**CS172 Program 2: Pig Game**

**Due : Dec 11th**

In this program, you will get to practice writing classes, using enum type, and reading from and writing to a file. Program 2 is similar to Program 1 but you will write Program 2 using classes. There are also some changes. Program 2 supports one to five players and it can save the state of the game. When a user chooses to quit a game, it gives the user the option to save the game. Later, when the game is run, it gives user the option to resume the game.

In this handout, we use the Unified Modeling Language (UML) notation to represent classes. A class is a table with the name of the class on the top left and the variables and methods below the class name. The minus (-) in front of a variable or method means it is a private, while the plus (+) means it is public. Variables, methods, and arguments are followed by a colon (:) and the data type. The right column contains the descriptions of the variables and methods. Below are suggested classes for Program 2 you can use. You are free to come up with your own classes but they must be reasonable.

RollResult enum type:

The RollResult enum (shorted for enumerate) type is used to represent the result of rolling two dice. When the player object rolls the dice, it will return the result as a RollResult type. The possibilities of a roll are: ONE\_PIG, TWO\_PIG, SAME, and DIFF. ONE\_PIG is when one die is a 1. TWO\_PIG is when both dice are one. SAME is when both dice have the same value. DIFF is when both dice have different values but none is one. If you plan to support more than two dice, you will want to change TWO\_PIG to ALL\_PIG. The solution to RollResult is below. Save it to a file named **RollResult.java**.

public enum RollResult{

ONE\_PIG,

TWO\_PIG,

DIFF,

SAME

}

The classes:

|  |  |
| --- | --- |
| Die | Description |
| -image:String [] | ASCII image of each side of a six-side die. |
| -value:int | The result of a roll. |
| -sides:int | The number of sides of the die (should be 6 for this game). |
| +Die() | Empty constructor, may initialize value and sides |
| +roll():int | Roll the die, return the result (between 1 and 6). |
| +getValue():int | Return the result of a roll. |
| +display():void | Print the result of a roll as ASCII image to screen. |

Example usage of Die class:

Die d=new Die(); //creating a die

System.out.println( “Rolling the die”);

int result = d.roll();

System.out.println(“The result is “ + result);

d.display(); //display die image of result.

|  |  |
| --- | --- |
| Player | Description |
| -name:String | Name of player (single word name) |
| -turnTotal:int | Player’s turn total score |
| -cumulativeScore:int | Player’s cumulative score |
| -lowScore:int | Player’s lowest accumulative score of all games |
| -highScore:int | Player’s highest accumulative score of all games |
| +Player(name:String) | Constructor to create a player object, takes a name |
| +getName():String |  |
| +setName(name:String):void |  |
| +roll(die1:Dice, die2:Dice): RollResult | Roll a set of dice, update turn total and accumulative score based on result of the dice. Return the result which is a RollResult enum type |
| +hold():void | Player holds, update score |
| +setTurnTotal(total:int):void | Update player’s turn total score |
| +getTurnTotal():int | Return player’s turn total score |
| +setCumulativeScore(score:int):void | Update player’s cumulative score |
| +getCumulativeScore():int | Get player’s cumulative score |
| +setLowScore(score:int):void | Update player’s low score |
| +getLowScore():int | Return player’s low score |
| +setHighScore(score:int):void | Update player’s high score |
| +getHighScore():int | Return player’s high score |
| +save(file:FileWriter) throws IOException:void | Save player’s stats in given file |

Example usage of Player class:

Player p = new Player(“John”);

Die d1 = new Die();

Die d2 = new Die();

RollResult result = p.roll(d1, d2);

if (result == RollResult.ONE\_PIG){

cout << “Player roll one pig.” << endl;

}

|  |  |
| --- | --- |
| GameState | Description |
| -gameStateFile: string | Name of file to save game state |
| -currentPlayerIndex: int | Current player’s array index |
| -players:Player[MAX] | Array of players (MAX is 5) |
| -numPlayers:int | Number of players in the game. |
| +GameState(file:string) | GameState’s constructor, initialize numPlayers to 0. |
| +addPlayer(name:string):Player | Create a player object for the given name, add player object to players array, return the Player object |
| +getCurrentPlayer():Player | Return the index of the current player |
| +nextPlayer():Player | Switch turn to next player. Hint: use the modulo operator to compute the index of the next player so that the players will rotate in a circle. |
| +getNumOfPlayers():int | Return the number of players |
| +newGame():void | Clear game state, empty players array, currentPlayerIndex set to 0 |
| +save() throws IOException: void | Save state to file and return true if successful. The state of the game include index of current player, the current tern total, and all player’s data. |
| +load() throws IOException:void | Load game state from file and return true if successful. The state of the game include index of current player, the current tern total, and all player’s data. |

Example usage of GameState clas:

Die d1 =new Die();

Die d2 = new Die(); //game uses two dice.

GameState state = new GameState(“game\_state.txt”);

//starting a new game with two players

state.newGame(); //start a new game.

state.addPlayer(“John”);

state.addPlayer(“Micheal”);

for(int i=0; i<state.getNumOfPlayers(); i++){

Player currentPlayer = state.getCurrentPlayer(); //current player

RollResult r = currentPlayer.roll(d1,d2); //make current player roll

If (r == RollResult.ONE\_PIG || RollResult.TWO\_PIG){ //if it is a pig,

currentPlayer = state.nextPlayer(); // it is the next player’s turn

//we don’t need to update the scores here because the player’s

//roll method does it already.

} else if (r == RollResult.SAME){ //if is the two of the same kind

continue; //make the current player roll again

} else {

//does user want to hold?

//if yes, update cumulative score and switch to next player

continue;

}

}

state.save(); //save game state

The main class or file:

* Loads game\_state.txt file. If successful, add “r) Resume” as an additional menu item to the default menu. The default menu is: N) New Game, E) Exit.
* If it is a new game, prompts for the number of players and then for their names. Create the necessary objects. Start the game with the first player as the current player.
* If it is a resumed game, create the necessary objects and initialize or update them to reflect the game state. Start with the current player (the player who was supposed to roll when the game was saved.
* While in the middle of a game, a user can choose to quit the game. The game should prompt the user whether they want to save the game. If yes, save the game state and quit. Otherwise, quit the game without any saving.
* The rest of the program’s logic is the same as in Program 1.

GameState persistent format:

First line is the index of the current player in the array. Second line is the total turn score of the current player. Third line is the number of players. Following the third line are rows of players’ data. The first column is the name of the player, the second column is player’s accumulative score, the third column is the lowest accumulative score, and the last column is the highest accumulative scores of all games. Each column is separated by a tab.

Format:

index\_of\_current\_player

turn\_total\_of\_current\_player

number\_of\_players

player\_name     cumulative\_score   lowest\_cumu\_score    highest\_cumu\_score

Below is a sample game\_state.txt file. The current player is 2, which is the index of the player in the array of players. This would maps to player “smith.” The turn total of current player is 50. There are three players in the game. The columns are player, player’s cumulative score, player’s lowest score, and player’s highest score.

Sample game\_state.txt:

2

50

3

Michael 43  0    0

Jack 31  0    0

Smith 13  0    0

Grading

Grading policy is the same as in Program 1. Remember to add comments to your code.

Submission

Your files should be in a folder Program2. Zip it by typing zip –r Program2.zip Program2. Use FileZilla or any SFTP client to get the file from stuwork1 server to your local computer. Upload Program2.zip to D2L Program 2 dropbox.