GRASP Patterns

Low Coupling

In order to achieve low coupling, we made sure that there were no unnecessary relations between classes. For instance, we considered having Card related to Game and Hand. Since we considered the Low Coupling pattern, we realized that we only needed Card to be coupled with Hand, since Card was already coupled with Hand and Hand was already coupled with Game.

High Cohesion

In order to have high cohesion, we made sure that our classes all have a number of highly related methods that do not too many operations. For instance, the only method that Player has is makeMove(). Before, we had considered putting methods about calculations of statistics in Player. However, based on the High Cohesion principle, we decided to have only highly related methods in Player.

Information Expert

We used the Information Expert pattern to identify what class should have any operations dealing with changing the GameStatistics class. Since Game has all the information about the game, such as number of players and player types, then Game was the appropriate choice based on the Information Expert pattern. Therefore, Game has the operations related to modifying and creating GameStatistics.

Creator

We used the Creator principle to decide what class should create the Players. Since the Game contains the Players, closely uses the Players, and has the data to initialize the Players, Game was a good choice for the Creator of the players (SD3).

Controller

We used the Controller pattern to decide what class should coordinate system operations. We decided to use a Façade Controller, because the operations are coming in over a single pipeline. Namely, all the operations are coming directly from the GUI. In addition, there are not many system operations. There really is only two main ones: setPreferences (SD1) and beginGame (SD2). Therefore, a Façade Controller was appropriate. Our Façade Controller is our GameController class. It delegates tasks to other objects and controls the activity in the system.

Polymorphism

We used Polymorphism when deciding how to deal with the different types of players. Since ParallelPlayer and SequentialPlayer were very similar but have a few different behaviors, we decided to have an abstract class, Player, and then two subclasses, ParallelPlayer and SequentialPlayer. This will also be useful for any future versions of the system. If a developer wants to add a different type of player, he can easily add another subclass to Player.

Indirection

Since we did not want to have Player directly coupled with the GameState, we used the Indirection pattern. We used an intermediate object, Game, to mediate between GameState and Player. That way, we did not have to have extra coupling between Player and GameState.

Pure Fabrication

Information Expert would lead us into high coupling with regard to presenting the details of the poker game to the user. For instance, each Player would have to output its data about the move decision to the GUI and the game would also have to be related to the GUI. Instead of following the Information Expert pattern for game statistics, we used the Pure Fabrication pattern. We made a GameStatistics class. GameStatistics is used to summarize the statistics about the game without having too high coupling. We were careful to avoid a common contraindication of Pure Fabtrication, which is too much data being passed to other objects for calculations. We made sure that GameStatistics does not request many calculations to be done by other classes.

Protected Variations

The most unstable part of the system is the neural network. Therefore, we used the Protected Variations pattern to create a stable interface around the neural network. We created a MoveGenerator class that is between the system components and the neural network. That way, if the neural network needs to be changed in the future, it can easily be altered without affecting the whole system.