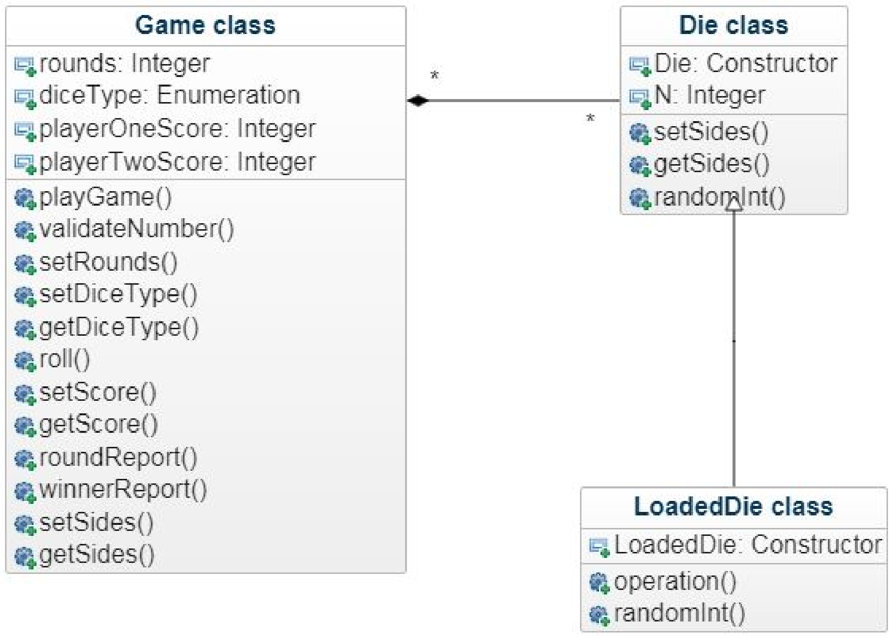
REFLECTION DOCUMENT

SANDRO AGUILAR
LAB 3 - SECTION 400

January 26, 2019

The overall design of this project is not very complicated. The lab requires that we have two dice classes, one of which is derived from the other and a game class to implement the game. This lab went a lot more smoothly than the first two however I believe that is because I have been able to reuse my validation functions. I have also found that some of the code that I reuse gets a little better as I find simpler ways to implements them.

The class hierarchy that I envision was like shown here:



The diagram shows that there is object composition between the Game class and the Base class where the Base class is inside the game class. The diagram also shows that the Sub-class is derived from the Base class. From our assignment, it states that the die and loaded die class should each have a “roll dice” function that return a normal and loaded value, respectively. However, since LoadedDie inherits from the Die class, it appears that the roll dice function from the LoadedDie class should be called instead of the roll dice function from the Die class. I interpret this to mean that the LoadedDie class should use function overriding to make this happen. Therefore, I will not require virtual functions for my initial design. This is also true since I am not using pointers to create my dice objects so I cannot have polymorphic behavior without pointers to the dice objects. I also think that I only need to create two dice objects, one of each since both users can share the same die (regular or loaded) if they choose to do so since both die will have the same number of sides. My design also includes the typical getters and setters for the private data variables that are required by each class.

During the implementation of the design, I did not need any overall structural changes to the layout above. However, the devil is always in the details. During the implementation, I found that my main class was handling activities that were more appropriate for my game class. For example, I was initially keeping track of who won round of dice in the main class when it is more appropriate to let the game class track. Doing this in turn allowed me to declutter my main function which adds to its readability.

A memorable challenging time during this lab was thinking about how to create a way to roll a random value that was skewed toward higher roll values. I found this challenge significant enough that I decided to instead push through with the project and saving that challenge for last since I realized that updating the implementation of of this challenging problem would not affect the overall structure of my program and would not require me to re-write any code. Lab 3 calls for a loaded dice that has a higher than average roll value than a normal dice with the same number of sides. The difficult part was thinking of a way to systematically create a structure where the changes of drawing a higher values was greater. Another curve ball is that the number of sides on the dice, N, must be taken into consideration when designing an algorithm. Luckily, I just took CS225 and I remembered the formula for the sum of an arithmetic sequence. This formula states that if you have values from 1 to N, then the sum of all of them is $\frac{(N+1)\,x\,N}{2}$. Why did I think of this formula? Because in my mind I thought “if a dice has 6 sides, then there is an equal chance of landing on any side. But if there are 2 sides with a 2, then my changes to get a 2 increase. And if there are 3 sides with a 3, then my chances of getting a three are even higher. So this created a pattern that results a pattern of number from 1 to 6 as follows: 1,22,333,4444,55555,666666. What this means is that ultimately, if you were to pick a random number from this list of numbers, than the chances of getting a higher number are significantly higher. And what is the length of this list of values? $\frac{(N+1)\,x\,N}{2}$:) so this gave me a good way to systematically create an an array of size $\frac{(N+1)\,x\,N}{2}$. This was important since the values in the list should not be greater than the number of sides on the dice.

TESTING PLAN

This test plan is provided to show how I tested the program. My goal was to create a test plan that is robust enough to provide assurance that the program behaves like it is meant to. The bulk of the test plan makes sure that the game parameters are properly set as this is what drives the rest of the program. Once all user data is entered, the game iterates for each round of game play, tracking the number of rounds played, round winner, updating scores, and showing a report of these values.

Test Scope	**Description**	**Expected Results**
Main menu validation	1. Show the main menu 2. Test validation by entering 1, 2, 0, 3, -1, -2, 01, 02, 1s, 2s, s1, s2, 10000, 20000, 111111, 22222, 1.0, -1.0, 1ss, 2ss, ss, !, @, 1!, 2@, ‘ ’ blank space, 1ss 1, 2 2. 3. Play game if option 1 4. Quit game if option 2	1. User is prompted to select option 1 to play, 2 to quit. 2. User input should only accept digits 1 or 2, reject otherwise 3. Continue with game with additional user input menu options. 4. Game ends upon

		selecting option 2.
Storing Rounds to Play	1. Show menu asking user to store the number of rounds to play. 2. A player should only be able to enter 1 to 1000 rounds to play. Tested with 0, 1001, -1, -1000, 100, -100, 1.2, 1000.2, 100r, 100 s*, r100, r 100, ‘ ‘ blank space,	1. Menu should show asking for rounds to be played. 2. Set the # of rounds to be played within a valid range of 1 - 1000.
Set Type of Dice to Use	1. Shows a menu asking user to set the type of dice to use. 2. Test validation by entering 1, 2, 0, 3, -1, -2, 01, 02, 1s, 2s, s1, s2, 10000, 20000, 111111, 22222, 1.0, -1.0, 1ss, 2ss, ss, !, @, 1!, 2@, ‘ ’ blank space, 1ss 1, 2 2.	1. Menu should ask user to enter the dice to be used. 2. Validate user’s input should select 1 or 2.
Set Total Sides on Dice (N)	1. Shows a menu asking user to set the total sides on the dice selected by user. 2. Test validation by entering 1, 2, 0, 3, -1, -2, 01, 02, 1s, 2s, s1, s2, 10000, 20000, 111111, 22222, 1.0, -1.0, 1ss, 2ss, ss, !, @, 1!, 2@, ‘ ’ blank space, 1ss 1, 2 2.	1. Shows menu. 2. Validate user’s input should only accept a value from 3 to 20.
Play all the rounds	1. A for loop is used to iterate through each round of game play.	1. A counter keeps track of the round the game is currently in and displayed in the report.
Tested loaded die Test - 1000 rounds, Player 1 uses Loaded Die, Player 2 uses regular die, 10 sides for each dice.	1. Test to see whether the loaded die is in fact rolling higher values than a regular die. Complete multiple test to make sure a player using a loaded die wins more rounds than a player using a normal die. 2. Verify that both die only roll numbers up to N sides.	1. Multiple tests were completed and a player using a loaded die wins more often than a player with a regular die. I ran the test several times and each time, Player 1 wins about 600 times with player

		two winning 300 times, all others being tied (60% vs 30%) 2. Both die show N as their max number of sides for each. Verified in the round report.
Keep Score	1. The winner of each round should be determined. 2. The winner should get a point. Tie game results in no score.	1. If player one wins, they get a point. 2. If player two wins, they get a point. 3. If players tie, no one gets a point.
Show Report	1. A report is display after each round showing the value rolled by each player, the winner of the round, the total score, and the game winner.	1. A report that shows current roll value, round winner, total score, game winner
Memory Leaks	N/A - new keyword was not used	N/A - new keyword was not used