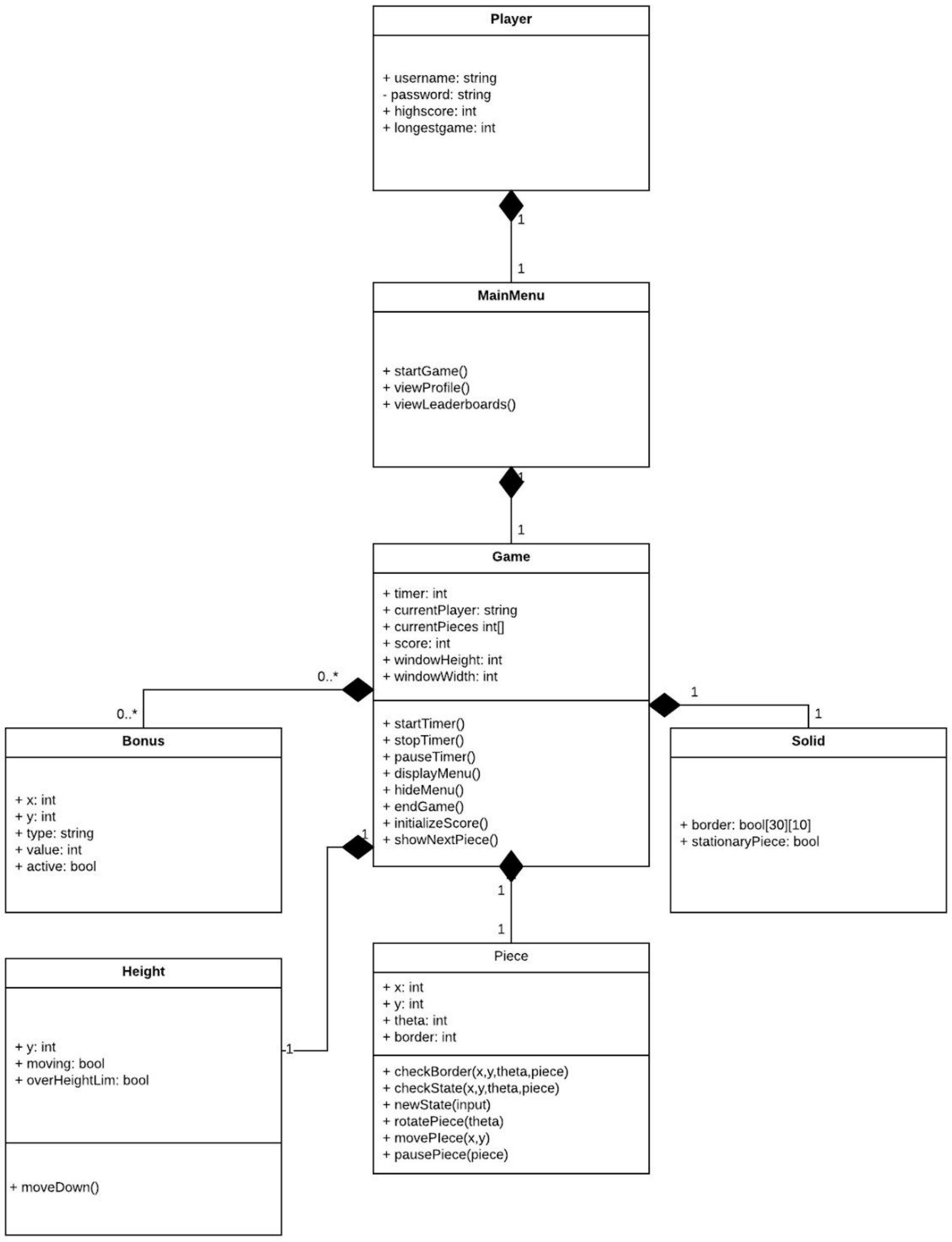
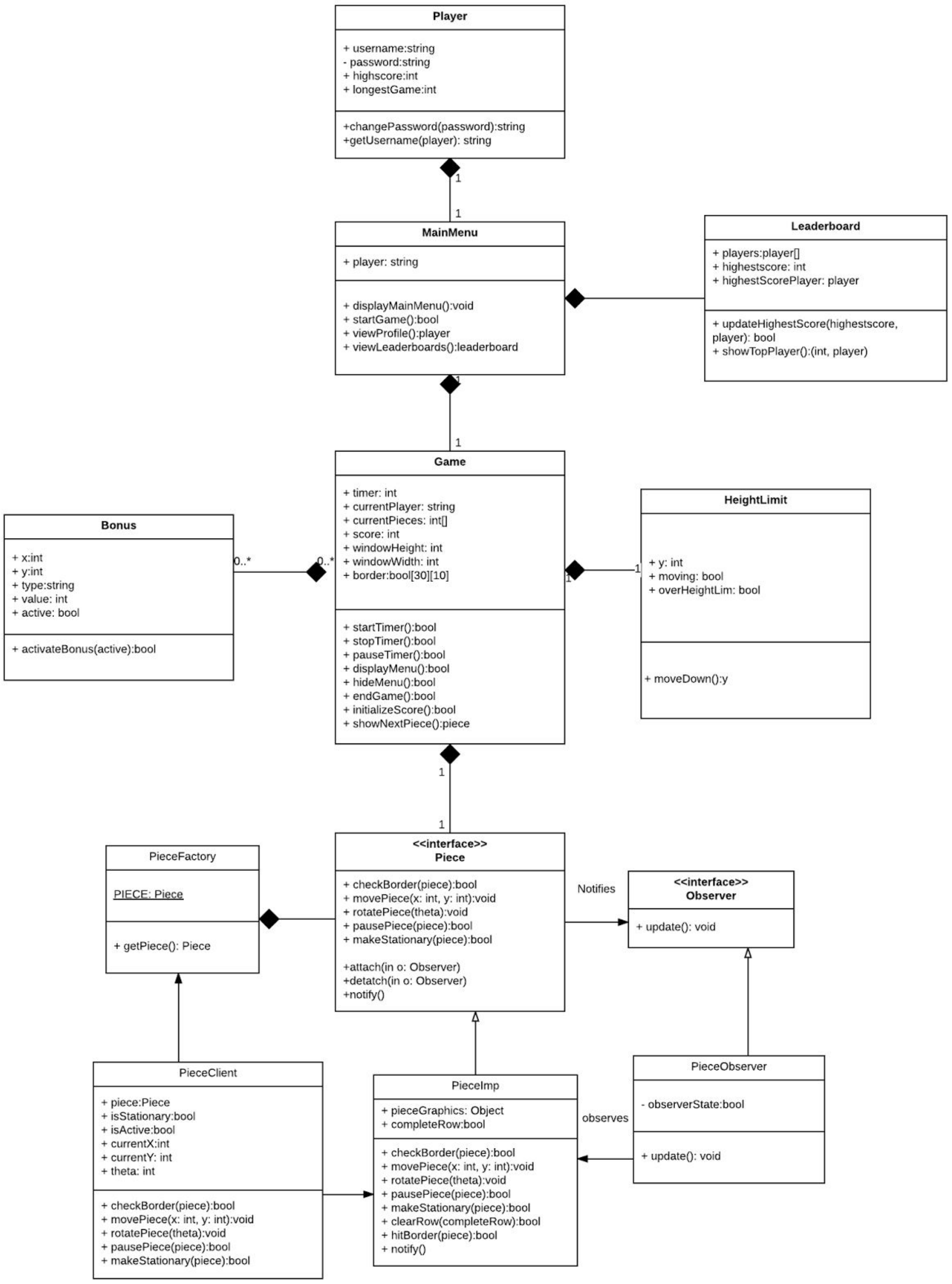
Tetris
Travis Roundy, Jake Traut, Justin Olson

Class Diagram (Part 2):



Updated Class Diagram:



Design Patterns and Refactoring:

We chose to implement the Flyweight and Observer design patterns for our Tetris project. This will fit our need of creating a large amount of similar objects (the tetris pieces), while being very efficient.The bottom half of our class diagram shows our application of these patterns. The client requests a new piece from the flyweight factory, in our case PieceFactory. This will create an instance of Piece that can be used by the client. After the piece is instantiated, it then can be controlled by the client and the PieceImp class. This class will be used for the overall motion of the piece (rotating and moving), color of the piece, and status of the piece. The piece interface will manage each piece within the game, current or stationary. To determine between an active and stationary piece, we use the PieceObserver class. The PieceObserver takes care of the current piece, making that piece the “observer” to the input keys for moving and rotating, then notifies the piece implementation class such that it can update the object's state accordingly.

The other major refactor we did was changing the way the state of the solid block object in tetris is represented. Rather than defining it as a single entity, it is composed of the collection of pieces. We renamed the class of Height to HeightLimit as it better depicts what the class represents. We also added a leaderboard class which will contain a collection of players and their respective highscores. This will be a stretch goal, however, as we will not have time to implement this. We were able to remove poltergeist classes that had only functions or only attributes, this allows us to condense our class diagram and remove this anti-pattern.