UML PRESENTATION

Alessandro Barbieri, Mariarosaria Cotrone, Silvia Denti, Marco De Vellis

In our UML, we used the design pattern of the **Model View Controller**, which divides the code into three parts: Model contains the data access methods, Controller observes the changes taking place in the View, communicates them to the Model, which in turn changes accordingly, View which displays the information to the client.
Eventually the Controller notifies the View of the change in the Model.

More specifically, in the Model we have implemented the classes that make up the game (GameBoard, PersonalGrid, Tile, Player and personal and common goal cards). Each of these classes has get and set methods.
We have decided for now to implement both the **PersonalGoal** cards and the **CommonGoal** cards with 12 subclasses. In the personal goal cards we have used a HashMap that contains the coordinates and the color that there must be at that specific position in the grid. In the common goal cards there is a method which is initialize which implements the logic to check whether that particular goal has been completed or not. Both types of cards have their own deck and the required number of cards is drawn randomly from each deck each time.
PointAssigner is the class responsible of keeping track of the points assigned at the completion of a common goal. It is essentially a stackList that is initialized according to the number of players (2 players 8-4, 3 players 8-6-4, 4 players 8-6-4-2). The list is in the same order of the list of common goals in Game and GameOrchestrator so for example stack in position one keeps track of the points of common goal in position one. The method assignPoints returns the correct amount of points the player has earned by completing the common goal and empties the stack each time. The gameBoard and the playerGrid are arrays of spots, initially empty spaces that can be filled by a tile.
Tiles are distinguished according to their color using an enumeration, they are contained in a bag (**TileBag**) that each time the board must be filled draws the number of tiles needed to fill it.
GameBoard, the board, is initialized depending on the number of players there are using the createFinalMatrix method. The boardCheck method tells me if the board needs to be filled. The boardCheckNum method returns the number of pieces needed to fill the board and finally the fill method fills the board with the set of tiles passed to it. The verifyPickable method sees if the tiles are adjacent and if they have a free side. The pick method eventually picks the pieces from the board.

As for the **PlayerGrid**, the personal grid of each player, the topUp method inserts the tile into the grid, the spaceCheck method checks if the selected column has enough space to put the taken tiles and the fullCheck method tells if the playerGrid is full or not.
The **Player** class in particular has three important methods, the verifyPersonalPoints and verifyExtraPoints which respectively calculate the points related to the personal goal and the points due to adjacent tiles of the same color, the other method is modifyCompleteCommonGoal which changes the completeGoal array if I have completed the common goal, so the pointAssigner will not award me any more points if the goal is by chance completed twice.

There are three main classes in the Controller: Game, GameOrchestrator and the GameState (which implements a state machine). **Game** is the class that acts as the lobby (here are the players who connect, we sort them and pass them to GameOrchestrator); also initializes all game elements and is the class that interacts with the View.
There are two types of interaction that the user can do with the game: the choice of the tiles to take from the GameBoard and the selection of the coordinates in which to place the tiles in the PersonalGrid. We have therefore implemented these two actions as two classes that extend the PropertyChangeListener interface. These two classes extend the propertyChange method which thanks to a series of switches and ifs and thanks to the Game that is aware of both the player who is playing, it excludes all events coming from other players and also knows the state we are in so it knows what kind of event to expect and therefore accept, discarding all others. In a nutshell, these series of ifs and switches understands if the event that arrives is what it expected and if so it propagates the changes in the ModelGame, through **GameOrchestrator** will propagate all changes also in the Model, GameOrchestrator is the class that manages the turns.

We have reduced the game scheme to a finite-state automaton in order to simplify both debugging and game management.
The states are:

1. **StartTurnState** - state that looks at whether we are in the last turn or not. In the event of a positive response, it causes the EndGameState to be called in the changeState. The connectionState is called instead.

2. **ConnectionState** - state that sees if it makes sense to execute the player's turn, i.e. checks whether the player is connected or not. In case of a positive response, the RefillState is executed.

3. **RefillState** - state that takes the tiles from the bag and places them on the board.

   At this point you are waiting for an event, i.e. you are waiting for the

player to choose the tiles to take from the board, after which you reach the VerifyGrillableState.

4. **VerifyGrillableState** - state that checks if there is a column in the player's grid that can accommodate the exact number of tiles taken by the player. If yes, it moves to the next state otherwise it waits for another input.

5. **VerifyBordableState** - state that checks that the tiles a player wants to take all have a free side, if this method is successful it automatically goes to the next state, otherwise it returns to the VerifyGrillableState, always waiting for another input.

6. **PickState** - state that takes care of putting the tiles inside the player, then I wait for input from the player who will have to tell me which column to put the tiles in, and finally, once the input is received, it goes to the next state.

7. **TopUpState** - state that takes care of placing the tiles in the PlayerGrid by checking if the chosen column has the actual vacancies, this state is called until you finish placing all the tiles, once all the tiles have been placed it automatically switches to the next state.

8. **VerifyCommonGoalState** - state that verifies whether the player has completed one of the common goals.

9. **FullGridState** - state that checks whether the player's grid has been completed, if so the boolean that says we are in the last round is modified. This boolean will be checked by the StartTurnState. In this state the transition to the next player also takes place.

10. **EndGameState** - state that is only called when the game is over and is responsible for adding up the points.

In the Controller we also manage the disconnection of the Client: through a keep alive system, managed in another thread, if five consecutive messages do not arrive from the Client, it is considered disconnected. When the Client is considered disconnected the keep alive thread takes over, changes the player's index to GameOrchestrator and then calls StartTurnState. When the player returns to the game, if he already had some pieces in his hand, he goes directly to the TopUpState.

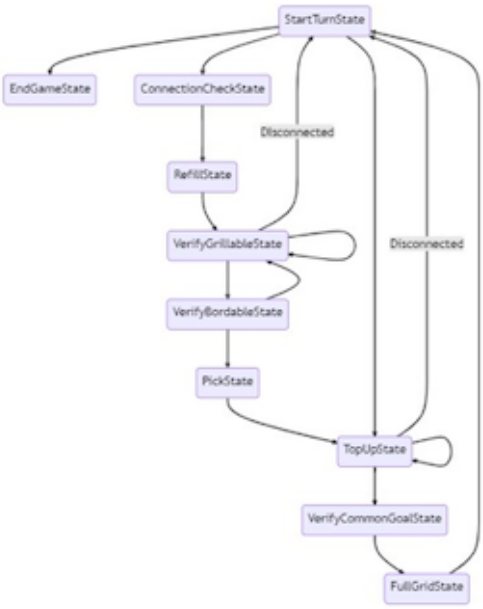
StartTurnState
EndGameState
ConnectionCheckState
Disconnected
RefillState
VerifyGrillableState
Disconnected
VerifyBordableState
PickState
TopUpState
VerifyCommonGoalState
FullGridState

Figure 1: FSM