Introduction:

This game is a cyclic, persistent program implementing openGL graphics and simplistic AI algorithms. The program was constructed using a Model-View-Controller design pattern while working under a Scrum design system. The MVC pattern segregates classes into the three subsets that it is named after. The project utilizes the Slick2d openGL wrapper library to render graphics, along with some minor classes in the LWJGL wrapper library to help render images and certain texts. Using Slick was an interesting experience given the time crunch. Slick allows for total graphics control on a basic level, achieved by using an openGL imbued version of Java's native graphics and geometry libraries. Doing this gave us move design control compared to Swing, but disallowed the use of Swing's convenient components and premade graphical structures.

User Stories:

- ✓ The user will want to see a table with cards and three other players.
- ✓ The user will want the table to have perspective.
- ✓ The user will want to see basic animation in the computer players.
- ❑ The user will want witty dialog when they interact with the computer players.
- ✓ The user will want the ability to check or bet, and call or raise.
- ✓ The user will want the game to function like a Texas hold 'em game of poker.
- ✓ The user will want to see the river and their hand in a separate area of the screen.
- ❑ The user will want the ability to cheat or gain perks after winning or losing
- ❑ The user will want to change the difficulty of the opponent
- ✓ The user will want to be able to close the game whenever they want
- ✓ The user will want to see their opponents cards when the game is over
- ❑ The user will want to be able to use a help window

Our system accomplishes many of the important user stories. We aimed first to have a clean and comprehensive poker experience for the user. Being able to see a table with three other players added to the full GUI experience. Giving the table and cards perspective, along with

glowing animations for the players helped give an intuitive design to the GUI. Also, showing the cards in the river and in the hands in a separate section of the game window allowed for us to give a clean experience in the game while still maintaining a sense of realism. This was all accomplished through many hours of photoshop and logic in the controller to draw images to the game window when appropriate. Often, switching between different photos (e.g. the 'glowing' players) is what gave the game its unique experience. Allowing the player to see their opponents cards when the game is over helped the player because they did not have to trust the computer that the result was true, even though the computer is never wrong. This peace of mind gave the user a much better experience.

The user, of course, also wanted the game to function like a typical game of Texas Hold 'em Poker. That is, they wanted all of the betting functionality and for the game to flow and end like a normal game. The betting logic was achieved through the turn controller, which only allows 'valid' betting options, depending on the state of the game (e.g. a player cannot check if someone has already bet). The ending of the game was achieved through the CardScorer class. This class takes in a players hand and the river and calculates the best poker hand they could have. It then returns a score determined by the hand they have, and the 'high card' of that hand to see who the winner of the round is. Once a winner has been chosen, that player gets the game pot and the next round is started.

Unfortunately, there were some user stories we did not get to implementing. These were low priority, but in the future we would like to see them done. For example, the user desired to have witty dialog between themselves and the AI. That is, they wanted multiple conversation options when they 'talk' to the AI to add another sense of realism to the game. While we had the dialog written, we did not have time to implement this. Additionally, we did not think of a good way to implement an on-demand help window, and we never provided 'perks' or 'cheats' for a player. Finally, the user wanted the ability to change the difficulty of the AI, but considering our focus was on the GUI and not the AI, this is not something that was accomplished.

Object Oriented Design:

Player/AI	
Knows about its money Can gamble Knows its hand	Table

River	
Knows about own cards Know when its full	Table Dealer

Deck	
Knows what cards are left Be able to deal Reset	Table Dealer

Table	
Knows Players Knows Dealer Knows River	Game

Card	
Needs to know own value Compare itself with other cards	Table Player River Dealer

Dealer	
Keeping the pot Knows the minimum bet Know win condition	Table

Game	
Runs and manages game	

Figure 1: CRC cards

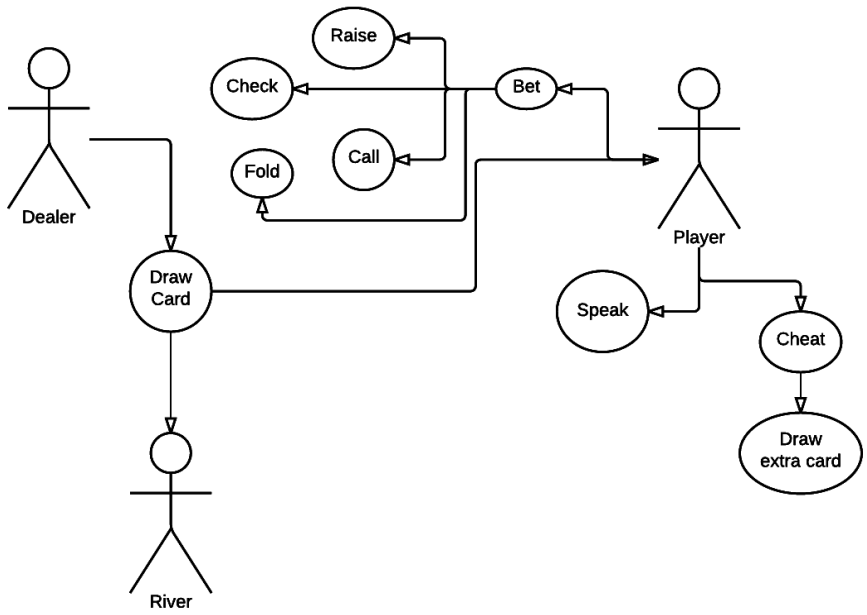

Figure 2: UML Case Diagram

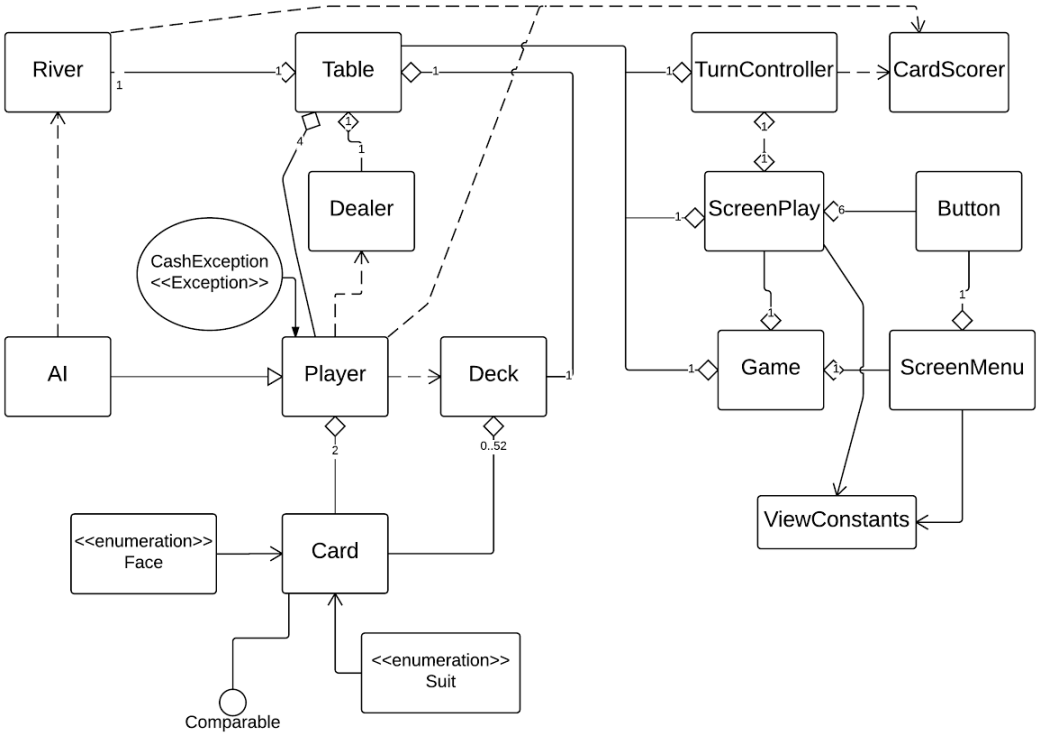

Figure 3: UML Class Diagram [Top Level]

As shown in the above diagram, we were able to maintain a proper MVC design pattern, but had to make some minor concessions. Firstly, due to the way Slick2d runs its game loop, the view and the controller are mixed together and must be teased apart to follow a proper MVC pattern. To tease these two apart we had both the view and the controller aggregate a copy of each other using a round-about method. Once that was accomplished the model fell snugly into place using the table as an interface buffer between the controller, the view, and the model. As can be seen in the diagram, the Game class starts off the whole process and collects all the pieces of the program and distributes them as needed before the game starts. Another major player in the game is the somewhat bloated ViewConstants class; this allows for more readable code in the view and makes maintaining the large collection of constants much simpler.

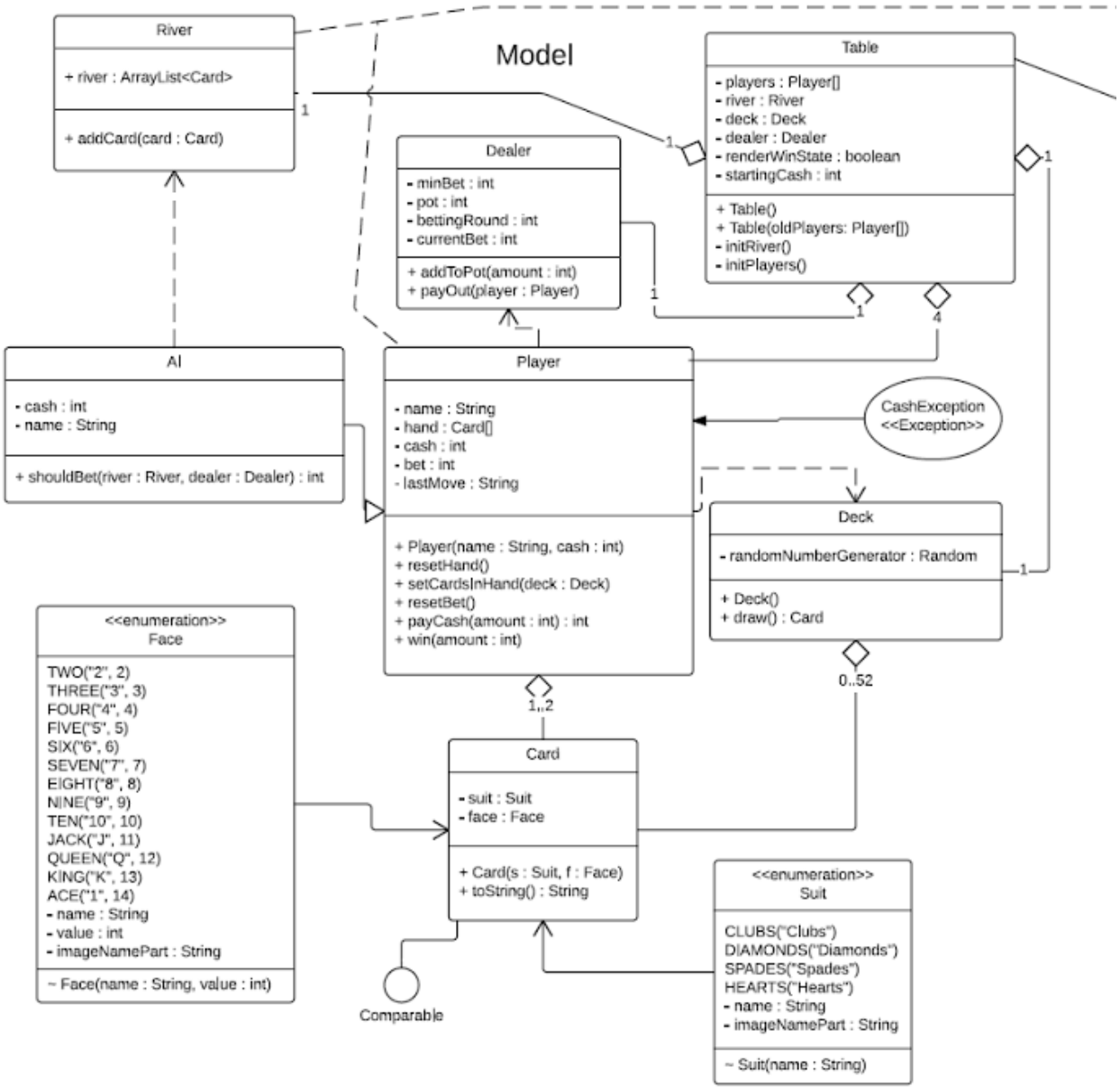

Figure 4a: UML Class Diagram [w/ important methods]: Model

The model itself it fairly straight forward and does not need too much explanation, except for the AI class. This class has within it a lot of thinking logic that the AI needs to determine the best move to make during the moment. The class contains a number of methods that analyse the current river and the AI's hand and can pick out the best move between conflicting hand patterns (ie. knows to play for the full house instead of the pair and triple).

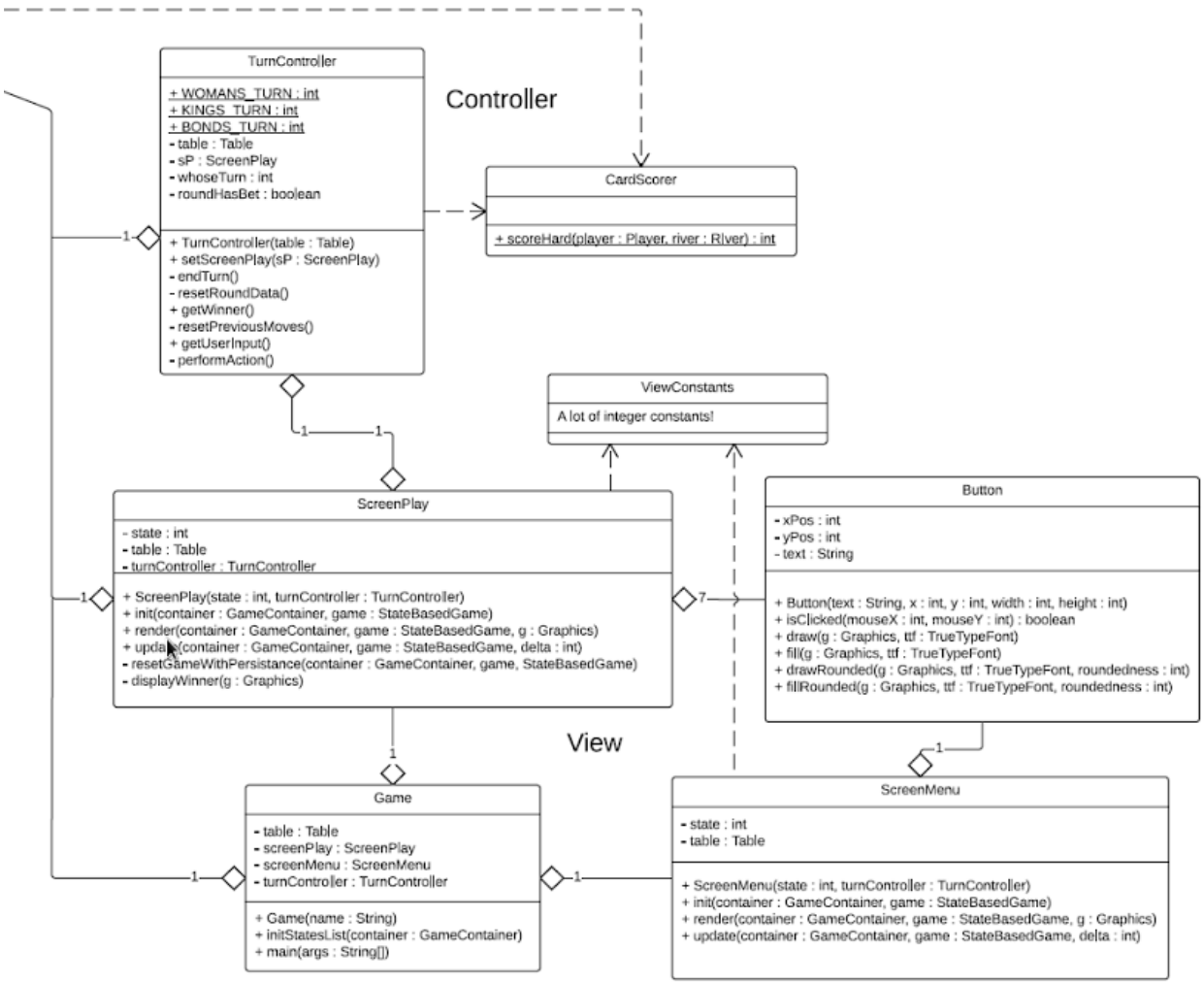

Figure 4b: UML Class Diagram [w/ important methods]: Controller, View

The game was made using Slick2d's StateBasedGame interface. This interface allows for simple switching between different rendering environments without having to load elements in at switch-time. This was essential for making smooth and responsive screen changes that are currently in place and for later implementations. The ScreenPlay and ScreenMenu classes are both examples of Slick2d's SimpleGameState interface, which is what the StateBasedGame uses to flip views. As can be seen, the two classes share many methods that are used to render and update the screen information as the user interacts with the game itself.

Due to Slick2D's simple architecture, we lacked the conveniences that exist in Swing and JavaFX, such as JButtons, JPanels, and many ActionListeners. So, to compensate, we wrote

some of our own, like with the Button class which contains methods to draw a button in several fashions, and an ActionListener to determine if the button is being interacted with.