Text-based RPG

## Vincent Alexander Seliang

## 2001585562

Simple Adventure

**Section 1**

**Description** This application is, pretty much self-explanatory, a text-based RPG. It follows the style of the one of the earliest RPGs (the 1980’s style) that does not use any, or at least use a minimal amount of graphical game objects. Instead, these types of video games heavily depend on text, obviously, and some newer version of those types of RPGs utilizes the ASCII characters to form interface and HUD (Head-Up Display) for visualization purpose. However, this project takes a different angle of RPG. Instead of making a rogue-like style like the one made in the 1980’s (it is a game genre, where the player is required to explore a certain area in order to complete objective, also it is usually implemented as a subgenre of RPG), it took the path of one of the earlier predecessor of RPG, which is the Dungeons and Dragons series, which were first publish in the early 1970’s. Dungeons and Dragons is a non-digital version of RPG that requires three or more players to participate the game, where one of them is the Dungeon Master, which is the player who sets the setting of the game and the one who decides the fate of the other players based on the number that appears on the dices. However, this game does only require only one player and have a mixture of linear and slight branched progression, inspired from the newer, digital version RPGs. However, the aspect of having multiple choices and the random chance of events’ occurrence still exists and it is also the main aspect of the game. Also, the aspect of battle and role –playing also exist.

**Section 2** 2A

Figure 1, Flowchart of how the game flows

Important notes:

* In order to play the game, players have to follow the instructions displayed on the screen. If there are a list of actions can be done, usually, the player has to input the integer that matches to the choice’s number in order to proceed. Failing to do so might cause the game to fail to operate.

However, if there are only two choices, the player can still input an integer such as one and zero, but the player is also allowed to input words, such as “true” and “yes”.

* The first step, opening, handles the choice of player, either the player can choose to play the game or simply exit the program.
* The second step, player creation, instructs the player to choose their avatar’s race and a job that will be used to initializing the avatar’s status and starting equipment.
* The final step, story, handles on the game’s story flow, where it will allow the player to choose their action with a slight element of Dungeons and Dragons. This includes managing the storyline and processing the battles and loots.

Figure 2.1.1, Class diagrams (Player), simplified

Figure 2.1.2, Class diagrams (Player), UML diagram

|  |
| --- |
| **Player** |
| * sPlayerName : string * nHealth : int = 0 * nMana : int = 0 * nCurrency : int = 500 * nJob : int * nRace : int * nLevel : int = 1 * nEXP : int = 0 * nAttack : int = 0 * nDefence : int = 0 * nMagicAttack : int = 0 * nMagicDefence : int = 0 * nAgility : int = 0 * nLuck : int = 0 * nWeapon : Weapon * nArmor : Armor |
| * Player(job : int, race : int, health : int, mana : int, attack : int, defence : int, magic\_attack : int, magic\_defence : int, agility : int, luck : int, weapon : Weapon\*, weaponnum : int, armor : Armor\*, armornum : int) * Player() * SetStatus(health : int, mana : int, attack : int, defence : int, magic\_attack : int, magic\_defence : int, agility : int, luck) : void * JobIncrementer(: int, health : int, mana : int, attack : int, defence : int, magic\_attack : int, magic\_defence : int, agility : int, luck : int, weapon : Weapon\*, weaponnum : int, armor : Armor\*, armornum : int) : void * SetName(name : string) : void * SetRace(race : int) : void * SetJob(job : int) : void * SetWeapon(weapon : int) : void * SetArmor(armor : int) : void * GetName() : string * GetLevel() : int * GetHealth() : int * GetMana() : int * GetAttack() : int * GetDefenec() : int * GetMagicAttack() : int * GetMagicDefence() : int * GetAgility() : int * GetLuck() : int * GetJob() : int * GetData() : void * Leveling(exp : int) : int |

Figure 2.2.1, Class Diagram (Enemy), simplified

Figure 2.2.2, Class Diagram (Enemies), UML diagram

|  |
| --- |
| **Enemy** |
| * sEName : string * nEHealth : int * nEMana : int * nEAttack : int * nEDefence : int * nEMagicAttack : int * nEMagicDefence : int * nEAgility : int * nELuck : int * nEXPGain : int |
| * Enemy(name : string, health : int, mana : int, attack : int, defence : int, magic\_attack : int, magic\_defence : int, agility : int, luck : int) * Enemy() * eGetName() : string * eGetHealth() : int * eGetMana() : int * eGetAttack() : int * eGetDefence() : int * eGetMagicAttack() : int * eGetMagicDefence() : int * eGetAgilty() : int * eGetLuck() : int * eGetEXP() : int |

Figure 2.3.1, Class Diagram (Weapon), simplified

Figure 2.3.2, Class Diagram (Weapon), UML diagram

|  |
| --- |
| **Weapon** |
| * nWName : string * nWHealth : int * nWMana : int * nWAttack : int * nWDefence : int * nWMagicAttack : int * nWMagicDefence : int * nWAgility : int * nWLuck : int |
| * Weapon(name : string, health : int, mana : int, attack : int, defence : int, magic\_attack : int, magic\_defence : int, agility : int, luck : int) * Weapon() * wGetName() : string * wGetAttack() : int * wGetMagicAttack() : int |

Figure 2.4.1, Class Diagram (Armor), simplified

Figure 2.4.2, Class Diagram (Armor), UML diagram

|  |
| --- |
| **Armor** |
| * sAName : string * nAHealth : int * nAMana : int * nAAttack : int * nADefence : int * nAMagicAttack : int * nAMagicDefence : int * nAAgility : int * nALuck : int |
| * Armor(name : string, health : int, mana : int, attack : int, defence : int, magic\_attack : int, magic\_defence : int, agility : int, luck : int) * Armor() * aGetName() : string * aGetDefense() : int * aGetMagicDefense() : int |

Figure 2.5.1, Class Diagram (InvalidRace), simplified

Figure 2.6.1, Class Diagram (InvalidJob), simplified

Note: class of figure 2.5 and figure 2.6 are used for exception handling.

Figure 2.5.2, Class Diagram (InvalidRace), UML diagram

|  |
| --- |
| **InvalidRace** |
| “Empty class”  Only for exception handling |

Figure 2.6.2, Class Diagram (InvalidJob), UML diagram

|  |
| --- |
| **InvalidJob** |
| “Empty class”  Only for exception handling |

2B

**Classes**

* Class Player This class is used to create and initialize the player’s status, such its race and job, which will be used to give the player’s initial status at level 1.
* Class Enemy This class is created for making the enemies that the player will face along the way. They are also the ones who will provide the player with EXP.
* Class Weapon and Class Armor

These classes are used to create weapons and armors that can be used by the player to aid them during battle. Weapon provides extra damage, while the armor dampens the incoming attack from the elements, such as enemies.

* Class InvalidJob and Class InvalidRace

These classes are used for error handling, the exception feature. Those objects are used for throwing and catching part.

**Functions**

* void BattleSystem(Player\*, Enemy\*)

This function is used to handle whatever is happening during encountering with an enemy. This includes attacking (conventional method), talking to it (pacifist method), using a potion to heal the player, and escape if the player does not or not able to fight it. The parameter pointer to Player and Enemy uses to accept any Player and Enemy objects.

* + void PlayerTurn(Player\*, Enemy\*) and void EnemyTurn(Enemy\*, Player\*)

This nested function of the battle system function is used to handle the calculations of the amount of damage inflicted on either side and calculate the probability of dodging incoming attacks.

* string DisplayArt(ifstream&)

This function is use to both store the text file and displaying the content of the file on the console. It is used only at the opening of the game. The parameter ifstream is used to accept any variable that does input.

* void Opening()

This function is used to emulate and opening menu for like most video games, where is display the ASCII art and the option whether to play the game of exit the game.

* string DisplayWeaponName(int) and string DisplayArmorName(int)

These functions are used to display the names of the equipment because class method, for some reason, could not access the arrays of object’s content. The parameter int is used for accessing the array’s index.

* void PlayerCreation()

This function is used to let the player choose their race and job to initialize their status and their starting weapon and armors.

* void Story()

This function is used to handle all the narrative parts of the video game. It also holds several nested functions that always start with the word Choice such as ChoiceOne() and ChoiceTwo().

* + Choice…()

These nested functions are used to hold the result of the player’s action, results of it could slightly vary.

* int main()

This function is essential for all C++ programs to run and, in this case, it is used to run some functions that are essential for the game to play

* void end()

This function is used for playing the ending part of the game.

**Section 3** 3A

The lessons obtained while making this project:

1. Making function’s prototypes are important because, with the usage of prototypes, the programmer does not need to rearrange to functions into the right position due to it is already declared before.
2. Making the game or program into the MCV (Model Control View) form is actually quite efficient compared to combining them into one huge file because it allows easier implementation for reusability. However, my program only has model and view due to having a control tends to cause multiple errors.

Note: this problem has been resolved by moving only the global functions into a separate header file.

1. It is impossible to directly declare a function and its content within another function, which could cause an error when it is done.
2. Making a get-set method for each class can be very useful, which can help in requiring just a single value from each property of the object.
3. Switch conditional is sometimes more efficient than the if-else conditional for receiving and evaluation an equivalence statement, the == statement.
4. A class method, for some reason, is not able to retrieve a part of object’s properties if the object is a part of an array.
5. Error-handling (exceptions) works almost similar to if-else conditional, if it is taken to its most basic level, but only up to a certain extent.
6. Using the fstream library is really useful, especially when it is used for storing a game object that has a high chance to be changed on a regular basis, such as ASCII arts.

3B

The problems encountered and how it was overcome while making this script:

1. It was, somehow, impossible to access an object’s property if it is a part of an array from the class’ method. In order to overcome that problem, using an external function, the function that is not a class method, is the excellent method for accessing the object’s property in an array.
2. Making and ASCII art is difficult and changing would require a lot of time. This can be solved by using the fstream library, which allows easy modification simply by changing the content of the text file.
3. Sometimes, a function won’t execute because it was not declared before the other function that is going to execute it. However, this can be solved by using function prototyping.
4. Instead of using multiple arrays that hold several values for storing the values of the game objects value, using the class to create an object can be a better alternative because the data within the object are connected, which mean accessing an object property would be more efficient than accessing multiple arrays to find each property.
5. The value of the avatar’s property does not increment, but it got change whenever the method JobIncrementer was called. It got fixed after the addition symbol, +, was added before the assignment symbol, =.
6. When the object was created, it shows an error saying it requires some parameters where it is clearly there is no need of using one. This was fixed by overloading another constructor that does not take any parameter.

Section 4

The script in MCV form