

**The Effects of Game Awareness on the Decisions
of Texas Hold 'em Poker Agents**

by
John Kilby

Departmental Honors Thesis
The University of Tennessee at Chattanooga

Project Director: Dr. Andy Novobilski
Examination Date: March 28, 2005

Committee Members:
Dr. Billy Harris
Dr. Jack Thompson
Dr. Gavin Townsend (Liaison)

Examining Committee Signatures:

Project Director

Department Examiner

Department Examiner

Liaison, Departmental Honors Committee

Chairperson, Departmental Honors Committee

Abstract

One of the problems facing the study of artificial intelligence is how an agent can react rationally in a situation where there is incomplete information. One such environment where this can occur is Texas Hold 'em Poker. Agents in this environment often make conclusions based off of details not immediately important to their current hand. The tested hypothesis of this project was that by examining his current position in the game standings, an agent that modifies its strategy based on its prior record of success will perform better at Texas hold 'em than an agent that bases its decisions solely on the current hand. An application was coded in order to run simulations that determined if keeping up with the current rate of success would help an agent make better decisions. An analysis of the data showed that while the best performance was not always obtained using this technique, it did work as a way of minimizing risk for the player. It was also found that the technique worked best when used frequently in cases with good hand odds and infrequently in cases with bad hand odds. This risk minimization warrants further examination using dynamic parameter changing, rule changes, and environment changes.

Introduction

Sometimes “correct inference is not *all* of rationality, because there are often situations where there is no provably correct thing to do, yet something must still be done” [1]. The decisions that are made in the game of Texas hold 'em poker, one of poker's most popular variants, provide an example of this [2]. The technology and understanding are available to determine which course of action has the best chance of resulting in a positive outcome, but there is always an element of uncertainty that permeates the game. This uncertainty provides for some interesting alternatives in deriving the best course of action.

Much of the research conducted today on Texas hold 'em agents concerns how they can compete with their opponents on a psychological level [3]. It observes the bets and hand outcomes of other players to determine what style of play each player is using. Some agents have done this by collecting statistical data on the actions of their opponents [4]. Other agents have observed the time it takes for other players to play [5]. Some simulations go as far as having the agents respond to input from other players [6]. These observational elements often form a great deal of the agent's strategy.

A common theme in all of these approaches is that “agents appraise their simulated world according to their simulated concerns according to their individual concerns” [6]. One of those individual concerns is how the game is going for the agent. By looking at the current standings, the agent might modify the decisions it makes in order to either catch-up with his opponent or to protect his lead. The hypothesis of this project is that by examining his current position in the game

standings, an agent that modifies its strategy based on its prior record of success will perform better at Texas hold 'em than an agent that bases its decisions solely on the current hand. This hypothesis was tested by running simulations of Texas hold 'em between two agents.

Game Rules

The basic rules of Texas Hold 'em are observed for this test. Players throw money into a pot to bet that they can form the best possible five-card hands. The game is played with a standard deck of 52 playing cards, with a 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king, and ace in each of the four suits: hearts, diamonds, clubs, and spades. There are no jokers and no cards are designated as wild cards. The ace is treated as a high card. Play begins by dealing each player two hole cards. Each player's hole cards are not observable to their opponents. As the game progresses, community cards are dealt face-up. The community cards can be used in combination with both, either, or neither of the two hole cards to create the highest-ranking five-card poker hand. The standard poker hands are used. From lowest to highest, they are: no pair (no cards make up any other hand), pair (two cards of the same rank), two pair (two sets of pairs), three of a kind (three cards of the same rank), straight (five cards with ranks in sequence), flush (five cards of the same suit), full house (three of a kind and a pair), four of a kind (four cards of the same rank), straight flush (a straight with cards of the same suit), and royal flush (straight flush that starts with a 10 and ends with an ace). If hands are of the same rank, ties go to the player with the cards of highest rank. For a full house, the rank of the three of a kind is checked first.

Otherwise, the cards that create the hand's rank (for example, the cards that give a player a pair) are compared first and followed by the cards not used to do so. If the tie is not broken by this method, the pot is split between the players. Otherwise, the player with the highest ranked hand takes the pot.

After the dealing of the hole cards and the first betting cycle, the first three community cards are dealt. These cards are called the flop. A round of betting occurs followed by the dealing of the fourth community card, or the turn. Again, a round of betting takes place before the final community card, the river, is dealt. One last round of betting takes place. If a player calls at this point, the hands of the non-calling players are revealed and the player with the highest ranking hand takes the pot. If at any time all players except one have folded, the remaining player takes the pot without revealing his cards.

The betting cycle begins with player next to the dealer. That player can either fold (remove himself from the game), check (not place a bet), or raise (put money into the pot). If that player checks, the bet passes to the next player who has the same options. If the player raises, the other players must match that bet in the pot or fold. Betting continues until all remaining players have placed the same amount of money in the pot. The only exception to this procedure is the first betting cycle. In this cycle, the first player to bet must bet the blind, or introductory bet. The next player can then either fold, call, or raise.

A couple of customarily observed rules were not taken into account in this project. Normally, players can opt to go "all-in" on a given bet. If only one player calls this bet, then no more betting takes place and all remaining cards are dealt. If

multiple players remain in the game, a new side pot is established for the remaining players. Another rule that is modified is the maximum amount that can be bet per cycle. Both of these rules are simplified to restrict the problem domain.

Software Design

The simulation software was written as a Java™ application [7]. The original program design included a graphical user interface and the use of Bayesian networks to aid the agent's decision-making process. This meant the built-in support for graphical user interfaces and support for Norsys Software's Netica™ software made Java an attractive choice [8]. Although the later two features were not included in the final project, they could be added with only a reasonable amount of overhead.

The structure of the program is shown in the UML diagrams in Figures 1 and 2 and includes nine classes. The CardSim serves as the class containing the main method and controls the flow of the game. A playing card is defined by the Card class and a deck of cards is defined by the CardDeck class. The PokerRules class contains the information needed to evaluate hands. The HandResults class is a data structure for storing the hand evaluation results. The Agent class is an abstract class that describes the basic strategy of the agent. Both the LogicalAgent and EmotionalAgent classes are derived from it. Finally, the SaveOutput class is used to record all output to the console to a log file.

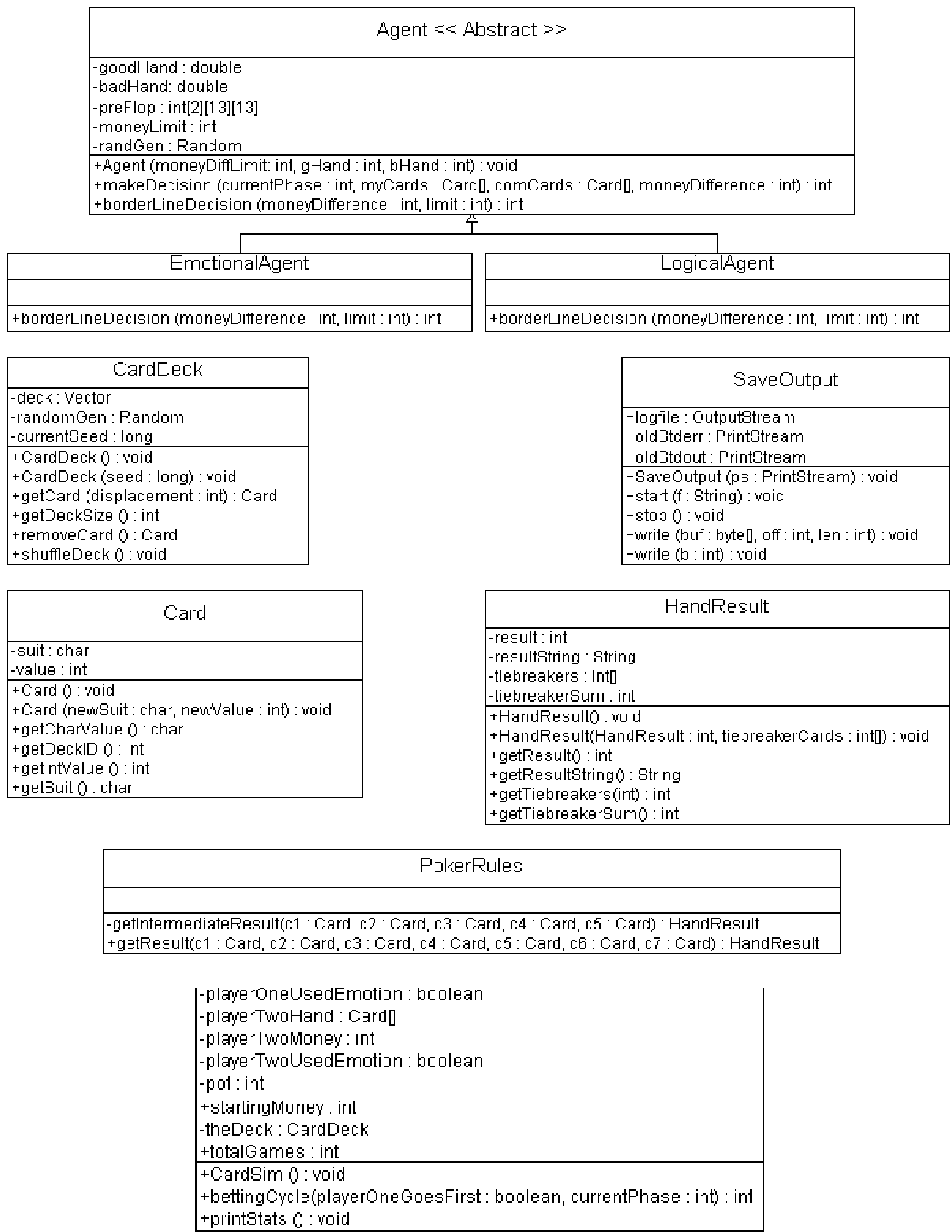


Figure 1. UML Diagram of CardSim Class

Figure 2. UML Diagrams of All Classes Excluding CardSim Class

One aspect of the game that must remain constant throughout the trials is the dealing of the cards. Each player must have the same cards to work with each hand. To create this effect, the CardDeck is created as a Vector of Card objects. The Deck is initially created with the cards in a specified order. Each CardDeck has a Random object that is seeded by a given number. To draw a Card, a random number that can be as big as the number of Card objects left in the CardDeck is chosen and used to pull the Card out of the CardDeck. The remaining Card objects stay in the order they were created. At the end of a hand, the CardDeck object is re-initialized, its previous seed is incremented, and its random number generator is re-seeded with the updated seed. Since each hand of Texas Hold 'em always uses two cards for each player along with five community cards, each player receives the same cards to work with every time. Three seeds were chosen based on the relative advantage each set of dealt cards gave to the players. The first seed, 10,000, is neutral, the second seed, 807, favors player one, and the third seed, 512,143, favors player two. This is demonstrated by the data in Table 1.

Table 1. Average Results For AgentCode = 1 By Seed

Seed	Difference Between Average Performance (Player One's Average Winnings Per Hand - Player Two's Average Winnings Per Hand) (\$)
------	---

(1) 10,000	2.00 ± 5.18
(2) 807	8.00 ± 3.79
(3) 512,143	-13.17 ± 5.42
Combined	-1.06 ± 10.24

Each Agent has the same basic method for decision making. At each state of the game, it makes a calculation of chance it has a higher ranking hand than its opponent. The result of the odds calculation is a number from 0 to 1000, where 0 indicates a 0% chance of winning and 1000 indicates a 100% chance of winning. This is the same as the percentage chance carried out to one decimal place and multiplied by ten. The odds are stored as an integer to increase performance and allow for more efficient storage. This is important as both performance and storage become issues in the calculation of odds. It would be nice if the Agent could brute force his way through the calculations, but when there are more than three unknown cards, the time to calculate the odds is very large, often taking over 30 minutes to make the calculation. Conversely, the farther into the game the agent gets, the harder it becomes for it to store all the odds in a data structure.

The Agent's logic for most phases of the game is different. Before the flop, a

look-up table is used to hold the odds. The table is efficient because all starting hands can be reduced to three indices: the rank of the first card, the rank of the second card, and whether or not they are of the same suit. The odds for the table were calculated using the trial version of the Gam Poker Analyzer [9]. The program's method of calculating the odds was not determined, but it looks like it uses a sampling method since results sometimes differ by 0.1% and it appears that the program is running through a number of iterations.

After the flop, a sampling method is carried out to determine the odds. Although there are approximately six-and-a-half million possible outcomes at this point of the game, randomly examining one million hands provides a good approximation. As shown in Table 2, the sampled result is accurate to within 1% of the actual value. This means the information is not perfect, but since both agents are using this same technique and the approximations are good, it is a fair method. For the rest of the game, a brute force method of calculating all the possible hands is used because the run-time is reasonable and devising with an efficient method of storing the data is difficult and consumes a great deal of memory.

Table 2. Sampling Approximations After The Flop

Seed	Sampled Odds (Chance of Winning %)	Fully-Calculated Odds (Chance of Winning %)	Difference Between Sampled and Fully- Calculated Odds (%)

Seed	Sampled Odds (Chance of Winning %)	Fully-Calculated Odds (Chance of Winning %)	Difference Between Sampled and Fully- Calculated Odds (%)
(1) 10,000	34.4	33.9	0.5
(2) 807	61.3	60.6	0.7
(3) 512,143	57.9	57.8	0.1

When an Agent is called upon to make a decision, it returns one of three results. A good hand is defined as having odds of winning above a specified percentage. If this occurs, the player enters an aggressive state. Whenever possible, he will raise his opponent. A bad hand is defined as having odds of winning below a specified percentage. If this occurs, the player enters an conservative state. The player only remains in the game if there is no cost to continue playing. The final state is where the two types of agents come into play. Each Agent is either a LogicalAgent or an EmotionalAgent. When the odds fall between the bad hand percentage and the good hand percentage, the LogicalAgent returns a neutral state. He is willing to call to stay in the game, but he does not intentionally raise the stakes. The EmotionalAgent differs in this regard. It examines the current money standings of the

game and its action is determined by its money, or amount that the player goes ahead or falls behind before modifying its strategy. If its lead or deficit is less than the money limit, a neutral state is returned. If its lead is above a certain amount or it trails by a certain amount, it returns either an aggressive or conservative state instead of the neutral state. This Agent is emotional in regard that it gets a “feel” for the game based on the current standings and modifies its decisions based on whether the game is going well or badly for him.

Method

The game was played out between two agents in a zero-sum manner. Each game included at most twenty hands. If either player ran out of money, the game was stopped. Each player was given \$20 for making bets. The game was initially tested with both agents as LogicalAgents. Trials were run for each seed using one of six possible settings for the definitions of a good hand or bad hand: an 80% chance of winning is a good hand and a 20% chance of winning is a bad hand, a 75% chance of winning is a good hand and a 25% chance of winning is a bad hand, a 70% chance of winning is a good hand and a 30% chance of winning is a bad hand, a 65% chance of winning is a good hand and a 35% chance of winning is a bad hand, a 60% chance of winning is a good hand and a 40% chance of winning is a bad hand, and finally a 55% chance of winning is a good hand and a 45% chance of winning is a bad hand. A second and third set of trials consisted of player one and player two taking turns being the EmotionalAgent. A trial using all of the seeds and good and bad hand definitions above were run for each turn as an emotional agent with two other varied

parameters: the money limit and the emotional code, or response to a given scenario. The money limit was tested for values ranging from zero to four dollars. There are two types of emotional responses tested. The first type of response makes the agent act aggressive when he is ahead and conservative when he is trailing. The other type is the exact opposite: the agent acts conservative when ahead and aggressive when behind.

Results

The detailed numerical results of the simulations are shown in the Appendix. The data in Table 1 suggests that for this simulation, player two had the better cards to work with. On average, he won about \$1 a hand from player one. The introduction keeping track of game results assisted player one greatly. Table 3 shows that he went from losing money to winning money on average each hand. This was an improvement of over \$1.50 a hand. Player two, on the other hand, ended up making less money. Table 4 shows that he dropped his profit per hand by almost a dollar. The important thing to notice is that he still did not lose money. That means that both players came out on the positive side using this technique, regardless of whether the cards favored him or not. Keeping track of the status of the game is akin to an insurance policy. By buying into the technique, the agent might not be as quick to make a certain profit, but the agent minimizes his risk by doing so.

Table 3. Average Results For AgentCode = 2 By Seed

Seed	Average Winnings Per Hand (\$)	Logical Agent's Winnings Per Hand (\$)	Difference (\$)
1	2.75 ± 7.14	2.00 ± 5.18	0.75 ± 12.32
2	6.13 ± 4.40	8.00 ± 3.79	-1.87 ± 8.19
3	-7.12 ± 4.76	-13.17 ± 5.42	6.05 ± 10.18
Combined	0.59 ± 7.90	-1.06 ± 10.24	1.65 ± 18.14

Table 4. Average Results For AgentCode = 3 By Seed

Seed	Average Winnings Per Hand (\$)	Logical Agent's Winnings Per Hand (\$)	Difference (\$)
1	3.13 ± 5.97	-2.00 ± 5.18	5.13 ± 11.15

Seed	Average Winnings Per Hand (\$)	Logical Agent's Winnings Per Hand (\$)	Difference (\$)
2	5.25 ± 5.46	-8.00 ± 3.79	13.25 ± 9.25
3	9.05 ± 4.09	13.17 ± 5.42	-4.12 ± 9.51
Combined	0.22 ± 8.19	1.06 ± 10.24	-0.84 ± 18.43

Although results from the effects of other parameters are not completely clear, they do reveal more detail about the optimal points at which this technique is used. Figures 3 and 4 provide a visual indicator of these points. When using the 80% / 20% odds standards, both agents struggle when they use this technique. The same can be said of the 65% / 35% standards. Both agents finish about even with or without this technique at the 75% / 25% and 65% / 35% points. The results did not agree with each other at the 55% / 45% mark, but it was clear that both agents worked best with the technique at the 70% / 30% mark. From observation of the trials, many of the odds are in this range. The LogicalAgent would often be playing neutral in this range while the EmotionalAgent would be able to keep the LogicalAgent in the game when he was playing badly and get out of the game when he was playing well. The other parameters consider when this technique should be taken advantage of relative

to the lead or deficit of the agent and how the response such be structured. At what point the technique should start being used depends on the cards dealt. For player one's case, the sooner he started using the technique (the lower the money limit), the better the results. On the other hand, player two got a bigger advantage of using the technique sparingly and only in extreme situations (a high money limit). A dynamic money limit might be useful for future agents that examines the state of the game and decreasing the money limit when doing badly and increases the money limit when doing well. For both players, the best response was to play conservative when ahead and aggressive when behind. Numerical evidence of this shown in Tables 5 and 6. This makes more sense that the opposite choice, because this strategy allows the player to play good hands and avoid losing money with bad hands.

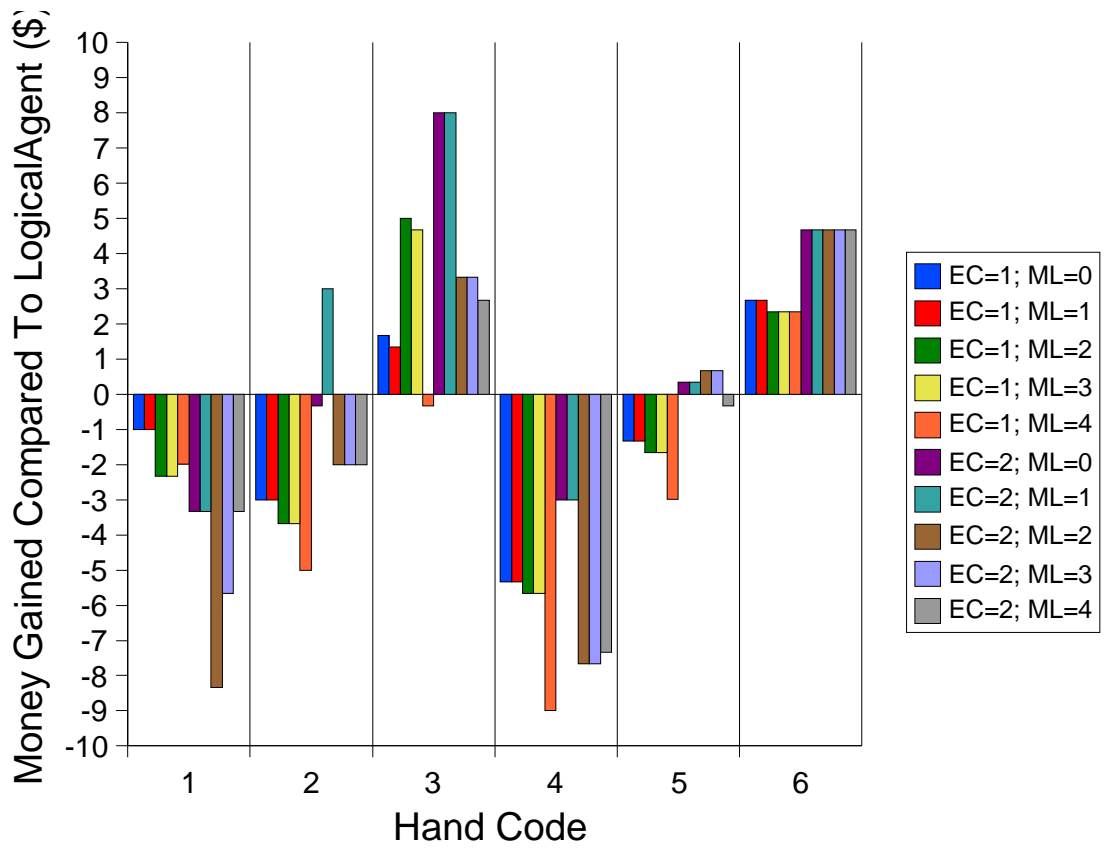


Figure 3. EmotionalAgent Versus LogicalAgent Performance For Player One
By EmotionCode (EC) and Money Limit (ML)

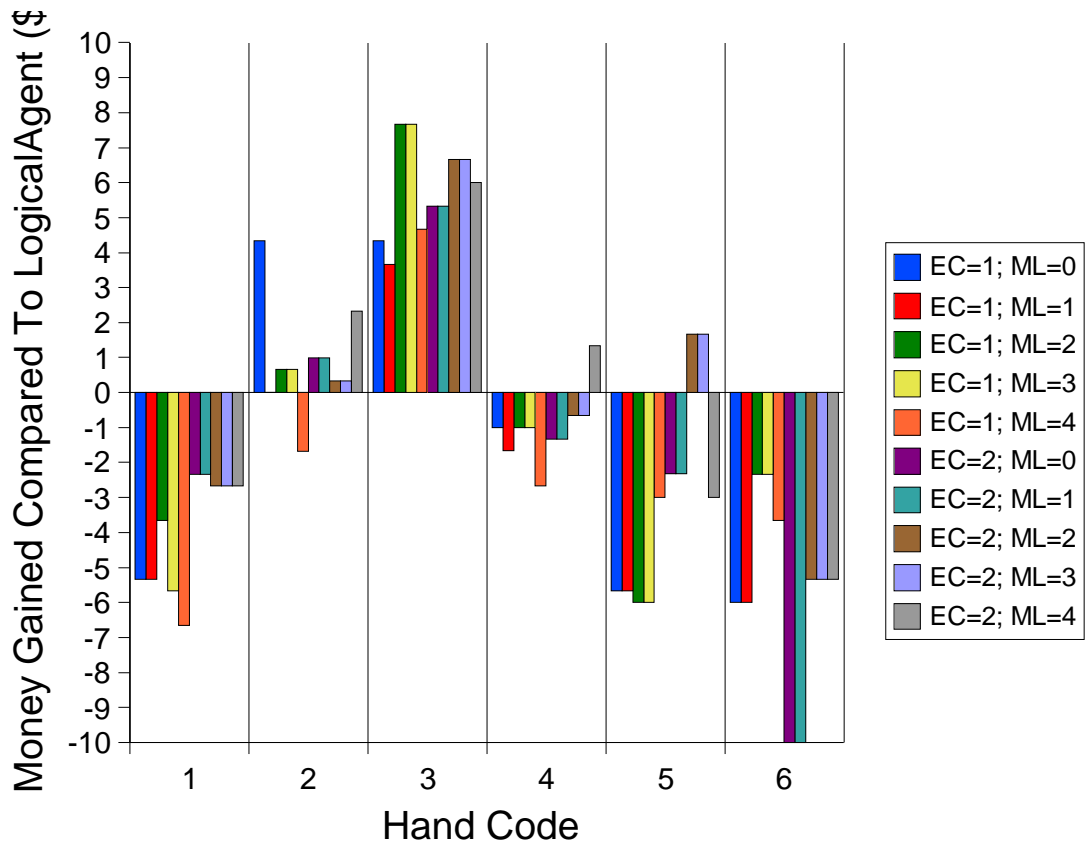


Figure 4. EmotionalAgent Versus LogicalAgent Performance For Player Two
By EmotionCode (EC) and Money Limit (ML)

Table 5. Breakdown of Results For AgentCode = 2 By EmotionCode

EmotionCode	Average Winnings Per Hand (\$)
1	-0.02 ± 8.26

EmotionCode	Average Winnings Per Hand (\$)
2	1.20 ± 7.52

Table 6. Breakdown of Results For AgentCode = 3 By EmotionCode

EmotionCode	Average Winnings Per Hand (\$)
1	-0.23 ± 8.01
2	0.68 ± 7.11

Conclusion

The hypothesis that an agent's play would be become better because of his ability to see into the status of the game was not proved. What the strategy actually did was allow the agent to play it safer and take fewer risks with his money. Knowing that this effect is achieved, a more focused study of this phenomenon can be conducted. Given that the standard deviation of the results were all very high, it would be nice to run more trial in attempt to gain more accurate data. As stated

earlier, the ability to dynamically change the agent's parameters such as the money limit would likely yield a performance increase. The additional of standard Texas Hold 'em rules such as the ability to go "all-in" would allow the agent to play more aggressively or conservatively and possibly perform better. Several aspects of the environment could be changed. If the games were allowed to run longer and the agents were allowed to bet more money per round, the agent keeping track of the game's status would likely be able to outperform his opponent by a much greater margin. If multiple agents with different behaviors were added, the skill of the agent that keeps track of his rate of success could be verified to a greater extent. The results of this research indicate that there is a potential that this effect could help to build a Texas hold 'em agent that is better suited for playing in the long run than in the short run by minimizing risk.

References

- [1] P. Norvig and S. Russel, *Artificial Intelligence: A Modern Approach*, 2nd Ed., Prentice Hall, 2003.
- [2] Poker and Its Many Versions, <http://www.usplayingcard.com/gamerules/poker.html>.
- [3] D. Boulton, *Bayesian Opponent Modeling in Poker*, July 2003, <http://www.csse.monash.edu.au/hons/projects/2003/Darren.Boulton/website/>.
- [4] L. Booker, *A No Limit Texas Hold'em Poker Playing Agent*, September 2004 <http://www.doc.ic.ac.uk/~dvd03/poker/documents/lrb.pdf>.
- [5] J. Gruenspecht, *Learning Texas Hold 'Em with Temporal Differencing*, December 2001, <http://zoo.cs.yale.edu/classes/cs490/01-02a/gruenspecht.joshua.jfg9/cs490f.html>.
- [6] C. Elliott and S. Marquis, *Emotionally Responsive Poker Playing Agents*, 1994, <http://condor.depaul.edu/~elliott/ar/ftp/ar/lll/aient94n.ps>.
- [7] *Java™ Technology*, <http://java.sun.com/>
- [8] *Netica™ Application*, <http://norsys.com/netica.html>
- [9] *Poker Analyzer*, http://gamsoft.ru/pkanalyzer_e.htm

Appendix

Table 7. Hand Code Interpretations

Hand Code	Good Hand Odds	Bad Hand Odds
1	> 80%	< 20%
2	> 75%	< 25%
3	> 70%	< 30%
4	> 65%	< 35%
5	> 60%	< 40%
6	> 55%	< 45%

Table 8. AgentCode Interpretations

AgentCode	Player 1 Agent Type	Player 2 Agent Type
------------------	----------------------------	----------------------------

AgentCode	Player 1 Agent Type	Player 2 Agent Type
1	Logical	Logical
2	Emotional	Logical
3	Logical	Emotional

Table 9. EmotionCode Interpretations

EmotionCode	When Ahead By More Than Money Limit, Play...	When Behind By More Than Money Limit, Play...
1	Aggressive	Conservative
2	Conservative	Aggressive

Table 10. Average Results For AgentCode = 1 By HandCode

Hand Code	Difference Between Average Performance (Player One's Average Winnings Per Hand - Player Two's Average Winnings Per Hand) (\$)
1	-1.00 ± 9.54
2	-3.00 ± 14.00
3	1.67 ± 4.16
4	-5.33 ± 12.50
5	-1.33 ± 10.69
6	2.67 ± 14.01

Table 11. Average Results For AgentCode = 2 By HandCode

Hand Code	Average Winnings Per Hand (\$)	Logical Agent's Winnings Per Hand (\$)	Difference (\$)
1	2.07 ± 2.24	-1.00 ± 9.54	3.07 ± 11.78
2	-1.05 ± 2.21	-3.00 ± 14.00	1.95 ± 16.21
3	-0.23 ± 2.73	1.67 ± 4.16	-1.90 ± 6.89
4	-2.63 ± 1.98	-5.33 ± 12.50	2.70 ± 14.48
5	0.93 ± 1.25	-1.33 ± 10.69	2.26 ± 11.94
6	4.90 ± 1.17	2.67 ± 14.01	2.23 ± 15.18

Table 12. Breakdown of Results For AgentCode = 2 By Money Limit

Money Limit	Average Winnings Per Hand (\$)
0	1.39 ± 7.26
1	1.64 ± 7.03
2	0.11 ± 8.60
3	0.31 ± 8.33
4	-0.50 ± 8.38

Table 13. Detailed Results For AgentCode = 2

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	0	1	4.33	5.33	-1.00

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	0	2	-2.33	0.67	-3.00
1	0	3	-2.33	-4.00	1.67
1	0	4	-2.00	3.33	-5.33
1	0	5	0.33	1.66	-1.33
1	0	6	4.00	1.33	2.67
1	1	1	4.33	5.33	-1.00
1	1	2	-2.33	0.67	-3.00
1	1	3	-2.67	-4.00	1.33

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	1	4	-2.00	3.33	-5.33
1	1	5	0.33	1.66	-1.33
1	1	6	4.00	1.33	2.67
1	2	1	3.00	5.33	-2.33
1	2	2	-3.00	0.67	-3.67
1	2	3	1.00	-4.00	5.00
1	2	4	-2.33	3.33	-5.66
1	2	5	0.00	1.66	-1.66

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	2	6	3.67	1.33	2.34
1	3	1	3.00	5.33	-2.33
1	3	2	-3.00	0.67	-3.67
1	3	3	0.67	-4.00	4.67
1	3	4	-2.33	3.33	-5.66
1	3	5	0.00	1.66	-1.66
1	3	6	3.67	1.33	2.34
1	4	1	3.33	5.33	-2.00

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	4	2	-4.33	0.67	-5.00
1	4	3	-4.33	-4.00	-0.33
1	4	4	-5.67	3.33	-9.00
1	4	5	-1.33	1.66	-2.99
1	4	6	3.67	1.33	2.34
2	0	1	2.00	5.33	-3.33
2	0	2	0.33	0.67	-0.34
2	0	3	4.00	-4.00	8.00

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
2	0	4	0.33	3.33	-3.00
2	0	5	2.00	1.66	0.34
2	0	6	6.00	1.33	4.67
2	1	1	2.00	5.33	-3.33
2	1	2	3.67	0.67	3.00
2	1	3	4.00	-4.00	8.00
2	1	4	0.33	3.33	-3.00
2	1	5	2.00	1.66	0.34

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
2	1	6	6.00	1.33	4.67
2	2	1	-3.00	5.33	-8.33
2	2	2	-1.33	0.67	-2.00
2	2	3	-0.67	-4.00	3.33
2	2	4	-4.33	3.33	-7.66
2	2	5	2.33	1.66	0.67
2	2	6	6.00	1.33	4.67
2	3	1	-0.33	5.33	-5.66

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
2	3	2	-1.33	0.67	-2.00
2	3	3	-0.67	-4.00	3.33
2	3	4	-4.33	3.33	-7.66
2	3	5	2.33	1.66	0.67
2	3	6	6.00	1.33	4.67
2	4	1	2.00	5.33	-3.33
2	4	2	-1.33	0.67	-2.00
2	4	3	-1.33	-4.00	2.67

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
2	4	4	-4.00	3.33	-7.33
2	4	5	1.33	1.66	-0.33
2	4	6	6.00	1.33	4.67

Table 14. Average Results For AgentCode = 3 By HandCode

Hand Code	Average Winnings Per Hand (\$)	Logical Agent's Winnings Per Hand (\$)	Difference (\$)
1	1.40 ± 7.27	1.00 ± 9.54	-0.40 ± 16.81
2	1.57 ± 6.72	3.00 ± 14.00	-1.43 ± 20.72
3	1.80 ± 6.47	-1.67 ± 4.16	3.47 ± 10.63

Hand Code	Average Winnings Per Hand (\$)	Logical Agent's Winnings Per Hand (\$)	Difference (\$)
4	2.33 ± 8.82	5.33 ± 12.50	-3.00 ± 21.32
5	-1.40 ± 6.72	1.33 ± 10.69	2.73 ± 17.41
6	-4.37 ± 10.76	-2.67 ± 14.01	-1.70 ± 24.77

Table 15. Breakdown of Results For AgentCode = 3 By Money Limit

Money Limit	Average Winnings Per Hand (\$)
0	-0.22 ± 7.92
1	-0.69 ± 7.73
2	1.00 ± 7.72

Money Limit	Average Winnings Per Hand (\$)
3	0.83 ± 7.94
4	0.19 ± 9.70

Table 16. Detailed Results For AgentCode = 3

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	0	1	0.00	5.33	-5.33
1	0	2	5.00	0.67	4.33
1	0	3	0.33	-4.00	4.33
1	0	4	2.33	3.33	4.33

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	0	5	-4.00	1.66	-1.00
1	0	6	-4.67	1.33	-5.66
1	1	1	0.00	5.33	-5.33
1	1	2	0.67	0.67	0.00
1	1	3	-0.33	-4.00	3.67
1	1	4	1.67	3.33	-1.66
1	1	5	-4.00	1.66	-5.66
1	1	6	-4.67	1.33	-6.00

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	2	1	1.67	5.33	-3.66
1	2	2	1.33	0.67	0.66
1	2	3	3.67	-4.00	7.67
1	2	4	2.33	3.33	-1.00
1	2	5	-4.33	1.66	-5.99
1	2	6	-1.00	1.33	-2.33
1	3	1	-0.33	5.33	-5.66
1	3	2	1.33	0.67	0.66

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	3	3	3.67	-4.00	7.67
1	3	4	2.33	3.33	-1.00
1	3	5	-4.33	1.66	-5.99
1	3	6	-1.00	1.33	-2.33
1	4	1	-1.33	5.33	-6.66
1	4	2	-1.00	0.67	-1.67
1	4	3	0.67	-4.00	4.67
1	4	4	0.67	3.33	-2.66

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
1	4	5	-1.33	1.66	-2.99
1	4	6	-2.33	1.33	-3.66
2	0	1	3.00	5.33	-2.33
2	0	2	1.67	0.67	1.00
2	0	3	1.33	-4.00	5.33
2	0	4	2.00	3.33	-1.33
2	0	5	-0.67	1.66	2.33
2	0	6	-9.00	1.33	-10.33

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
2	1	1	3.00	5.33	-2.33
2	1	2	1.67	0.67	1.00
2	1	3	1.33	-4.00	5.33
2	1	4	2.00	3.33	-1.33
2	1	5	-0.67	1.66	2.33
2	1	6	-9.00	1.33	-10.33
2	2	1	2.67	5.33	-2.66
2	2	2	1.00	0.67	0.33

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
2	2	3	2.67	-4.00	6.67
2	2	4	2.67	3.33	-0.66
2	2	5	3.33	1.66	1.67
2	2	6	-4.00	1.33	-5.33
2	3	1	2.67	5.33	-2.66
2	3	2	1.00	0.67	0.33
2	3	3	2.67	-4.00	6.67
2	3	4	2.67	3.33	-0.66

Emotional Code	Money Limit (\$)	Hand Code	Emotional Average Winnings (\$)	Logical Average Winnings (\$)	Difference (\$)
2	3	5	3.33	1.66	1.67
2	3	6	-4.00	1.33	-5.33
2	4	1	2.67	5.33	-2.66
2	4	2	3.00	0.67	2.33
2	4	3	2.00	-4.00	6.00
2	4	4	4.67	3.33	1.34
2	4	5	-1.33	1.66	-2.99
2	4	6	-4.00	1.33	-5.33

Logged Output Naming Scheme

Folder names:

trial[a][b][c][d]

[a]: Agent Code:

- 1: both logical
- 2: player 1 emotional, player 2 logical
- 3: player 1 logical, player 2 emotional

[b]: Emotional Code:

- x: not applicable
- 1: emotional agent plays aggressive when ahead, conservative when behind
 - 2: emotional agent plays conservative when ahead, aggressive when behind

[c]: Money Limit:

- x: not applicable
- 0 to 4: amount of money emotional agent leads or falls behind before it modifies its strategy

[d]: Good / Bad Hand Definition:

- 1: Good hands are defined as at least an 80% chance of winning, bad hands are defined as at most a 20% chance of winning
- 2: Good hands are defined as at least an 75% chance of winning, bad hands are defined as at most a 25% chance of winning
- 3: Good hands are defined as at least an 70% chance of winning, bad hands are defined as at most a 30% chance of winning
- 4: Good hands are defined as at least an 65% chance of winning, bad hands are defined as at most a 35% chance of winning
- 5: Good hands are defined as at least an 60% chance of winning, bad hands are defined as at most a 40% chance of winning
- 6: Good hands are defined as at least an 55% chance of winning, bad hands are defined as at most a 45% chance of winning

Inside the folders:

- results1.txt: result using seed 1 (10,000)
results2.txt: result using seed 2 (807)
results3.txt: result using seed 3 (512,143)

The log files themselves are fairly self-explanatory, but one item that might be confusing is the output of the "Positive Chance." The chance for player 1 winning is displayed first, followed by player 2's odds.

readme.txt

To run the program, do the following:

- 1) Ignore this step if no EmotionalAgent will be used. In the EmotionalAgent.java file, comment out the correct behavior as explained in the comments.
- 2) In the CardSim.java, edit the tweakable variables as desired and comment out the appropriate Agent setup variables as explained in the comments.
- 3) Compile all the classes and run the CardSim class.

-- JK

SaveOutput.java

```
/**
 * SaveOutput.java
 *
 * The SaveOutput class logs file output to a text file. It is taken directly
 * from Patrick Chan's SaveOutput class found at:
 * http://java.sun.com/developer/TechTips/1999/tt1021.html#tip2
 */

import java.io.*;

class SaveOutput extends PrintStream {
    static OutputStream logfile;
    static PrintStream oldStdout;
    static PrintStream oldStderr;

    SaveOutput(PrintStream ps) {
        super(ps);
    }

    // Starts copying stdout and
    //stderr to the file f.
    public static void start(
        String f) throws IOException {
        // Save old settings.
        oldStdout = System.out;
        oldStderr = System.err;

        // Create/Open logfile.
        logfile = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream(f)));

        // Start redirecting the output.
        System.setOut(
            new SaveOutput(System.out));
        System.setErr(
            new SaveOutput(System.err));
    }

    // Restores the original settings.
    public static void stop() {
        System.setOut(oldStdout);
        System.setErr(oldStderr);

        try {
            logfile.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // PrintStream override.
    public void write(int b) {
        try {
            logfile.write(b);
        }
    }
}
```

SaveOutput.java

```
    } catch (Exception e) {
        e.printStackTrace();
        setError();
    }
    super.write(b);
}
// PrintStream override.
public void write(
    byte buf[], int off, int len) {
    try {
        logfile.write(buf, off, len);
    } catch (Exception e) {
        e.printStackTrace();
        setError();
    }
    super.write(buf, off, len);
}
}
```

PokerRules.java

```
/**
 * PokerRules.java
 *
 * The PokerRules class takes a hand and determines the ranking of
 * the hand and the tiebreaker cards that go along with it. The
 * hand ranking is numerically encoded as follows:
 *
 * 10: Royal Flush
 * 9: Straight Flush
 * 8: Four Of A Kind
 * 7: Full House
 * 6: Flush
 * 5: Straight
 * 4: Three Of A Kind
 * 3: Two Pair
 * 2: Pair
 * 1: High Card
 *
 * It returns the results of the hand as a HandResult object. It has
 * no constructor, so it uses the default one (it has nothing to initialize).
 */

import java.util.*;

public class PokerRules
{
    // Get the result of a given seven-card hand (two player cards + five community cards)
    public HandResult getResult(Card c1, Card c2, Card c3, Card c4, Card c5, Card c6, Card c7)
    {
        int highestResult = 0;

        HandResult results[] = new HandResult[21];

        // Divide the calculation into each possible five-card hand
        results[0] = getIntermediateResult(c1, c2, c3, c4, c5);
        results[1] = getIntermediateResult(c1, c2, c3, c4, c6);
        results[2] = getIntermediateResult(c1, c2, c3, c4, c7);
        results[3] = getIntermediateResult(c1, c2, c3, c5, c6);
        results[4] = getIntermediateResult(c1, c2, c3, c5, c7);
        results[5] = getIntermediateResult(c1, c2, c3, c6, c7);
        results[6] = getIntermediateResult(c1, c2, c4, c5, c6);
        results[7] = getIntermediateResult(c1, c2, c4, c5, c7);
        results[8] = getIntermediateResult(c1, c2, c4, c6, c7);
        results[9] = getIntermediateResult(c1, c2, c5, c6, c7);
        results[10] = getIntermediateResult(c1, c3, c4, c5, c6);
        results[11] = getIntermediateResult(c1, c3, c4, c5, c7);
        results[12] = getIntermediateResult(c1, c3, c4, c6, c7);
        results[13] = getIntermediateResult(c1, c3, c5, c6, c7);
        results[14] = getIntermediateResult(c1, c4, c5, c6, c7);
        results[15] = getIntermediateResult(c2, c3, c4, c5, c6);
        results[16] = getIntermediateResult(c2, c3, c4, c5, c7);
        results[17] = getIntermediateResult(c2, c3, c4, c6, c7);
        results[18] = getIntermediateResult(c2, c3, c5, c6, c7);
        results[19] = getIntermediateResult(c2, c4, c5, c6, c7);
    }
}
```

```

                                PokerRules.java
results[20] = getIntermediateResult(c3, c4, c5, c6, c7);

// Search for highest result; default to results[0] if none found
for(int i = 1; i < 21; i++)
{
    if( results[i].getResult() > results[highestResult].getResult() )
        highestResult = i;
    else if ( (results[i].getResult() == results[highestResult].getResult()) &&
(results[i].getTiebreakerSum() >
results[highestResult].getTiebreakerSum()) )
        highestResult = i;
}

return results[highestResult];
}

// Get the result and tiebreakers for a given five-card hand
private HandResult getIntermediateResult(Card c1, Card c2, Card c3, Card c4, Card c5)
{
    int tiebreakers[] = new int[5];
    for(int i=0; i < 5; i++)
        tiebreakers[i] = 0;

    // Get card data
    char suitArray[] = new char[5];
    suitArray[0] = c1.getSuit();
    suitArray[1] = c2.getSuit();
    suitArray[2] = c3.getSuit();
    suitArray[3] = c4.getSuit();
    suitArray[4] = c5.getSuit();

    int valueArray[] = new int[5];
    valueArray[0] = c1.getIntValue();
    valueArray[1] = c2.getIntValue();
    valueArray[2] = c3.getIntValue();
    valueArray[3] = c4.getIntValue();
    valueArray[4] = c5.getIntValue();

    int sortedArray[] = valueArray;
    Arrays.sort(sortedArray);

    // Check for royal flush or straight flush
    if ( suitArray[0] == suitArray[1] && suitArray[1] == suitArray[2] &&
        suitArray[2] == suitArray[3] && suitArray[3] == suitArray[4])
    {
        // Royal flush
        if ( sortedArray[0] == 10 && sortedArray[1] == 11 && sortedArray[2] == 12 &&
            sortedArray[3] == 13 && sortedArray[4] == 14)
            return (new HandResult(10,tiebreakers));
        // Straight flush
        if ( sortedArray[1] == (sortedArray[0] + 1) && sortedArray[2] == (sortedArray[0] + 2)
&&
            sortedArray[3] == (sortedArray[0] + 3) && sortedArray[4] == (sortedArray[0] + 4) )
        {
            tiebreakers[0] = sortedArray[4];

```

```

        PokerRules.java
        return (new HandResult(9,tiebreakers));
    }
}
// Four of a kind
if (sortedArray[0] == sortedArray[1] && sortedArray[1] == sortedArray[2] &&
    sortedArray[2] == sortedArray[3] )
{
    tiebreakers[0] = sortedArray[2];
    tiebreakers[1] = sortedArray[4];
    return (new HandResult(8,tiebreakers));
}

    if (sortedArray[1] == sortedArray[2] && sortedArray[2] == sortedArray[3] &&
        sortedArray[3] == sortedArray[4] )
    {
        tiebreakers[0] = sortedArray[2];
        tiebreakers[1] = sortedArray[0];
        return (new HandResult(8,tiebreakers));
    }

// Full house
if ( sortedArray[0] == sortedArray[1] && sortedArray[1] == sortedArray[2] )
{
    if ( sortedArray[3] == sortedArray[4] )
    {
        tiebreakers[0] = sortedArray[0];
        tiebreakers[1] = sortedArray[4];
        return (new HandResult(7,tiebreakers));
    }
}
else if (sortedArray[2] == sortedArray[3] && sortedArray[3] == sortedArray[4] )
{
    if ( sortedArray[0] == sortedArray[1] )
    {
        tiebreakers[0] = sortedArray[4];
        tiebreakers[1] = sortedArray[0];
        return (new HandResult(7,tiebreakers));
    }
}

// Flush
if ( suitArray[0] == suitArray[1] && suitArray[1] == suitArray[2] &&
    suitArray[2] == suitArray[3] && suitArray[3] == suitArray[4] )
{
    tiebreakers[0] = sortedArray[4];
    tiebreakers[1] = sortedArray[3];
    tiebreakers[2] = sortedArray[2];
    tiebreakers[3] = sortedArray[1];
    tiebreakers[4] = sortedArray[0];
    return (new HandResult(6,tiebreakers));
}

// Straight
if ( sortedArray[1] == (sortedArray[0] + 1) && sortedArray[2] == (sortedArray[0] + 2) &&
    sortedArray[3] == (sortedArray[0] + 3) && sortedArray[4] == (sortedArray[0] + 4) )
{
    tiebreakers[0] = sortedArray[4];
    return (new HandResult(5,tiebreakers));
}

```

PokerRules.java

```

}
// Three of a kind
if (sortedArray[0] == sortedArray[1] && sortedArray[1] == sortedArray[2] )
{
    tiebreakers[0] = sortedArray[0];
    tiebreakers[1] = sortedArray[4];
    tiebreakers[2] = sortedArray[3];
    return (new HandResult(4,tiebreakers));
}
if (sortedArray[2] == sortedArray[3] && sortedArray[3] == sortedArray[4] )
{
    tiebreakers[0] = sortedArray[4];
    tiebreakers[1] = sortedArray[1];
    tiebreakers[2] = sortedArray[0];
    return (new HandResult(4,tiebreakers));
}
// Two pair or just a pair
int pairCount = 0;
boolean hasPair[] = new boolean[5];
for(int i=0; i < 5; i++)
    hasPair[i] = false;
if ( sortedArray[0] == sortedArray[1] )
{
    hasPair[0] = true;
    hasPair[1] = true;
    pairCount++;
}
if ( sortedArray[1] == sortedArray[2] )
{
    hasPair[1] = true;
    hasPair[2] = true;
    pairCount++;
}
if ( sortedArray[2] == sortedArray[3] )
{
    hasPair[2] = true;
    hasPair[3] = true;
    pairCount++;
}
if ( sortedArray[3] == sortedArray[4] )
{
    hasPair[3] = true;
    hasPair[4] = true;
    pairCount++;
}
if (pairCount == 2) // Two pair
{
    int tiebreakerIndexA = 0;
    int tiebreakerIndexB = 2;
    for(int i = 4; i >= 0; i--)
    {
        if ( hasPair[i] )
        {
            tiebreakers[tiebreakerIndexA] = sortedArray[i];
            i -= 1;
        }
    }
}

```

```

        PokerRules.java
        tiebreakerIndexA ++;
    }
    else
    {
        tiebreakers[tiebreakerIndexB] = sortedArray[i];
        tiebreakerIndexB ++;
    }
}
return (new HandResult(3,tiebreakers));
}
else if (pairCount == 1) // Pair
{
    int tiebreakerIndex = 1;
    for(int i = 4; i >= 0; i--)
    {
        if ( hasPair[i] )
        {
            tiebreakers[0] = sortedArray[i];
            i -= 1;
        }
        else
        {
            tiebreakers[tiebreakerIndex] = sortedArray[i];
            tiebreakerIndex ++;
        }
    }
    return (new HandResult(2,tiebreakers));
}
// High Card
tiebreakers[0] = sortedArray[4];
tiebreakers[1] = sortedArray[3];
tiebreakers[2] = sortedArray[2];
tiebreakers[3] = sortedArray[1];
tiebreakers[4] = sortedArray[0];
return (new HandResult(1,tiebreakers));
}
} // End class PokerRules

```

LogicalAgent.java

```
/**
 * LogicalAgent.java
 *
 * The LogicalAgent class is a subclass of the abstract Agent class. It fills in the
 * method that handles a borderline decision by returning the call/check decision
 * (returning 0).
 */

public class LogicalAgent extends Agent
{
    // Constructor
    public LogicalAgent(int moneyDiffLimit, int gHand, int bHand)
    { super(moneyDiffLimit, gHand, bHand); }

    // Handle borderline decisions
    public int borderLineDecision(int moneyDifference, int limit) { return 0; }
} // End class LogicalAgent
```

HandResult.java

```
/**
 * HandResult.java
 *
 * The HandResult class is a data structure for holding the
 * value of the hand. It stores the numerically encoded result
 * of the hand and the cards used to settle a tie (see PokerRules.java
 * for more detail about the result encodings).
 *
 * It has functions to:
 * - get the numerical result of the hand
 * - get the string result of the hand
 * - get the tiebreaker cards
 * - get the sum of the tiebreakers
 */

public class HandResult
{
    private int result;
    private int tiebreakers[];
    private int tiebreakerSum;
    private String resultString;

    // Default constructor (CANNOT CREATE EMPTY HANDRESULTS)
    public HandResult()
    {
        System.out.println("Error! Cannot create empty results!");
        System.exit(0);
    }

    // The only valid constructor
    public HandResult(int handResult, int tiebreakerCards[])
    {
        result = handResult;
        tiebreakers = tiebreakerCards;
        tiebreakerSum = 0;
        for(int i = 0; i < 5; i++)
            tiebreakerSum += tiebreakerCards[i];

        switch(result)
        {
            case 10: // Royal flush
                resultString = "a royal flush";
                break;
            case 9: // Straight flush
                resultString = "a straight flush";
                break;
            case 8: // Four of a kind
                resultString = "four of a kind";
                break;
            case 7: // Full house
                resultString = "a full house";
                break;
            case 6: // Flush
                resultString = "a flush";
        }
    }
}
```

```

        HandResult.java
        break;
    case 5: // Straight
        resultString = "a straight";
        break;
    case 4: // Three of a kind
        resultString = "three of a kind";
        break;
    case 3: // Two pair
        resultString = "two pair";
        break;
    case 2: // Pair
        resultString = "a pair";
        break;
    case 1:
        resultString = "a high card";
        break;
    default:
        System.out.println("ERROR: Unknown result");
        System.exit(0);
        break;
    }

}

// Get result as an integer value
public int getResult() { return result; }

// Get result in printable string format
public String getResultString() { return resultString; }

// Get a specific tiebreaker card
public int getTiebreakers(int index) { return tiebreakers[index]; }

// Get the sum of the tiebreakers
public int getTiebreakerSum() { return tiebreakerSum; }

} // End class HandResult

```

EmotionalAgent.java

```
/**
 * EmotionalAgent.java
 *
 * The EmotionalAgent class is a subclass of the abstract Agent class. It fills in the
 * method that handles a borderline decision by returning a decision based on how
 * far behind or ahead the agent is. It returns a 2 for the raise decision, a -2 for
 * the fold/check decision, and a 10 for the fold/check decision.
 */

public class EmotionalAgent extends Agent
{
    // Constructor
    public EmotionalAgent(int moneyDiffLimit, int gHand, int bHand)
    { super(moneyDiffLimit, gHand, bHand); }

    // Handle borderline decisions
    public int borderLineDecision(int moneyDifference, int limit)
    {
        // For emotion code = 1, uncomment this
        if ( moneyDifference > limit )
            return 2;
        else if ( moneyDifference < (-1 * limit) )
            return -2;
        else // -limit < moneyDifference < limit
            return 10;

        // For emotion code = 2, uncomment this
        /*
        if ( moneyDifference > limit )
            return -2;
            else if ( moneyDifference < (-1 * limit) )
                return 2;
            else // -limit < moneyDifference < limit
                return 10;
        */
    }
} // End class EmotionalAgent
```

CardSim.java

```
/**
 * CardSim.java
 *
 * The CardSim class serves as the main method for the application.
 * It handles the running of the game, including the betting cycle (how
 * to interpret the agent decisions) and keeping track of the game and stats.
 */

import java.io.*;
import java.util.*;

public class CardSim
{
    // Tweakable variables
    int emotionalPlayer = 0,
        emotionalMoneyLimit = 0,
        goodHand = 80 * 10,
        badHand = 20 * 10,
        totalGames = 20,
        startingMoney = 20;
    long deckSeed = 10000;

    // Custom set variables

    // if (emotionalPlayer == 0 ), uncomment this
    private LogicalAgent agent1 = new LogicalAgent(emotionalMoneyLimit, goodHand, badHand);
    private LogicalAgent agent2 = new LogicalAgent(emotionalMoneyLimit, goodHand, badHand);

    // if (emotionalPlayer == 1 ), uncomment this
    //private EmotionalAgent agent1 = new EmotionalAgent(emotionalMoneyLimit, goodHand, badHand);
    //private LogicalAgent agent2 = new LogicalAgent(emotionalMoneyLimit, goodHand, badHand);

    // if (emotionalPlayer == 2 ), uncomment this
    //private LogicalAgent agent1 = new LogicalAgent(emotionalMoneyLimit, goodHand, badHand);
    //private EmotionalAgent agent2 = new EmotionalAgent(emotionalMoneyLimit, goodHand, badHand);

    private CardDeck theDeck = new CardDeck(deckSeed);
    private int playerOneMoney = startingMoney, playerTwoMoney = startingMoney,
        gamesLeft = totalGames;

    // Variables unchanged for each trial
    private PokerRules gameRules = new PokerRules();
    private Card communityCards[] = new Card[5], playerOneHand[] = new Card[2], playerTwoHand[] = new
Card[2];
    private int pot;
    private boolean playerOneUsedEmotion, playerTwoUsedEmotion;

    /**
     *
     * Stats
     *
     * Key (Replace x with player number):
     *
     * pxW - Total wins by player x
     */
}
```

CardSim.java

```
* pxL - Total losses by player x
* pxT - Total ties by player x
* pxWI - Total wins by player x caused by purely logical play
* pxLI - Total losses by player x caused by purely logical play
* pxTI - Total ties by player x caused by purely logical play
* pxWe - Total wins by player x caused by mixed logical/emotional play
* pxLe - Total losses by player x caused by mixed logical/emotional play
* pxTe - Total ties by player x caused by mixed logical/emotional play
* pxCI - Total logical choices made by player x
* pxCa - Total emotional choices made by player x that were aggressive (caused raise decision)
* pxCu - Total emotional choices made by player x that were unchanged (caused call/check decision)
* pxCc - Total emotional choices made by player x that were conservative (caused fold/check decision)
* pxMI - Total money won/lost by purely logical play
* pxMe - Total money won/lost by mixed logical/emotional play
*
*/
private int p1W = 0, p1L = 0, p1T = 0, p1WI = 0, p1LI = 0, p1TI = 0, p1We = 0, p1Le = 0, p1Te = 0,
           p1CI = 0, p1Ca = 0, p1Cu = 0, p1Cc = 0, p1MI = 0, p1Me = 0,
           p2W = 0, p2L = 0, p2T = 0, p2WI = 0, p2LI = 0, p2TI = 0, p2We = 0, p2Le
= 0, p2Te = 0,
           p2CI = 0, p2Ca = 0, p2Cu = 0, p2Cc = 0, p2MI = 0, p2Me = 0;

public CardSim() throws IOException
{
    int i, winner, splitPot, previousMoneyPlayerOne = playerOneMoney,
        previousMoneyPlayerTwo = playerTwoMoney, temp;
    boolean validInput, keepPlaying = true, playerOneGoesFirst = false;
    String input;

    System.out.println("Parameters For This Run:");
    if ( emotionalPlayer == 0 )
        System.out.println("Neither player is the emotional agent.");
    if ( emotionalPlayer == 1 )
        System.out.println("Player 1 is the emotional agent.");
    else if ( emotionalPlayer == 2 )
        System.out.println("Player 2 is the emotional agent.");
    if ( emotionalPlayer != 0 )
        System.out.println("Emotional Money Limit: $" + emotionalMoneyLimit);
    temp = goodHand / 10;
    System.out.println("Good Hand: " + temp + "% odds or better");
    temp = badHand / 10;
    System.out.println("Bad Hand: " + temp + "% odds or worse");
    System.out.println("Total Games To Play: " + totalGames);
    System.out.println("Starting Money: $" + startingMoney);
    System.out.println("Deck Starting Seed: " + deckSeed);
    System.out.println();

    while (keepPlaying)
    {
        // Display game status

        System.out.println("Detailed Current Stats:");
        printStats();
    }
}
```

CardSim.java

```
System.out.println("Hands Left:" + gamesLeft);
temp = totalGames - gamesLeft + 1;
System.out.println("Now starting hand #" + temp);
System.out.println();

////////////////////////////////////
// Initialize game
////////////////////////////////////

// Stat status reset
playerOneUsedEmotion = false;
playerTwoUsedEmotion = false;
previousMoneyPlayerOne = playerOneMoney;
previousMoneyPlayerTwo = playerTwoMoney;

// Shuffle the deck
theDeck.shuffleDeck();

// Reset pot
pot = 0;

// Toggle starting player
playerOneGoesFirst = !playerOneGoesFirst;

// Deal player cards
playerOneHand[0] = theDeck.removeCard();
playerTwoHand[0] = theDeck.removeCard();
playerOneHand[1] = theDeck.removeCard();
playerTwoHand[1] = theDeck.removeCard();

// Show hands
System.out.println("Player 1 has:");
System.out.println("Suit: " + playerOneHand[0].getSuit() +
                    ", Value: " +
playerOneHand[0].getCharValue());
System.out.println("Suit: " + playerOneHand[1].getSuit() +
                    ", Value: " +
playerOneHand[1].getCharValue());
System.out.println("Player 2 has:");
System.out.println("Suit: " + playerTwoHand[0].getSuit() +
                    ", Value: " +
playerTwoHand[0].getCharValue());
System.out.println("Suit: " + playerTwoHand[1].getSuit() +
                    ", Value: " +
playerTwoHand[1].getCharValue());

////////////////////////////////////
// Start playing
////////////////////////////////////

// Find result of first betting cycle
winner = bettingCycle(playerOneGoesFirst, 0);

// Continue if no winner
if ( winner == -1 )
```

```

CardSim.java
{
    // Flop - deal first three community cards
    communityCards[0] = theDeck.removeCard();
    communityCards[1] = theDeck.removeCard();
    communityCards[2] = theDeck.removeCard();

    // Show community cards so far
    System.out.println("After the flop, the community cards are:");
    for(i = 0; i < 3; i++)
        System.out.println("Suit: " + communityCards[i].getSuit() +
+ communityCards[i].getCharValue());
                                ", Value: "

    // Find result of second betting cycle
    winner = bettingCycle(playerOneGoesFirst, 3);

    if ( winner == -1 ) // Continue if no winner
    {
        // Turn - deal next community card
        communityCards[3] = theDeck.removeCard();

        // Show community cards so far
        System.out.println("After the turn, the community cards are:");
        for(i = 0; i < 4; i++)
            System.out.println("Suit: " +
communityCards[i].getSuit() +
", Value: " + communityCards[i].getCharValue());

        // Find result of third betting cycle
        winner = bettingCycle(playerOneGoesFirst, 4);

        // Continue if no winner
        if ( winner == -1 )
        {
            // River - deal final community card
            communityCards[4] = theDeck.removeCard();

            // Show the community cards
            System.out.println("After the river, the community
cards are:");
            for(i = 0; i < 5; i++)
                System.out.println("Suit: " +
communityCards[i].getSuit() +
", Value: " + communityCards[i].getCharValue());

            // Find the final result
            winner = bettingCycle(playerOneGoesFirst, 5);
        }
    }

    // Determine winner
}

```

CardSim.java

////////////////////////////////////

```
System.out.println();  
System.out.println("Game result:");
```

```
switch (winner)  
{
```

```
    case 0: // Tie
```

```
        System.out.println("Draw");  
        splitPot = pot / 2;  
        playerOneMoney += splitPot;  
        playerTwoMoney += splitPot;
```

```
        p1T++;  
        p2T++;
```

```
        if ( playerOneUsedEmotion )  
        {
```

```
            p1Te++;  
            p1Me += playerOneMoney -
```

previousMoneyPlayerOne;

```
        }  
        else
```

```
        {  
            p1Ti++;  
            p1Mi += playerOneMoney -
```

previousMoneyPlayerOne;

```
        }
```

```
        if ( playerTwoUsedEmotion )  
        {
```

```
            p2Te++;  
            p2Me += playerTwoMoney -
```

previousMoneyPlayerTwo;

```
        }  
        else
```

```
        {  
            p2Ti++;  
            p2Mi += playerTwoMoney -
```

previousMoneyPlayerTwo;

```
        }
```

```
        break;
```

```
    case 1: // Player one wins
```

```
        System.out.println("Player 1 Wins");  
        playerOneMoney += pot;
```

```
        p1W++;  
        p2L++;
```

```
        if ( playerOneUsedEmotion )  
        {
```

```
            p1We++;  
            p1Me += playerOneMoney -
```

previousMoneyPlayerOne;

```

CardSim.java
}
else
{
    p1Wl++;
    p1Ml += playerOneMoney -
previousMoneyPlayerOne;
}

if ( playerTwoUsedEmotion )
{
    p2Le++;
    p2Me += playerTwoMoney -
previousMoneyPlayerTwo;
}
else
{
    p2Li++;
    p2Mi += playerTwoMoney -
previousMoneyPlayerTwo;
}
break;
case 2: // Player two wins
System.out.println("Player 2 Wins");
playerTwoMoney += pot;

p1L++;
p2W++;

if ( playerOneUsedEmotion )
{
    p1Le++;
    p1Me += playerOneMoney -
previousMoneyPlayerOne;
}
else
{
    p1Li++;
    p1Mi += playerOneMoney -
previousMoneyPlayerOne;
}

if ( playerTwoUsedEmotion )
{
    p2We++;
    p2Me += playerTwoMoney -
previousMoneyPlayerTwo;
}
else
{
    p2Wi++;
    p2Mi += playerTwoMoney -
previousMoneyPlayerTwo;
}
break;
}
}

```

CardSim.java

```
System.out.println();
System.out.println("--END OF HAND--");
System.out.println();

////////////////////////////////////
// Results
////////////////////////////////////

System.out.println("Player One now has $" + playerOneMoney + ".");
System.out.println("Player Two now has $" + playerTwoMoney + ".");
System.out.println();

////////////////////////////////////
// Keep Playing?
////////////////////////////////////

gamesLeft--;

if ( gamesLeft == 0 )
{
    System.out.println("End of the game...");
    if ( playerOneMoney > playerTwoMoney )
        System.out.println("Player One Wins!");
    else if ( playerOneMoney < playerTwoMoney )
        System.out.println("Player Two Wins!");
    else
        System.out.println("It's a tie!");
    keepPlaying = false;
}
else if ( playerOneMoney == 0 )
{
    System.out.println("Player One is out of money; Player Two wins!");
    keepPlaying = false;
}
else if ( playerTwoMoney == 0 )
{
    System.out.println("Player Two is out of money; Player One wins!");
    keepPlaying = false;
}

}

System.out.println("Thanks for playing!");
System.out.println("");
System.out.println("Final Detailed Stats:");
printStats();

}

// Carries out a cycle of betting depending on what phase of the game is being played
public int bettingCycle(boolean playerOneGoesFirst, int currentPhase)
{
    int i, playerOneDecision, playerTwoDecision;
```

CardSim.java

```
// Give agents data and let them make a decision
playerOneDecision =
    agent1.makeDecision(currentPhase, playerOneHand, communityCards,
playerOneMoney - playerTwoMoney);
playerTwoDecision =
    agent2.makeDecision(currentPhase, playerTwoHand, communityCards,
playerTwoMoney - playerOneMoney);

// Track type of decision made by player 1
if (playerOneDecision == 2)
{
    p1Ca++;
    playerOneDecision = 1;
    playerOneUsedEmotion = true;
    System.out.println("Player 1 is making an aggressive decision.");
}
else if (playerOneDecision == -2)
{
    p1Cc++;
    playerOneDecision = -1;
    playerOneUsedEmotion = true;
    System.out.println("Player 1 is making a conservative decision.");
}
else if (playerOneDecision == 10)
{
    p1Cu++;
    playerOneDecision = 0;
    System.out.println("Player 1 is making an unchanged emotional decision.");
}
else // logical decision made (-1, 0, or 1)
    p1Cl++;

// Track type of decision made by player 2
if (playerTwoDecision == 2)
{
    p2Ca++;
    playerTwoDecision = 1;
    playerTwoUsedEmotion = true;
    System.out.println("Player 2 is making an aggressive decision.");
}
else if (playerTwoDecision == -2)
{
    p2Cc++;
    playerTwoDecision = -1;
    playerTwoUsedEmotion = true;
    System.out.println("Player 2 is making a conservative decision.");
}
else if (playerTwoDecision == 10)
{
    p2Cu++;
    playerTwoDecision = 0;
    System.out.println("Player 2 is making an unchanged emotional decision.");
}
else // logical decision made (-1, 0, or 1)
    p2Cl++;
```

CardSim.java

```

// Show decision made by each player
System.out.println("Player 1's Status: " + playerOneDecision);
System.out.println("Player 2's Status: " + playerTwoDecision);

// Pre-flop cycle (use blind)
if ( currentPhase == 0 )
{
    if ( playerOneGoesFirst )
    {
        System.out.println("Player 1 bets the blind ($1).");
        playerOneMoney--;
        pot++;

        if ( (playerOneDecision == -1 && playerTwoDecision == -1) ||
              (playerOneDecision == 0 && playerTwoDecision == -1) ||
              (playerOneDecision == 1 && playerTwoDecision == -1) )
        {
            System.out.println("Player 2 folds.");
            return 1;
        }
        else if ( (playerOneDecision == -1 && playerTwoDecision == 0) ||
                  (playerOneDecision == 0 && playerTwoDecision
== 0 ) ||
                  (playerOneDecision == 1 && playerTwoDecision
== 0 ) )
        {
            System.out.println("Player 2 calls ($1).");
            playerTwoMoney--;
            pot++;
            return -1;
        }
        else if ( (playerOneDecision == -1 && playerTwoDecision == 1) ||
                  (playerOneDecision == 0 && playerTwoDecision
== 1 ) )
        {
            System.out.println("Player 2 raises ($2).");
            playerTwoMoney -= 2;
            pot += 2;
            System.out.println("Player 1 folds.");
            return 2;
        }
        else if ( playerOneDecision == 1 && playerTwoDecision == 1 )
        {
            System.out.println("Player 2 raises ($2).");
            playerTwoMoney -= 2;
            pot += 2;
            System.out.println("Player 1 calls ($2).");
            playerOneMoney--;
            pot ++;
            return -1;
        }
    }
    else // !playerOneGoesFirst

```

```

        CardSim.java
    {
        System.out.println("Player 2 bets the blind ($1).");
        playerTwoMoney--;
        pot++;

        if ( (playerTwoDecision == -1 && playerOneDecision == -1) ||
            (playerTwoDecision == 0 && playerOneDecision == -1) ||
            (playerTwoDecision == 1 && playerOneDecision == -1) )
        {
            System.out.println("Player 1 folds.");
            return 2;
        }
        else if ( (playerTwoDecision == -1 && playerOneDecision == 0) ||
            (playerTwoDecision == 0 && playerOneDecision
== 0 ) ||
            (playerTwoDecision == 1 && playerOneDecision
== 0 ) )
        {
            System.out.println("Player 1 calls ($1).");
            playerOneMoney--;
            pot++;
            return -1;
        }
        else if ( (playerTwoDecision == -1 && playerOneDecision == 1) ||
            (playerTwoDecision == 0 && playerOneDecision
== 1 ) )
        {
            System.out.println("Player 1 raises ($2).");
            playerOneMoney -= 2;
            pot += 2;
            System.out.println("Player 2 folds.");
            return 1;
        }
        else if ( playerTwoDecision == 1 && playerOneDecision == 1 )
        {
            System.out.println("Player 1 raises ($2).");
            playerOneMoney -= 2;
            pot += 2;
            System.out.println("Player 2 calls ($2).");
            playerTwoMoney--;
            pot++;
            return -1;
        }
    }

}

else if ( currentPhase == 5 ) // Post-river cycle (evaluate hands)
{
    if ( playerOneGoesFirst )
    {
        if ( (playerOneDecision == -1 && playerTwoDecision == -1) ||
            (playerOneDecision == -1 && playerTwoDecision == 0) ||
            (playerOneDecision == 0 && playerTwoDecision == -1) ||
            (playerOneDecision == 0 && playerTwoDecision == 0) )

```

CardSim.java

```
{
    System.out.println("Player 1 checks.");
    System.out.println("Player 2 checks.");
}
else if ( playerOneDecision == -1 && playerTwoDecision == 1 )
{
    System.out.println("Player 1 checks.");
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 folds.");
}
else if ( playerOneDecision == 0 && playerTwoDecision == 1 )
{
    System.out.println("Player 1 checks.");
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 calls ($1).");
    playerOneMoney--;
    pot++;
}
else if ( playerOneDecision == 1 && playerTwoDecision == -1 )
{
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 folds.");
}
else if ( playerOneDecision == 1 && playerTwoDecision == 0 )
{
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 calls ($1).");
    playerTwoMoney--;
    pot++;
}
else if ( playerOneDecision == 1 && playerTwoDecision == 1 )
{
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 raises ($2).");
    playerTwoMoney -= 2;
    pot += 2;
    System.out.println("Player 1 calls ($2).");
    playerOneMoney--;
    pot++;
}
}

else // !playerOneGoesFirst
{
    if ( (playerTwoDecision == -1 && playerOneDecision == -1) ||
```

```

CardSim.java
(playerTwoDecision == -1 && playerOneDecision == 0) ||
(playerTwoDecision == 0 && playerOneDecision == -1) ||
(playerTwoDecision == 0 && playerOneDecision == 0) )
{
    System.out.println("Player 2 checks.");
    System.out.println("Player 1 checks.");
}
else if ( playerTwoDecision == -1 && playerOneDecision == 1 )
{
    System.out.println("Player 2 checks.");
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 folds.");
}
else if ( playerTwoDecision == 0 && playerOneDecision == 1 )
{
    System.out.println("Player 2 checks.");
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 calls ($1).");
    playerTwoMoney--;
    pot++;
}
else if ( playerTwoDecision == 1 && playerOneDecision == -1 )
{
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 folds.");
}
else if ( playerTwoDecision == 1 && playerOneDecision == 0 )
{
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 calls ($1).");
    playerOneMoney--;
    pot++;
}
else if ( playerTwoDecision == 1 && playerOneDecision == 1 )
{
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 raises ($2).");
    playerOneMoney -= 2;
    pot += 2;
    System.out.println("Player 2 calls ($2).");
    playerTwoMoney--;
    pot++;
}
}
}

```

CardSim.java

```

HandResult playerOneResult = gameRules.getResult(playerOneHand[0],
playerOneHand[1],
communityCards[0], communityCards[1],
communityCards[2],
communityCards[3],
communityCards[4]);
HandResult playerTwoResult = gameRules.getResult(playerTwoHand[0],
playerTwoHand[1],
communityCards[0], communityCards[1],
communityCards[2],
communityCards[3],
communityCards[4]);

System.out.println("Player 1 has " + playerOneResult.getResultString() + " (" +
playerOneResult.getResult() + ")");
System.out.println("Player 2 has " + playerTwoResult.getResultString() + " (" +
playerTwoResult.getResult() + ")");

if (playerOneResult.getResult() > playerTwoResult.getResult())
    return 1;
else if (playerTwoResult.getResult() > playerOneResult.getResult())
    return 2;
else // Go to tiebreaker
{
    for(i = 0; i < 5; i++)
    {
        System.out.println("Tiebreaker) P1: " +
playerOneResult.getTiebreakers(i) + "; P2: " +
playerTwoResult.getTiebreakers(i));
        if ( playerOneResult.getTiebreakers(i) >
playerTwoResult.getTiebreakers(i) )
            return 1;
        else if ( playerOneResult.getTiebreakers(i) <
playerTwoResult.getTiebreakers(i) )
            return 2;
        else if ( i == 4)
            return 0;
    }
}

else // Post-flop (3) and post-turn (4) cycles
{
    if ( playerOneGoesFirst )
    {
        if ( (playerOneDecision == -1 && playerTwoDecision == -1) ||
(playerOneDecision == -1 && playerTwoDecision == 0) ||
(playerOneDecision == 0 && playerTwoDecision == -1) ||
(playerOneDecision == 0 && playerTwoDecision == 0) )
        {
            System.out.println("Player 1 checks.");
            System.out.println("Player 2 checks.");
            return -1;
        }
    }
    else if ( playerOneDecision == -1 && playerTwoDecision == 1 )

```

```

CardSim.java
{
    System.out.println("Player 1 checks.");
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 folds.");
    return 2;
}
else if ( playerOneDecision == 0 && playerTwoDecision == 1 )
{
    System.out.println("Player 1 checks.");
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 calls ($1).");
    playerOneMoney--;
    pot++;
    return -1;
}
else if ( playerOneDecision == 1 && playerTwoDecision == -1 )
{
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 folds.");
    return 1;
}
else if ( playerOneDecision == 1 && playerTwoDecision == 0 )
{
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 calls ($1).");
    playerTwoMoney--;
    pot++;
    return -1;
}
else if ( playerOneDecision == 1 && playerTwoDecision == 1 )
{
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 raises ($2).");
    playerTwoMoney -= 2;
    pot += 2;
    System.out.println("Player 1 calls ($2).");
    playerOneMoney--;
    pot++;
    return -1;
}
}
else // !playerOneGoesFirst
{
    if ( (playerTwoDecision == -1 && playerOneDecision == -1) ||

```

```

CardSim.java
(playerTwoDecision == -1 && playerOneDecision == 0) ||
(playerTwoDecision == 0 && playerOneDecision == -1) ||
(playerTwoDecision == 0 && playerOneDecision == 0) )
{
    System.out.println("Player 2 checks.");
    System.out.println("Player 1 checks.");
    return -1;
}
else if ( playerTwoDecision == -1 && playerOneDecision == 1 )
{
    System.out.println("Player 2 checks.");
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 folds.");
    return 1;
}
else if ( playerTwoDecision == 0 && playerOneDecision == 1 )
{
    System.out.println("Player 2 checks.");
    System.out.println("Player 1 raises ($1).");
    playerOneMoney--;
    pot++;
    System.out.println("Player 2 calls ($1).");
    playerTwoMoney--;
    pot++;
    return -1;
}
else if ( playerTwoDecision == 1 && playerOneDecision == -1 )
{
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 folds.");
    return 2;
}
else if ( playerTwoDecision == 1 && playerOneDecision == 0 )
{
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 calls ($1).");
    playerOneMoney--;
    pot++;
    return -1;
}
else if ( playerTwoDecision == 1 && playerOneDecision == 1 )
{
    System.out.println("Player 2 raises ($1).");
    playerTwoMoney--;
    pot++;
    System.out.println("Player 1 raises ($2).");
    playerOneMoney -= 2;
    pot += 2;
    System.out.println("Player 2 calls ($2).");
}

```

```

        CardSim.java
        playerTwoMoney--;
        pot++;
        return -1;
    }
}

return -1;
}

public void printStats()
{
    System.out.println();
    System.out.println("Player 1's Record: " + p1W + "-" + p1L + "-" + p1T);
    System.out.println("Player 1's Record From Purely Logical Play: " +
        p1WI + "-" + p1LI + "-" + p1TI);
    System.out.println("Player 1's Money Won From Purely Logical Play: $" + p1MI);
    System.out.println("Player 1's Record From Mixed Emotional/Logical Play: " +
        p1We + "-" + p1Le + "-" + p1Te);
    System.out.println("Player 1's Money Won From Mixed Emotional/Logical Play: $" + p1Me);
    System.out.println("Player 1's Purely Logical Choices: " + p1CI);
    System.out.println("Player 1's Emotional Aggressive Choices: " + p1Ca);
    System.out.println("Player 1's Emotional Unchanged Choices: " + p1Cu);
    System.out.println("Player 1's Emotional Conservative Choices: " + p1Cc);
    System.out.println();
    System.out.println("Player 2's Record: " +
        p2W + "-" + p2L + "-" + p2T);
    System.out.println("Player 2's Record From Purely Logical Play: " +
        p2WI + "-" + p2LI + "-" + p2TI);
    System.out.println("Player 2's Money Won From Purely Logical Play: $" + p2MI);
    System.out.println("Player 2's Record From Mixed Emotional/Logical Play: " +
        p2We + "-" + p2Le + "-" + p2Te);
    System.out.println("Player 2's Money Won From Mixed Emotional/Logical Play: $" + p2Me);
    System.out.println("Player 2's Purely Logical Choices: " + p2CI);
    System.out.println("Player 2's Emotional Aggressive Choices: " + p2Ca);
    System.out.println("Player 2's Emotional Unchanged Choices: " + p2Cu);
    System.out.println("Player 2's Emotional Conservative Choices: " + p2Cc);
    System.out.println();
}

// Create instance of application
public static void main(String args[]) throws IOException
{
    // Record output
    SaveOutput.start("results.txt");

    // Begin application
    CardSim CardSimLauncher = new CardSim();

    // Stop recording output
    SaveOutput.stop();

    // Exit system
    System.exit(0);
}

```

CardSim.java

```
} // End class CardSim
```

CardDeck.java

```
/**
 * CardDeck.java
 *
 * The CardDeck class simulates a standard deck of 52 playing cards using a Vector. The
 * deck always remains in order by card ID (as explained in the Card class). To simulate
 * the drawing of cards from the deck, a card is randomly picked and removed from the
 * ordered deck. The random number generator can be seeded.
 *
 * It has two constructors:
 * - default constructor with no seed specified
 * - constructor with seed specified
 *
 * It has methods to:
 * - reseed the deck
 * - shuffle the deck
 * - randomly remove cards from the deck
 * - get a specific card from the deck
 * - get the current number of cards left in the deck
 */
```

```
import java.util.*;
```

```
public class CardDeck
{
    private Vector deck = new Vector(52);
    private Random randGen;
    private long currentSeed;

    // Default constructor
    public CardDeck()
    {
        currentSeed = 0;
        randGen = new Random();
        shuffleDeck();
    }

    // Seeding constructor
    public CardDeck(long seed)
    {
        currentSeed = seed;
        randGen = new Random(seed);
        shuffleDeck();
    }

    // Re-initialize deck
    public void shuffleDeck()
    {
        randGen.setSeed(currentSeed);
        currentSeed = currentSeed + 1;

        // Erase the vector
        deck.clear();

        char currentSuit;
```

CardDeck.java

```
for(int i = 1; i <= 4; i++)
{
    switch(i)
    {
        case 1:
            currentSuit = 'H';
            break;
        case 2:
            currentSuit = 'D';
            break;
        case 3:
            currentSuit = 'C';
            break;
        case 4:
            currentSuit = 'S';
            break;
        default:
            currentSuit = 'X';
            System.out.println("Error! Invalid suit type during reshuffling.");
            break;
    }

    for(int currentValue = 2; currentValue <= 14; currentValue++)
        deck.add(new Card(currentSuit, currentValue));
}

return;
}

// Randomly remove card from the Vector
public Card removeCard()
{
    Card thisCard;
    int deckSize = deck.size();
    int cardIndex;

    if (deckSize < 1)
    {
        System.out.println("Error! Attempted to remove card from empty deck.");
        System.exit(0);
    }

    cardIndex = randGen.nextInt(deckSize);
    thisCard = (Card) deck.get(cardIndex);
    deck.removeElementAt(cardIndex);

    return thisCard;
}

// Get specific card from the deck (parameter is displacement from front of deck)
public Card getCard(int displacement)
{
    Card thisCard;
    int deckSize = deck.size();
```

CardDeck.java

```
    if (deckSize < 1)
    {
        System.out.println("Error! Attempted to get card from empty deck.");
        System.exit(0);
    }
    else if (displacement >= deckSize)
    {
        System.out.println("Error! Attempted to get card from deck that doesn't exist.");
        System.exit(0);
    }

    thisCard = (Card) deck.get(displacement);

    return thisCard;
}

// Deck size accessor function
public int getDeckSize() { return deck.size(); }

} // End class CardDeck
```

Card.java

```
/**
 * Card.java
 *
 * The Card class represents a playing card from a standard deck of 52
 * cards. It contains a suit represented by a character and a value
 * represented by an integer. More specifically, they are:
 *
 * Suits:
 * H: Heart
 * D: Diamond
 * C: Club
 * S: Spade
 *
 * Values:
 * 2 - 10: Face Value
 * 11: Jack
 * 12: Queen
 * 13: King
 * 14: Ace
 *
 * Each card has an implied ID from 0 - 51. In order (from 2 to Ace)
 * by suit, the ID's are:
 *
 * Hearts: 0 - 12
 * Diamonds: 12 - 25
 * Clubs: 26 - 38
 * Spades: 39 - 51
 *
 * The class has methods to:
 * - obtain the suit
 * - obtain the value as a character
 * - obtain the value as an integer
 * - obtain the card's ID
 *
 * Note that the default constructor should never be called. Calling it
 * causes an error that suspends program execution. Always call the
 * constructor with suit and value arguments.
 */

public class Card
{
    private char suit;
    private int value;

    // Default constructor (SHOULD NOT BE CALLED)
    public Card()
    {
        System.out.println("Error! Cannot create empty card!");
        System.exit(0);
    }

    // The only valid constructor
    public Card(char newSuit, int newValue)
    {
```

Card.java

```
        suit = newSuit;
        value = newValue;
    }

    // Suit accessor method
    public char getSuit() { return suit; }

    // Value as an integer accessor method
    public int getIntValue() { return value; }

    // Integer ID accessor method
    public int getDeckID()
    {
        if ( suit == 'H' )
            return value - 2;
        else if ( suit == 'D' )
            return (value + 13) - 2;
        else if ( suit == 'C' )
            return (value + 26) - 2;
        else // suit == 'S'
            return (value + 39) - 2;
    }

    // Value as a character accessor method
    public char getCharValue()
    {
        char returnChar = ' ';

        switch(value)
        {
            case 10:
                returnChar = 'T';
                break;
            case 11:
                returnChar = 'J';
                break;
            case 12:
                returnChar = 'Q';
                break;
            case 13:
                returnChar = 'K';
                break;
            case 14:
                returnChar = 'A';
                break;
            default:
                int temp = 48 + value;
                returnChar = (char)temp;
                break;
        }

        return returnChar;
    }
} // End class Card
```

Agent.java

```
/**
 * Agent.java
 *
 * The Agent class handles the decision-making process for each player. It
 * uses a table to determine the odds of winning before the flop, a sampling
 * of results to determine the odds of winning after the flop but before the
 * turn, and a brute force calculation to determine the odds of winning after
 * the turn. Its most important method is the makeDecision method, which actually
 * determines how the agent will react to a given scenerio.
 */

import java.util.*;

public abstract class Agent
{
    private double goodHand, badHand;
    private int preFlop[][][] = new int[2][13][13], moneyLimit;
    private Random randGen = new Random();

    // Constructor
    public Agent(int moneyDiffLimit, int gHand, int bHand)
    {
        moneyLimit = moneyDiffLimit;
        goodHand = gHand;
        badHand = bHand;

        /**
         * Initialize pre-flop odds:
         *
         * preFlop[a][b][c]
         * a: 0 for on-suit, 1 for off-suit
         * b: value of first card, starting with 0 for 2 and 12 for Ace
         * c: value of first card, starting with 0 for 2 and 12 for Ace
         *
         * A -1 as the value of the odds indicates an impossible combination,
         * specifically, two cards of the same value on-suit.
         */

        preFlop[0][0][0] = -1;
        preFlop[0][0][1] = 361;
        preFlop[0][0][2] = 368;
        preFlop[0][0][3] = 378;
        preFlop[0][0][4] = 377;
        preFlop[0][0][5] = 382;
        preFlop[0][0][6] = 403;
        preFlop[0][0][7] = 424;
        preFlop[0][0][8] = 448;
        preFlop[0][0][9] = 474;
        preFlop[0][0][10] = 501;
        preFlop[0][0][11] = 532;
        preFlop[0][0][12] = 573;

        preFlop[1][0][0] = 503;
    }
}
```

Agent.java

```
preFlop[1][0][1] = 323;  
preFlop[1][0][2] = 332;  
preFlop[1][0][3] = 343;  
preFlop[1][0][4] = 341;  
preFlop[1][0][5] = 346;  
preFlop[1][0][6] = 368;  
preFlop[1][0][7] = 391;  
preFlop[1][0][8] = 416;  
preFlop[1][0][9] = 443;  
preFlop[1][0][10] = 472;  
preFlop[1][0][11] = 505;  
preFlop[1][0][12] = 549;
```

```
preFlop[0][1][0] = 361;  
preFlop[0][1][1] = -1;  
preFlop[0][1][2] = 387;  
preFlop[0][1][3] = 397;  
preFlop[0][1][4] = 396;  
preFlop[0][1][5] = 400;  
preFlop[0][1][6] = 410;  
preFlop[0][1][7] = 433;  
preFlop[0][1][8] = 457;  
preFlop[0][1][9] = 482;  
preFlop[0][1][10] = 511;  
preFlop[0][1][11] = 541;  
preFlop[0][1][12] = 582;
```

```
preFlop[1][1][0] = 323;  
preFlop[1][1][1] = 537;  
preFlop[1][1][2] = 352;  
preFlop[1][1][3] = 363;  
preFlop[1][1][4] = 361;  
preFlop[1][1][5] = 366;  
preFlop[1][1][6] = 375;  
preFlop[1][1][7] = 400;  
preFlop[1][1][8] = 426;  
preFlop[1][1][9] = 453;  
preFlop[1][1][10] = 483;  
preFlop[1][1][11] = 514;  
preFlop[1][1][12] = 559;
```

```
preFlop[0][2][0] = 368;  
preFlop[0][2][1] = 387;  
preFlop[0][2][2] = -1;  
preFlop[0][2][3] = 415;  
preFlop[0][2][4] = 414;  
preFlop[0][2][5] = 419;  
preFlop[0][2][6] = 427;  
preFlop[0][2][7] = 440;  
preFlop[0][2][8] = 466;  
preFlop[0][2][9] = 490;  
preFlop[0][2][10] = 518;  
preFlop[0][2][11] = 549;  
preFlop[0][2][12] = 590;
```

Agent.java

```
preFlop[1][2][0] = 332;  
preFlop[1][2][1] = 352;  
preFlop[1][2][2] = 569;  
preFlop[1][2][3] = 382;  
preFlop[1][2][4] = 380;  
preFlop[1][2][5] = 386;  
preFlop[1][2][6] = 395;  
preFlop[1][2][7] = 408;  
preFlop[1][2][8] = 435;  
preFlop[1][2][9] = 462;  
preFlop[1][2][10] = 491;  
preFlop[1][2][11] = 524;  
preFlop[1][2][12] = 567;
```

```
preFlop[0][3][0] = 378;  
preFlop[0][3][1] = 397;  
preFlop[0][3][2] = 415;  
preFlop[0][3][3] = -1;  
preFlop[0][3][4] = 432;  
preFlop[0][3][5] = 437;  
preFlop[0][3][6] = 446;  
preFlop[0][3][7] = 458;  
preFlop[0][3][8] = 472;  
preFlop[0][3][9] = 500;  
preFlop[0][3][10] = 529;  
preFlop[0][3][11] = 558;  
preFlop[0][3][12] = 599;
```

```
preFlop[1][3][0] = 344;  
preFlop[1][3][1] = 363;  
preFlop[1][3][2] = 381;  
preFlop[1][3][3] = 603;  
preFlop[1][3][4] = 400;  
preFlop[1][3][5] = 406;  
preFlop[1][3][6] = 414;  
preFlop[1][3][7] = 427;  
preFlop[1][3][8] = 443;  
preFlop[1][3][9] = 472;  
preFlop[1][3][10] = 501;  
preFlop[1][3][11] = 532;  
preFlop[1][3][12] = 576;
```

```
preFlop[0][4][0] = 377;  
preFlop[0][4][1] = 396;  
preFlop[0][4][2] = 414;  
preFlop[0][4][3] = 432;  
preFlop[0][4][4] = -1;  
preFlop[0][4][5] = 453;  
preFlop[0][4][6] = 463;  
preFlop[0][4][7] = 474;  
preFlop[0][4][8] = 490;  
preFlop[0][4][9] = 506;  
preFlop[0][4][10] = 535;  
preFlop[0][4][11] = 566;  
preFlop[0][4][12] = 599;
```

Agent.java

```
preFlop[1][4][0] = 341;  
preFlop[1][4][1] = 361;  
preFlop[1][4][2] = 380;  
preFlop[1][4][3] = 399;  
preFlop[1][4][4] = 634;  
preFlop[1][4][5] = 423;  
preFlop[1][4][6] = 432;  
preFlop[1][4][7] = 445;  
preFlop[1][4][8] = 461;  
preFlop[1][4][9] = 479;  
preFlop[1][4][10] = 510;  
preFlop[1][4][11] = 543;  
preFlop[1][4][12] = 577;
```

```
preFlop[0][5][0] = 382;  
preFlop[0][5][1] = 400;  
preFlop[0][5][2] = 419;  
preFlop[0][5][3] = 437;  
preFlop[0][5][4] = 453;  
preFlop[0][5][5] = -1;  
preFlop[0][5][6] = 480;  
preFlop[0][5][7] = 492;  
preFlop[0][5][8] = 506;  
preFlop[0][5][9] = 524;  
preFlop[0][5][10] = 542;  
preFlop[0][5][11] = 576;  
preFlop[0][5][12] = 610;
```

```
preFlop[1][5][0] = 346;  
preFlop[1][5][1] = 366;  
preFlop[1][5][2] = 385;  
preFlop[1][5][3] = 405;  
preFlop[1][5][4] = 424;  
preFlop[1][5][5] = 663;  
preFlop[1][5][6] = 450;  
preFlop[1][5][7] = 462;  
preFlop[1][5][8] = 479;  
preFlop[1][5][9] = 497;  
preFlop[1][5][10] = 519;  
preFlop[1][5][11] = 552;  
preFlop[1][5][12] = 588;
```

```
preFlop[0][6][0] = 368;  
preFlop[0][6][1] = 410;  
preFlop[0][6][2] = 427;  
preFlop[0][6][3] = 446;  
preFlop[0][6][4] = 463;  
preFlop[0][6][5] = 480;  
preFlop[0][6][6] = -1;  
preFlop[0][6][7] = 508;  
preFlop[0][6][8] = 524;  
preFlop[0][6][9] = 539;  
preFlop[0][6][10] = 560;  
preFlop[0][6][11] = 583;
```

Agent.java

```
preFlop[0][6][12] = 620;
```

```
preFlop[1][6][0] = 368;  
preFlop[1][6][1] = 376;  
preFlop[1][6][2] = 395;  
preFlop[1][6][3] = 415;  
preFlop[1][6][4] = 433;  
preFlop[1][6][5] = 450;  
preFlop[1][6][6] = 692;  
preFlop[1][6][7] = 481;  
preFlop[1][6][8] = 497;  
preFlop[1][6][9] = 515;  
preFlop[1][6][10] = 536;  
preFlop[1][6][11] = 560;  
preFlop[1][6][12] = 599;
```

```
preFlop[0][7][0] = 424;  
preFlop[0][7][1] = 433;  
preFlop[0][7][2] = 440;  
preFlop[0][7][3] = 458;  
preFlop[0][7][4] = 474;  
preFlop[0][7][5] = 492;  
preFlop[0][7][6] = 508;  
preFlop[0][7][7] = -1;  
preFlop[0][7][8] = 541;  
preFlop[0][7][9] = 557;  
preFlop[0][7][10] = 576;  
preFlop[0][7][11] = 599;  
preFlop[0][7][12] = 629;
```

```
preFlop[1][7][0] = 391;  
preFlop[1][7][1] = 401;  
preFlop[1][7][2] = 406;  
preFlop[1][7][3] = 427;  
preFlop[1][7][4] = 445;  
preFlop[1][7][5] = 463;  
preFlop[1][7][6] = 481;  
preFlop[1][7][7] = 720;  
preFlop[1][7][8] = 515;  
preFlop[1][7][9] = 532;  
preFlop[1][7][10] = 554;  
preFlop[1][7][11] = 579;  
preFlop[1][7][12] = 607;
```

```
preFlop[0][8][0] = 448;  
preFlop[0][8][1] = 457;  
preFlop[0][8][2] = 466;  
preFlop[0][8][3] = 472;  
preFlop[0][8][4] = 490;  
preFlop[0][8][5] = 506;  
preFlop[0][8][6] = 524;  
preFlop[0][8][7] = 541;  
preFlop[0][8][8] = -1;  
preFlop[0][8][9] = 575;  
preFlop[0][8][10] = 594;
```

Agent.java

```
preFlop[0][8][11] = 617;  
preFlop[0][8][12] = 646;
```

```
preFlop[1][8][0] = 417;  
preFlop[1][8][1] = 426;  
preFlop[1][8][2] = 436;  
preFlop[1][8][3] = 443;  
preFlop[1][8][4] = 461;  
preFlop[1][8][5] = 479;  
preFlop[1][8][6] = 497;  
preFlop[1][8][7] = 515;  
preFlop[1][8][8] = 750;  
preFlop[1][8][9] = 553;  
preFlop[1][8][10] = 573;  
preFlop[1][8][11] = 597;  
preFlop[1][8][12] = 627;
```

```
preFlop[0][9][0] = 474;  
preFlop[0][9][1] = 482;  
preFlop[0][9][2] = 490;  
preFlop[0][9][3] = 500;  
preFlop[0][9][4] = 506;  
preFlop[0][9][5] = 524;  
preFlop[0][9][6] = 539;  
preFlop[0][9][7] = 557;  
preFlop[0][9][8] = 575;  
preFlop[0][9][9] = -1;  
preFlop[0][9][10] = 602;  
preFlop[0][9][11] = 626;  
preFlop[0][9][12] = 654;
```

```
preFlop[1][9][0] = 444;  
preFlop[1][9][1] = 453;  
preFlop[1][9][2] = 462;  
preFlop[1][9][3] = 472;  
preFlop[1][9][4] = 478;  
preFlop[1][9][5] = 497;  
preFlop[1][9][6] = 515;  
preFlop[1][9][7] = 532;  
preFlop[1][9][8] = 553;  
preFlop[1][9][9] = 774;  
preFlop[1][9][10] = 582;  
preFlop[1][9][11] = 606;  
preFlop[1][9][12] = 635;
```

```
preFlop[0][10][0] = 501;  
preFlop[0][10][1] = 511;  
preFlop[0][10][2] = 518;  
preFlop[0][10][3] = 529;  
preFlop[0][10][4] = 535;  
preFlop[0][10][5] = 542;  
preFlop[0][10][6] = 560;  
preFlop[0][10][7] = 576;  
preFlop[0][10][8] = 594;  
preFlop[0][10][9] = 602;
```

Agent.java

```
preFlop[0][10][10] = -1;  
preFlop[0][10][11] = 634;  
preFlop[0][10][12] = 662;
```

```
preFlop[1][10][0] = 473;  
preFlop[1][10][1] = 482;  
preFlop[1][10][2] = 491;  
preFlop[1][10][3] = 502;  
preFlop[1][10][4] = 510;  
preFlop[1][10][5] = 518;  
preFlop[1][10][6] = 536;  
preFlop[1][10][7] = 553;  
preFlop[1][10][8] = 573;  
preFlop[1][10][9] = 582;  
preFlop[1][10][10] = 799;  
preFlop[1][10][11] = 614;  
preFlop[1][10][12] = 646;
```

```
preFlop[0][11][0] = 532;  
preFlop[0][11][1] = 541;  
preFlop[0][11][2] = 549;  
preFlop[0][11][3] = 558;  
preFlop[0][11][4] = 566;  
preFlop[0][11][5] = 576;  
preFlop[0][11][6] = 583;  
preFlop[0][11][7] = 599;  
preFlop[0][11][8] = 617;  
preFlop[0][11][9] = 626;  
preFlop[0][11][10] = 634;  
preFlop[0][11][11] = -1;  
preFlop[0][11][12] = 670;
```

```
preFlop[1][11][0] = 505;  
preFlop[1][11][1] = 514;  
preFlop[1][11][2] = 523;  
preFlop[1][11][3] = 533;  
preFlop[1][11][4] = 542;  
preFlop[1][11][5] = 552;  
preFlop[1][11][6] = 561;  
preFlop[1][11][7] = 578;  
preFlop[1][11][8] = 597;  
preFlop[1][11][9] = 606;  
preFlop[1][11][10] = 613;  
preFlop[1][11][11] = 824;  
preFlop[1][11][12] = 653;
```

```
preFlop[0][12][0] = 573;  
preFlop[0][12][1] = 582;  
preFlop[0][12][2] = 590;  
preFlop[0][12][3] = 599;  
preFlop[0][12][4] = 599;  
preFlop[0][12][5] = 610;  
preFlop[0][12][6] = 620;  
preFlop[0][12][7] = 629;  
preFlop[0][12][8] = 646;
```

Agent.java

```

preFlop[0][12][9] = 654;
preFlop[0][12][10] = 662;
preFlop[0][12][11] = 670;
preFlop[0][12][12] = -1;

preFlop[1][12][0] = 549;
preFlop[1][12][1] = 558;
preFlop[1][12][2] = 567;
preFlop[1][12][3] = 576;
preFlop[1][12][4] = 576;
preFlop[1][12][5] = 588;
preFlop[1][12][6] = 599;
preFlop[1][12][7] = 607;
preFlop[1][12][8] = 627;
preFlop[1][12][9] = 636;
preFlop[1][12][10] = 644;
preFlop[1][12][11] = 653;
preFlop[1][12][12] = 852;
}

/**
 * The makeDecision method decides whether how an agent will react to a given scenerio by returning
 * one of the following values:
 *
 * -2: Agent decided using emotion that it wants to play conservatively
 * -1: Agent decided using only logic that it wants to play conservatively
 * 0: Agent decided using only logic that it wants to play neutrally
 * 1: Agent decided using only logic that it wants to play aggressively
 * 2: Agent decided using emotion that it wants to play aggressively
 * 10: Agent decided using emotion that it wants to play neutrally
 */
public int makeDecision(int currentPhase, Card myCards[], Card comCards[], int moneyDifference )
{
    int positiveChance = 0, i = myCards[0].getDeckID(), j = myCards[1].getDeckID(),
        positiveResults = 0, negativeResults = 0, k, l, n, p, q, sampleSize;
    Card oppCards[] = new Card[2];
    HandResult myResult, oppResult;
    CardDeck fullDeck = new CardDeck();
    PokerRules gameRules = new PokerRules();

    switch (currentPhase)
    {
        // Pre-flop, use look-up table
        case 0:
            int firstCard = myCards[0].getIntValue() - 2, secondCard =
myCards[1].getIntValue() - 2;
            if ( myCards[0].getSuit() == myCards[1].getSuit() ) // On-suit
                positiveChance = preFlop[0][firstCard][secondCard];
            else // Off-suit
                positiveChance = preFlop[1][firstCard][secondCard];
            break;
        case 3: // Post-flop, calculate odds of a small sample of hands
            for(sampleSize = 0; sampleSize < 1000000; sampleSize++)
            {

```

```

Agent.java
k = randGen.nextInt(52);
l = randGen.nextInt(52);
p = randGen.nextInt(52);
q = randGen.nextInt(52);
// Check for duplicate cards in players' hands
if ( !(p == q || p == i || p == j || p == k || p == l ||
      q == i || q == j || q == k || q == l) )
{
    oppCards[0] = fullDeck.getCard(k);
    oppCards[1] = fullDeck.getCard(l);
    comCards[3] = fullDeck.getCard(p);
    comCards[4] = fullDeck.getCard(q);

    myResult = gameRules.getResult(myCards[0], myCards[1],
    comCards[0], comCards[1], comCards[2],
comCards[3], comCards[4]);

    oppResult = gameRules.getResult(oppCards[0], oppCards[1],
    comCards[0], comCards[1], comCards[2],
comCards[3], comCards[4]);

    if (myResult.getResult() > oppResult.getResult())
        positiveResults ++;
    else if (oppResult.getResult() > myResult.getResult())
        negativeResults ++;
    else // Go to tiebreaker
    {
        for(n = 0; n < 5; n++)
        {
            if ( myResult.getTiebreakers(n) >
oppResult.getTiebreakers(n) )
            {
                positiveResults ++;
                n = 5;
            }
            else if ( myResult.getTiebreakers(n) <
oppResult.getTiebreakers(n) )
            {
                negativeResults ++;
                n = 5;
            }
            else if ( n == 4)
                positiveResults ++;
        }
    }
}

if ( (positiveResults + negativeResults) > 0 )
    positiveChance = (int)( (float)((float)positiveResults /
(float)(positiveResults + negativeResults)) * 1000);
else
    positiveChance = -10;
break;
//
case 4: // Post-turn, calculate possibility of winning by brute force

```

```

Agent.java
// Iterate through opponent's possible hands
for(k = 0; k < 52; k++)
for(l = 0; l < 52; l++)
// Check for duplicate cards in the opponent's hand before continuing
if ( !(k == l || i == k || i == l || j == k || j == l) )
for(p = 0; p < 52; p++)
// Check for duplicate cards in players' hands
if ( !(p == i || p == j || p == k || p == l) )
{
    oppCards[0] = fullDeck.getCard(k);
    oppCards[1] = fullDeck.getCard(l);
    comCards[4] = fullDeck.getCard(p);

    myResult = gameRules.getResult(myCards[0], myCards[1],
        comCards[0], comCards[1], comCards[2],
comCards[3], comCards[4]);

    oppResult = gameRules.getResult(oppCards[0], oppCards[1],
        comCards[0], comCards[1], comCards[2],
comCards[3], comCards[4]);

    if (myResult.getResult() > oppResult.getResult())
        positiveResults ++;
    else if (oppResult.getResult() > myResult.getResult())
        negativeResults ++;
    else // Go to tiebreaker
    {
        for(n = 0; n < 5; n++)
        {
            if ( myResult.getTiebreakers(n) >
oppResult.getTiebreakers(n) )
            {
                positiveResults ++;
                n = 5;
            }
            else if ( myResult.getTiebreakers(n) <
oppResult.getTiebreakers(n) )
            {
                negativeResults ++;
                n = 5;
            }
            else if ( n == 4)
                positiveResults ++;
        }
    }
}

if ( (positiveResults + negativeResults) > 0 )
    positiveChance = (int)( (float)((float)positiveResults /
(float)(positiveResults + negativeResults)) * 1000);
else
    positiveChance = -10;
break;
// Post-river, calculate possiblity of winning by brute force
case 5:
    // Iterate through opponent's possible hands

```

```

Agent.java
for(k = 0; k < 52; k++)
for(l = 0; l < 52; l++)
// Check for duplicate cards in the opponent's hand before continuing
if ( !(k == l || i == k || i == l || j == k || j == l) )
{
    oppCards[0] = fullDeck.getCard(k);
    oppCards[1] = fullDeck.getCard(l);

    myResult = gameRules.getResult(myCards[0], myCards[1],
comCards[3], comCards[4]);
    oppResult = gameRules.getResult(oppCards[0], oppCards[1],
comCards[3], comCards[4]);

    if (myResult.getResult() > oppResult.getResult())
        positiveResults ++;
    else if (oppResult.getResult() > myResult.getResult())
        negativeResults ++;
    else // Go to tiebreaker
    {
        for(n = 0; n < 5; n++)
        {
            if ( myResult.getTiebreakers(n) >
oppResult.getTiebreakers(n) )
            {
                positiveResults ++;
                n = 5;
            }
            else if ( myResult.getTiebreakers(n) <
oppResult.getTiebreakers(n) )
            {
                negativeResults ++;
                n = 5;
            }
            else if ( n == 4)
                positiveResults ++;
        }
    }

    if ( (positiveResults + negativeResults) > 0 )
        positiveChance = (int)( (float)((float)positiveResults /
(float)(positiveResults + negativeResults)) * 1000);
    else
        positiveChance = -10;
    break;
    default:
        positiveChance = 0;
}

// Display chance of winning the hand
System.out.println("Positive Chance: " + positiveChance);

if ( positiveChance > goodHand ) // Good Hand - Raise

```

Agent.java

```
        return 1;
    else if ( positiveChance < badHand ) // Bad Hand - Fold
        return -1;
    // Borderline Hand - Stay or Call? - Depends on Agent
    return borderLineDecision(moneyDifference, moneyLimit);
}

// Method to handle borderline decisions; depends on what type of agent
public abstract int borderLineDecision(int moneyDifference, int limit);
}
```