  
  
  
  
CS 319

OBJECT-ORIENTEDSOFTWAREENGINEERING  
Design Report  
  
Dread of Evil Wizard  
Group 20  
Afra DÖMEKE

İlteber AYVACI  
Tuğberk TOPALLAR  
Utku OYMAK

Course Instructor: Uğur DOĞRUSÖZ

1. **Introduction**
   1. **Purpose of the system**

Dread of the Evil Wizard aims to provide a unique gaming experience with presenting a timeless concept with a new infrastructure, a brand new implementation for a text based RPG. The main purpose of this game is to save the sister of our main character from the evil wizard. The way to defeat the evil wizard is completing the missions given by the characters in the game and fighting the monsters and become victorious along the way.

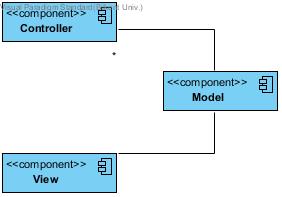
* 1. **Design goals**
* **Usability:** Text based nature of Dread of the Evil Wizard provides a very user friendly and easy to understand game for our players. The explanations regarding storyline, items, game dynamics, creatures, powers and quest goals provided in the help page using plain English. No special interface experience or knowledge about special key combinations are needed since the game is text based but player needs to know the valid commands provided in the help section.
* **Robustness:**  To have a reliable infrastructure Dread of the Evil Wizard especially command interpreter must differentiate valid and invalid commands entered by the user and only execute the valid commands. Command validation depends on the current situation of the character for example attack commands outside of battle mode ignored by default. Also while loading game if the save file is missing or corrupt game starts with default settings since dying results with deletion of the save file game is already compatible with missing files. These properties of File Manager increase the fault tolerance of the game.
* **Extensibility:** Additional characters or quests can be added or modified in the game later on. In order to achieve this, creating a good documentation regarding the inner dynamics and object models of the game was necessary. This documentation would minimize the compatibility problem may occur extending the features of the game.
* **Efficiency:** The throughput is limited to only one move at a time (since the game is turn based) so calculations and responses are not pose a big problem and to make the game more efficient the map itself (objects that are non-interactive or stays still) is not rendered but displayed as an image in the form of background then the objects that are dynamic is rendered by the game on top of this background. This method must be applied for both battle mode and regular game mode.
* **Portability:** The Java implementation provides a platform independent game where every system compatible with running Java software would be able to run this game successfully.

**Trade-offs:** Some trade-offs needed to be decided to implement these design goals that listed below.

* **Efficiency vs. Portability:** By implementing the system with Java we obtain a portable game which would be functional in any system capable of running Java but less efficient compared to beforehand compiled languages like C/C++.
* **Space vs Speed:** Handling the actions with two separate components with Battle Manager and Map Manager instead of just one. This eats up more space but runs more quickly because of the purpose built nature of the separate components.
* **Functionality vs. Usability:** The game provides the player with options about classes, character attributes, status affects etc. This options affect the player’s choices regarding the selection of a character type, a skill or an attack to be used in an arbitrary moment of the game. To simplify the learning curve relating the matter number of these factors must not overwhelm the user but simultaneously present a diversity of options to maximise the gaming experience.

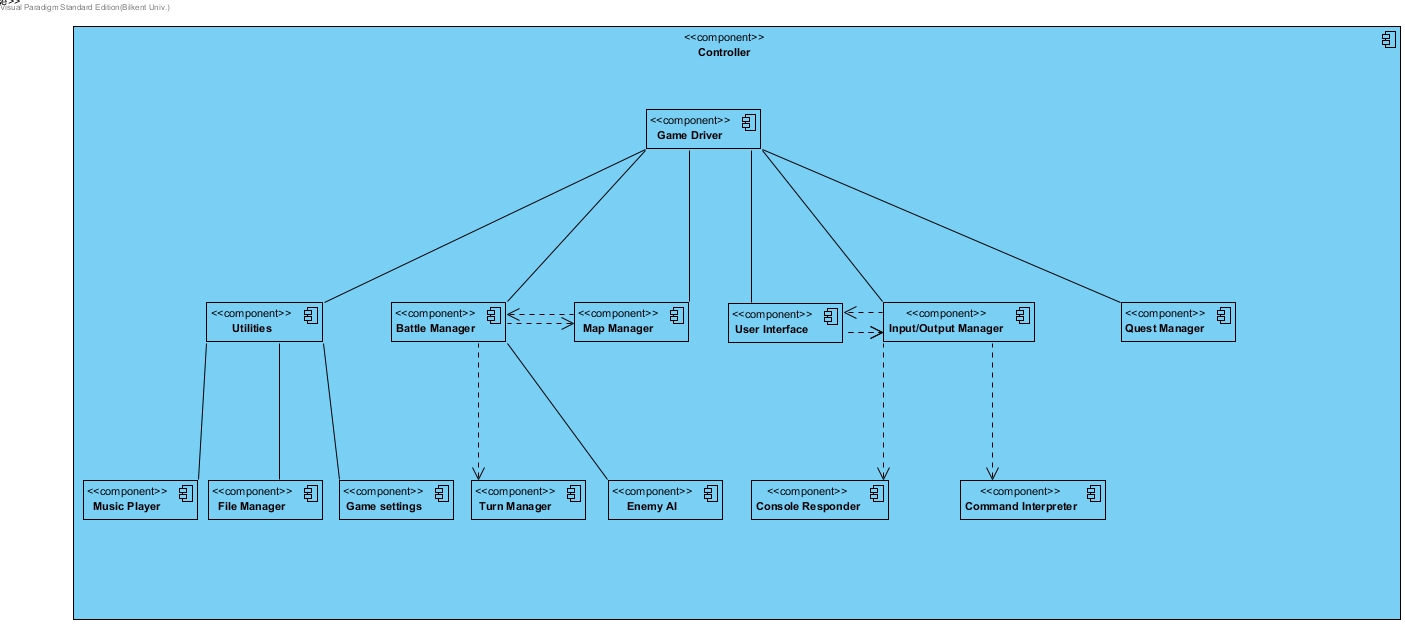
1. **Software Architecture**
   1. **Subsystem Decomposition**

We divided our system into relatively independent Sub-systems to better understand and explain it. For our architectural choice, we went with MVC. This means that we divided the program into 3 big subsystems. Model part hold our game entity objects, View part hold the user interfaces and lastly, the controller part hold the system logic (Business Intelligence). Controller part has its own sub-systems. These sub-systems are responsible for the core features of the game.



**Figure 1**

Figure 1 shows the interactions between model-view-controller parts. Figure 2 Shows the inside of the Controller part. We used 3 layered architecture to design this part. 3 layered architecture gives us better maintainability and flexibility, so if we need to change something or add something to these systems, it will be relatively easy.



**Figure 2**

* 1. **Architectural Styles**
     1. **Model View Controller**

In our project, we chose MVC (Model-View-Controller) as architectural pattern. MVC pattern is a pattern that the main system is divided into 3 systems. One is model, where the data flow is stored at. View is the part where Interaction with User can be done. Controller is useful for our game because:

MVC pattern provides the necessary abstraction and encapsulation for us to hide the game logic from Users. User (Player) only interacts with View part. Also he is aware of the game only as much as developers let him.

Also, MVC pattern creates a head start for the game Dread of Evil Wizard in terms of performance and keeping its implementation stage simple since high coherence has been satisfied.

In this game, the game logic is stored in the Model part. User can interact with the program and ask for change in an object's states, Controller is the part where User's intention is taken into consideration and the object's data (Model) is modified. Model responds to the update calls that are made by controller.

It is preferred to build each Model-View-Controller subsystems in 3 layers with closed architecture. This approach enables us to keep to the implementation maintainable since changing a tier in closed architecture would only cost to change the layer above and below. Also 3 layered architectural design enables developers to implement a cleaner system, everything ordered and the subsystems that do the similar jobs grouped into the same layer.

* 1. **Hardware/Software Mapping**

The game requires only Java software to execute. No special hardware devices are needed to play the game except a regular computer system with a monitor a mouse and a keyboard. Our game does not have high performance demands since the game is 2D and throughput is limited to one turn at a time. Also Dread of Evil Wizard is a single player game so there is no need for an external connection like internet. For gameplay no special keys mapped to special functions since the game is text based. Players will be using predetermined commands to play the game. Any other input than this set of commands will be functionless. Outside the game session regular mouse controls will be used for most functions such as navigation in the menu, load/save operations and so on.

* 1. **Persistent Data Management**

Dread of Evil Wizard uses its own file extension to save/load the games to storage. Reason behind this is to avoid misread or corrupt other files which may be in the same save directory. Also death of the main character results in deletion of previous save files so the game must differentiate which files are relevant. A Database system is not needed to implement this specifications since there won’t be any concurrent file accesses.

* 1. **Boundary Conditions**

In case of data loss or corrupt file game returns to default settings and starts the story from the scratch. To be able to save the game player’s character must be in the town center so any attempt to use the save function of the game will be resulted in a message saying the character must be in the town center. Player will be able to have multiple save files with different characters. And these saves of different characters would not be affect one another. Program does not allow to instantiate multiple games at the same time.

1. **Subsystem Services**

In this section we will talk about our sub-systems and their functions. We used MVC for our basic division of our sub-systems and 3 layered opaque architecture to design the Controller part.

* 1. **Game Driver**

This our core sub-system. This is where our main class will be. Game Driver is our 1st layer of the 3 layer architecture. This will be responsible for communicating any necessary information between other the sub-systems and other parts of the program. We can think this sub-system as the boss of the whole system.

* 1. **Battle Manager**

Battle Manager is responsible for anything battle related in the game. It uses map manager, turn manager and enemy ai to advance the battles. It gets the user info from the game driver and acts accordingly.

* 1. **Map Manager**

Map manager is responsible for anything map related in the game. Map manager will handle all the things related to both the zone maps and the battle arena maps.

* 1. **Turn Manager**

Turn manager is responsible for the turn advancements and turn based effects on the characters. Turn manager keeps track of whose turn it is in the battle and relays this information to the battle manager so that it can do its job.

* 1. **Enemy AI**

Enemy ai will be responsible for controlling the opponents in battles. It will act as a user in a sense but its outputs will directly go to the battle manager. Turn manager will tell whose turn it is, if it’s an opponent’s turn battle manager will ask enemy ai to play the specific characters turn.

* 1. **Utilities**

This subsystem is responsible for managing the functions that are outside the game such as save files, music or settings of the game. Utilities invoke the relevant subsystems to these needs accordingly and the invoked subsystems makes the necessary arrangements about their specified tasks.

* 1. **Music player**

Music Player handles the music will be played or not. When user changes the settings in the menu Game Settings informs the Utilities and Utilities activate Music Player according to this settings. When the player is battling Music Player plays a different song when the player is roaming in the map.

* 1. **Game Settings**

Game Settings manages the player’s preferences about the game. This preferences are whether the music is on or and the difficulty level user is selecting to play the game in.

* 1. **File Manager**

File Manager controls the data management of this game. Player saving or loading is done through this subsystem. The controls about files are made by File Manager. Control conditions for loading are checking if the file exists, if exists checking the extension of the file to make sure the file is game relevant and the last step checking the file for corruption. If any of these steps fails the load operation is failed and Utilities is informed to display the relevant problem to the user. The save case is much simpler only control is about the main character’s location on the map and if so game creates the save file. The death of the player results in deletion of all saves of the player.

* 1. **Input/Output Manager**

Input/Output manager subsystem is on the 2nd layer of controller subsystem. Input/Output manager is responsible for receiving the commands entered by the user and sending the commands to Game Driver in a way that it can be processed. Also this subsytem is responsible for informing the user via displaying appropriate messages on the screen when it is necessary. Input/ Output Manager has relationship with 4 different subsystems.

It has partition relationship with User Interface subsystem so that it can use the services to receive commands entered through User Interface.

It has runtime dependency with Command Interpreter subsystem, so it uses the services of Command Interpreter Subsystem to convert the commands taken from the user as input to a form that Game Driver can understand.

It has run time dependency with Console responder. Console will be ready after the user selects to play game via either selecting Starting a new game or successfully loading a saved game. Input/Output Manager Subsystem will get the commands from user through Console Responder. Furthermore, Input/Output manager will send appropriate messages to the user through Console responder as well.

Lastly, Game Driver has compile time relationship with the Input/Output manager to keep Interaction with user by processing the commands entered by the user In a way to keep the game alive!

* 1. **Command Interpreter**

Command Interpreter is responsible turning the commands entered by the user into appropriate form that Game Driver can understand. Having such a subsystem enables user have a fault tolerance. In other words, if the user enters a command that does not match exactly the same input asked from the user, command Interpreter still can convert it into a form that game will understand. For example, if the user wants to move to the North and enters the command without case sensitivity, or if the user enters “nOrtH” instead of “North”, Command manager can still correct it and send it to Input/Output Manager. Adding such a subsystem increases the robustness of overall system.<s

* 1. **Console Responder**

Console responder subsystem is responsible for giving the necessary response information to the user. When user enters a command and game driver does what it supposed to do, console responder gives an appropriate feed back to the user via writing. Since the only visual feedback user will be getting will be from the map we will rely heavily on the console responder. These feed backs can be exemplified like, giving a description of what is around the user when user says “Look around” or telling how much damage the user did to a monster after he attacked.

* 1. **Quest Manager**

Quest Manager provides services to the Game Driver about Quest functionalities. It is responsible for keeping track of the overall Quests of the Player. In other words, Quest Manager keeps track of the ongoing and completed Quests. This subsystem is responsible for updating the states of each quest that player is still doing. Furthermore, whenever player tries to start a quest, it’s Quest Manager’s responsibility to compute the necessary requirements for completing the quest and updating the quest to “ongoing” status.

1. **Low-level design**
   1. **Object design trade-offs**

**Factory:**

We used this design pattern number of reasons as follows, with single interface type created by factory it is much easier to test. Also it is easier to extend since the code has low coupling. When extending the features of the system existing features do not require additional modification thus ease of extendibility is achieved with factory. We decided that the complication added by this design pattern is worth the positive outcomes.

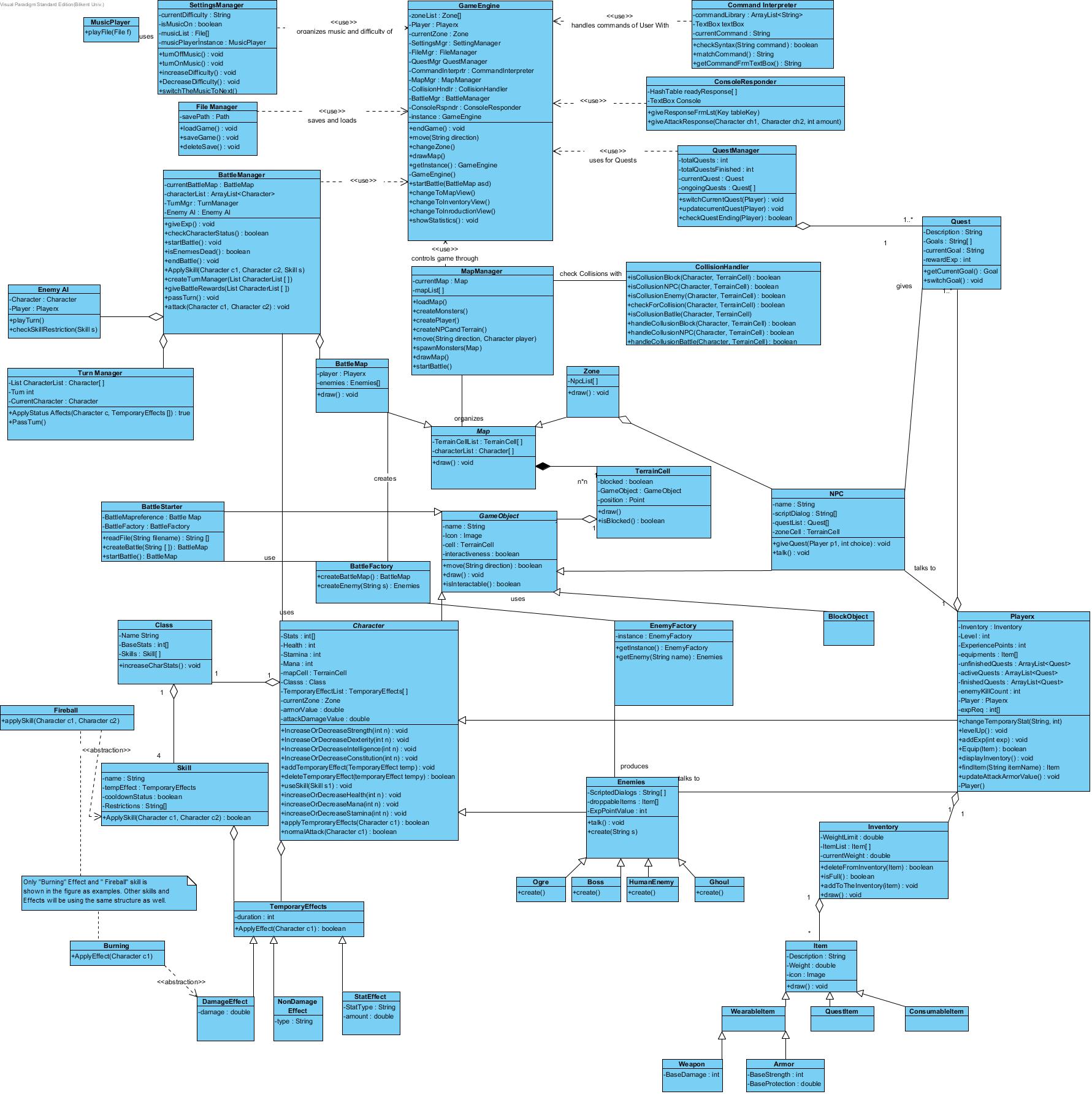
**Singleton:**

In this game several subsystems are working on different aspects of the game but the game as a whole is affected by each and every one of them. At this point the singleton design pattern is the most suitable one for this purpose. Because there must be only one game at a time and the different components must communicate with this only instance of the game. By having only one instance of the game there cannot be a confusion about for the subsystems that which game they are responding to. Also by implementing singleton the game holds the information throughout its lifespan.

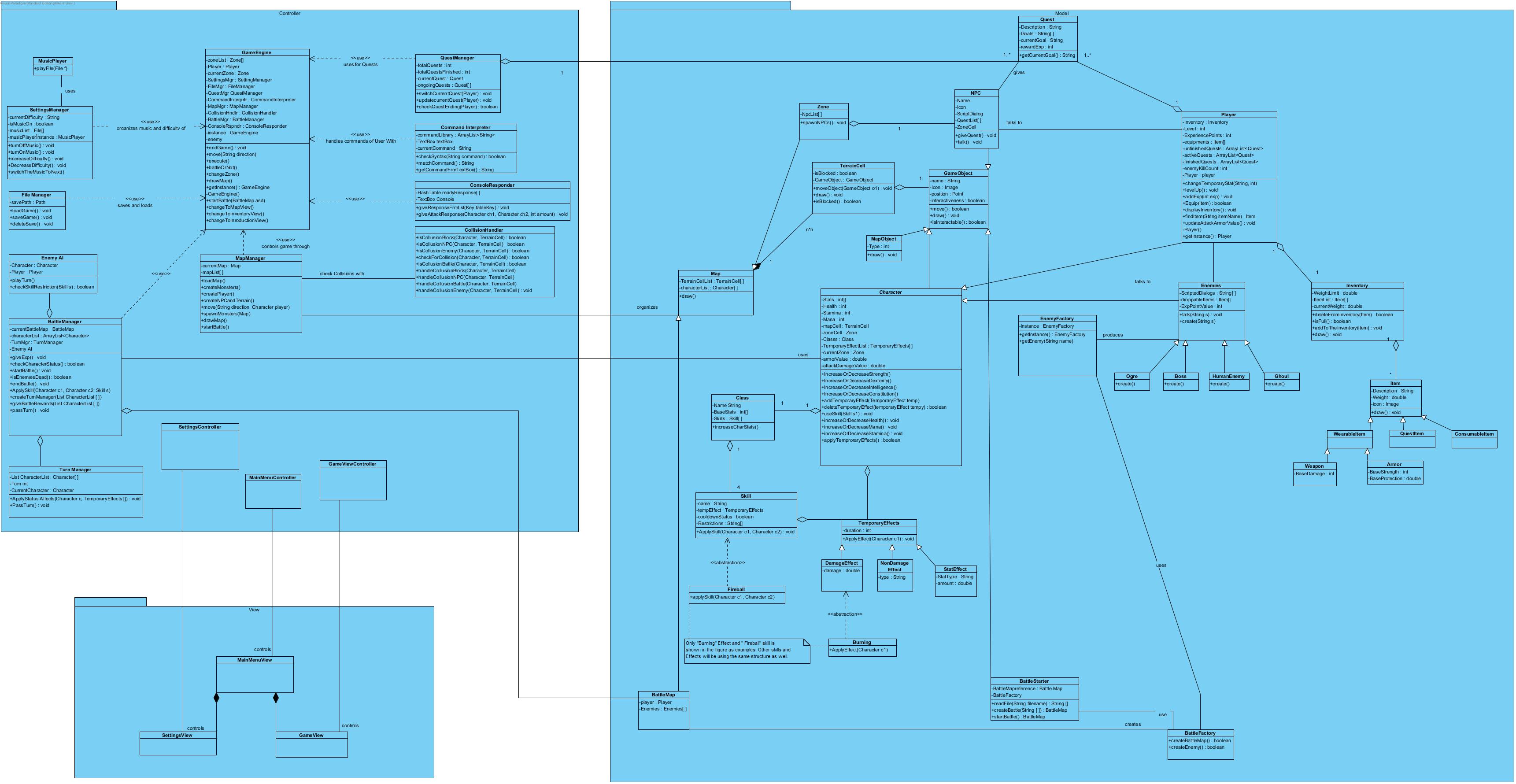
**Abstract occurrence:**

Abstract occurrence pattern is the best way to represent similar objects (i.e. objects that are sharing multiple properties but differ in a slight way) without duplicating common information for every object. This pattern is used because it provides memory benefits by making less duplications for common points.

* 1. **Final object design**

****

* 1. **Packages**

****

* 1. **Class Interfaces**

**Inventory Class**

This class is used for organizing the inventory objects of the player.

**Attributes:**

private intweightLimit: This shows the maximum weight that player's inventories can achieve. Players can not exceed this weight limit.

private Item[] itemList: This shows the list of items that user currently have.

privateintcurrentWeight: This attribute holds the current weight of the player's items to compare the whether the user exceeds the weight limit or not.

**Operations:**

publicbooleandeleteFromInventory(Item): This is used for deleting an item from the inventory. When the user drops an item or gives it to NPC item will be deleted from his inventory with this function.

publicbooleanisFull(): This function shows whether the current items in player's inventory is equal to the weightLimit or not. It operates with the addToTheInventory() function.

public void addToTheInventory(Item): This is used for adding an item to the inventory. If isFull() function returns false, then user can add item. Otherwise, he can't.

public void draw(): This function draws each item on the console that player currently have. When the user calls changeToInventoryView(), which is an operation of GameEngine class, this function will be called and console will show the current inventory.

**Item Class**

This class depends on the Inventory class. It keeps the operations and attributes of each item in the player's inventory.

**Attributes:**

Private String Description: This is used for describing the item's features.

privateint weight: It keeps the weight of items since each item has specific weight. This attribute will be used when calculating the currentLimit and checking weightLimit of inventory.

Private Image icon: It shows the icon of the item in the inventory view in console.

**Operations:**

public void draw(): This is used for drawing the item onto the console when the changeToInventoryView() is called.

**WearableItem Class**

This class depends on the Item class. It keeps the wearable items in the inventory which are Weapons and Armors.

**Weapon Class**

This class holds the attribues of weapons.

**Attributes:**

privateintbaseDamage: This valueholds the damage of the selected weapon. This value is added to the player's attackDamageValue which is an attribute of Charachter class..

**Armor Class**

This class holds the attribues of armors.

**Attributes:**

Private intbaseStrength: This attribute holds the base sterngth value of selected armor. When player use an armor, the str value of the player will increase.

privateintbaseProtection: This attribute holds the base protection value of selected armor. When player use an armor, the protection value of the player will increase.

**QuestItem Class**

This class depends on the Item class. It keeps the quest items in the inventory that are used during a quest.

**ConsumableItem Class**

This class depends on the Item class. It keeps the consumable items in the inventory.

**BlockObject Class**

BlockObject is an object that only blocks the way of the player.

It overrides all the methods of the GameObject Class.

**Character Class**

Private int[] stats: An integer list of size 4. Each integer corresponds to Strength, Intelligence, Dexterity, Constitution. These all have impact on the damage a player can deal and the damage absorption that player can have.

Private int Health: The health of character. More health means harder to kill.

Private int mana: Mana of character. If a character has more mana, he/she can cast more skills.

Private int Stamina: Stamina of character.

Private TerrainCellmapCell: The Cell that character is on.

Private Class Classs: the Class that character belongs to.

Private temporaryEffect[] tempEffectList: current temporary effects that character has at that moment.

Private Zone currentZone: the Zone the character is on.

Private double ArmorValue: the Armor value that player has.

Private double attackDamageValue: The amount of damage that player can hit with normal Attack.

**Operations:**

Public voidIncreaseOrDecreaseStrength/Dexterity/Intelligence/Constitution/Health/Mana/Stamina(int n): These are 7 different methods. But they all add the amount passed as parameter to the corresponding attribute.

Public void addTemporaryEffect(TemporaryEffect s): adds the passed temporary effect to the list of character.

Public boolean deleteTemporaryEffect(TemporaryEffecttempy): deletes the temporary effect passed as parameter from the list of character and returns true. If it is not found on the list, returns false.

Public void useSkill(Skill s1): Character uses the skill passed as parameter.

Public boolean applyTemporaryEffects(Character c1): applies temporary effects stored in the character’s temporary effect list to the character. Returns true if the TemporaryEffects list is not empty. If it is empty it returns false.

Public boolean normalAttack(Character c1): This method uses the main character’s attack point and attacks the character passed as parameter.

**Enemies Class**

Enemy class has the attributes of the enemies. It extends the Character Class.

**Attributes:**

Private String [] ScriptedDialogues: Dialogue list of the Enemy.

Private Item[] droppableItems: List of Items that can be given to the Player if he/she kills the enemy.

Private intexpPointValue: the amount of experience points that the player will get after killing this enemy.

**Operations:**

Public void talk(): Randomly selects one of the scripted dialogues and displays it on the User’s screen.

Public void create(String s): Takes the name of Enemy to be created as parameter and returns the Corresponding enemy.

**There are 4 types of enemies. They are extending the Enemies class. They are :Ogre,Boss,HumanEnemy,Ghoul. They have their own create methods.**

**Player Class**

**Attribute:**

Private Inventory inventory : This is the Inventory of the Player. It contains the items of player.

Private int level: this indicates the level of the player. Player’s enemies will be created according to the player’s level. Like 2 level Player will encounter with enemies with level 1 and level 2.

Private int experiencePoints: this is the amount of experience Points that player has.

Private Item equipments[]: These are the items that are equipped by the player. Equipped Items will be changing the stats of the Player.

Private ArrayList<Quest>unfinishedQuests: These are the quests that Player has not taken yet by not talking to the related NPC.

Private ArrayList<Quest>activeQuests: These are the ongoing Quests of the Player.

Private ArrayList<Quest>finishedQuests: List of finished Quests. These quests are the finished. Number of finished Quests will be used for displaying statistics.

private int enemyKillCount: this is the total number of killed by enemy.

Private Player player: This is the instance of the Player. Essential since we do not want multiple copies of the player. Part of using Singleton Design Pattern.

Private int[] expReq: List of required experience points for all the levels.

**Operations:**

Public void changeTemporaryStat(String,int amount):Changes temporary Stat with the given amount of amount.

Public void levelUp(): Increases the level of the Player if it exceeds the required amount of leveling up.

Public void addExp(intexp): adds the given amount of experience Points to the player’s total experience points.

Public booleanequip(Item I): Adds the item to the equipments if it is in the inventory and returns true. Returns false if it is not in the inventory.

Public void displayInventory(): displays the contents of the inventory.

Public Item findItem(String itemName): Takes the item name and returns the item from the inventory.

Public void updateAttackArmorValue(): Attack and Armor values of the player will be updated each time he/she equips an item. This method is called by equip method.

Private Player(): Private Constructor. Part of Singleton pattern.

Public static Player getInstance(): returns the player instance if it is already created. Creates a new player instance if it is not created yet

**BattleStarter Class**

BattleStarter is a kind of object that if Player collides with it , a battle is engaged. It is a GameObject so it overrides all the methods of the GameObject class. Overriden methods are not listed below.

**Attributes:**

Private BattleMap battleMapReference: This is the reference battleMap that the battle will occur.

Private BattleFactory battleFactory: this is the BattleFactory object that both creates enemies and the corresponding battleMap

**Operations:**

Public BattleMap startBattle(): This method calls for readFile and createBattleMap methods. Then returns the battleMap that created by calling the listed 2 methods.

private String[] readFile(String filename): This method reads the enemy information and the map information stored on the map from the file whose name is passed as parameter. And returns all the information in a String list.

private BattleMap createBattleMap(String [] infoList):This method takes the information list that is returned by the readFile method, calls the methods of battleFactory to get a BattleMap and Enemies. It sends the information of enemies to the createEnemy() method of BattleFactory. Takes an empty BattleMap and fills it with player and the enemies.

**EnemyAI Class**

**EnemyFactory Class**

This factory creates Enemies with given names. This is part of using Factory Method design pattern.

**Attribute:**

Private EnemyFactory instance: instance of EnemyFactory. Part of using Singleton Design Pattern.

**Operations:**

Public static EnemyFactorygetInstance(): If the instance is created, returns the instance. If it is not created, creates a new Enemy Factory instance and returns it.

Public Enemy getEnemy(String s): creates appropriate Enemy and returns it to the caller back.

**BattleFactory Class**

This class creates the Battle Components.

**Operations:**

Public BattleMap createBattleMap(): Creates an empty Battle map and returns it.

Public Enemies createEnemy(String s): creates and returns an enemy with given name.

**BattleMap Class**

BattleMap extends Map class so it has all the attributes that Map Class has. In addition, this map has the instance of Player on the BattleMap and has instances of Enemies that Player will face on the BattleMap as attributes.

**Operation:**

Public draw():overrides the draw method of Map class in a way that BattleMap with appropriate images will be drawn.

**Map Class**

Map is an abstract class. Map is the place that player will be wandering on mostly. There are 2 kinds of Maps. One is BattleMap that player is placed when the player is engaged in a battle. The other one is the Zone, the map that player is on when he/she is not engaged in a battle.

**Attributes:**

Private TerrainCellterrainCellList[]: This is the list of terrain cells on the map. Every map is consisting of terrain cells that hold the game objects on them.

Private Character characterList[]: This is the list of characters that are on the map.

**Operations:**

Public void draw(): This operation draws the Map on the Graphical User Interface.

**GameObject Class**

This is an abstract class.Game Objects are the objects that player can interact or collide with. There are 4 types of GameObjects: Character ,NPC,Block Object and BattleStarter.

**Attributes:**

Private String name: Name of the game object.

Private Image Icon: this is the Image of the object that is displayed on the map.

Private terrainCell cell: this is the cell that game object is on.

Private boolean interactiveness: If a game object is not block object, then it is interactable so it is true. It is false if the object is a block object.

**Operations:**

Public void move(String direction): It used to move the object to the given

Public void draw(): Draws the gameObject on the map with its icon displayed on the corresponding TerrainCell.

Public boolean isInteractable(): returns true if the object is interactable. False if it is not interactible.

**TerrainCell Class**

This class is used for dividing maps into cells and putting each a GameObject to Cells when it is needed.

**Attributes:**

Private Point position: It has the position of the given TerrainCell.

Private boolean blocked: If the cell contains a gameObject it is true, otherwise it is false.

Private GameObject gameobject: this is the GameObject that is on the specified cell. it is NULL, if the TerrainCell is empty and it is not blocked. If it is not NULL, then it means specified TerrainCell is not empty.

**Operations:**  
public void draw(): It draws the TerrainCell on the map.

Public boolean isBlocked(): It returns whether the cell is blocked or not. Returns true if blocked, false otherwise.

**Zone Class**

Zone class extends the Map class. Zone is the map that player can wander if he/she is not in a battle. NPCs, and other staff will be here.

**Attributes:**   
Private NPC Npclist[]: This is the list of NPCs on the zone.

**Operations:**

Public draw(): overrides the draw method of Map class in a way that Zone Map with appropriate images will be drawn.

**NPC Class**

NPC is a class is a class that is responsible for giving quests to the Player and giving rewards to the Player if the player has completed a quest. NPC is a game object. So it overrides the methods of the GameObject Class. The overriden methods are not listed below.

**Attributes:**

Private String name: this is the name of the NPC

Private String ScriptDialogue[]: This is the dialogue List of of the NPC. If the player selects to get a quest, Quest related dialogue will be displayed, otherwise introductory dialogue will be displayed

Private Quest QuestList[]: This is the list of the Quests that the NPC can give. Player can select any of them(if available)

Private TerrainCell zoneCell: This is the cell of the NPC on the map.

**Operations:**

Public giveQuest(Player p1,int choice): the chosen quest on the list will be added to the Player’s ongoing quests. Int Choice will be the index of the Quest on the list.

Public talk(): this will be menu method that displays the introductory dialogue of the NPC. IT will ask for the user whether he wants to take a quest or exit the dialogue. The appropriate action will be taken according to the user’s input. This

**Quest Class**

**Attributes:**

Private Description: Type is string. short description of the quest. Contains information about the quest like how many monster to kill/ where to go to complete the quest etc.

Private Goals: Type is a list of Strings. The goals for completing the quest will be stored here. If the player completes a goal, it will be deleted from the goal List.

Private currentGoal: Type is string. It is the current stage of quest that the player is on.

Private rewardExp:Type is int. The amount of experience point that will be given as reward to the player for completing the quest. This will be taken when the quest I completed by the player.

**Operations:**

SwitchGoal(): This operation passes the current Goal to the next goal on the list.This method will be used when the player passes the current Goal.

**CollisionHandler Class**

Collision Handler class is responsible for checking and handling the collisions that occur when the user wants his/her player to move towards a direction. In such a case, If a player is moving to another direction, move method of Game Engine class is being called and move() method of game engine calls for the methods of CollisionHandler Class. So that if a collision occurs, the appropriate action taken by the game Engine.This class has no attributes, instead it has several operations.

**Operations:**

Public boolean checkForCollusion(Character, TerrainCell): This method takes the Player instance and the Terrain Cell object that player wants to move to( Terrain Cell is a cell of the map that either contains a game object or not). Then checks for the possible collisions that the player may encounter. Player can either collide with a block object,with an NPC or with a BattleStarter object if the player is on the zone map( whenever a player collides with a battle starter object, a battle is engaged between player and his/her enemies). Or player can collide with an Enemy if the player is on a battle map. For all the cases, if the player collides with a game object, handleCollision() methods are called for each object

Public boolean isCollusionBlock(Character, TerrainCell): This method takes the Player instance and the Terrain Cell that player wants to move to. If the TerrainCell ,that player wants to move to, has a Block object on it, it returns true. Otherwise it returns false.

Public boolean isCollusionNPC(Character,Terrain Cell): This method takes the Player instance and the Terrain Cell that player wants to move to. If the TerrainCell ,that player wants to move to, has an NPC object on it, it returns true. Otherwise it returns false.

Public boolean isCollusionBattle(Character,Terrain Cell): This method takes the Player instance and the Terrain Cell that player wants to move to. If the TerrainCell ,that player wants to move to, has a BattleStarter object on it, it returns true. Otherwise it returns false.

Public boolean isCollusionEnemy(Character,Terrain Cell): This method takes the Player instance and the Terrain Cell that player wants to move to. This method is called if the player is on a battle map. If the TerrainCell ,that player wants to move to, has an enemy object on it, it returns true. Otherwise it returns false.

Public boolean handleCollusionBlock(Character, TerrainCell): This method takes the Player instance and the Terrain Cell that player wants to move to. This method is called by checkForCollusion() method if the result from isCollusionBlock() is true. The player will be informed with appropriate message that he/she hit a block. No other action will be taken. So this method returns nothing.

Public boolean handleCollusionNPC(Character, TerrainCell): This method takes the Player instance and the Terrain Cell that player wants to move to. This method is called by checkForCollusion() method if the result from isCollusionNPC() is true. The player will talk with NPC in such a case by calling talk() method of NPC. No other action will be taken. So this method returns nothing.

Public boolean handleCollusionBattle(Character, TerrainCell): This method takes the Player instance and the Terrain Cell that player wants to move to. This method is called by checkForCollusion() method if the result from isCollusionBattle() is true. The player will be engaged in a battle, startBattle()operation of the BattleStarter object will be called.

Public boolean handleCollusionEnemy(Character, TerrainCell): This method takes the Player instance and the Terrain Cell that player wants to move to. This method is called by checkForCollusion() method if the result from isCollusionEnemy() is true. Player will attack to the enemy by using normalAttack() method of the Character.

**QuestManager Class**

This Class is responsible for handling the quests that player is completing/doing and storing the quests that are active/inactive for the player.

**Attributes:**

Private totalQuests : Type is int. It stores the total numbers of quests in the whole gaming process. This information is used when the user wants to see his/her statistics.

Private totalQuestsFinished: Type is int. Used for preparing statistics of the user.

Private currentQuest: Type is Quest. This is the current Quest that player is doing. Whenever user wants to display his/her quest information, the information is taken from this instance.

Private ongoingQuests: this is a list of Quests. These are the quests that player hasn’t finished.

**Operations:**

switchCurrentQuest(Player): This operation returns nothing and switches the quest that player is doing to the next quest on the Quest List. So that whenever user wants to display information about the Quest he is doing, the switched Quest’s information will be displayed.

updateCurrentQuest(Player): updates the Player’s current Quests’ Goal to the next goal on the Quest’s Goal List.

checkQuestEnding(Player): checks the Player’s current Quest’s ending state and deletes the quest from ongoing quest List if the player has completed all the goals in the quest.

**MusicPlayer Class**

This class is responsible for playing the music according to the user’s preferances. It has no attributes, an instance of this class is stored in Settings Manager Class. Whenever a music will be played, the operation of the MusicPlayer instance is used.

**Operation:**

Public playFile(File f): This method returns nothing, takes a Music File that will be played and Plays the music.

**FileManager Class**

This class is used for file processes like saving the game, loading the game.

**Attributes:**

Private savePath: Type is String. This is the path that the game file will be saved at.

**Operations:**

Public loadGame(): This operation loads the game information stored at the file located at the savePath.

Public SaveGame(): This operation uses the Player and Map information stored at the game Engine, stores it in a data file located on the save Path of the hard disk.

Public deleteSave(): This operation deletes the save file located at the save path of the harddisk. This operation is used when the player dies while he/she was playing on hardest difficulty.

**SettingManager Class**

This class is responsible for changing the settings like current music,difficulty level according to the user’s preferances

**Attributes:**

Private currentDifficulty: Type is string.This is the current difficulty that player wants to play the game on.

Private isMusicOn: Type is boolean. This is the attributed that will be used to turn off the music or turn on the music

Private MusicList: Type is File List. Contains the list of music files to be played during the entire game.

Private musicPlayerInstance: This is an Instance of Music Player. Used for Playing the Music.

**Operations:**

Public turnOffMusic(): If the music is on, this method turns off the music. If the music is off, it displays an error message as “The music is already off”.

Public turnOnMusic(): If the music is off, this method turns on the music. If the music is on, it displays an error message as “The music is already on”.

Public IncreaseDifficulty(): This method increases difficulty by 1 level. There are 3 levels of difficulties on this game. If the player is on the highest difficulty it displays appropriate message and does not change the difficulty level.

Public decreaseDifficulty(): This method decreases difficulty by 1 level. There are 3 levels of difficulties on this game. If the player is on the highest difficulty it displays appropriate message and does not change the difficulty level

Public switchTheMusicToNext(): switches the music to the next music in the music list.

**Game Engine Class**

This class controls the flow of game by using the appropriate controllers like Map,File,Settings,Battle,Quest etc. Managers. This class is a singleton class so its constructor is private

**Attributes:**

Private zoneList: This is a List of Zones. Zones will be taken from this list to draw the Maps User will see and act on while he/she is not in battle.

Private Player: This is the main character.

Private currentZone: This is the zone that player is currently displaying and acting on.

Private Instance: This is the GameEngine instance. This attribute is used to avoid creating multiple instances of GameEngine.

SettingsMgr,FileMgr,QuestMgr,CommandInterprtr,CollisionHndlr,MapMgr,BattleMgr,ConsoleRspndr attributes are used for holding the appropriate controllers in the game engine.

**Operations:**

endGame(): If the player chooses to quit, This method is called for ending the current game.

Move(String direction): This method takes a String parameter “direction” that is taken from the CommandInterpreter Class. Moves the player to the given direction if there is no collision between the cell Player located at and the cell Player wants to go. If there exists a collision, than the appropriate actions is taken according to the Type of the collision.

changeZone(): Uses the attribute zoneList to change the Zone player acting on to the next Zone on the List.

drawMap(): draws the Map that Player should be displaying according to the Player’s status. For example if the Player is battling, it draws the battle map. If the Player is on the zone, it draws the Zone Map.

getInstance(): static method to get the instance of Game Engine. It is essential since Game Engine class is Singleton. If the Game Engine is already created, it returns the current Game Engine. If the game engine has not been created yet, it creates a new Game Engine and returns it.

GameEngine(): private constructor for Game Engine class. It is private because Singleton pattern is applied on this class so that there will not be multiple copies of the game Engine.

startBattle(BattleMapasd): It takes the BattleMap from the BattleStarter class and current Map is changed to the taken BattleMap and Battle is started. Game Engine uses the operations of Battle Manager to operate the flow of the battle.

changeToMapView(): changes the player’s view to the map view So that player can switch to the map view from inventory View easily.

changeToInventoryView(): changes the player’s view to the inventory view So that player can display his/her inventory.

changeToIntroductoryView(): Whenever Player is encountered to a battle, this method is called to display the Icon of the Enemy to the Player.

**TemporaryEffect Class**

This class is for representing the temporary effects a character currently affected by. Three classes extends this class DamageEffect, NonDamageEffect, StatEffect.

**Attributes:**

Private duration: duration is an int type it holds the duration of the current TemporaryEffect

**Operations:**

Public Boolean ApplyEffect(Character C) : ApplyEffect takes a character object as a parameter and changes the effect of the character after completion returns a Boolean according to the success of the operation for example an effect cannot be applied two times than ApplyEffect returns false.

**DamageEffect:**

**Attributes:**

Private damage: damage is a double type member and holds the value which damage of the character will multiplied with.

**NonDamageEffect:**

**Attributes:**

Private type: type’s data type is string and hold the values regarding effects such as invisibility, heal etc.

**StatEffect:**

**Attributes:**

Private StatType: holds a string which states that which skill is going to be changed.

Private amount: holds a double value to hold whichever stat will be changed by the amount.

**Skill Class**

Attributes:

Private name: holds the name of the skill as a String

Private tempEffect: holds a value type of TemporaryEffects this is the effect which when activated skill would provide to the character

Private cooldownStatus: holds a Boolean value false while a skill has recently used and a time needed to pass to use the skill again has not passed yet.

Restrictions: holds a string array for the restrictions. These restrictions are the required mana, the required stamina, and a maximum distance which an attack can be effective at the most.

**Operations:**

Public booleanApplyskill (Chaarcter c1, Character c2): returns a Boolean regarding the success of the attack c1 is the attacker and c2 is the receiver.

**Class Class**

**Attributes:**

Private Name: holds a string for the class.

Private baseStats: an int array holding the base stats of that particular Class.

Private Skills: the skills a Class can be use holds an array of type Skill.

**Operations:**

Public Boolean increaseCharStats(): increases the stats according to the race of the character for example some classes have more strength inherently some are more clever

**BattleManager Class**

This is used to operate the battles. It handles everything that happens in a battle.

**Attributes:**

currentBattleMap : This is a reference to the current battle map.

characterList : This is a list that holds the characters in the battle.

TurnManager: An instance of the turn manager. Battle manager uses this to handle turn related issues like pass turn and turn based effects.

EnemyAi: An instance of the enemy ai. Battle manager uses this to handle the moves of the enemies.

**Methods:**

giveExp(): At the end of the battle this method goes through the character list and gives exp respect to the enemies in the battle.

checkCharacterStatus() : It check whether the player is dead or not.

startBattle() : Initiates the battle by making necessary assignments to the character list and current battle map.

isEnemiesDead() : Checks whether the enemies in the battle are dead or not.

endBattle(): It ends the battle.

ApplySkill() : This is called after user enters a command to use a skill or an enemy uses a skill. This method calls the apply skill method of the characters.

createTurnManager(): Creates a turn manager object instance.

giveBattleRewards(): Gives the battle rewards to the player after the battle is concluded.

passTurn(): This method lets the turn manager know that turn has ended.

**EnemyAI Class**

This class is used to operate the enemies in the battle. When turn manager passes the turn to an enemy, that enemy’s reference is sent to here and then enemy ai decides what to do for the turn of this enemy.

**Attributes:**

Character: This is a reference to the current enemy character.

Player: A reference to the player. Enemy ai uses this to get information about the players current status and make a better battle decision.

**Methods:**

playTurn(): Handles the turn of the current enemy character. Decided what to do for this turn.

checkSkillRestrictions(): If the enemy ai decides to use a skill, it uses this method to check whether it can use that skill or not.

**Turn Manager Class**

This is the class that handles the turn advancements and everything turn related in battles. Every time a new battle starts a new turn manager is created.

**Attributes:**

CharacterList: Holds the character list in the initiative order.

Turn: Turn number.

CurrentCharacter: Hold a reference to the current character. Every time a turn passes this cycles through the character list.

**Methods:**

ApplyStatusEffects(): When a new turn starts, this method applies the necessary temporary effects to the current character.

passTurn(): Passes the turn.

**CommandInterpter Class**

This class is used for taking commands that users have written in the text box, process the commands and turn them in a format that game can understand.

**Attributes:**

private commandLibrary: This used for matching commands from user with the predetermined commands.

public textBox: This is a reference to the text box that user writes in.

private currentCommand: We save the commands into this variable in order to easily process it

**Operations:**

public boolean checkSyntax(String command): This is used for checking the syntax of the command. This checks the first word and uses the matchCommand function.

public string matchCommand(): This is where we match current command with the command library and disect the command into variables.

public String getCommandFrmTextBox(): This is used for get command from text box.

**ConsoleResponder**

This class is used for giving responses to user via console (uneditable textBox). Since we are not using visuals for our game, this is where the user gets most of his information.

**Attributes:**

private readyResponse: This is where we store the predetermined responses that we will give to the user. We use a hash map to find the reponse we want easily.

private Console: This is a reference to the text box that user reads information from.

**Operations:**

public giveResponseFrmList(): This finds the necessary response from the response list and prints it into the text box.

Public giveAttackResponse(): This is used for battle information. When an attack happens this prints the damage values and enables user to know how much damage done and how much mana/hp/stamina left after the attack.