Rationale

In my design choices for my object model, I chose to focus on the lowest coupling, lowest representational gap but highest cohesion. As obvious as that sounds like a software system should have, I think I make some clear decisions as to why I composed some classes with instances of other objects or why I completely separately others. For example, Game will have an instance of Shop, Board, BagofTiles and many instances of Players. This will keep game as the way to tie together everything. There are other design choices I made within each class including the following:

The SpecialTiles class is a very independent class from LetterTile because even though at first the both seem similar, I realized that they have nothing in common in terms of functionality. I had SpecialTiles be an abstract interface that the rest of the types of SpecialTiles can implement. These tiles have nothing to do with the game, but rather the Shop in which they can be bought from, until the Player takes them out of their SpecialBag, which contains all the SpecialTiles already bought. I thought to use the Strategy pattern for this part because it will solve issues with having to experience the upheaval associated with change: no impact when the implementation of a SpecialTiles changes. This is key since we will have to implement another unknown SpecialTile that we don't know how the implementation works. The Strategy Pattern works because we want to encapsulate interface details in a SpecialTile, and bury implementation details in its derived classes.

The board is a 2D list of Squares. Each square has a letterMultiplier and a wordMultiplier, which when the game recognizes a valid move is made with the letterTile in this square, then the letterMultiplier is used to multiply the letter's point value with the letterMultiplier. Subsequently, we take the sum of all of the letterTiles similarly and check each letterTile's wordMultiplier to multiply with the sum we accumulated after iterating through the letters. This design choice was made to simplify the classes and not require an extra Bonus tile class. That would extend the functionality in an unnecessary way when all we need to know about that bonus tile is the letter or word multiplier.

Additionally, I chose to have a Shop class. This is where players can buy Special tiles. I'm going to allow 5 of each type of the tile to be placed. This is so that Special tiles can be balanced still and not allow only the players with a lot of points have a lot of advantage. Instead of the game itself containing the bag of special tiles players can buy, this division of Game and Shop was necessary because the shop is a separate entity on its own that players will have access to, but not the game.

A gameplay choice I made was to allow challenges to be made immediately after a word is placed on the board, but before any special tiles are revealed or activated. This way we don't need to waste any special tiles on players who lost a challenge, and thus will remove their tiles and lose their turn.

The Player only directly interacts with the Shop and the Game, which makes sense because everything else that is relevant to the Player is composed within these two classes. Even things like starting a game is initialized by the Game class, which will request a number of players and player names. Placing tiles is done by the Game class to the board, once a list of LetterTiles is established to place.