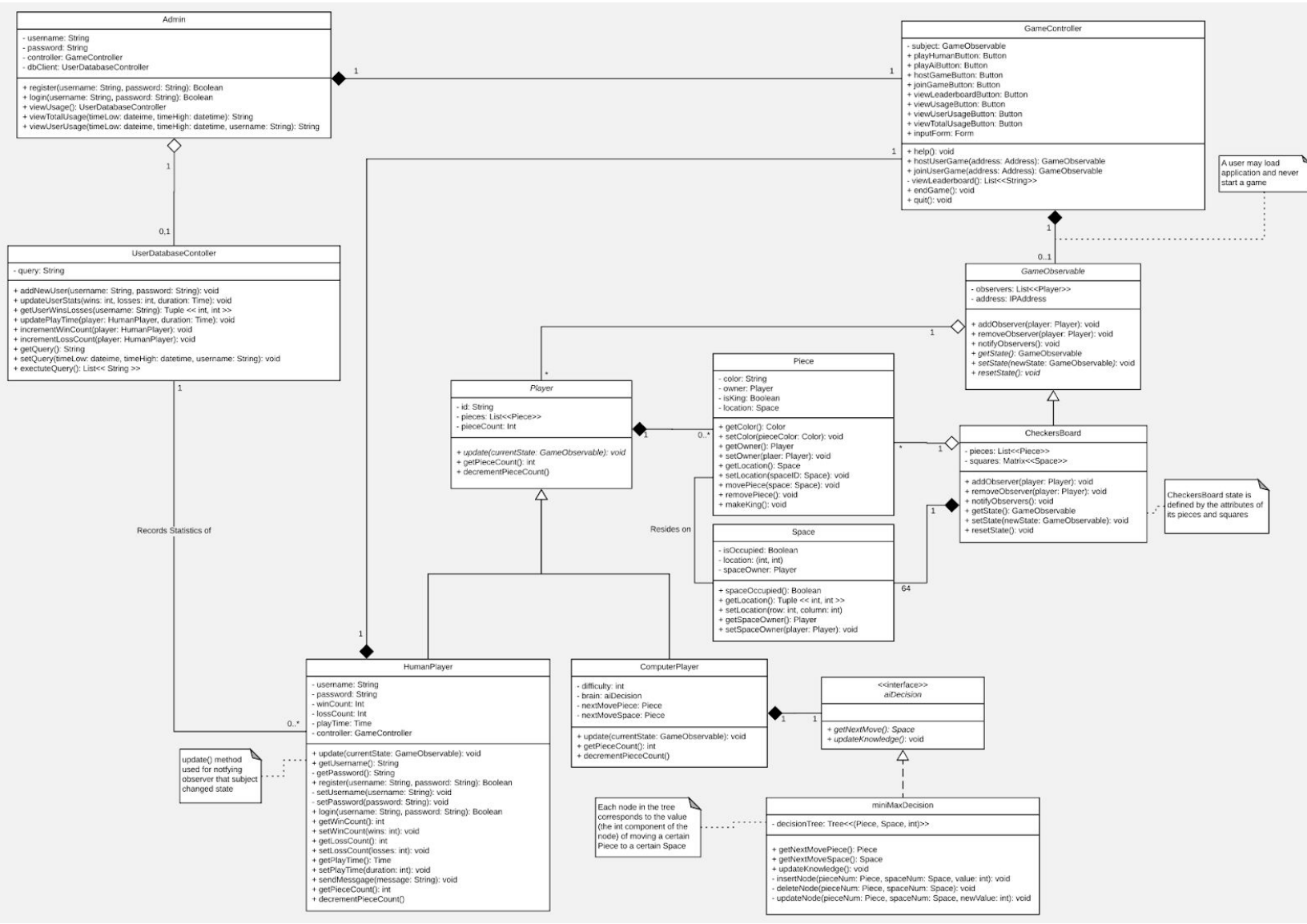
User Requirements		
ID	**Title**	**Description**
US-01	Host Game	As a user, I want to be able to host a game so that I can initialized a game to play with another user.
US-02	Join Game	As a user, I want to be able to join an already initialized game so that I can play a game against other users on the network.
US-03	Play against user	As a user, I want to be able to play a game of checkers against another user so that I can play against my friends.
US-05	Play AI-Player	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-06	Display Game	As a user, I want to see a visual representation of the game being played so that I can easily see the current state of the game.
US-07	View Usage	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-08	View Help	As a new user, I want access to a "Help" page so that I can learn the rules of the game and use of the application.
US-09	Change AI Difficulty	As a frequent user, I want to be able to change the difficulty of the computer player so that I can continue to challenge my skills as I progress.
US-10	View Ranking	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.

3. List the features were not implemented from Part 2 (table with ID and title).

User Requirements		
ID	**Title**	**Description**
US-04	Chat with Opponent	As a user, I want to be able to chat with my opponent so that we can communicate with each other without being in the same location.

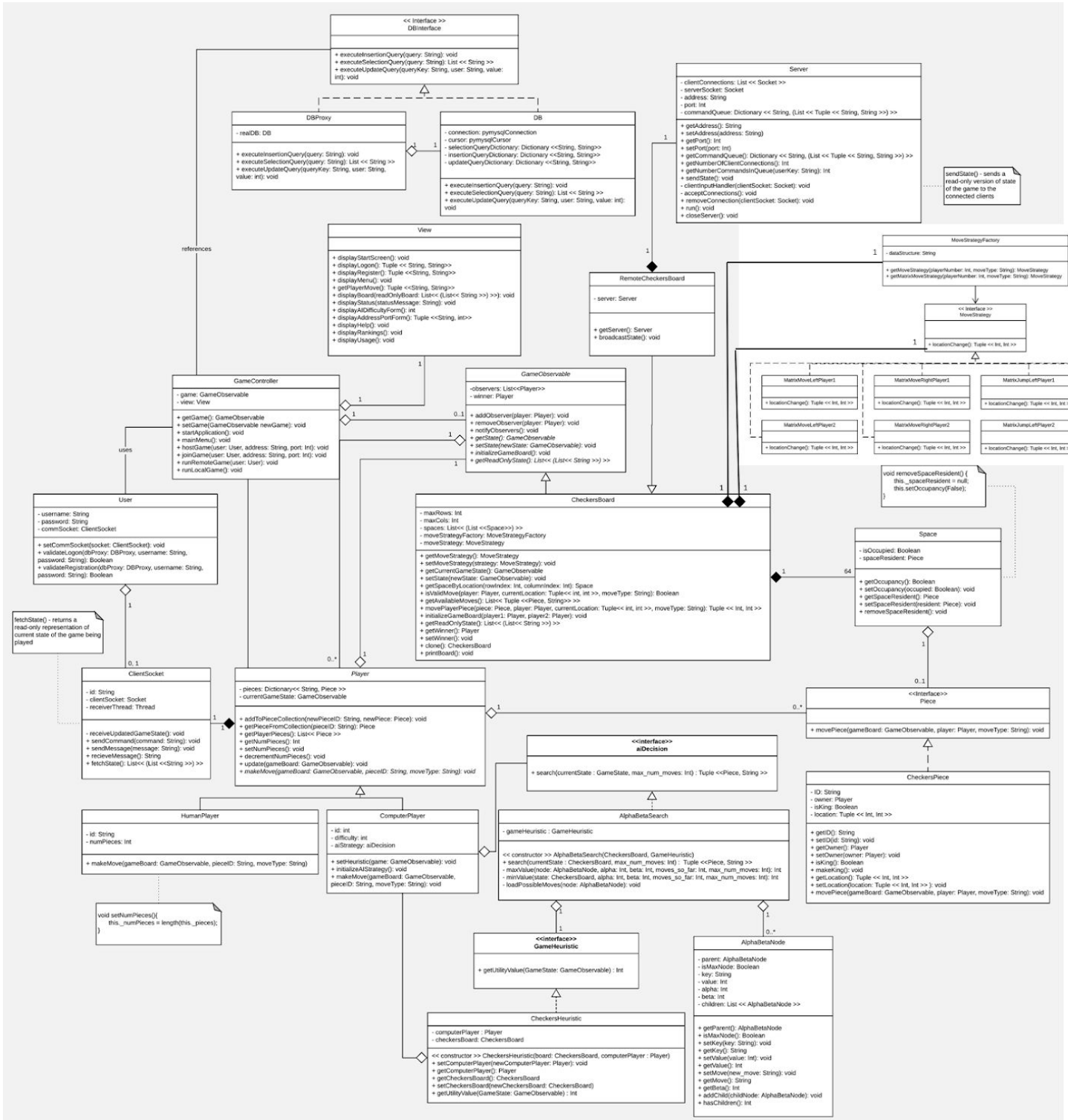
4. Show your Part 2 class diagram and your final class diagram. What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

Part 2 Class Diagram:



Completed Class Diagram:

(See completed_class_diagram_part7.png in our github home directory for a larger view of our completed class diagram.)



Since creating our initial system design in Part 2 of this project, our design has changed considerably. The two primary reasons for our design changes were to better support the SOLID design principles and to utilize appropriate design patterns that were learned about since our initial system design.

Shortcomings in support for the single responsibility principle and the open-closed principle were flaws in our design that our team identified once we began implementation of our system. Components of our original design that demonstrate these flaws include, but are not limited to, the methods included in the Admin and HumanPlayer classes, and the implementation dependent coupling between the CheckersBoard, Piece, and Space classes. In our original system design, the Admin and HumanPlayer classes failed to support the single responsibility principle. The Admin class contained methods such as viewUsage(), viewUserUsage(), and viewTotalUsage(). These methods are responsible for retrieving the system usage statistics from the system database. Although administrative users are intended to have the privilege to view these statistics, they are not pertinent to the Admin class itself. In revising our system design, these method functionalities were moved from the Admin class to the GameController class, which delegates the work of retrieving information from the database to the DBProxy class. Along the same lines, the HumanPlayer class in our original design includes methods that are of interest to system users, such as setUsername() and setPassword(). In order to improve cohesion within the HumanPlayer class and better support the single responsibility principle, these methods were moved to a User class that was created in our updated system design. Coupling between the CheckersBoard, Piece, and Space classes in our original system design resulted in issues with supporting the open-closed principle. During implementation, we found that these classes were originally highly dependent on each class' specific implementation. Whenever we changed the details of one class, the other associated classes were required to change as well. By introducing the Strategy pattern and leveraging sources of composition, these coupled classes became not only less dependent on one another's implementations, but also better encapsulated.

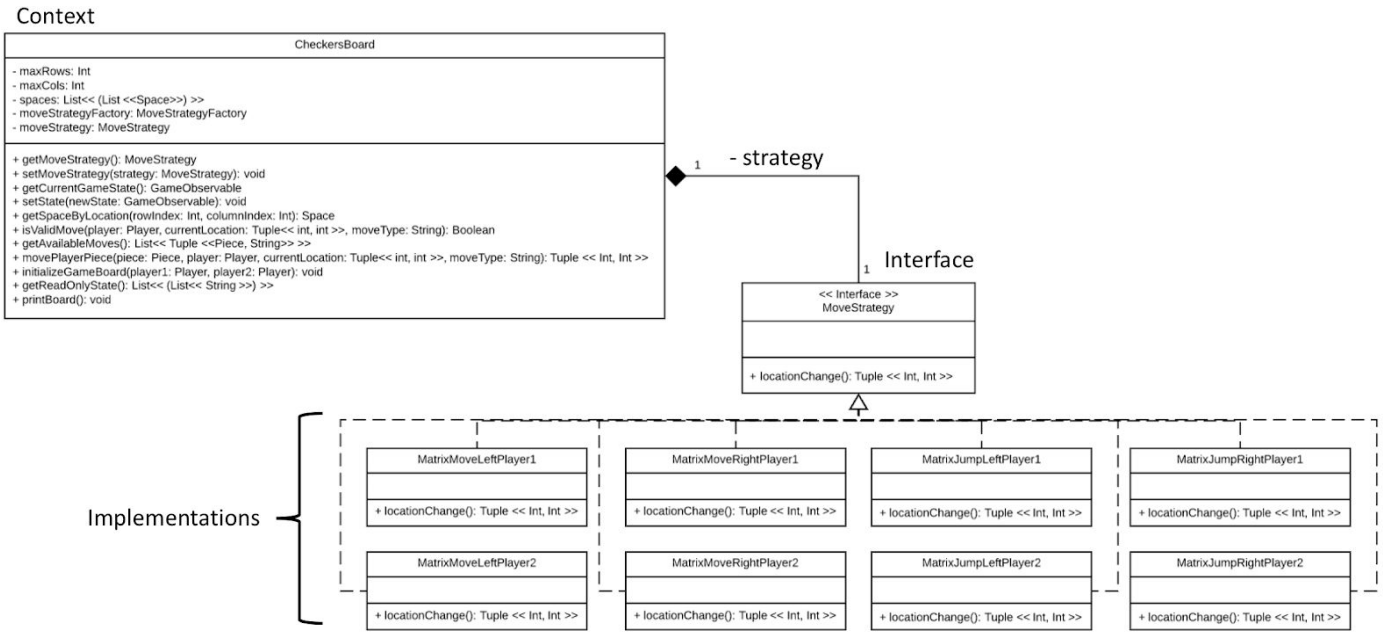
As previously mentioned, other significant changes in our system design were due to the incorporation of design patterns. The design patterns that we utilized in our system include the Strategy pattern, the Factory pattern, the Observer pattern, and the Proxy pattern. Detailed explanations of how these patterns were incorporated into our system design are included in the next section of this document.

Although the changes we made to our original system design resulted in the addition of numerous classes, this has allowed our overall system to become more flexible since each class is better encapsulated. Our team found that the tradeoff of having more classes, and a higher degree of coupling due to composition between many of these classes, for a higher degree of cohesion within the classes and better support of the the SOLID principles was beneficial to our system overall. Having spent a significant amount of time revising our system design during the implementation process, we learned that the cost of taking the time to develop a sound system design up front is minute compared to the effort required if a system must be redesigned during the implementation phase.

5. Show the classes from your class diagram that implement each design pattern (each design pattern as a separate image in the .PDF).

Strategy Pattern:

Classes from our class diagram that implement this pattern:

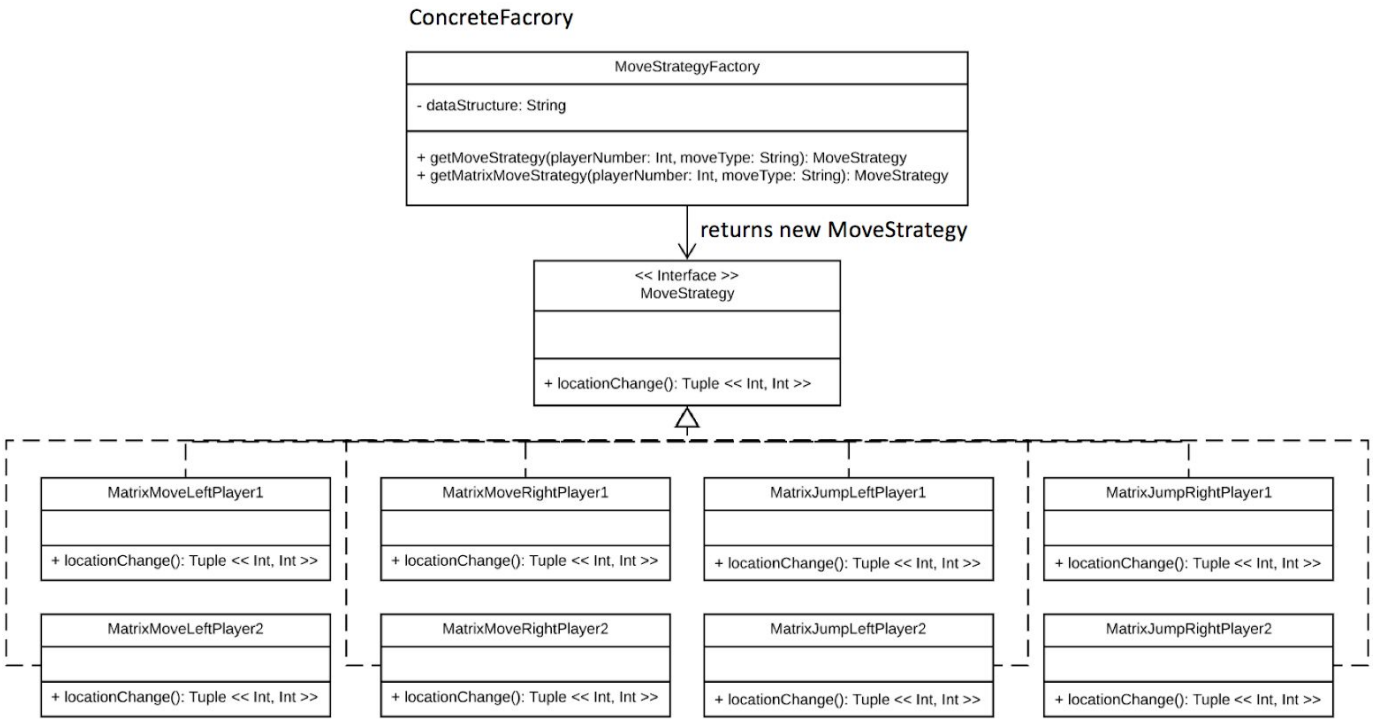


How this pattern was used in our system:

We adopted the Strategy pattern by creating MoveStrategy, an interface which includes the algorithm for determining how a game piece is moved on the game board. A MoveStrategy object is incorporated into the CheckersBoard 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. 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 be added to enable the movement of pieces within this new data structure would be the addition of new movement strategies.

Factory Pattern:

Classes from our class diagram that implement this pattern:

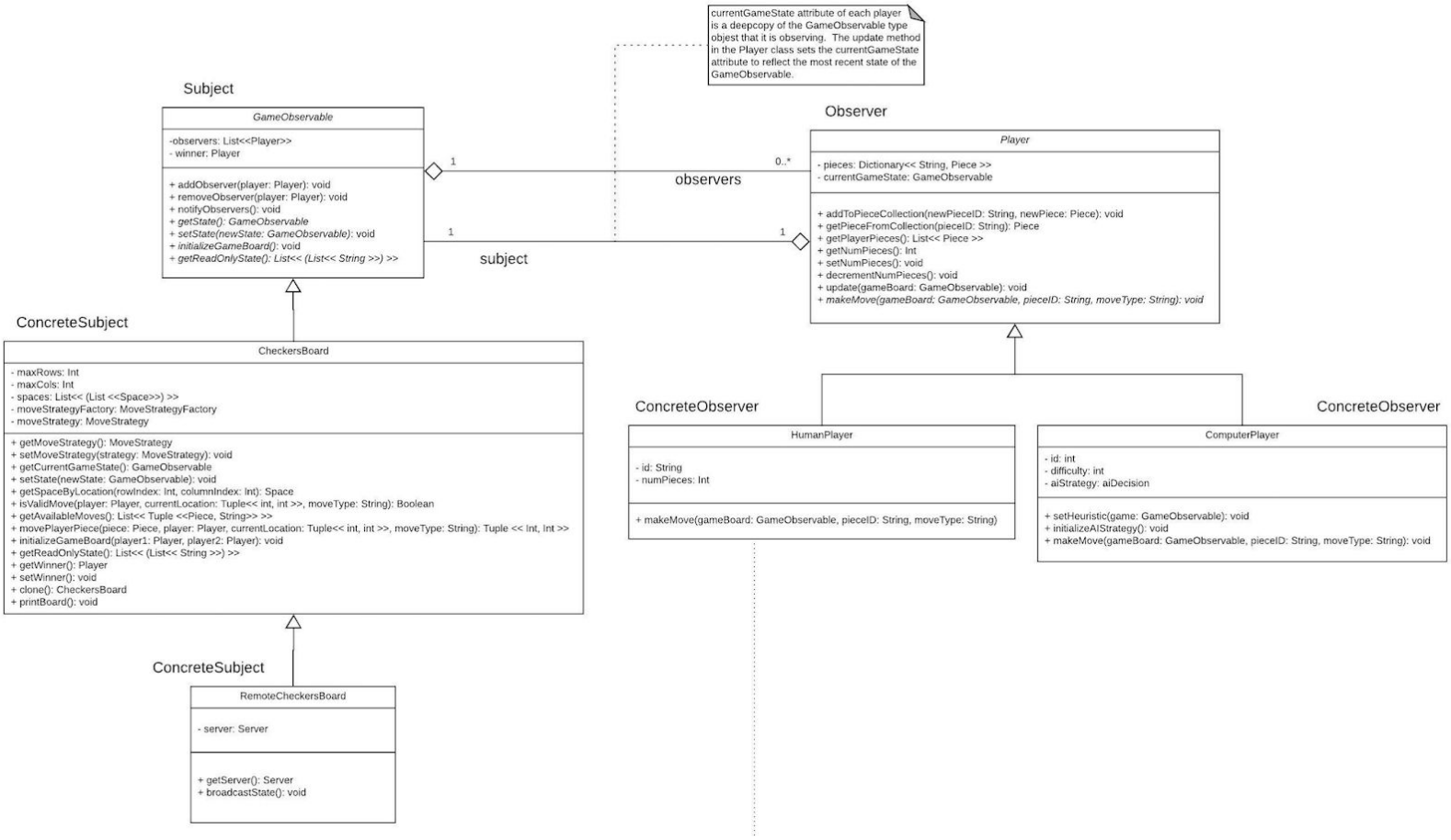


How this pattern was used in our system:

The MoveStrategyFactory class returns a MoveStrategy object when a game player moves a game piece. 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 the factory class, MoveStrategyFactory.

Observer Pattern:

Classes from our class diagram that implement this pattern:

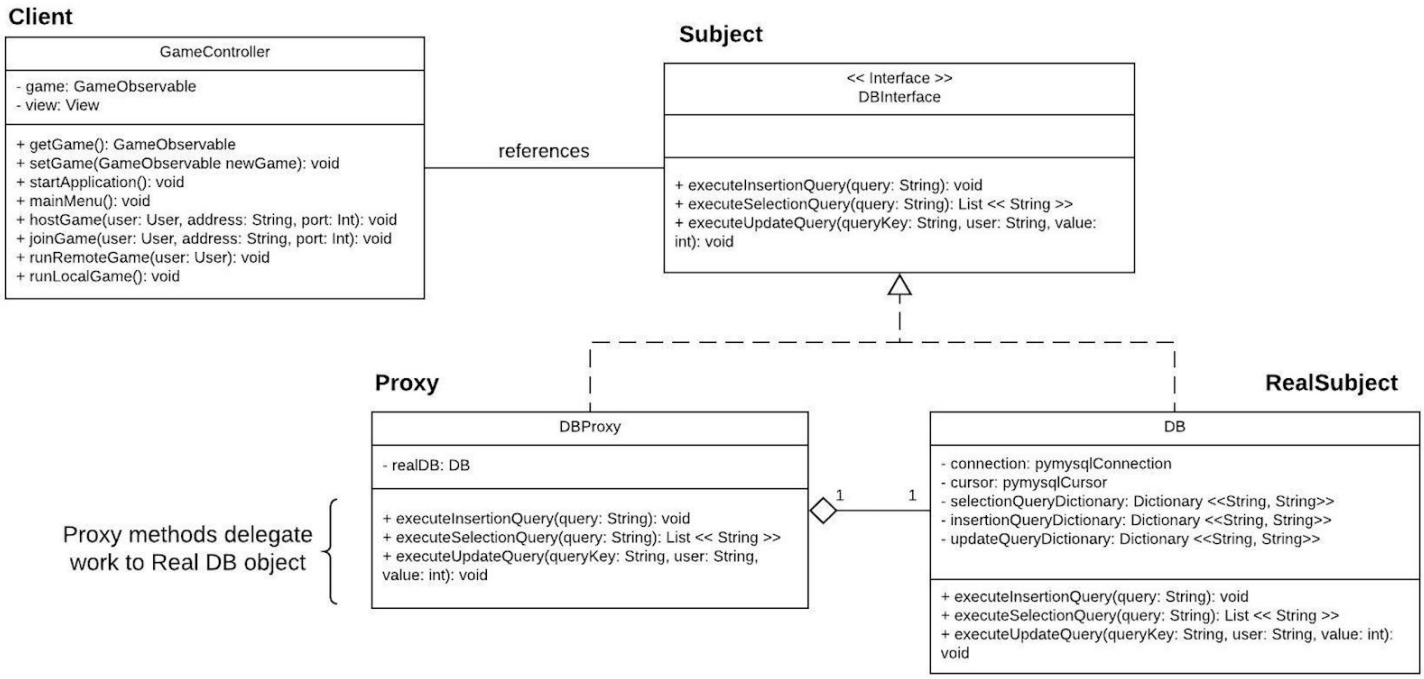


How this pattern was used in our system:

The subject in this pattern is a class that inherit from the GameObservable abstraction. 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.

Proxy Pattern:

Classes from our class diagram that implement this pattern:



How this pattern was used in our system:

In our system, the Proxy Pattern is used for access to the databases used by our system. The database interface, DBInterface, is implemented by both a database proxy class, DBProxy, and a class which contains the real database connection, 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(...), executeSelectionQuery(...), and executeUpdateQuery(...) are made on the DBProxy object, which delegates these same operations to DB, the real database object.

6. What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

Throughout the process of working on this project, our group has achieved a much better understanding of the theory behind good object oriented design and analysis, how to apply this theory to a real system, and the tradeoffs associated with design and implementation decisions.

Prior to creating an overall system design, the first step of this project was defining system requirements and associated use case documents, activity diagrams, and sequence diagrams. After taking the time to work through each of these tasks, we were able to obtain a much better understanding of the layout of our system. Specifically, the use case documents and activity diagrams provided us with a stronger grasp of how we intended a user to utilize the system, and the sequence diagram helped guide our design by forcing us to think through interactions between individual system components. Overall, this first iteration of the project gave us a better understanding of how to map abstract requirements to a concrete system design.

It was in the first iteration of system implementation where we learned most about the object oriented design and analysis. After beginning to implement the system we had originally designed, we were soon running into crucial issues with object oriented principles, most notably lack of support for the single responsibility and open-closed principles, low levels of code reuse, and the fact that our original design featured a high degree of coupling between the implementations of classes throughout the system (which hinted at the failure to use polymorphism to code to an interface/abstraction). These principle violations made it incredibly difficult to implement new features. They also made the system intolerant to change in the underlying implementation. At this point in the project, we took it upon ourselves to redesign much of our system. During system redesign, we employed careful thought into analyzing our original system and designing new solutions that would be more more extensible and maintainable as we continued development. This redesign process helped us learn a lot about the motivation behind the design principles that we had been studying in class.

As we continued to implement the functions of our system, we became keen on recognizing areas in our system that would benefit from certain design patterns. Discovering these areas where we could leverage the use of an already solved problem was both exciting to us and beneficial to our system implementation. Tailoring these design patterns to incorporate them into our system yielded a better understanding of how each aspect of these design patterns work.

Two tasks during this project gave us greater insight into the cost-benefit trade offs within development of an object oriented software system. The first of these tasks was the system redesign that our group performed. Redesigning our system after already beginning implementation took a considerable amount of effort. From this we learned that the up front cost of developing a well designed system in the first place is negligible compared to the time spent modifying system design after implementation has begun. The second of these tasks was the code refactoring that we decided to continuously work on. Although the refactoring that we

executed took time away from implementing features, it did, in the end, permit more efficient coding as we added new features and allowed us to track down potential bugs.

After completing our project, our group took the time to look at some of the metrics used to evaluated object oriented systems. Doing this enabled us to identify areas of our system that could be improved. Using these metrics to review our system as a whole compounded the object oriented analysis skills that we developed while modifying and inspecting our system design in previous iterations.