Tucker Lavell

CS 162 Fall 2017

Lab 3 Reflection

Reflection

At first I wasn't really sure what to do about a game, so I practiced inheritance by writing my Die and LoadedDie classes, and just printing rolls. I used hard coded numbers to ensure each class was being used correctly. Then I had to sit down and decide how to make the game. I decided making another class for the game itself seemed like a simple and easy way to move data around. At first I had an input function and a Game constructor that took in all the info about the game and the dice. The input function was similar to my Project 1, and that looked hideous. Once I got the game working, without trying to break it, I realized that I had not accounted for ties. I went back and added a statistic for ties and an end condition where player 1 and 2 can tie. The Game constructor worked, but it was messy and not the most intuitive implementation. Then I had to go back and do input validation.

This is where I had to do a complete overhaul. Using my input validation from Project 1, was not satisfactory, it worked, but it didn't work how I wanted. Fixing it brought up tons of new issues, including the fact that it was only integer validation, and this time I thought I needed string input validation. I wrote the string input validation, but it was messy. In the end both validations ended up getting the program stuck in loops that required multiple inputs to break. I had to go back to square one, and after tons of research I decided to make all menu type input options in the program basically a switch, using an input validation function. I settled on a function that takes in the cout messages, the max and mins of allowable inputs, and prints and loops until valid input. For the type of die input I gave the user the option of 1. Normal or 2. Loaded. Though they may not be the prettiest function calls, with sentences as inputs, they worked exactly how I wanted. Because of the way I setup my input validation, I had to rehaul the Game constructor. This time I made it much nicer, it prints out that the game is starting, then it calls a function that "builds" the game by asking the input questions and setting the die's, then it plays the game once the die are set.

After going on Piazza to see what my classmates said about this project, I'm glad I found a topic about how to load the dice. Originally I was just adding 10% of the number of sides on the dice to the roll, but I saw in the topic that was not desired. I have now made it so that it add the results of 3 rolls together then divides it by 2.5, dividing by 3 gave too much of a 50/50. I played with the formula for hours before I finally came to something viable. It seems to win 58-63% of the time, based on large number of turns testing.

Test Table

Test Case	Input	Expected Output	Actual Output
Die and LoadedDie use separate roll()	Hard coded 0 for Die::roll() and 4 for LoadedDie::roll()	Normal die: 0 Loaded die: 4	0, 0. It took tweaking and research about inheritance to get it.
Small game	Turns = 5, P1 = normal P1sides = 10 P2 = loaded P2sides = 10	P2 wins	P1 wins (they actually tied, but no condition for ties). The size of the die also allows for normal die to be competitive with loaded die, based on current load function.
Loaded die is smaller, more turns	Turns = 5, P1 = normal P1sides = 50 P2 = loaded P2sides = 10	P1 wins or tie	P1 wins, the die might be loaded, but P1 can roll lots of numbers higher than the loaded max.
Tons of turns, same size die	Turns = 10000, P1 = normal P1sides = 50 P2 = loaded P2sides = 50	P2 wins	P2 wins