Rationale:

The design includes a Top-level GameController Class (Carcassonne Class) that implements the controller pattern that coherents the game subroutines and the observer pattern the coherents the GUI elements; The major game elements are composed of Players, the Game Rule Controller, the Board, and the Deck; There can be 2-5 player objects for the game, each having a name a score and at most 7 placed meeples objects owned at the same time; the game rule controller is a hot-swappable and extensible class/interface that scores the board at runtime and end game and reinforces the system invariants during the meeple placements; 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;
The GUI design will be implementing the observer pattern to establish a connection to the base game; the base game will not be aware of the GUI elements and will be notifying the subscribers of all game play events, including the GUI elements; The GUI elements may be linked to some actions on the board, such as a function from the game may be called when a button is pressed in the GUI;

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

<u>Interaction Implementation in Game Controller Class vs Player Class:</u>
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;
In general, the open and closed design made the design more clear to readers and made better enforcement of the invariants present in the carcassonne game rules; it also adheres to the “fail fast” rule in framework design so that wrong items do not propagate too far down the implementations;

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 has been added in the implementation to serve as the hashkey and will also be used by the GUI for defining positioning on the board for better readability and cohesion;

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;

Feature as Objects vs Feature as Enums with Real Time Calculation:
In the implementation I was constantly thinking about the advantages and issues of making features objects that can store the tiles contained in this feature and perform calculations using its own functions; I thought of this during the implementation of 4B, yet I do come to believe that this design is inferior according to my evaluations; I choose not to use this design as this would make the scoring process more embedded and coupled, and less likely to be extended and modified for performance; Also, such an approach can have a heavy performance impact, especially if there are many player-meeples and many features to constantly maintain; My approach seeks to represent feature as nothing more than a collection of tiles in a stateless format; This gives great flexibility in both general scoring processing and applying specific scoring rules that will give the most optimal performance; In general, although it makes the code slightly more complex, I do believe that the flexibility and performance is worth the trade off;

Unused by still present CornerSegment Classes:
In the original design, I have intended that the scoring process used not only edges but also corners for scoring, as I initially thought that corners will be necessary to score cities that are disconnected and distincted over tiles; in the implementation process I discovered a clever trick that eliminated the use of the corner segments by using pathfinding over the tiles; however, I still left the corner segment and all fully functional corner segment manipulation functions within the code base as these may become useful in later expansion packs for carcassonne, and may also be very important for the integrity of the framework;

Observer Pattern:
The observer pattern makes sure that the game object does not depend on the implementation details of the GUI, and the GUI is not extremely embedded or coupled to the game

implementation; This allows for more flexibility in design and revision as changing items in the core implementation and the GUI implementation will likely not affect each other; overall, the core design does not have to account for communicating directly to the GUI but only have to update and send notifications in the form of listeners, and also the GUI implementation does not have to send data back and forth with the core implementation, but only needed to process the received data from listener methods; this decrease overall coupling and is the best way that this problem can be solved;