The first decision I should justify is that of not splitting Segments into the subtypes of segments despite there being road, city, field, and cloister segments. I initially planned to separate them into subclasses, however realized that they share the large majority of implementation between them. The only thing in my segment implementation that depended on a specific segment type was two one line functions that checked if a city segment had a pennant and one which checked if a road segment was a road end (at an intersection or cloister). Yes, splitting the segments into subclasses would make sense to reduce the representational gap, but I didn't feel that it was worth the overhead and code repetition.

The main difference between segment types came into play with features, not the segments themselves. For features, the functionality to check if a feature has been completed and to score it varies greatly depending on which type of feature it is. Therefore I chose to make Feature an abstract class and have the 4 types of features inherit from it. Each of the subclasses implemented their own scoring system and feature completion check. I chose to use an abstract class here instead of other alternatives mainly because of code reuse. The large majority of feature implementation lives in the abstract class, not in the subclasses. If I were to use the strategy pattern for example it would have been more difficult to attain the same sort of code reuse. It also made sense to me conceptually to have the different feature types only do the things that they do differently from each other and leave the constants to the superclass.

I made the decision to represent my board as a 2-dimensional list rather than a map. I did this for two reasons. One, I don't have much experience using maps in programming while I have done a lot with lists. Therefore I felt more comfortable reasoning about a list and developing logic that I knew could get complex in certain areas. I didn't feel that it'd be a good idea to experience with an unfamiliar data structure in the first project I've ever done of this scale. Also, I did it as a list because it makes it extremely easy to visualise and that helped me a lot in developing the logic. I also chose to represent my tiles as a 2-dimensional list of segments for the same reason. There was the added benefit here of having tiles and the board represented in same way which made the code base more cohesive.

The follower object does not really have much useful functionality, but I wanted it to be its own class so they could be passed around. I knew that figuring out the scoring would be the hardest part of this assignment, so with that in mind I needed it to be as easy as possible to figure out which segment/feature/tile a follower was on and that was most easily achievable with follower being their own object. I also had a focus on real world representation as all of my objects are essentially taken directly from the rulebook.

The board class is the workhorse of the project because it has access to enough information to do useful things but does not need to make things public like the Carcassonne class. The board has access to all tiles and features which in turn has access to all segments so it can do operations which involve multiple tiles for example. That's a situation where the two tiles don't know about each other but the board is in the spot where it does. The Carcassonne class is only really there to put together all of the functionality in a user-friendly way so the game can be played with fluidity.