Sean Mulholland ID No: 932652783 mulholls@onid.oregonstate.edu

Assignment 3 Design Document-

For assignment 3, I'm going to create a Dice class that is similar to the one we've used before. I plan to have a constructor that initializes the sides as a constant integer, and I plan to define a function rollDice(int numRolled) that will return a "dice roll" of the total value of throwing the die D numRolled number of times (or if we want to simulate 3 six sided dice, we can roll a six sided die 3 times and add the values). This object will form the foundation of my dice throws and make the attacks and defense incredibly easy to implement as each character has only 1 die type for each attack or defense.

For assignment 3, the hierarchies seem fairly clear cut:

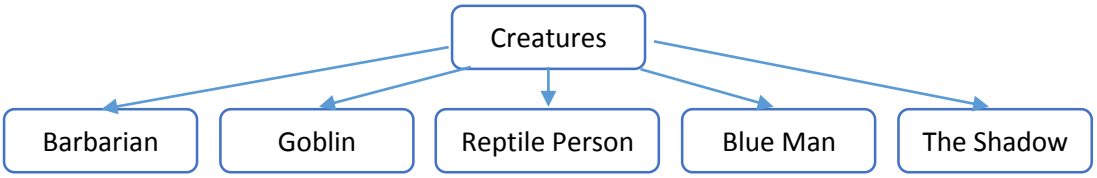

Figure 1. Inheritance Layout

The creatures base class will hold the following protected elements:

integers: armor, strength

double: attackMod

std::string: creatureName

virtual integers: attack(), defend()

Along with these protected elements, they will have a number of get functions for each value, and one modify function so that strength can be changed as the creatures attack one another. There will be an additional creature function modifyAttackMod() that will be utilized to modify attack of the creatures. Attack mod will be just a coefficient to the returned attack() function so that if an effect modifies the creatures attack strength (i.e. Achilles sets attackMod = ½) the set function can set that coefficient. In later uses of this code, this could include an increase of attack power too. Each of these components will be inherited by the sub-classes and the virtual functions are pure virtual functions so they will ensure that the sub-classes define the functions prior to use of the class. The attack() and defend() functions will be composed of the dice objects I designed for simplifying the functions. It will make the functions

As for the Barbarian, it only needed an attack() and defend() function definition and a constructor. The inherited traits should define the rest of the creature. The attack() function will return a 2D6 roll, and the defend() function will as well.

The goblin is more complicated. Although his basic attack() and defend() functions are similar, the only solution for the Achilles skill I could come up with would be an external evaluation of conditions from the instances. In other words, I will check if the attacking creature is a goblin doing 12 damage and if the opposing creature is not a goblin the second creature can have it's attackMod set to .5.

The reptile man is fairly simple. He will have 3D6 for his attack, and 1D6 for his defense function. He has no special abilities.

Sean Mulholland ID No: 932652783 mulholls@onid.oregonstate.edu

The Blue Man is the first using 2D10 for it's attack() function, but the dice function should handle this easily and create an instance with 10 sides instead of 6. Other than that, the implementation of the class is almost identical to the Barbarian and Reptile Man.

The Shadow uses the same attack type as the Blue man, but has an interesting armor special ability. 50% of the time The Shadow isn't where its enemy expects it and dodges the attack no matter how much damage is supposed to be done. I'm implying from the written direction that other than that 50% of the time it would have 0 armor. So, I plan to write a modified getArmor() function that will override the inherited getArmor() function. It will have a random element that will produce 10000 armor 50% of the time (and effectively "dodge" the attack), and the other 50% of the time will return a 0 for the armor. Other than that, the Shadow still gets a defense roll and can function in the same manner as the other types.

For the combat test system, I plan on implementing a simple menu to either view the creature type's info or to battle two creatures. I will use the polymorphic properties of pointers to accept any variety of subclass into the "slots" for my functions. The "slot" in the view creature function will be a Creature pointer that, depending on user input, will build a new instance of a sub-class assigned to that pointer. After the instance is instantiated, the user can get the program will output information about it and then delete the instance releasing the memory to the system. For combat, there will be two "slots" where the sub-classes will fit in. The user will be able to instantiate a class in a Creature* Char1, and a separate Creature* Char2. This should allow the same set of rules to be applied to any of the combinations of creatures the user could pick.

The rules will be simple. A coin will be flipped prior to combat, and that creature will get to attack first. The first creature will roll their attack, the second creature will roll their defense, and the proper damage will be applied after armor is taken into account. As stated before, there will be a conditional statement checking the Achilles condition in the loop as well. The combat will continue until one of the creatures is vanquished, combat will be over and the program will return to the main menu. Very little room for user input, or breaking the program. I plan to implement pauses at each step but std::cin.ignore() will handle the pauses.

Test Plan:

This program will have very little user input. They will make menu choices, and hit enter to proceed in the battle screen at informational pauses. There isn't much to break on the user's part. I plan on having the user input retrieved utilizing std::strings and the std::getline() function. I will also have a function to convert the input into a common format for parsing (all lowercase letters, and no whitespaces). Comparing the formatted input to a known bank should leave little room for error, and all non-matched input will be rejected and the user will be able to try again.

The biggest breaking points will be in the inheritance reliance of the program. I don't foresee any issue, but I will test a combination of each of the character types against one another (25 battles) and will ensure that the special attributes of the Goblin and the Shadow work. I plan on simplifying the goblin by just ensuring it returns a 12 attack by temporarily altering its attack return. If this works the opposing creature should hit a limit of ½ their max attack. For the Shadow, I plan on utilizing the Blue Man vs. the Shadow for several rounds. The Blue Man and the Shadow both have 2D10 attacks making

them the strongest, but hopefully using a Blue Man instead of another Shadow should allow me to more readily tell if it's working right instead of potentially having the creatures die incredibly quickly.

Reflection:

My implementation went well. I ran into issues with what needed to be included where, as the number of files being managed was much greater than I had worked with before. But, in actuality once I figured out the barbarian.h and barbarian.cpp it was easy enough to copy the templates into 4 new sets of files for the other 4 classes. A little doctoring of the values and I had functioning creature classes in a short amount of time. The hardest part was implementing the combat system and viewing character information section (even though it wasn't required).

In terms of the combat system, I faced issues with giving the user the appropriate information for what's going on and the number of variables I was going to use. I built the creatures class to function fairly internally. The attack() and defend() functions were pure virtual protected functions, so to retrieve them and simplify the code I set up getAttack() and getDefense() functions that would return this->attack(). This effectively encapsulated the dice and rolling functions of the attack, so I had to find a simple solution for informing the user. It basically came down to a balance of the already encapsulated functions, and the storing the attack integer and defense integers in local variables to the combat function. I decided it wasn't important for the user to know what each of the rolls were as long as the rolls performed as they were supposed to. In addition, I encapsulated the application of damage to the creature instance because it didn't matter if the user saw every step.

For the viewing of the characters, I just decided to write a few extra functions into the classes. I added in a function to retrieve the instance name (which I planned a variable for, but never planned anything to do with that variable for some reason), and a getAttackInfo() and getDefenseInfo() function that just returned a string literal defining the information about each class. It wasn't particularly elegant, but it worked and gave the user the information they may want.

The testing worked as expected. I had no hitches in user input or creatures battling one another. I had defined the special cases well enough ahead of time that they did not cause any snags in the testing or functioning of the program, excepting that the creatures were clearly not balanced and this caused some battles to go really long or really short!

All in all this was a great illustration of the benefit of polymorphism. It all came together incredibly quickly once I got the syntax down. The inheritance made defining the creature types really easy, and I could readily see how a creature could hold experience as a variable and begin to level up. Further, it would be easy to add an inherited class to each of the 5 predetermined classes so that if a creature is a certain level, it can transform into a new better class with improved stats.