Assignment #2
Group: Benedict + 3

Command-Line Rock-Paper-Scissors Game

Encapsulation

All of the code that we wrote is encapsulated into classes that help to segment the responsibilities of the program, thus we will be able to change, modify, or add to the code as needed in order to accomplish all that we set out to do. We have a main class that just makes a new instance of the game then tells it to run, then the program goes to another class that is in charge of controlling the game and its flow. We then are able to call other classes' methods that then execute their code in order to progress the game. We also have one class for players that had 2 child classes: one for the computer and its attributes and one for the human player and their attributes that do not apply to the computer. Thus, if we need to update either the computer, human user, or both, we will be able to without having to have repeated code or make some attributes not apply to the other class.

One of the easiest explicit examples of how our code is encapsulated is with the weapons class. Since the game will call the weapon class method to set the players' weapons, we will be able to update the number and types of weapons offered to the players. Thus we can change the game from just Rock-Paper-Scissors to the expanded game Rock-Paper-Scissors-Lizard-Spock. This will increase the number of ENUM objects without having to rewrite a bunch of code to incorporate the other two options. In future iterations of this game, we will likely come across some aspects of the classes that are not encapsulated well enough and will need to make more classe to keep the number of purposes of a class down to one or two.

Law of Demeter

For each class that was designed in this program, the Law of Demeter was used. This was accomplished by giving specific tasks to certain classes, which allows for encapsulation of our classes. With this encapsulation we are able to ensure that no class can go outside of the bounds of it's "friend zone" and start using functions or variables that does not belong to it. We also made sure that every member function can only use member variables located within its class along with any parameters that are passed through it.

One example of the Law of Demeter that was used was in our user class. The user class contains the name variable which can not be accessed except by its close

friend the Player class. This variable can not be changed by any class other than the User class. In order for anything to be changed in the user class, it must be done through itself or it's close friend the Player class, ensuring no strangers have access to this class. Another example of how we used the Law of Demeter would be our Stats class. This class contains the percentages for both winning and losing for the player and the CPU. Due to this class being "close friends" with the player class it is able to see the wins and losses of the player class showing how only "close friends" have access to this information. This class also does not talk to any other class to set it's data aside from it's close friend the Player class.

One other example of how the Law of Demeter was used is in our Weapons class. This class does not allow any other class aside from itself to edit it's member variable playerWeapon. The only classes able to access the set weapon functions are the "close friends" of the weapon class being the Player class. None of the other classes can interact with the weapons class and the only class that has access to those functions and variable is the Player class.

Cohesive Classes

When programing the RPS game application, we tried to keep every class focused on only one objective without using many classes that could be consolidated together and still have a single purpose. In total we had seven classes, each having a purpose that did not encroach on another class's purpose. The player class has the objective of keeping track of the players wins, losses, and ties with methods to edit and return the values of each of the variables. The computer class is a child of the Player class and only has one method, besides the constructor, that chooses its weapon when called. This class can be further expanded later to include some machine learning that will make it more difficult for the user to win. The user class also inherits attributes from the player class and has methods to ask for the user's name and to ask for the user's choice of weapon.

The stats class is there to keep a record of the user's win and loss percentage as well as the computer's win and lose percentage. It also has a function to print both the computer and user stats as well as the game stats and who won the round. The weapons class is to keep track of the weapons available to the players and to interact with the weapon list, like setting the players' weapon choice or listing the available choices to the user. The RPSControl class is the main game progression class and is meant to control the flow of the game and interact with the other classes so the game can be played. It has many functions that inform the user of the status of the game and what step they currently are on through console messages. It also keeps track of the number of rounds and repeats the game until that number is reached. Finally the

GameLogic class is used to determine who wins the round. This class only has one method in it but it is a series of if-else statements that determine which player is the rond winner or if they tied.

Loosely-Coupled Classes

When implementing the classes for this iteration of the RPS game we attempted to make all of the classes loosely-coupled, however for a game it is rather difficult to accomplish this for the first iteration of the project. Currently in our implementation of RPS we have one loosely-coupled classes. The one loosely-coupled class is the weapons class as any changes made to this class does not affect the player class. In a future iteration it is definitely possible to make a lot of our classes loosely-coupled. Most of our classes are not entirely tightly coupled, but in a state somewhat between loosely-coupled and tightly-coupled. Many changes that occur in one class do not affect any other class, however the parameters of certain functions can change which would require a change to the function calls in other classes. Thus, many changes made to classes do not affect the vast majority of other classes, however, parameter changes will affect other classes. For example, we can create a new class which handles the weapon comparisons and change the GameLogic class to call on the comparison class which will allow for any changes in the actual weapon comparison to not affect the GameLogic class.