Adventure Game

Alston Listenberger, CSCI 23000

**Purpose**

The purpose of making this game is to demonstrate various techniques learned in CSCI 23000 and to showcase them functioning together to illustrate understanding of the material.

**Project Summary**

The project will be an object-oriented, function-driven, text-based, adventure RPG with the working title “adventure game.” Consisting of a mostly linear structure, progression through the game is achieved through a progression of functions, which define the world and the user’s interactions with it. To progress towards the next location, certain conditions, tracked by Boolean variables, must be met within the current function. The game will have a scalable difficulty, selected upon character creation, and a character class selection system, also selected upon character creation, which will determine which weapons the player acquires throughout the quest. Combat will occur between the player character and enemies determined by the location function. A save and load feature will be implemented for players to resume their previous playthrough. If time permits, differences in weapons will be implemented into the combat system. A cheat system will also be added if time permits.

**Use Case Analysis**

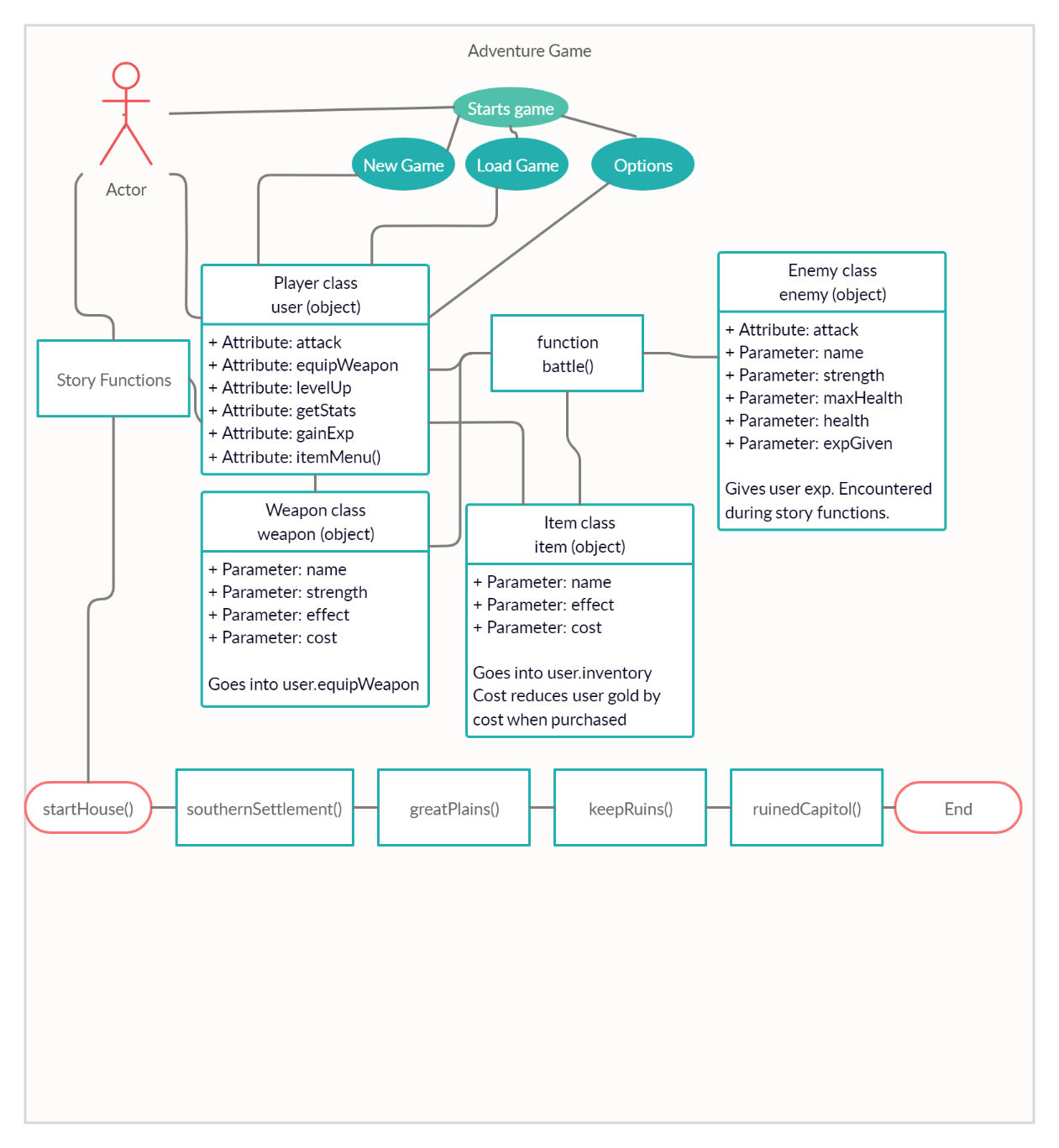
The anticipated final version of this game will have two cases. Once is a player who wants to play the game in either its standard difficulty “medium,” or the “easy,” difficulty to experience the full playthrough in either one or multiple sittings. The other case will be either new players or returning players seeking a challenge by playing on the “hard” or “hurt my feelings” difficulty settings.

**Technologies**

Game written entirely in the Python language. The final release of this game will need to use files in conjunction with the dill module as a save/load feature for object serialization. However, in addition to object serialization, dill serializes the state of the program as a whole. Very useful for my purposes as there are many factors that track progress. The final release will also factor in random number generators provided by the random module to affect gameplay features.

**Data Design**

The game is designed with an object-oriented approach and is primarily about the Player class object, user, and the other objects from the weapon, item, npc, or enemy classes of which the player interacts with. The battle function ties in every object in every class created for the game by pitting the user object against an enemy object. Functions drive the story by affecting the user object controlled by the player. The user’s inventory is an array of objects, which can be accessed by the user at any time. The user’s equipped weapon is an object from the Weapon class. The user object will need to be persistent in order to allow the player to save and load the game. This will be accomplished through the use of the dill module, a modified version of pickle, to write the game state to a document and then read the document upon loading.



**Algorithm/In-depth Description**

**Classes**

The game consists of three “primary” classes of which the user will have significant interaction with and three “minor” classes which will serve to equip the user throughout the game or to determine progression and determine the player’s overall experience.

**Player class**

The player class is an object of which the user assumes control. The Player class has the parameters of name, level, strength, maximum health, health, weapon equipped, amount of gold, inventory, experience points, and progress code. Upon starting a new game, the player will create an instance of the Player class, user, which inherits the parameters of the Player class.

A name will be assigned to the user by the player, along with the character class of user. The character class determines what weapon will be equipped upon starting, as well as what weapons will be found later in the game. The strength, maximum health, and health are determined by the user’s experience points and level. As the user gains experience through combat, their level will increase, increasing strength and health attributes. The amount of gold is variable and tracked as an integer. Through combat, the user will find gold, and through interactions with merchants, the user will lose gold dependent upon what is purchased. The user’s inventory is an array which will keep track of the various items the user acquires. The inventory will function both in and out of combat, allowing the user free access to this array to see and use what items the user has. The progress code of the user object serves to drive the game progression, and aids in saving and loading, so progress can be restarted upon the reading of the progressCode using a function.

The user class also has 6 attributes, other than initialization. They are: attack, equipWeapon, levelUp, getStats, gainExp, and itemMenu.

The attack attribute of Player takes an input of enemy and is used in battle. A random number generator is used to determine success or failure of the attack. If successful, damage is dealt to the enemy class’ health based upon the character’s strength and the equipped weapon’s strength.

The equipWeapon attribute takes a weapon as input and sets the user’s parameter weaponEquip to the input weapon.

The levelUp attribute works in tandem with the experience attribute to increase user’s stats as sufficient experience is accumulated.

The gainExp attribute is called upon victory in a battle and takes an input of enemy and gives the user the value of the enemy’s expGiven parameter. Works in tandem with levelUp to progress the user’s stats.

The getStats attribute displays the user’s max health, strength, and level. This is called upon leveling up and through the user’s in game menu.

The itemMenu attribute is called by the user when accessed through the in-game menu or during combat. The itemMenu iterates through the items in the user’s inventory and displays them along with giving the user the option to use the items. Once used, an item disappears from the inventory and its effect is applied through a function.

**Enemy class**

The enemy class serves to provide the player with battles throughout the game’s progression. Certain battles against the enemy class will be have to be successfully completed to allow progression. Differing types of enemies will be created as objects of the player class, inheriting the parameters of the class and functions of the class. The Enemy class has five parameters: name, strength, maxHealth, health, and expGiven. The name parameter simply specifies the name of the object. The strength parameter determines how much damage is dealt to the player upon a successful attack. The maxHealth parameter is the health possessed by the enemy object upon starting a battle. The health parameter varies based upon a successful attack from user. Upon reaching zero, the enemy object is defeated and the user is awarded exp based upon the enemies expGiven parameter.

There are five instances of the enemy object, and as time permits, more will be added to provide variety. Certain “boss” enemies are modified enemy objects that are tougher to defeat than standard enemy objects and their defeat will be required to progress through the game. The enemy object has one attribute, attack, and this attribute is a mirrored version of the player class attack attribute, to allow the enemy object to attack the user object.

**Item class**

The item class serves the provide the user with tool to make the journey manageable. Consisting of potions in the present iteration, the class has 3 parameters: name, effect, cost.

Name is simply the name of the item. Effect is an integer which determines the amount of user’s health parameter which is healed. Cost is the amount of gold of which the user will have to pay to purchase the item from the merchant.

There are currently 4 objects within the item class: health potions in sizes small, medium, and large, and tears of denial. The tears of denial object cannot be purchased, and operates on a Boolean true/false conditional statement. If the user has tears of denial in the inventory at the time of death, the user will be revived, and the battle can resume. This object is spent in the process and the Boolean is reverted to false.

**Location class**

The location class is currently unused within the game and mostly serves, in its current state, as a repository of information about the individual locations. Total rework of the game’s scripting would need to be done to incorporate this class.

**NPC class**

The NPC class currently contains the four merchants that will be encountered throughout the game, but as time permits, more will be added for the user to acquire weapons from.

The NPC class has two parameters, name and inventory. Of which, the name simply is the npc object’s name and the inventory is what can be given to the user. This feature works in tandem with the Item class to determine pricing.

**Weapon class**

The weapon class exists to provide the user object with a variety of approaches to combat. There are four parameters to this class: name, strength, effect, and cost. Name is what is displayed as the name for the weapon, strength is the boost in combat damage the weapon object provides to the user, effect is a string which describes what unique contribution the respective object provides to combat(displayed in the weapon menu), cost is how much gold the user will have to spend to purchase the item.

There are eight distinct weapons in the game currently, with more to be added as time permits. The sword, dagger, bow, and staff are character class specific starting weapons. Their legendary variants are stronger versions of these weapons, making a total of eight. As time permits, effects will be added to the weapon objects so that each provides a unique experience when dealing with combat.

**Functions**

Functions are the core of the structure of this game. Functions carry out various user tasks such as combat, opening the item menu, equipping a weapon, or using an item, as well as serve as the game’s locations. Locations are simply functions enacting their code, and in this sense the game follows this structure for story progression and area exploration. The functions handle progression through generating progressionCodes, reading progressionCodes, and passing various information between themselves. Below I have categorized the functions into two categories, general functions and story functions.

**General Functions**

These functions handle the baseline functioning of the game.

**setName()**

Allows the player to determine the name of the user object. Returns user.name and global variable “name”

**cheatMenu()**

Allows the user to input cheats into the game to vary the playing experience

**difficultyChoice()**

Allows the player to select a scalable difficulty setting, effecting the user object’s starting stats and equipment. Returns global variable “gameDifficulty”

**chooseCharClass()**

Allows the player to select a “class” of character to play as. Returns charClass.

**setCharClass(charClass)**

Takes an input generated from the chooseCharClass() function and returns user.weaponEquip and global variable weaponEquip.

**setProgressCode(code)**

Takes an input of code from a story function and sets user.progressCode to code. Used for saving and loading the game as well as in tandem with the getProgressCode() and setLocation() functions to determine location. Returns user.progressCode

**getProgressCode(self)**

Takes an input of self and assigns the variable progressCode to value user.progressCode. Returns progressCode.

**setLocation(progressCode)**

Takes the output of getProgressCode as an input and uses it to determine what location the player is placed into.

**newGame(gameDifficulty)**

Takes the output of difficultyChoice as an input and sets the players stats and equipment in the user object of the Player class to a setting determined by the difficulty. Calls the setName() function and assigns the name to global name variable. Calls the chooseCharClass() function and assigns it to variable charClass. Calls the setProgressCode() function to set the progress code to the first area. Calls the setLocation() function to read the progressCode and load the first area. Returns global name and returns progress code.

**optionsMenu()**

Allows the player to access the cheat menu or return to the main menu.

**mainMenu()**

Displays upon starting the game. Gives the user the option to select whether they want to start a new game, load a previous game, access the options menu, or quit the game, by calling their respective functions. If new game is selected, calls difficultyChoice() function and passes the result to newGame(gameDifficulty).

**usePotion(potion)**

Works in tandem with the Player class attribute itemMenu(), which returns potion. Takes an input of potion and increases the user’s health parameter by the effect parameter of the potion. Pops the potion from the inventory after use.

**battle(enemy)**

Takes an input of enemy and starts a battle between the user object and the input enemy object determined either by a directly scripted event, in the case of boss battles, or by random encounter through the explore() function. Utilizes a “keepGoing” while loop. Gives the player the choice between attacking, using an item by calling the itemMenu Player attribute, or fleeing. Uses two random number generators to determine whether the user object or the enemy object attacks first by comparing the two numbers and determining which is greater. Calls the Player attack attribute and the Enemy attack attribute in the order determined by the roll at the beginning of the battle. Fleeing is also determined by a random number generator and the success of flight is contingent upon the results of this roll. A failed flee attempt calls the Enemy class attack attribute without giving the player the chance to retaliate for that turn. Checks user health and enemy object health every turn. If the user reaches zero health, calls the death() function and breaks the loop. If the Enemy object reaches 0 health, calls the gainExp(enemy) function, breaks the loop, and sets the Enemy object back to full health for the next encounter.

**death()**

Prints a death message inspired by Dark Souls and returns the player to the mainMenu() function.

**saveGame(self)**

Saves through object serialization. Takes the user object by taking an input of self and uses dill to write binary to a document. Makes the Player object persistent. Dill takes this a step further and writes the state of the program as well. Ensures all factors in progress are persistent.

**loadGame()**

Uses dill module to read binary the document created in saveGame() and returns the user object and program state. An advantage to pickle, this no longer needs to read the user.progressCode to place the player where they left off.

**explore()**

Uses a random number generator to select a random number. Works in tandem with story functions to provide the user with a variety of events specific to the area of which the user is exploring. Returns exploreNum.

**inGameMenu()**

Provides the user with a menu to access the following features in-game:

user.itemMenu()

user.getStats()

saveGame()

return to game

return to main menu

quit game

**Story Functions**

These functions drive the story and handle the player’s interactions with the world. Utilizes the explore() function to find items and generate random encounters. In addition to random events, also contains various Boolean conditions acting as objectives and other scripted events specific to the area.

**startHouse()**

prints various story statements

uses Player class variable to determine completion of objective for area

**southernSettlement()**

prints various story statements

uses Player class variable to determine completion of objective for area

uses battle() to fight area enemy

**greatPlains()**

prints various story statements

uses Player class variable to determine completion of objective for area

uses battle() to fight area enemies

uses battle() to fight area boss and tracks completion with Player class variable

**keepRuins()**

prints various story statements

uses Player class variable to determine completion of objective for area

uses battle() to fight area enemies

uses battle() to fight area boss and tracks completion with Player class variable

**ruinedCapitol()**

prints various story statements

uses Player class variable to determine completion of objective for area

uses battle() to fight area enemies

uses battle() to fight area boss and tracks completion with Player class variable