Phase 3 Documentation

1. Menu

The *Menu* class is what our player will see when first running the program. In it, they will be asked to select an option such as starting a new game, loading a saved game, save their current game when given the chance, and the option to quit. The menu will be used when the program is first ran, but also throughout the game whenever the player wants to “pause” their current game and access these options again.

* +showOptions() – This is the main function of the menu class as it will host the other options available to the player.

* +Load() – The load function will allow the player to load a previously saved game and will notify the player if they try to do that when there is no saved file to be loaded.

* +Save() – This function will let the player save their current playthrough so that they can pick up here they left off at any time. We plan to do this by having set checkpoints throughout the game, such as at the beginning and end of each floor, and then writing the players current character data into a text file that is to be read whenever they try to load a game.

* +Quit() – Simple function to let the player quit the game for any reason.

2. Game

The *Game* class is the foundation for our game. It will serve as the portion of our program in charge of running the game, calling event functions when neccessary, and managing player requests. To do this, it will use several member functions of other class, but they will only be called in this class.

* +runGame() – This function is where all the game will be ran. Calls to other Game functions and other class functions will be made here.

* +Intro() - The beginning of the game will begin in this function. Character creation will also occur in this function.

* +makePlayer() – This function will be called in *Intro*. The player will be provided with different options in order to create their character. Origin, name, and stat allocation will take place here.

3. Story

The **Story** class will hold all story narrative for our entire game. It will be composed of functions relating to sections of the story we want to display to the player and will at times ask for their choice in making a story decision that could lead to different events.

It will have many functions so to avoid oversaturating our uml with a bunch of story functions, we've omitted their inclusion in our diagram but the class itself is still present.

****4. Item****

The **Item** interface will be used to implement the various types of items within our game. It contains several functions that will be utilized by all items, as well as a string data member for item name.

* +useItem() – Pure virtual function to be defined by deriving classes as needed.
* +description() – Pure virtual function that serves to inform the player on what the item is.
* +getName() - Function to return the item's name.

****5. Consumables****

The **Consumables** class will be derived from the item interface. Healing items and Agility boosting items will be made from this class. Contains an **effect** and **type** data member to quantify the benefit of the item and determine the item respectively.

* +useItem() - Returns the effect of the item.
* +description() - Tells the player what type of consumable the current item is.

****6. Weapons****

The **Weapons** class will be derived from the item interface. All weapons will be created from this class. It contains a **damage** and **crit** data member to quantify weapon damage and crit values.

* +useItem() - Calls and returns the getDmg and doCrit function to determine total attack value during combat.
* +getDmg() - Returns the damage data member.

* +doCrit() - Determines if crit value should be added to total attack value depending on the player's luck stat.

* +getCrit() - Returns crit data member.

7. Inventory

The *Inventory* class will be used to store any items the player picks up during their playthrough. It contains vectors for consumables and weapons for easy access.

* +checkInv() - Displays all items in inventory and asks player if they want to use anything.

* +useHeals() - Adds HP to user depending on which healing item they choose.

* +useAglBoost() - Temporarily increases agility stat for player depending on which boost item they choose.

* +equipWep() - Allows the player to equip any weapon they have.

8. Event

The *Event* interface will be used to create the different event types for our game. It contains a single pure virtual runEvent() function to be defined by the different event types.

* +runEvent() - Defined by deriving classes.

9. CombatEV

The *CombatEV* class will be used to create all combat events within our game.

* +chckAgl() - Checks who has the higher agility stat between the player and enemy in order to determine who begins the combat event.

* +fight() - Loops combat until the player or enemy dies.

10. LootEV

The *LootEV* class will be used to create all looting events for our game. The two instance in which an object of this class will be created is after a combat event after an enemy has been defeated, and when the player comes along a chest or something similar that contains loot.

* + runEvent() - Generates a random number to determine what item drops after combat or what appears in the chest the player finds.

11. EnemyStats

The *EnemyStats* class will hold all required stats for the enemy. Contains only three of the six stats the player has. Only has a constructor to set stats.

12. CharacterStats

The *CharacterStats* contains all the stats the player can utilize during their playthrough. Contains setters and getters as stats can potentially change throughout the game's duration.

13. EntityData

The *EntityData* class contains basic entity info such as HP, currently equiped weapon, and character name. Both the enemy class and character class inherit this.

14. Character

The *Character* class is where the player's character object will be created from. It contains only a checkInv class and a doCombat class. All other character functionality is handled by inherited data and stats classes.

* +checkInv() - Uses inventory member to access inventory class functions.

* +doCombat() - Menu to be displayed during combat events for player with options to choose from.

15. Enemy

The *Enemy* class is where all instances of enemies in our game will be created from.

* +equipWep() - Function to generate a weapon for enemy depending on what we want them to have.

* +getDrop() - Generates a loot event to get an item drop.

SOLID principle changes

1. Single-Responsibility

* Implementing this principle was one of the more obvious and straightforward changes we could've made. Initially, our *Game* class was called *Level Design* and was in charge of creating events for each level we wanted while also introducing narrative elements to the player. This class which was meant to only host the level was also implementing features of our game that could've been handled by separate classes. To fix this, we created a story class and an event interface to create event types from so that *Game* can simply run the game while its features are handled by separate classes.

2. Open-Close Principle

* This principle came naturally after the Single-Responsibility fix as by creating an event interface, not only did we take some responsibility away from *Game*, we also created a system that allowed for the easy addition of new event types that might be added at a later time. The pure virtual function *runEvent()* can be defined by any event type and anything extra that is needed for that type can be included in its individual class.

3. Interface-Segregation

* The issue that was resolved with the addition of this principle wasn't clear to us at first, but as we began developing our program further we quickly realized how we would have a violation of this principle and how to fix it. Initially we were going to have the *Enemy* and *Character* class both inherit from a *EntityData* and *EntityStats* class. The issue with this was that the enemy only needed three of the six stats that the player would have and thus would be inheriting several data members and member functions that it woulnd't be using. To fix this, we created separate stat classes for the character and enemy so that both only receive the data and functions they require.