1. Overview:

The project has a Card class, containing all the information that a card needs, like suit and value. The card class also has some basic functions, one to determine if a card can be played on top of another, and a getValue() function gives the output string of the card. There is another function that is used to change the content of the card for testing.

The Pile class is a class that is used to store cards. It contains cards and allows the user to shuffle the cards in the pile, draw a card, add cards to it, and look at the content on the top of the pile. This is the class that is used to represent draw piles, discard piles, and the heads on the board.

The Player class represents a player in the game. Each player owns a draw pile, discard pile, and a reserve slot. The player has the capability to draw a card, reshuffling their draw pile when required, swapping a card they are holding with their reserve, adding cards to their draw and discard pile, adding their reserve card back to their draw pile, as well as checking whether they have any cards or not.

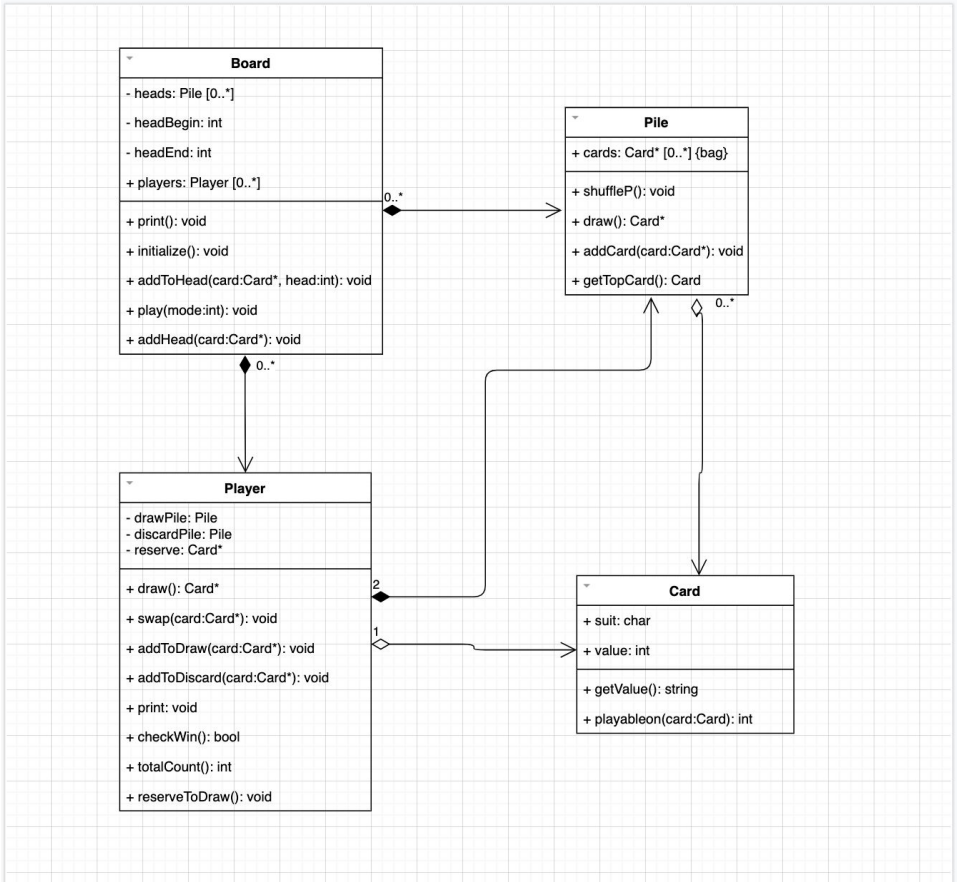
The Board class owns a list of players and a list of heads (piles) and keeps track of which player's turn it is. The board is capable of printing its contents (the heads and players), initialize itself (shuffling and plays hee first head), add cards to a head, and adding heads. Board has a Board.play(mode) function, which is the main function where it will loop and have players play until they win. It also handles all the inputs that players are supposed to put in and then outputs the result. The Board.play(0) function is the function that combines the other functions to calculate and display the correct outputs. The play also has a testing mode, which is called using Board.play(1) and lets the user enter the testing mode. The initializing and the play function will throw an exception if the user enters the eof character which will be caught using the main function.

The main function takes in command argument, decides the seed, and whether the board will be in testing mode or not, then it will initialize the board and play. The main function will make the random engine and pass a reference to it for players (in the Board.initialize(rng, mode) step) so that the players will use the same engine instead of using new ones.

2. Design:

Due to the nature of this project, if one doesn't use STL containers or smart pointers, it can cause a memory leak quite easily. Therefore std::vector is used for all instances where one would otherwise use an array. std::unique_ptr is used for any instances where one would use a pointer. I felt that using shared_ptr is unnecessary since a card can only belong to one pile, a pile can only belong to a player or a board as a head but not both, and a player has to belong to one board. There are no instances where an object needs to belong to two things at once, so I decided to use unique_ptr, which is safer since using shared_ptr can potentially still cause a memory leak if one accidentally adds a cycle.

Here is the UML diagram



Note that each pointer here is a unique_ptr, and every collection of objects with [0..*] items are in vectors.

The reason I am showing some of the private variables for Board is that originally, I had these as public variables.

Private variables are used food the Board class and Player class to increase cohesion. By setting draw pile, discard pile and the reserve as private variables, the user can only access

these using functions that they would actually need. For example, a player won't need to draw a card from the middle pile or draw from the discard pile, so you won't be able to use them on players.

Changes to the design from the original plan:

drawCount and discardCount from the player class have been made to private, to decrease coupling. By not letting the user/coder explicitly change the drawCount and discardCount, it makes the code safer (less segmentation fault chance) and also if the drawing mechanic change somehow, if I only update the drawCount and discardCount in the Player class, it won't affect the other class if the Player class.

The same thing can be said for my decision to make heads, headBegin and headEnd private in Board. By not letting classes like Players to access the heads, if there is a change in the cutting heads portion of the ruleset, the other class won't be that affected.

Some other functionality has been added as well, such as reserveToDraw() and addHead(). The reason is for the resilience to change, which I will explain in the next section.
Card.playableon(card) returns an integer now. The reason will be discussed in the section "Resilience to change".

Originally, I planned on having the main function contain the main loop, but then I thought it makes a lot more sense just to have the board function contain it, this way they can just call Board.play(0) in main, and Board has access to all its private variables for easy implementations.

I have also added some more accessors such as getDrawCount() and getDiscardCount() for a player, this way I can access them when needed for a calculation, but also makes it so that you can't change their value explicitly outside of the player class.

Changes to schedule from the original plan:
Due to the changes to the new designs, here is the new schedule.

Date	Description
12/01	Finish the UML diagram and the initial plan of attack
12/02	Implement all the .h file and Makefile
12/03 - 12/04	Implement the initialize function and all of the functions in Pile and Card, as well as Player.addToDraw() and Player.addToDiscard(). Test the initializing is done properly.
12/05 - 12/07	Setting up the -testing option, implementing Board.print() and Player.draw() (without reshuffling when needed), set up the main function and Board.play() enough so that I can draw cards and add cards to the heads in a loop.
12/08	Set up the rest of the functions for Player, all of Player.draw()
12/09	Update the new design with new functions for Player.
12/10	Implement the new functions for Player.
12/11	Set up Board.play() enough so that player can perform cutting off a head and swap.
12/12	Set up any -testing features that aren't implemented already, test functions
12/13	More testing, work on demo plan (all base features)
12/14	Finish demo plan and final design document. Work on the additional features if I have time.
12/15	Catching up on anything I didn't do, if I have time, work on additional features
12/16	Last-minute bug tests if I am still alive after final exams.

3. Resilience to change:

As mentioned above, some changes to the design are meant to increase the resilience to change.

By making drawCount and discardCount from the Player class private, I make sure that no other class will be able to change the drawCount and discardCount. This means that if I change the ruleset and make a player draw more cards, the drawCount will be properly updated, since everytime I perform draw, the count is automatically updated. Same thing for discardCount, if for any reason a card is added to discard, the addToDiscard() function has to be called an the discardCount automatically updates.

Similarly, by making heads private and making it so that you have to add heads using Board.addHead(), if I cut off the head, and more heads emerge, then I just need to call more Player.draw() and Board.addHead()

reserveToDraw() is a function I added that handles what to do when the turn ends and there still is a reserve. If the ruleset changes and there are some other ways of handling the reserve, I can simply just change the contents of this particular function. Since this function is called by the Player class, you have access to all piles that the player owns.

Functions from the original design:

Card.getValue() makes it so that if you change the output of a card (e.g. 3H KD), I just need to change its output here to change all of it.

Card.playableon(card) determines whether a card can be played on top of another. This is what allows me to change the ruleset easier. For example, if I need to change the ruleset where black card goes up, I just need to change this function. Note that this function returns an integer value, I have it return 1 if the card that the player is holding is strictly larger than the faced up card, and return 2 if the card is exactly equal, return 0 it is unplayable.

```
int f=c->playableon(heads[m-headBegin]->getTopCard());
if (f>0){
      addToHead(move(c), m);


 if (f==2){
      immed=true;
}
[...]
}
```

Code location: board.cc (line 191 to 204)
The above code is a snippet of the code in Board.play(). C is a card. m is the user input (the user's move). Immed is a switch that will break the loop.
This way, if there is a ruleset change such as "if you play joker, you get an extra turn", I can make the Card.playableon(card) returns 3, then I can have a case where

```
 if (f==3){
        // do something
}
```

Here is another snippet of code in Board.play(). The code is found in board.cc (line 87 to 269)

```
while (cont){
   for (int h=headBegin;h<=headEnd;h++){
      [...]

      while(true){
            [...]
            if (!((m<=0)||(m>0 && m<headBegin) || (m>headEnd))){


            [...]
            }
            else if (m==0){
                if (swaped==false){
                  bool r=players[turn].getReserveCount();
                  if ((r==false) && ((headEnd-headBegin+1)<=1)){
                    cerr << "Invalid input, please try again." << endl;
                  }
                  else{
                    players[turn].swap(c);
                    swaped=true;
                    if (!r){
                      break;
                    }
                  }
          }
          else{
              cerr << "Invalid input, please try again." << endl;
          }
```

```
        }
[...]
}
[...]
}
```

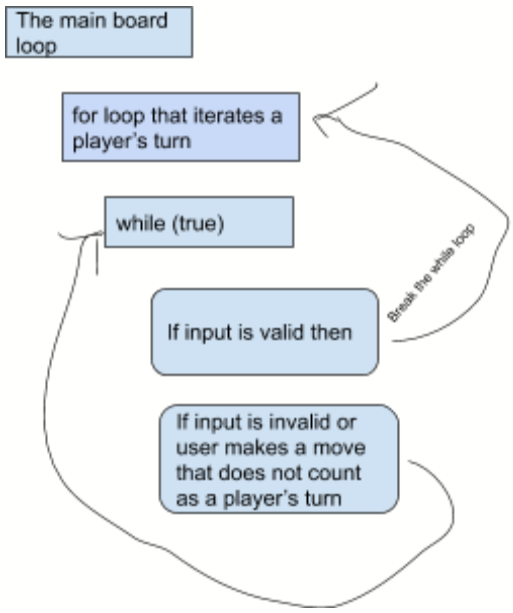
Explanation:

`while (cont){` is the main loop for board.play(). As long as cont is set as true, it will continue

`for (int h=headBegin;h<=headEnd;h++){` starts the loop for a single player's actions

`while(true){` makes sure that when an invalid move is made, the board will repeat the same sequence. However, this has another purpose. When m==0, which means the user inputs a 0, indicating a reserve switch, the turn doesn't actually count (unless there is no reserve), in which case the board will ask the player to make a move without incrementing h. When the user makes a move that will count as a turn, we just use break the break out of the while loop.

This allows me to change the program easily if there is a rule change such that if the user enters a certain number that allows the user to make a play without it counting as a move (like swapping for reserve), let's say the user can type -1 to put the card at the top of his discard pile to draw another card (for a limited number of time), I can special case (m==-1), simply use the Player.addDiscard() and Player.draw() function for implementation, and it won't count as an extra turn.



4. Answers to questions: (Italics represent a change to answer from the original plan)

Question:
What sort of class design or design pattern should you use to structure your game classes so that changing the interface or changing the game rules would have as little impact on the code as possible?
Explain how your classes fit into this framework.

I designed the code in a way that for each of the different actions you can take (cut off a head, adding to a head, etc), there is a different function, this way one can just change that particular function if the rules for this action gets changed. Player and pile are classes, this way if we were to change what they can do, we can do that easily. *After working on the project, I realize that private variables should be used for the piles a player owns and piles on the board as heads, to make sure we don't explicitly change the values of the pile otherwise.*

Question:
Jokers have a different behaviour from any other card. How should you structure your card type to support this without special-casing jokers everywhere they are used?

For the value field of the Joker, the value field will have a default of 2, so that if the Joker is used in a situation where the user doesn't declare the value, then the Joker will have a value of 2. Its suit is J, so whenever we need to see if a card is a Joker, we can just use that.

Question:
If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

I would make computer players inherit from the normal players class, since the actions they can perform is the same, and in my design, all the prompting questions calls are called from the board (so that when computer player plays, in Board.play(), instead of calling for input from cin, I can use an algorithm to determine the play. The computer player can also have a public field for defining what kind of player it is, and I can implement a different algorithm for the different types. The player will look at key fields that the board has to change strategies throughout the game. We can use an observer pattern to notify these computer players of a strategy change when the board is changed.

Question:
If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

In this case, I would use a static cast to cast the human player to a computer player for easy conversion. I would make sure that the human class already has all the values that the computer player has, this way the static case won't create an incomplete object.

5. Extra credit features

No memory leaks. Eof character ends program immediately.

6. Final Questions

Q1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

I worked along. The lesson I learned from writing large programs is planning is important. Some of my design originally was not optimal and changing them midway through takes away a huge chunk of time. Sometimes when I implement one function, I forget to think about how it would work in conjunction with another feature. For example for Board.play(mode), when I first code this, I just thought of implementing the default mode correctly. Then I implemented the testing mode (when the user runs ./hydra testing), which made me realize that I could have implemented the default mode in a way that I would need to change fewer things too get the testing mode working correctly.

Q2. What would you have done differently if you had the chance to start over?

I would have made a function that is just Player.play(), and Board.play() will just loop through players and call Player.play(). This would allow me to change what the player can do easier. The downside of this would be the Player object would need a pointer to the Board object, and changes to the Board object will impact the player object more. However, I think this would still be worth it, as it allows for adding computer players a lot easier.
Similarly, a Card.play() that gets called when Player.play() gets called could work if I make a Joker class inherit from the Card class, then I wouldn't have to special case the Joker class when one tries to add it on top of a pile.