Anh Huynh
Danish Khan
Thanh Nguyen
Simon Altamirano

Assignment 3: Rock, Paper, Scissors

Class: **Game**	Class: **Player**
scorePlayer: int scoreCPU: int roundNumber: int Rounds: vector<vector<int>> winner: enum	move: int
Game(): constructor executeMatch(): void updateRound(result: int): void calculateResult(pmove:int, cpumove: int): int	Player(): constructor setMove(moveInput: int): void getMove(): int printPlayerMove(): void

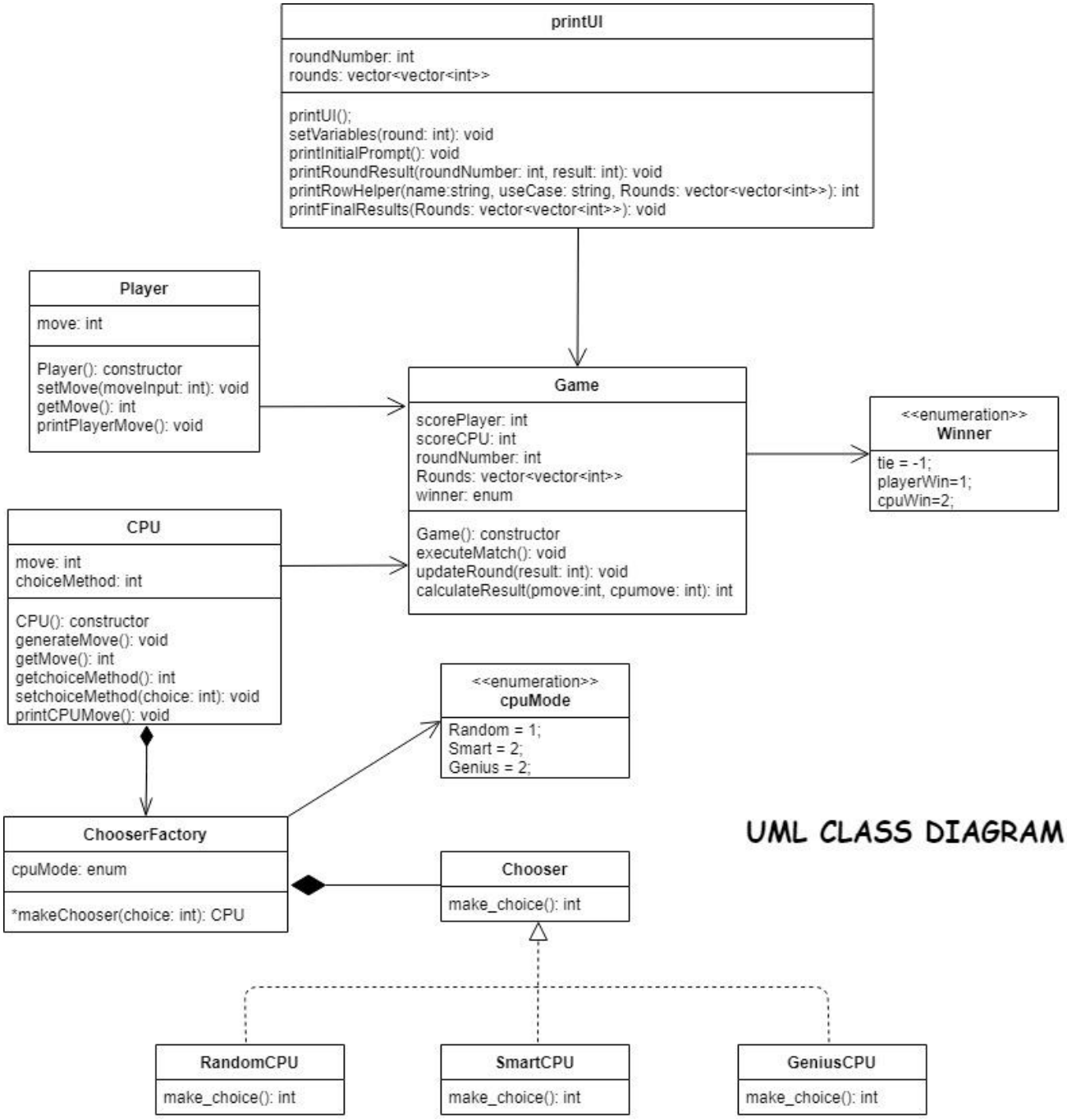
Class: **printUI**	Class: **CPU**
roundNumber: int rounds: vector<vector<int>>	move: int choiceMethod: int
printUI(); setVariables(round: int): void printInitialPrompt(): void printRoundResult(roundNumber: int, result: int): void printRowHelper(name:string, useCase: string, Rounds: vector<vector<int>>): int printFinalResults(Rounds: vector<vector<int>>): void	CPU(): constructor generateMove(): void getMove(): int getchoiceMethod(): int setchoiceMethod(choice: int): void printCPUMove(): void

Class: **ChooserFactory**	Class: **Chooser**
cpuMode: enum	
*makeChooser(choice: int): CPU	make_choice()=0: virtual int

Class: **SmartCPU**	Class: **GeniusCPU**	Class: **RandomCPU**
make_choice(): int	make_choice(): int	make_choice(): int

Enum: **cpuMode**	enum: **winner**

Random = 1
Smart = 2
Genius = 3
	Tie = -1;
	playerWin = 1;
	cpuWin = 2;



Encapsulated Code that will Change

The functions we have all contain private variables and functions that manipulate those private variables. That means if we ever needed to add more functionality or extra variables to be manipulated later down the line, it wouldn't be difficult to do. We also utilized interfaces through the Chooser interface and ChooserFactory so we can add different methods for how the CPU chooses its move later.

Law of Demeter in Our Code

In essence, the law of Demeter is a programming style where the programmer can use only variables and functions declared in its class while having limited knowledge of other classes. In our implementation, we used the law of Demeter by only using the appropriate variables and function calls. At the same time, we can instantiate other classes by including them in the header. For example, in our Game class, we implemented the law of Demeter by focusing on the main functionality of each class. The game class will only use functions and private variables within its class, and when it calls on other classes like the player class or the CPU class, it does not dig into how those class's functions do their job, just that they can do it.

Another example in our code respecting the Law of Demeter is our CPU class. To obtain the CPU mode - Random, Smart, or Genius - in our Game class, the chooser class is called. Therefore, we only tell the chooser to select and return the mode the user wanted instead of the gaming class telling each individual class if they are being called or not.

Cohesion in Our Code

Each of our classes is independent of each other and have their own specified jobs.

- The Player class only sets the player move, gets the move, and prints the player inputs to ostream.
- The CPU class only generates moves, gets moves, set and gets the version of CPU which the player picks, and also print the current move picked
- The Game class executes and updates the match, calculates and outputs results. This class instantiates Player, CPU, and printUI classes in order for the Rock Paper Scissors game to start.
- printUI only prints the UI and scores based on the input vector.

Loosely-Coupling in Our Code

We made sure that all of our code functions don't depend purely on a specific class's set of variables. For example, in our print class, we made sure that you can pass what you want to print so that it may be used later for other things, rather than hard coding the variable of the game class that we needed to print to allow for loosely-coupled classes. Another example is when we calculate results for the game. The calculation does not hard code values from the CPU class and the Player class but instead passed into the function for calculation.