Design rationale

Design goals: To design a game system which can interact with the external user, and update and show real-time information (e.g. all the tiles, features, followers on board, scores of different players).

Design principle: There are 11 main classes: Game, Player, Follower, Board, Segment, Position, Tile, Features (City Feature, Road Feature, Cloister Feature, Farm Feature). During one game, there is only **a Game object and a Board object**. Two to five Player objects share the same Board. **Segment object** is used to save the edge information of tiles, and check if two abut tiles have the same segment. **Position object** is used to record the x, y position of the tile. **A Tile object** is a single tile with fields including position, segment list (to record edge information of the tile), inside feature list (Feature information of the tile, discussed later in this document), and number of banner. **Feature object** is an aggregation of tile objects, it has a list of covering tiles, which indicates all the tiles the feature covered; it also has a set of neighboring positions, which indicates the all positions needed to complete this Feature. Once the neighbor position set has no element, this Feature is complete. Different types of Feature (road, city, cloister, farm) has different adding covering/neighboring tile, removing neighboring tile, and scoring strategies. **Feature object** also has a list of Follower objects. Follower object has one field *master* to indicate which Player it belongs to. Appendix A describes the relationship between tile, segment, and feature in detail.

System processes:

1. Initialization and add player
   When calling startGame(), the game begins, and the user can call addPlayer(Player) to expand *playerList*. The initial *tileStack* is a stack of random order of 72 tiles. Position of each tile in *tileStack* is (0, 0). For Player object, the initial *score* is 0, the initial *numOfUnusedFollower* is 7. The initial *tileList* and *featureList* of the Board has no elements.

2. A player draws and places the first tile(follower)
   The user calls drawTile() to pop a tile from the *tileStack*. Since *tileList* on Board is empty, any tile is valid. Then the system returns the tile to the user.
   The user calls placeTile(Player, Tile) to place the tile for the player. Then findAbutTiles(Tile) is called to find abut tiles of this placement. When *tileList* is empty, findAbutTiles(Tile) returns true. Then The user can choose whether to place a follower on the feature in this tile by calling placeFollower(Player, Tile, Feature). Since it is the first tile, any placement of follower is valid. If the user chooses not to place a follower, the input Feature is null and the method will return directly.

3. Place another tile(follower)
   The user calls drawTile() to pop a tile from *tileStack()*. If the tile cannot be placed in any position and rotation (discussed later), it will be discarded and call drawTile() again, until the tile can be placed and return the tile to the user.
   The user calls placeTile(Player, Tile) to place the tile for the player. Then findAbutTiles(Tiles) is called to find abut tiles of this placement (search for up, down, left, right positions). abutTiles is in

the form of [upTile, leftTile, downTile, rightTile]. If all elements in *abutTiles* are null, the method will return false and system will ask user to call placeTile(Player, Tile) again. The user can rotate the Tile by changing the order of segment list.
If *abutTiles* is not all null, validateAbutTiles(Tile, abutTiles) will be called, and check the adjacent segments of the tile and abutTiles. If any adjacent segment is not the same, the method will return false.
If all adjacent segments are the same, findAbutCloister(Tile) will be called and search for all CloisterFeature in *featureList*. If the position of the tile is in *neighborPos* of a CloisterFeature, the position will be removed from *neighborPos*, and the tile will be added into *coveringTiles* of the CloisterFeature. Then addToFeatures will be called, and merge *insideFeatures* of the tile into *featureList* of Board. (See Appendix B for detailed description). Then addToTiles(Tile) will be called, adding the tile into *tileList*.
If the user decides to player a follower on the tile, placeFollower(Player, Tile, Feature) will be called. If the tile is not in the coveringTiles of the Feature, or the *master* of *followers* of the Feature is not the Player, the placement is illegal. Otherwise the Follower will be added into *followers* of the Feature. After placing a follower, each Feature in *featureList* will call isComplete() to check if *neighborPos* is empty. If empty, find the dominant Player among *followers*, and count the number of *coveringTiles* and *banner*, then call addScore(number or number * 2) to add number to the Player.
After each follower placement, isGameOver() will be called to check if tileStack is empty. If empty, isGameOver() will be called.

4. Game over and get final scores
   When game is over, getFinalScores() will be called, and it will call CalculateFinalScores(). It will loop through all the incomplete Features. In each incomplete Feature, find the dominant Player among *followers*, count the number of *coveringTiles* and *banner*, then call addScore(number) to add number to the Player.
   For each FarmFeature, find dominant Player among *followers*, loop through all the complete CityFeatures, and check if the FarmFeature and CityFeature have at least a common tile in *coveringTile*. If yes, then call addScore(3) for the Player.

Appendix A

There are three Segment objects in total. (cloister doesn't appear on the edge, so it is not a segment)
CitySegment, name of which is "city";
RoadSegment, name of which is "road";
FarmSegment, name of which is "farm".
The order of segment list of a tile is up→left→down→right.

Suppose the position of the tile in Figure A.1 is (x1, y1).
Its segment list is [CitySegment, FarmSegment, CitySegment, FarmSegment]
Its inside feature list is [FarmFeature1, CityFeature, FarmFeature2]
For all the features in this list, they only cover this tile.
For FarmFeature1, its neighboring position set is [left], which is [(x1 – 1, y1)].
For CityFeature, its neighboring position set is [up, down], which is [(x1, y1 + 1), (x1, y1 – 1)].
For FarmFeature2, its neighboring position set is [right], which is [(x1 + 1, y1)].

Suppose the position of the tile in Figure A.2 is (x2, y2).
Its segment list is [CitySegment, FarmSegment, RoadSegment, RoadSegment]
Its inside feature list is [FarmFeature1, FarmFeature2, CityFeature, RoadFeature]
For all the features in this list, they only cover this tile.
For FarmFeature1, its neighboring position set is [left, down, right], which is [[(x2 – 1, y2), (x2, y2 – 1), (x2 + 1, y2)]
For FarmFeature2, its neighboring position set is [down, right], which is [(x2, y2 – 1), (x2 + 1, y2)]
For CityFeature, its neighboring position set is [up], which is [(x2, y2 + 1)]
For RoadFeature, its neighboring position set is [down, right], which is [(x2, y2 – 1), (x2 + 1, y2)]

Suppose the position of the tile in Figure A.3 is (x3, y3).
Its segment list is [FarmSegment, FarmSegment, FarmSegment, FarmSegment]
Its inside feature list is [FarmFeature, CloisterFeature]
For all the features in this list, they only cover this tile.
For FarmFeature, its neighboring position set is [left, right, up, down], which is [(x3 – 1, y3), (x3 + 1, y3), (x3, y3 + 1), (x3, y3 – 1)]
For CloisterFeature, its neighboring position set is [upleft, up, upright, left, right, downleft, down, downright], which is [(x3 – 1, y3 + 1), (x3, y3 + 1), (x3 + 1, y3 + 1), (x3 – 1, y3), (x3 + 1, y3), (x3 – 1, y3 – 1), (x3, y3 – 1), (x3 + 1, y3 – 1)].


Figure A.1


Figure A.2


Figure A.3

Appendix B

In Figure B, suppose the right tile is newly-placed, and position of the left tile is (0, 0).
Before place the new tile, *featureList* of the board is [FarmFeature1, FarmFeature2, CityFeature1]
For the new tile, *insideFeatures* is [CityFeature2, FarmFeature3, FarmFeature4, RoadFeature]
When merging *insideFeatures* into *featureList*, do the following loop:

```
for newFeature in insideFeature:
    matchedFeature = new ArrayList
    for oldFeature in featureList:
        if newFature.type.equals(oldFeature.type):
            if (any position in neighborPos of oldFeature is (1, 0)) && (any position of coveringTiles of
            oldFeature is the one of neighborPos of newFeature):
                matchedFeature.append(newFeature)
    if matchedFeature.size() == 0: //when there is no matched feature
        add newFeature into featureList
    if matchedFeature.size() == 1: // when there is one matched feature
        add neighborPos of newFeature into neighborPos of oldFeature, if the position is in coveringTiles
        of oldFeature, skip it
        remove (1, 0) from neighborPos of oldFeature;
        add the new tile into coveringTiles of oldFeature;
    if matchedFeature.size() > 1: // when there are multiple matched features, merge all features as one
        set the first element in matchedFeatures as the base feature, and loop through the other features,
        adding neighborPos of otherFeatures to baseFeature, adding coveringTiles of otherFeatures to
        baseFeature, and remove position of coveringTiles of otherFeatures from baseFeature;
        after the loop, merge newFeature to baseFeature
```

For the case of Figure B,
For CityFeature2, its *neighborPos* = (0, 0), matches the position of *coveringTiles* of the CityFeature1, but it is in *coveringTiles* of the CityFeature1, so it will not be added to *neighborPos* of CityFeature1. Then remove (1, 0) from *neighborPos* of CityFeature1, and add the new tile into *coveringTiles* of CityFeature1.
For FarmFeature3, its neighborPos = [(0, 1), (0, -1), (1, 0)], doesn't match any position *coveringTiles* of FarmFeature1 and FarmFeature2, so add FarmFeature3 into *featureList*.
Since there's no Road type Feature in *featureList*, add RoadFeature into *featureList*.

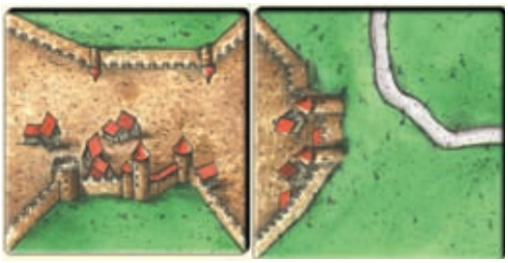

Figure B