Rationale

In my design, there is a Game class, and Player class, LetterTile class, SpecialTile class, Dictionary class etc. Player can interact with the game by manipulating these objects. At the start of the game, Game class has a method, initialGame, which can start a new game. addPlayer method can add new player. The actions a player can perform are pass, exchange letter tiles and play tiles.

If the player chooses play tiles, he move tiles in the GUI. After playing one tile, then the Move class in player class will add a new placement. Move is a class which contains two main attributes, a list of LetterTile placement and a list of SpecialTile placement. Placement class contains location and a letter tile or a special tile. LetterTile is a class which show the letter and score of this tile. Location class treat row and column as a Location, and I also override the equals and hashCode method in this Location class. After the player placed multiple tiles, the ArrayList<Move> should have several members.

In the isValid method, I check if these letter tile placements are collinear. I check if this position is on board or not and if these potion has been occupied. Then I check if the tiles in this turn are adjacent to existing tiles. Then check if these letter tiles are consecutive. After checking validation of this play, in the Board class, there is a method called getWord(), which will loop over each Move of this turn, and get horizontal and vertical words of this Move. In this getWord() method, starts from this Move, check left, right, up and down until there is empty squares. This achieved **low coupling and high cohesion.**

After checking validation, move() method will take care of all the effects of special tiles before calculating scores. If there is bomb tile, then remove all related tiles from board. If there is reverse tile, then reverse the order of player. If there is negative tile, set ratio in Word class as -1.

After processing all special tiles, then calculate score. If there is empty tile, then this player's score in this turn is 0.

The square is in charge of calculating score of this square alone. If a tile is placed on

double letter square or triple letter square, the score of this square should be the score of the tile on this square times 2 or 3. When looping over all squares of each word, if there is a double word or triple word square, I will set ratio of this word as 2 or 3. The attribute ratio is belongs to each word (Word is a class). After calculating basic score of this word, then I will use basic score times this ratio.

A player can acquires special tile from Game, call buySpecialTile(String) method, if the player has enough score to buy this tile, then add this tile to player's specialTileList. If the player chooses play in this turn, he can place special tile on board. Then directly modify corresponding square if this square is empty. If a word triggers multiple special tiles, all of these special can be triggered in this turn. The order of these special tile should follow this rule: boom tile blow up related tiles, then all rest tiles calculate total score, if there is negative tile, only negative the remain score, if there is zero tile, the player will get 0 score in this turn no matter if there is negative tile or not. Other special tiles will not influence reverse tile. After calculating the score, switch tiles will make effect. In order to achieve **information hiding and code reuse.** I use an abstract class called SpecialTile, then other special tiles inherit this abstract class. In this abstract class, there are two abstract method, makeEffectBefore(Game, Square) and makeEffectAfter(Game, Square) because different special tiles has different implementation of these two methods. This abstract class also has concrete method like getName, getPrice etc. This abstract make my code **extensible and flexible, it's design for change**.

Game if in charge of managing turns. After player exchanging or passing, game will trigger next turn. After player played valid tiles, also trigger next turn. Game is also responsible for knowing who the current player is.

For GUI part, I use **observer pattern.** GamePanel class implements GameChangeListener class.