Updated Rationale

AndrewID: chenxili

In this Carcassonne game software, there are mainly eight classes that constructs the whole software system core. They are “GameSystem”, “GameBoard”, “Player”, “Tile”, “Position”, “Segment”, “Feature”. For the GUI part, the game applied Swing to realize the UI. A TileButton class that extends JButton would used to represent tiles on the GameBoard. A GameBoardPanel class that extends JPanel would used to represent the whole GameBoard and mainly accept the information from the backend. A GUI class that extends JFrame that holds the whole game frame and GameBoardPanel.

“GameSystem” API is designed to maintain the general gaming process of the whole game. It will be serve as the “Controller” of the whole program and user should only implement with “GameSystem” API without knowing information of other classes. There would be a boolean flag to indicate if the system run out of tiles. If this flag turns to be true, the game would automatically ended and calculate the final score based on special rules. In one turn in the game, first the “GameSystem” would update the tile and playerID, and then call the GameBoard to validate and take the users’ input of proposed tile position and meeple position. After that, the GameSystem would check if any feature is complete and finally update the scores. As an API, it should not directly expose any other classes through the accessor so that the user do not have to learn any other classes except the “GameSystem” API.

“GameBoard” class is designed to represent the relationship among tiles, segments, positions, and features. When player is attempting to place a tile or a Meeple, the “GameBoard” could check if the placement is valid or not by the existing relationship. Also, GameBoard stores the all the features in one single list and use polymorphism to differentiate. Anytime when a tile is added, first the feature list would be updated as there would be new segments to be added in to existing feature and new feature to be added into existing list. Then, the GameBoard would check if there are some features are just complete with this update. The basic logic of checking complete of RoadFeature and CityFeature is to check if all the segments in this feature have already connected with some other segments and not “opening”. The way to realize this is to maintain a list called “realSegmentList” to store the original “un-merged” segments on the Tile. The “merged” segment is used to calculate score, while the “un-merged” segments is used to check the completeness of the feature. We can simply go through this list of “un-merged” segments. As each Segment object has Position and orientation information, it can easily find its adjacent Position. Check if this position has already has a Tile on it. If not, then this segment is still “opening”, which means the feature is not complete. If all the Segment in the “realSegmentList” is not “opening”, then we can know this feature is complete. This logic is only applied to “RoadFeature” and “CityFeature”. For the MonasteryFeature, the logic of check complete is much simpler.

Just check if the number of its surrounding Tiles is eight. If so, the MonasteryFeature is Complete. After checking complete and update the score, the meeples are also needed to be turned back. The GameBoard maintains a Queue called “returnedMeeplePos”. Every turn when the GameSystem call “getReturnedMeeplePos()” to acquire this Queue, the process of deep copy the Queue will automatically empty the “returnedMeeplePos” Queue in GameBoard so that the GUI would not need to reset the image of one Position for multiple times. After GameSystem get this Queue, it would be sent to the GUI and the GUI would reset the Image of the TileButton on these position to show that the meeple is turned back vividly.

“Player” class is an object class that represents the players. It has some general attributes like “score”, “ID”, and “leftMeeple”. It is important to know that it is user to implement the game, rather than “Player” object. This class is just used to represent a player so it should have actual implementing methods like “getTile”, and “placeTile”. The action of “placeTile” and “placeMeeple” would be directly implemented through GameSystem. “Score” is used to store the current score of a user and “ID” is used to represent the order. “leftMeeple” represents the number of left meeples (originally a Player would have 7 meeples). If “leftMeeple” turns to be 0, this user would not be able to place a meeple on the segment. And everytime when a feature is complete, the meeple would be turned back by adding the “leftMeeple” by one.

“Tile” class is an object class that represents the tiles in the Carcassonne game. A tile could have several different segments on its four edges and its center. So the “Tile” class has feature like “segmentList”. The index of the “segmentList” would represents the different edges. As a tile can be rotated in different orientation before being placed in the gameboard, the “orientation” of each Segment on this tile needs to be changed when this Tile was rotated 90 degrees. After a user placed this tile, ii would has a position. For this game, 56 tiles was generated automatically by parsing JSON configuration files at the beginning of the game. This list of tiles should always start with “D” type tile and the rest of them are randomly assigned.

“Position” is specifically used for record a tile’s position on the game board. As the game board is 2 dimensional, the “Position” has “x” and “y” to locate a specific point. Notice, as the GUI panel uses fourth quadrant rule, the Position should also follows fourth quadrant rule.

Although technically there are three variants in “Segment” that construct “RoadFeature”, “CityFeature” and “MonasteryFeature” separately. But there is no need to have three class to represent each of them but use an attribute called “Type” to separate them. Except the “Type” attribute, it also has “orientation” to represent which edge of the Tile that this segment sits on. After a user placed this tile, it would has a position. The reason why the segment would need a Position attribute is for checking feature complete usage. Also, for the tile that has “City segment, it may have at most

one “Shield” on it so the class has boolean attribute “hasShield”. After a Segment is added to a feature, its “featureID” would be set as the “featureID” of that feature. In this way, a segment can easily find which feature it belongs to through GameBoard.

As there are several variants in “Feature”, the software system chose to use “Strategy Pattern” design pattern to realize better code reuse and information hiding. There are three kinds of Feature in this game - “RoadFeature”, “CityFeature” and “MonasteryFeature”. By “Strategy Pattern” design pattern, an interface called “Feature” declares several methods that a general feature would have. Then a concrete class called “DelegationFeature” implements this interface and serves as the delegation for the three variants. A “Feature” is constructed by several corresponding “Segment”. Different check complete rules are implemented in the GameBoard class.

For the GUI design, A TileButton class that extends JButton would used to represent tiles on the GameBoard. A GameBoardPanel class that extends JPanel would used to represent the whole GameBoard and mainly accept the information from the backend. A GUI class that extends JFrame that holds the whole game frame and GameBoardPanel. The Observer Pattern was applied to realize the GUI so that GUI only accept the information from the GameSystem API without influencing any behavior of the Game Core.