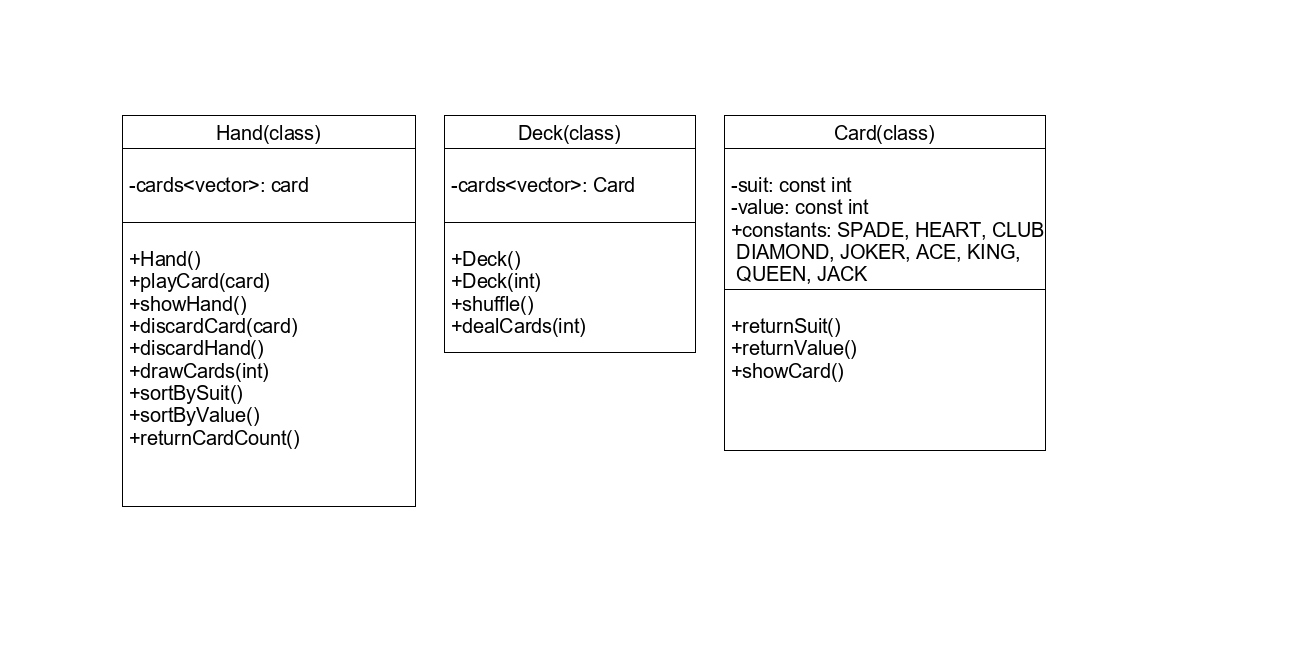
**Card Deck for Playing Games (Prof. Frank Emanuel)**

When we think about a deck of cards the simplest conception of the deck is as an array of card objects each having a suit and a value. A traditional card deck has 52 unique cards of four suits with 13 possible values in each suit, and a deck may also contain jokers. We can create a deck object that will hold our deck of cards, some utility functions like shuffle(), and constructors that will allow us to create a variety of different decks making our class generic and thus reusable. In object-oriented design (OOD) the idea is to create discreet reusable classes. When we are building our classes, we want to encapsulate the data and functions that work on that data (methods) in a single self-contained object. At this stage we are not building complete objects (classes) but rather identifying the components and how they are used. For example, most data elements in an object are made private so that we can strictly control how the data is accessed and manipulated. We can think of such objects as *black boxes* with a set of public methods allowing us to work with each object.

Below I have imagined one way of approaching the deck of cards problem, I have a card class which gets used by the deck and by each player’s hand. I use private vectors for the cards in each as vectors are a flexible form of array that are part of C++, but in other languages we could simply use an array.



In the build to test approach to programming we can begin implementing our classes starting with the cards. This way we can build and test each of the methods before adding in decks and hands. It is always good to break down our coding challenges into small testable units. It is also good to test things in the simplest way possible and add complexity after we have the logic working correctly. A great example of this is in the card class where we can implement the showCard() function to just print out the cards value in the terminal. When we know we are identifying the correct card we can add in a link to a card image. Once we know we are returning the correct image by printing the file name in the terminal then we can add in the functionality to display the graphic on the screen, likely requiring us to set scaling and location which would be parameters our game would pass into the card object. This may seem tedious but each of those steps adds libraries and complexity, so if you are not positive that the logic works in the simpler cases it will be a lot harder to debug when you are trying to move graphics around on a window.

The deck is the next object to build because it has very basic functionality and will give us some of the logic, we need to create the hand class. By default, we will construct a standard 52-card deck with no jokers but overloading the constructor will allow us to create all kinds of custom decks for different kinds of card games.

Once we have a deck, we will need to be able to deal cards into hands, hands here are any collection of cards that can be interacted with as a collection – so in many trick taking games the kitty becomes a hand and if you win the bidding then you can simply add that hand to your own hand. We can easily overload some operators to make this work. I have identified a small set of functionalities in my UML diagram, as we continue to work with our cards, we can extend the hand class to accommodate future functionality. We will want to build and test these classes before we move onto thinking about the rest of the game engine, the goal will be the creation of three core classes that can be used in any card game we would like to implement. As such we should place the cards into a header file such as cards.h which we can include whenever we want to create a card game.