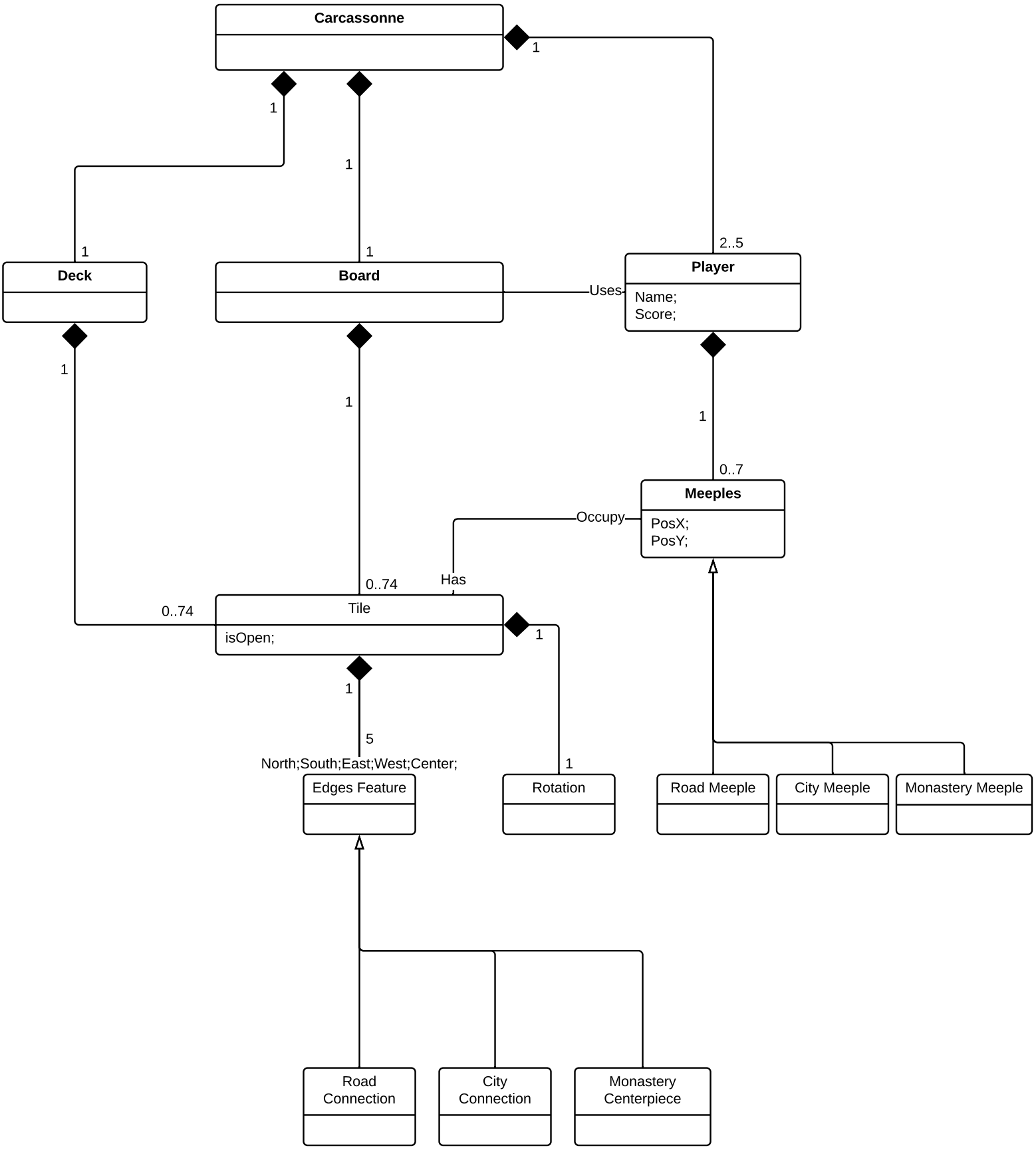
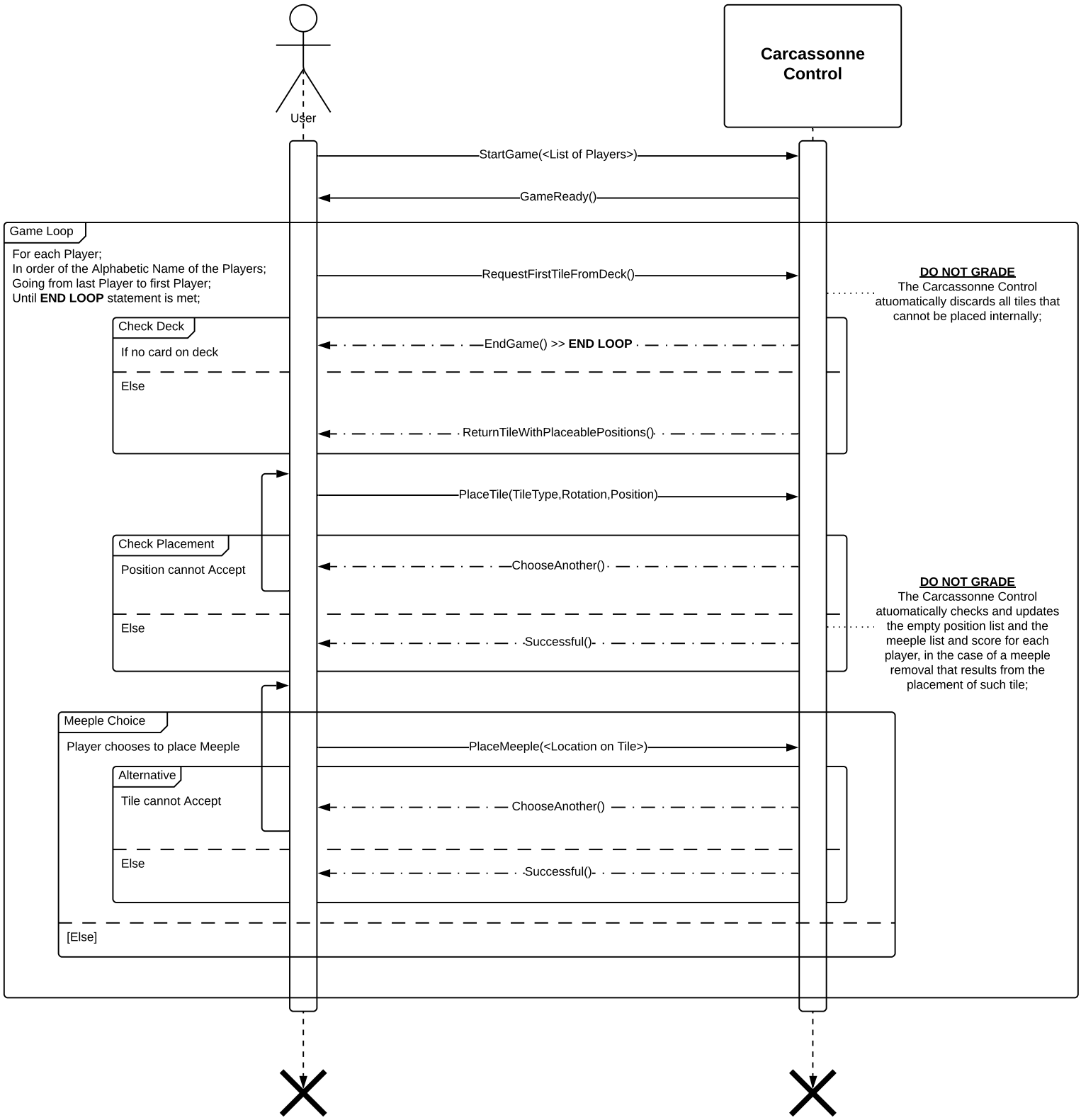
# Carcassonne Domain Model

Roger Zheng | October 4, 2019



# Carcassonne System Sequence Diagram

Roger Zheng | October 4, 2019



Behavior Contracts:

*User Attempts to Place a Tile;*

Note:
In this design, the behavior does not require the player to select a valid Position and Rotation to place the tile; the placement will be processed as long as the player is holding a placeable tile; in the instance when the player seeks to place the tile with invalid Position or Rotation, the Game Control will return false and prompt the player to reconsider; This is by design and not a mistake;

Precondition:

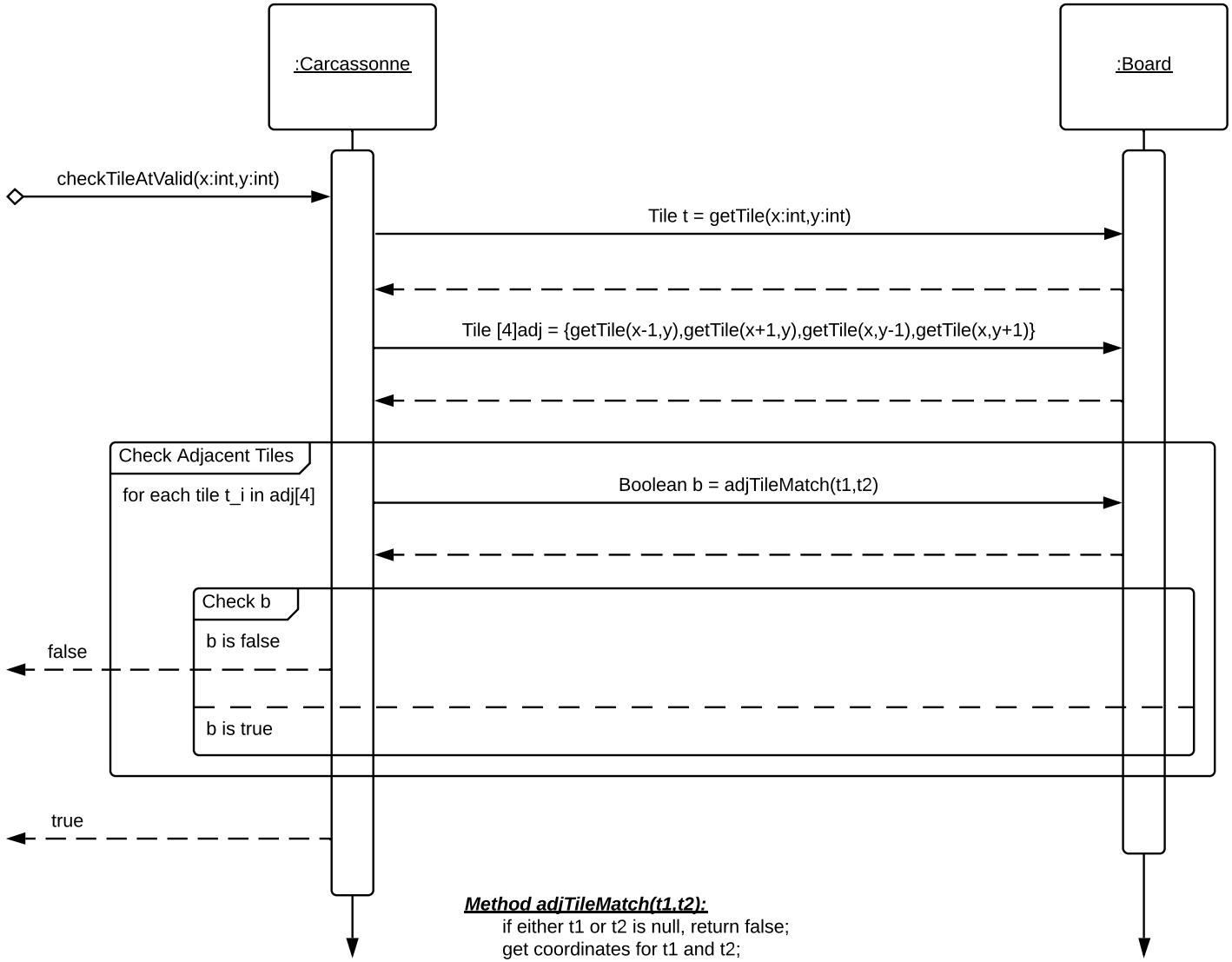
1. The Game is not in the initiation state or has already ended;
2. The Game is Ready (valid initiation with correct deck and board information) and is Running in the Gameplay loop that loops on all of the players;
3. The user is holding one and only one ***placeable*** Tile card that was returned by the Game Control prior to this call; (*attribute of placeable* is a postcondition of a prior function call return, see system sequence diagram);

Postcondition:

1. The behavior will return Positive Response signaling successful placement if and only if after the placement of the specified tile at the specified position with the specified rotation, all the invariants of the Carcassonne board holds, such that 1> for all arbitrary and fixed tiles on the board, exist at least one tile surrounding such tile such that it is properly abutted to this fixed tile (aka, not diagonally touching), 2> for all undirected bituples of tiles that abut each other, the adjacent segments of abutting tiles must have the same feature types, being one of the following: road, field, city, or monastery;
2. Otherwise, the behavior will return Negative Response signaling unsuccessful placement; In this condition, the board will be exactly the same as was before the behavior is called;
3. In both cases, after the behavior is called, all the invariants of the Carcassonne board hold, such that for all meeples currently placed on the board, exist no meeple that is occupying a finished feature;

# Interaction Tile Validation

Roger Zheng | October 4, 2019

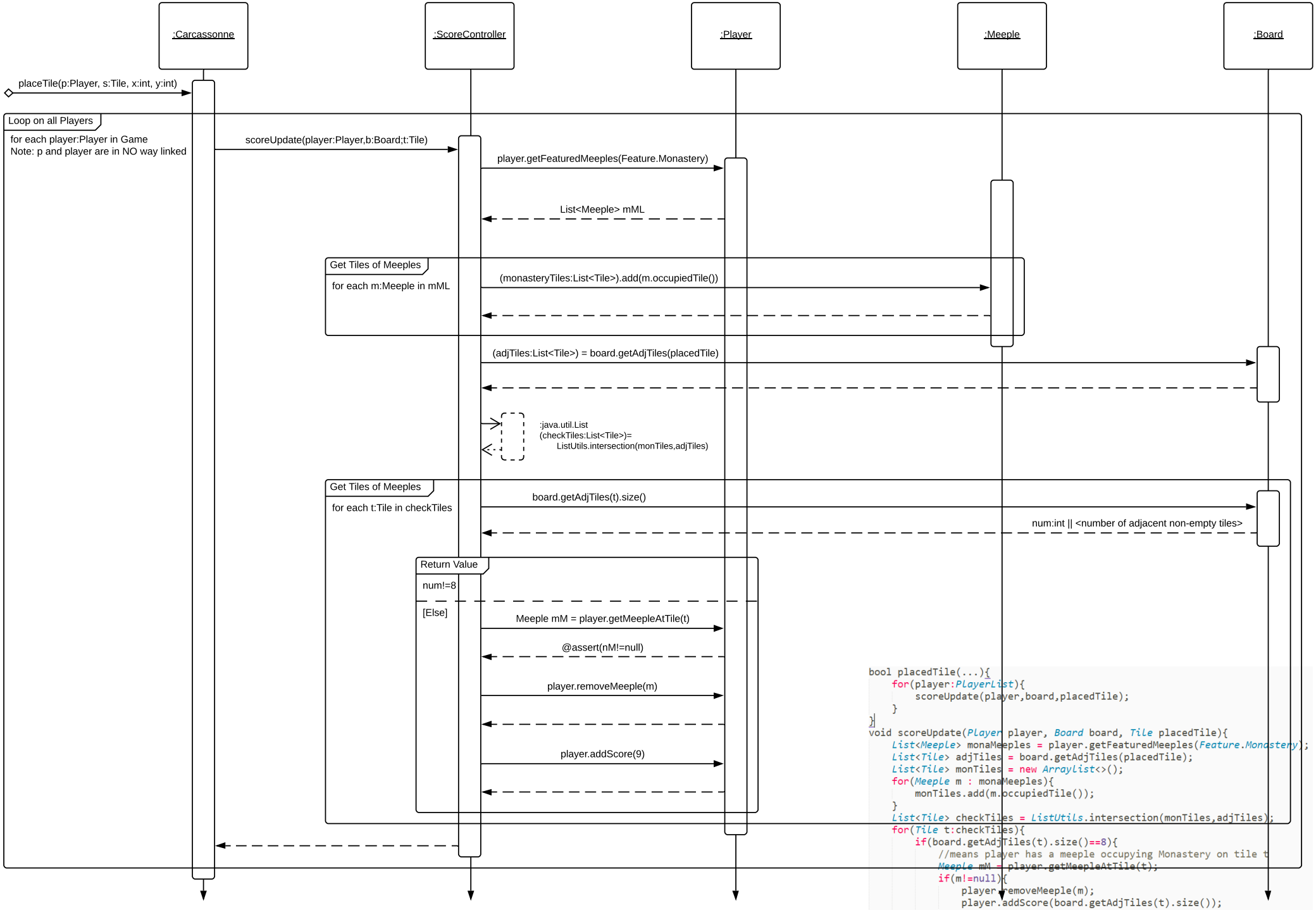



***Method adjTileMatch(t1,t2):***

if either t1 or t2 is null, return false;
get coordinates for t1 and t2;
if t1 and t2 not adjacent, return false;
find adjacent edges of t1 and t2, get features f1 f2 on both edge;
call featureMatch(f1,f2), if false return false; else return true;

# Interaction Monastery Scoring

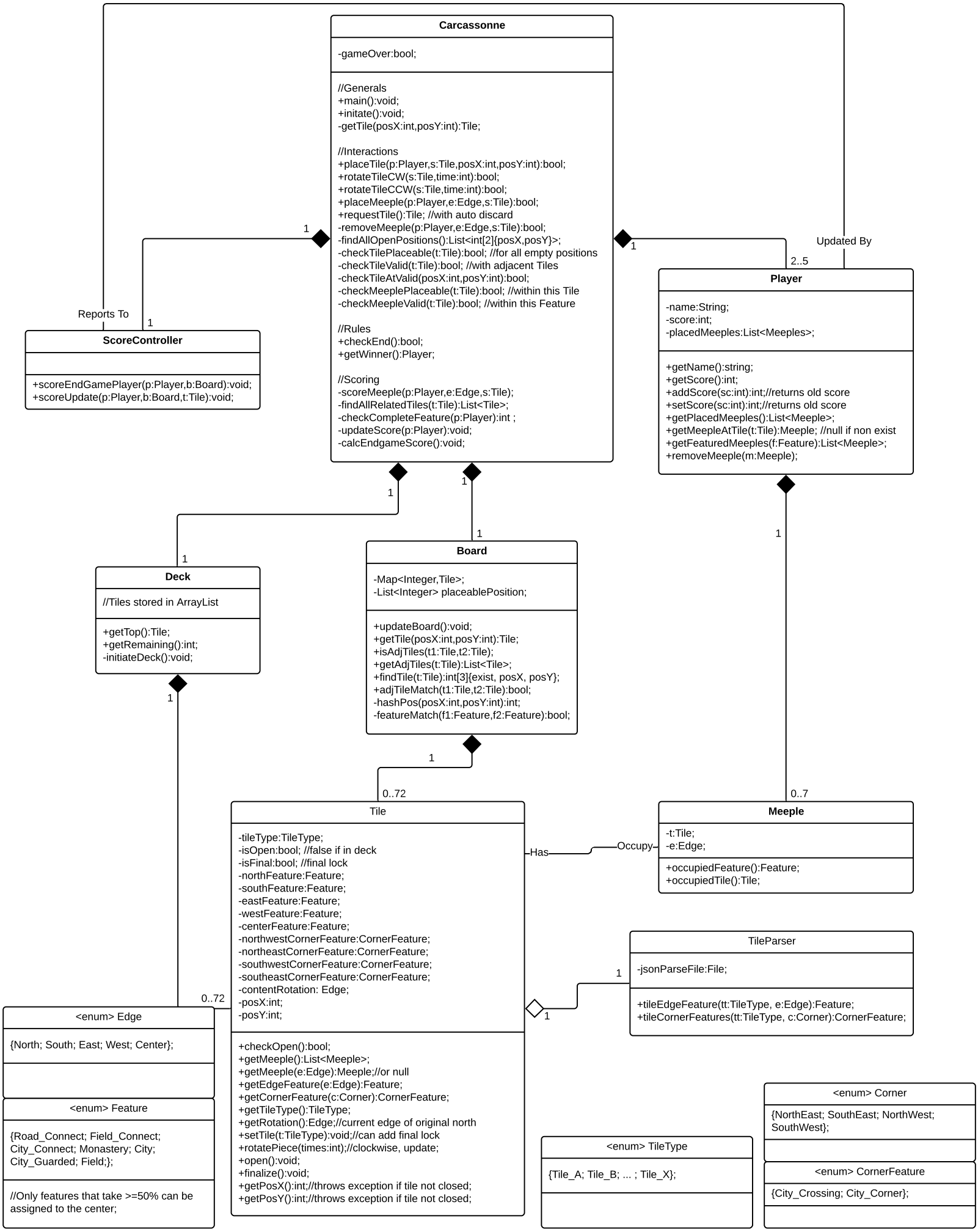
Roger Zheng | October 4, 2019




```
bool placedTile(...){
    for(player:PlayerList){
        scoreUpdate(player,board,placedTile);
    }
}
void scoreUpdate(Player player, Board board, Tile placedTile){
    List<Meeple> monaMeeples = player.getFeaturedMeeples(Feature.Monastery);
    List<Tile> adjTiles = board.getAdjTiles(placedTile);
    List<Tile> monTiles = new ArrayList<>();
    for(Meeple m : monaMeeples){
        monTiles.add(m.occupiedTile());
    }
    List<Tile> checkTiles = ListUtils.intersection(monTiles,adjTiles);
    for(Tile t:checkTiles){
        if(board.getAdjTiles(t).size()==8){
            //means player has a meeple occupying Monastery on tile t
            Meeple mM = player.getMeepleAtTile(t);
            if(m!=null){
                player.removeMeeple(m);
                player.addScore(board.getAdjTiles(t).size());
            }
        }
    }
}
```

# Carcassonne Class Diagram

Roger Zheng | October 4, 2019



## ***Rationale***:

The design includes a Top-level GameController Class (Carcassonne Class), which has composition Player, ScoreController, Board, and Deck; There can be 2-5 player objects for the game, each having a name and at most 7 placed meeples objects owned at the same time; the score controller is a hot-swappable and extensible class/interface that scores the board at runtime and end game; the board and deck holds a total of 72 Tile objects, which will be parsed by the Tile Parser at the beginning of the game and all shuffled into the Deck; the tiles have information of the corners, edges, and center, as well as its position on the board if and only if it is placed in the board (see #2 justification for more details); the meeple will have information of which tile it is placed on and which edge (including center) of such tile it is placed on;
Enums will be used heavily to represent TileTypes and Features; see class diagram for more information on Enums;
The user will interact only with methods declared in the GameController class to play to game; checks will be run automatically in interaction method calls and the user will be prompted when an action is invalid; Scoring is two staged and done automatically after each tile placement and at the end of the game; score of each player will be communicated automatically by the ScoreController to each player object; the design is optimized for performance and least board traversal at runtime; the only time board traversal will be required is during the end game scoring;

Below are several design choices that I would like to justify respectively:

**Interaction Implementation in Game Controller Class vs Player Class:**
In my initial design, all of the board manipulation methods were embedded in the Player class instead of the Carcassonne class; for instance, the “rotatePiece()” method would be implemented as a member method of the Player class, and can only be called by an initiated player instance; at first, this seems to show a low representation gap, as the players are the real life objects that should be interacting with the game;
However, such a design would cause many issues; firstly, in the computer game situation, the object that initiates the actions must be user that interacts with the top module of the game (such as GUI), and to have several different users with different interaction implementations would lead to large amounts of code duplication and very low cohesion; secondly, it will be extremely hard to implement such a design, as information will have to be passed over to and from the player for all evaluations after an action is performed; for instance, after performing a tile placement, the player will have to notify the Game class to check for the validity of the placement, as the player does not (and should not) have a private aggregation or composition of the board; this would require passing around essential variables by reference and cause the code to be very coupled and confusing to maintain; there are also many other issues with this design which I will not list here;
Therefore, in this design, the player class will only be considered as a storage container, storing the name, current score, and current meeple list of the player; only meeple control methods that involve manipulation of the meeple of that particular player will be included in the player class for the sake of information hiding and cohesion; all other high level interaction methods will be included instead in the Carcassonne class which acts as a game controller and the connection point of the user experience (and GUI in later assignments);

**Tile Composition and States:**

In the current version of the design, an instance of the tile class can be used to represent both an external tile that exist in the deck, a lifted tile that is held by the player, or a tile that is placed on the board; therefore, I have included several private member variables in the tile class to convey the state of each tile;
In the game initialization period, all 72 tiles needed are parsed from JSON, initiated as instances, and placed into the deck; at this stage, all tiles are “closed”, meaning that no changes can be made to them, and attempting to access their board related attributes will yield undefined behavior (return of null or return with exception); accessors will maintain that these attributes are not accessible; after the tile is placed on the board, it will be “opened”, and we can now rotate to tile to gain a desired result; the coordination attributes will remain inaccessible as enforced by the accessor methods; after the placement is decided by the user and validated by the Game Control, the tile will be finalized and have its coordinates on the board written stored as immutable data; after a tile is finalized, all board related attributes are now accessible, but the tile is considered totally immutable;
Such a design ensures that the game is perfectly safe, and nothing that should not be accessed can be accessed, nothing that should not be changed can be changed; it clearly defines the boundaries of the Tile object within regards of the Board and Deck, and minimizes the complexity of the system by only having a single Tile type instead of two types for the board and for the deck; overall, this design makes the Tile class well defined, implementation detached, easy to maintain, and safe to use;
###: The open vs closed checking mechanism may be removed, as it does not need to exist in the Tile class, and can rather be checked by the Board class and the Deck class; changes will be decided and made in implementation and resubmitted in 4c;

**Data Structure for Tile Storage in Board Class:**
The Board will use a sparse data structure instead of using the conventional dense representation of a 2D array for each tile; this will decrease the memory needed greatly, as the dense representation requires 72 times more space than the sparse representation; implementation details are yet to be decided, but an object of the interface Map will be used and the implementation will be swappable to achieve different design goals (agile design); a position class will be added in the implementation to serve as the hashkey;

**Meeple Attribution in Player Class:**
Meeples are attributed to the player class as the player is responsible for and is the owner of the meeples; alternatively, meeples can have a private field that specifies the player it belongs to, but this will make responsibility assignment more confusing and increase the representation gap; it will also make the meeple class more coupled to the player and less extensible, which is not necessary;

**Delegated Scoring Algorithm as Composition vs Embedded:**
A delegated scoring algorithm will provide hotswappability, decrease the coupling and dependencies, increase extensibility and robustness, and encourage information hiding; should we want to implement later expansion packs into the scoring algorithm (such as scoring rivers, fields), we will only need to provide another ScoringAlgorithm class to do so, without changing anything in the Carcassonne class;
###: a reference to the board may be passed in by constructor and stored in the ScoringAlgorithm class to decrease the need to copy the board every time a method in ScoringAlgorithm is called; this may or may not be optimal; changes will be decided and made in implementation and resubmitted in 4c;

Readme:

Note that all designs are subjected to change;
I have written some code already for this design and most of the aspects are well defined and cohesive;
most of the functions you see in the class diagram may have confusing naming or return values; for the purpose of this assignment, they are most likely will not be needed; all of the essential functions represented in the interaction diagrams should have pretty clear namings;
if you are really interested, please refer to the comments in the source code for clarification;