Rationale for homework 4c: Full Implementation with GUI

Name: Xinyu Bao   Andrew ID: xinyub

In the whole design process of Carcassonne game, there are bunch of important design decisions I made. Here I will address them as much as I can with respect to each class in the core implementation.

For Tile class, the type of each part of a tile is represented by an enum class, FeatureType, which is better than using constants since enum facilitates compile-time type check and thus it’s less error prone. Second, considering there could be multiple feature types in the center of a tile, for example both a city and a road in tile P, the center attribute is designed to be a list. The last key point about Tile class is that, as many instructions have suggested, it is an immutable class. Immutability is good because it’s thread safe, easier to test and spares programmers from verbose defensive copies.

For Deck class, there’s only one design point I need to mention. The key operation of drawing a tile is achieved by generating a random number rather than shuffling the whole deck because I think this makes testing easier since we already know the specific tile in a given index position, then it is easy to check whether a tile draw is correct.

Then there is a Segment class. This class is necessary since clearly every tile is composed of different segments, and each feature is a collection of different segments on the board. A single Tile class should not be responsible for all of these responsibilities, which corresponds to design principles of high cohesion and low representational gap. A Segment instance only holds a (x, y) coordinates rather than a tile because the game board has all information for all placed tiles already, so we do not need another copy of tiles in order to save memory and also follow the information expert heuristic. As in the Tile class, an enum class Orientation is applied to represent different orientations of a tile, which is better than constants. Finally, even though it seems that there should be a Meeple class corresponding to meeple in the domain model, I found it unnecessary since I can represent a meeple simply by an integer field in Segment class to differentiate its owners and another integer field in Player class to indicate how many meeples are available for each player. This saves some overhead for an extra Meeple class.

Moving on to Feature class, this is where I applied the template method pattern to have an abstract Feature class and 3 child class to represent road, city, and monastery feature respectively. Since a lot of operations such as *addSegment*, *containsSegment* are identical to for each feature, and at the same time the *checkComplete* method are also the same for CityFeature and RoadFeature, which can be cleared completed if no other segment can extend the current feature, applying template method pattern can achieve a lot of code reuse. I only need to override the *computeScore* method for different subclasses and the *checkComplete* method for MonasteryFeature. Besides, the 3 child classes are indeed a “Feature”, and they are also behavioral subtype of Feature class, which also justify the use of inheritance. Another design decision for MonasteryFeature is that a coordinates (x, y) should be maintained to record the place of the monastery.

For Board class, I choose to use a 143*143 2-D array of Tile to represent the game board. The reason I used an array is that array operations are easy and simple, and the size 143*143 is enough for any extreme tile placement cases. The Board class maintains a list of features, which represents all uncompleted features in current board, and is updated every time a new tile is placed. The Board is responsible for create and maintain feature list because it has all information needed for feature list, as indicated by information expert heuristic. With a feature list, Board is also responsible for almost all crucial operations such as placing a tile, placing a tile with meeple, checking whether a tile/meeple placement is legal, and updating the feature list. Placing a tile only and placing a tile with meeple is differentiated by two methods, where placing with meeple requires one extra argument which represents the placement orientation of the meeple. The feature list is updated by first handing the possible monastery features, then splitting the new tile into different segments, adding each segment to a possible existing feature in the list, creating a new feature if one segment is not added into any feature, merging features if a segment is added to multiple features, and finally deleting completed features.

Finally there is a GameSystem class, which is in charge of coordinating all different components’ behavior and merging them together. This is a typical case for the controller heuristic of responsibility assignment, so that low representation gap and high cohesion is achieved. GameSystem is responsible for creating and managing all necessary components of a game including Deck, Board, and Player, and so on. It also serves as an API for possible client such as the GUI application.

Then I will continue to talk about some design decisions I made for the GUI part. First and foremost, the GUI part is implemented using the observer pattern. Specifically, A GameChangeListener interface is added to the core code, which represents parts of client code that should be executed when something changed with the core. Then, GameSystem maintain a listener list, allowing one or more listeners to be registered. When data associated with client changes, all listeners will be notified to execute corresponding method through this listener list. In my implementation, a listener will be notified when the players’ info changed, a tile is placed on the board, the round is changed, one feature is completed, or the game is ended. Through observer pattern, the core game decoupled from the client code, in this case the GUI application, so the dependency relation is unidirectional from GUI to core code. The core doesn’t depend on the GUI implementation and as a result, low coupling and high cohesion is achieved.

There are also some key designs related to the interaction with players. The number of player should be indicated first, and then the main game board is displayed. One player is able to rotate a tile by simply clicking on the tile on the tile panel, or on any place on the board if the new drawn tile is there. Players are only allowed to place a tile on buttons adjacent to current existing tiles, and all other buttons on the board is invisible.

Finally, for the exception handling part, every possible wrong operation is either prevented from happening or an informative dialogue will show up to notify the user that it is an illegal operation. I didn’t filter to show only the legal placement positions either for the tile and meeple for that I think I don’t want to feel like the program is putting too much restriction on me if I were the player, so I designed it this way such that the players will be notified only after a tile/meeple placement is illegal to mimic the real-world board game experience. I have other exception statements in my GUI as well, but they are not supposed to happen and are indications of programming mistakes which should be handled by the programmer.