Team Rabbits

Assignment 3

CMPE 135

Team Members

Shreya Aggarwal

Alberto Chavez

Nhan Ha

Ngoc Jess Mai

Encapsulation of Code:

Chooser class is an abstract base class that defines the interface for choosing a move. Additionally, the Chooser class is for randomly choosing the move and takes place between the computer player and human player so that it is a public function. It will manipulate the data member such as the humanchooser function. We have encapsulated the code in the program is when making the Chooser class, as this is the main form that we use encapsulation to keep hidden the details that go into all the derived classes from Chooser. In our case, we can see that it helps to encapsulate the

Next, the RandomChooser is set as a virtual function that overrides the behavior of the bass class. Also, this encapsulates the code that is useful when implementing the move from both player's and the computer's move. Besides that, another encapsulation of code is that the "RandomChooser " interface defines a single "make_choice." With that, the way we have implemented *make_choice* allows for encapsulation, given that now the caller will have no access to how the actual choice is being made. This can help in the future so that there are no incorrect calls or misuse of the choice function.

The chooserFactory class creates an instance of a Chooser subclass based on a string input using a static method when it gets called. Also, it is to encapsulate the creation of the random move of the Chooser in order to return the value as unique pointers to the Chooser.

The Law of Demeter:

The program followed the Law of Demeter by only calling methods on directly related objects in ChooserFactory class, only called the make_choice method on instances of the Chooser class, which helps and reduces coupling between classes and makes code more maintainable and flexible. Thus, it makes the code more modular and easier to test in order to improve the maintainability when running the game.

Not only that, but we have tried to implement the principle moreover when accessing the variables in our classes. When looking into the *RandomChooser* class, you can see that we have the variables under the private tag because we do not need them to be accessed by things outside of that class. That is why we don't have get() or set() functions for the variables since they are unnecessary for this class because we are only currently getting directly called from within that function and nowhere else.

Cohesive Classes:

When looking into the cohesion of classes, we are trying to see if the classes maximize in trying only to implement one single concept in order to keep the code modular and straightforward to build upon. For this assignment, we have implemented a significant level of cohesion that will help and aid us down the line when we go through and build upon this project and make it more complex. With this, the *Chooser* class focuses solely on trying to choose a move in the game of RPS, for which it is an abstract class.

Another class that follows this type of principle is the *RandomChooser* class we have for the computer to make a move. This class is solely responsible for trying to generate a random move that the computer can use in the game of RPS. The same goes with the *HumanChooser* class that we have in the *ChooserFactory,* in here, the class is solely responsible for getting and recording the move that the human player is making when playing the game.

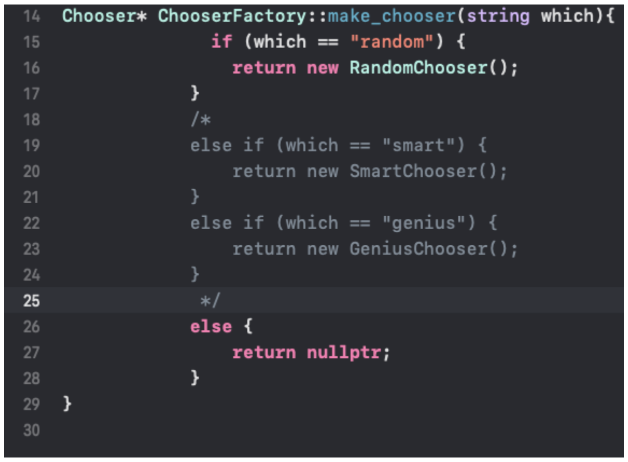
For the future iterations of the project, we have already made skeletons for two other different ways the computer will be able to make moves using two different algorithms to make moves. And this will all be able to be selected in the *ChooserFactory*. Overall the current project exhibits enough suitable class cohesion given that currently, we have a very simple game, but it is enough for us to build upon in the future.

Loosely-Coupled Classes:

When designing the classes we tried to make them loosely coupled in order to make them able to edit, expand, and also for building test cases for each class; that way, we are not really affecting any of the other sections in the program or classes. Along with this, we have also made the *Chooser* class an abstraction for this very concept we are trying to implement. We are implementing the class as an interface for the program in order to make a choice in the game, either by using the *HumanChooser* class or the *RandomChooser* class that we have implemented currently for the possible ways in which a choice can be made.

Of course, this is all done by using the *ChooserFactory,* which is where we are actually instantiating the *Chooser* objects that we are going to be using because of this method of implementation. The *chooserFactory* class only depends on *Chooser* abstract base class, not concrete subclasses, making code more modular and easier to maintain. This will allow us to quickly swap, add, or edit the different ways we implement the *Chooser* objects.

To add to what the previous paragraph discusses, we have also loosely coupled the ChooserFactory class. As seen in the figure below:



```
Chooser* ChooserFactory::make_chooser(string which){
              if (which == "random") {
                return new RandomChooser();
            }
            /*
            else if (which == "smart") {
                return new SmartChooser();
            }
            else if (which == "genius") {
                return new GeniusChooser();
            }
             */
            else {
                return nullptr;
            }
}
```

From here, you can see that the function inside of the cpp file can be used to quickly and efficiently swap between different types of ways the computer could make a choice. Right now, we currently have just let the computer make a random choice, but with the loosely-coupled classes, we are looking to add in a simple choice algorithm and a very sophisticated algorithm to make the game more competitive. Thus giving us a way to quickly expand the game and make modifications to the classes without breaking the entire program.