Rationale

I have a game class as the controller. A game would have a board, a dictionary, a tile bag, a special bag, and several players. Players can interact with game and indirectly get access to these objects. And a board would have 225 squares of each has a word tile, a special tile, and bonus information in it, as well as its coordinate. A player would have a word rack which holds word tiles, and a special tile which holds special tiles. Special tiles consist of 5 types. By this means, object oriented design is achieved.

A game class has a newGame() method that can start a new game of certain number of players, a getCurrentPlayer() to get the player whose round is in turn currently. And nextPlayer to set next player's turn to true, reset each button state to false, set newWords and currentSquares and wordSquares to empty list, set currentScore and countPass to 0. Game can also let player purchase a special tile, using a special name. It would call special bag's method to create that certain special tile and check if the player has enough money to purchase it. Game can let the user use a special tile to a certain coordinate, and check to set that square the special tile if valid (no word tiles in the square and coordinate in board). Game can let a player place word before he press play button (he doesn't pass the turn or swap the tiles and pass) and swap certain number of words, or complement player's tile to 7 if the player finishes the turn and does not press pass or swap button. Game also has isValid() to check check whether tiles are placed in valid locations and whether they form actual words. Game would let board check if the if squares are in board, sort the squares, check if the word is in dictionary. And move() method might or might not trigger the evaluation of special tiles, compute points, adjust the score, and trigger the next round in the game. It would first calculate a plain score, and then modify it by bonus squares, using methods in board to get, for example, the new word list and new word squares list. Then it would check in order of boom, negative, bonus, reverse types of special tiles to modify the current score. Then it would add scores to the player, end this turn, and complement the tiles. It finally check if the game is over, if not, change to next player. Game's retreat() method would retreat the tiles in the square within current turn, and set certain attributes, for example, the button state, to original state, and let the player do again. Game has a field of countPass, which counts the continuous pass or swap action by players. If it reaches 4, the game is over. Or if the player's word rack is empty after complementing tile, the game is over. Then game would compare the winner.

Board has field of squares which are all squares of the board. It also has currentSquares to store the squares the player fills within this turn, and newWords for the new words generated this turn, and wordSquares (List<List<Square>>) for the new words' squares for this turn, and currentScore for this player's score within this turn. Board also has some special methods. getScoreByLetter() return a score number given a certain letter string. getLetterBonus() is easy that just calculate the square's tile, getWordBonus() would ask and find from wordSquares to find where are the squares that would be influenced by the word bonus square under the bonus letter square's influence, and calculate the bonus part to add to the currentScore. getIntoSquares() would get the player's input, list of tiles and list of coordinates into game board's

squares. Putting special tiles are treated differently in the useSpecialTile method. checkIsLine would check the coordinates that if they are ascending one by one. checkAdjacentWord should be executed after checking if the line has already formed a word. It returns the word to the opponent direction of X or Y, or null if there is no word formed. So newWords.add(aWord) should check if aWord is null. Inside the checkAdjacentWord method, wordSquares would be builded by adding each generated word square. They don't need to be real words because they would be reset if they are not valid, they would be retreated, as well as the current player's attributes in the board class.

Square class has a bonusFlag in it, which is a string like "DL" and "TW". By getting this flag information, we can decide which get bonus method to use inside the board class (square as a parameter). There are no special methods in square class.

Player would have a boolean value of whether in turn, a name, a score, and two racks. The racks can modify the tiles in them, differently. Tile bag should have a shuffle method. SpecialTile is an abstract class. All special tiles share same things except the effect method. The parameter should be the game and coordinate it affects. The special tiles are triggered inside move() method in the game.

There is no special method in other classes.

I answered all the questions including the two formal scenarios and the informal questions that you provided. This is a valid design program because it covers all aspects of the game with maximum code reuse. I achieved the goal of reuse by providing abstract classes for SpecialTiles. There is good division of labor, since I almost separate everything into different classes. Since the design is pretty clear, it achieved the goal of understanding and maintenance. With this design, I have made sure to have high cohesion, by providing a different class for each new step in the game. This also reduces the dependencies and lowers coupling.

Updated Design in Milestone B

I added two classes from milestone A.

1. ScoreWord: There are a pair consists of a square and a score, the score is responsible for the change of scored due to letter bonus, negative tiles, and etc., I do not want the score inside square to store that changed score.
2. CurrentMove: Assigned most of the check, sort and modification work to the CurrentMove class. there are three instance variables, currentSquares is a list of squares, which are the squares the player placed at that round; scoreWordSquares is a List<List<ScoreWord>>, it stores all possible valid (or not) words, a List<ScoreWord> consists a word; newWords is a List<String> corresponding to the scoreWordSquares, newWords is responsible for checking valid from dictionary.

Then I decreased some of the methods from Board class. Because board should not be responsible for the current move action. So Game would have an instance of CurrentMove class, and CurrentMove would be responsible for all current move actions, including check isValid()'s helper methods, letter and word bonus calculation, and different special tiles' effect to the current squares.

I don't need to tell each kind of bonus square, just first execute letterBonus() then wordBonus(). Also, I don't need to tell each kind of special tiles, just execute the effect() method. Inside the effect(), it would do a lot of things that I showed on the diagram.

I deleted bonusFlag and changed to two integers, letterBonus and word Bonus. They are default as 1. They are multipliers that applies to all squares. If we call getLetterBonus() and getWordBonus() from CurrentMove class, they would automatically calculate the bonus points.

I added a method called checkFirstWordCol(game) inside CurrentMove class, to check if the first word is valid.

As for the diagrams, I did several changes. For Domain Model, I changed each arrow to the line segment without arrow. I added cost to the special tile. I added eggiest to each special tile. I added a CurrentMove as I also did in object model, which handles the moves from the current player's round. I updated my special tile, called RevengeTile now, because this is more creative than the old one. When the current player triggers the revenge tile, it will remove all the special tiles of the player that are already placed on the board.(not influence the holding special tiles). For SSD: I deleted the interactions that the game would do to itself. (Eg: countPass, nextPlayer, compareWinner). I changed the methods that need to be captured and sent to the game. (Eg: if any button is pressed). I changed the loop specification as English, not implementation code.

Updated Design in Milestone C

I added the limit of special squares a player can purchase, which is 10. Because it makes the gui easier to manage.

I added the score switch tile, which will be explained in the discussion.

I deleted the button attributes from game.java, because gui can handle that.

I added a method “isAdjacentToOldWords(Game game)” to the CurrentMove class, also added it to the object model. It is to check if the new tiles are adjacent to old words. For example, you cannot place a word far away which is not adjacent to the old words. Because I forgot to check that inside the isValid().