CRITICAL ANALYSIS OF THE PROJECT WITH RESPECT TO THE IMPLEMENTATION OF 6 OBJECT-ORIENTED PRINCIPLES

1. ***Program to an interface, not implementation:*** Implementation of each and every method can be a very tedious job for a programmer. As a programmer, we always strive for making our code a better one. So, we must ensure that we have a healthy practice of using interfaces rather than implementing each and every method as modifications to it can be very time taking. In the code. We have a class called cards which contains methods of Color, it could so have been possible that rather than putting the color inside the Class Cards we could have directly used it in its two subclasses Normal cards and Special Cards but it could have been a repetition of the same color attribute and the method. So, by using the interface option we created a window for multiple inheritances which proved as an advantage for us as we can simply make use of our superclass and need not be implemented over and over again. Yes, it follows the principle.

2. ***Favor composition over inheritance:*** composition must be preferred over inheritance as it is easier to modify later. With composition, it's easy to change behavior on the Dependency Setters. Inheritance is more rigid as most languages do not allow you to derive from more than one type. In the code, the Utility class is composed of the classes Discard and Draw. Yes, our code follows this principle

3. ***Classes should be open for extension and close to modification***: Programmers generally do not start from scratch they prefer building up from the point where the other programmer has stopped. Classes must always be open to extension so that the child classes do not need

implementing the same class over and over again at the same time we must also ensure that the parent class is concrete as keeping it always available for modification is tedious job and requires multiple time testing. In our code, we did not keep the utility class concrete and it may sometimes be subjected to modification. A better version could have been if we could have kept the utility class concrete.

4. ***Dependency on abstraction rather than concrete classes:***
   Implementation of abstraction is always a better alternative than building up concrete classes as it can easily be viable for modification. Our, code has an implementation of abstraction in the cards as an interface
   And also we have made many functions as a result of which our main class becomes shorter, lesser complex as faster to compile

5) ***Encapsulate what varies***: In our code encapsulation, is not being used as we have not declared the attributes to be private neither did we use the getter nor the setter functions. But, it is a healthy practice to encapsulate the variation. This principle is not being followed in the code.

Design Pattern:

The design pattern refers to ideation rather than implementation.

1) ***Strategy Pattern***-It is not being implemented as we have not used encapsulation much and also we did not define a family of functionality and encapsulated them to make them interchangeable.

2) ***Builder Pattern:*** In the builder pattern we construct a complex object from simple objects using a stepwise approach. We have implemented this pattern clearly in our code as we preferred building of small building blocks /methods rather than the building of complex functions we focused on small functions as a result of which our main function is lightweight and it reduces complexity. Here, the object was not created in a single step and we followed the de-centralization method.

3)***Decorator Pattern:*** Decorator pattern uses composition rather than inheritance. In our code, the utility is a composition of the draw and discard. Via this, we are able to add flexible responsibilities to an object dynamically and also add responsibilities to an object that we wish to change in the future.

4)***Observer Pattern***: It is not being followed as there is no one-to-one dependency to which an object must relate.