Bilkent University

CS 319 Spring 2020

Iteration 2 - Project Design Report

Group 1B-SS

Can Cebeci	21703376
Cem Cebeci	21703377
Gizem Karal	21703094
Gökberk Boz	21602558
Sena Sultan Karataş	21604078
Mehmet Ali Altunsoy	21702531

Contents

1. Introduction

1.1 Purpose of the System

Slay the Spire is a roguelike deck building game where the goal is to follow a map through three levels while fighting various monsters and collecting or upgrading the deck.

1.2 Design Goals

1.2.1 Modifiability

Since the fact that the game has a wide variety of classes, relics and cards constitutes a large part of its attraction, it is paramount that the system we build can support the addition of new cards, classes, relics and mechanisms.

1.2.2 Understandability

If the player had to spend hours to figure out how the game is played, the enjoyability of the game would decrease significantly and it would lose much of its attraction.Therefore, it is very important that the game and its interface are easily understandable. We value understandability over functionality and therefore exclude complicated card effects and mechanisms.

1.2.3 Low Cost

Since this is a term project and the time is very limited, it's crucial that the project can be developed in a few months which requires the project to be low cost. Also, the target customers are not multi-billion dollar companies but indie gamers, a low cost project is more likely to succeed in this environment.

2. Subsystem Decomposition

The system is decomposed into three subsystems. The Navigation Subsystem handles the user's navigation through the menus in the game and manages the saved persistent state. The Run Management subsystem controls and modifies the run state according to the player's actions and the Combat Management subsystem controls what happens in a combat.

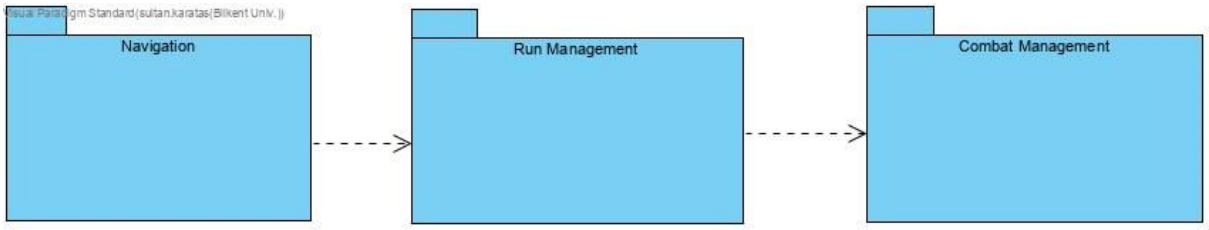

Figure 1: UML representation of the subsystem decomposition

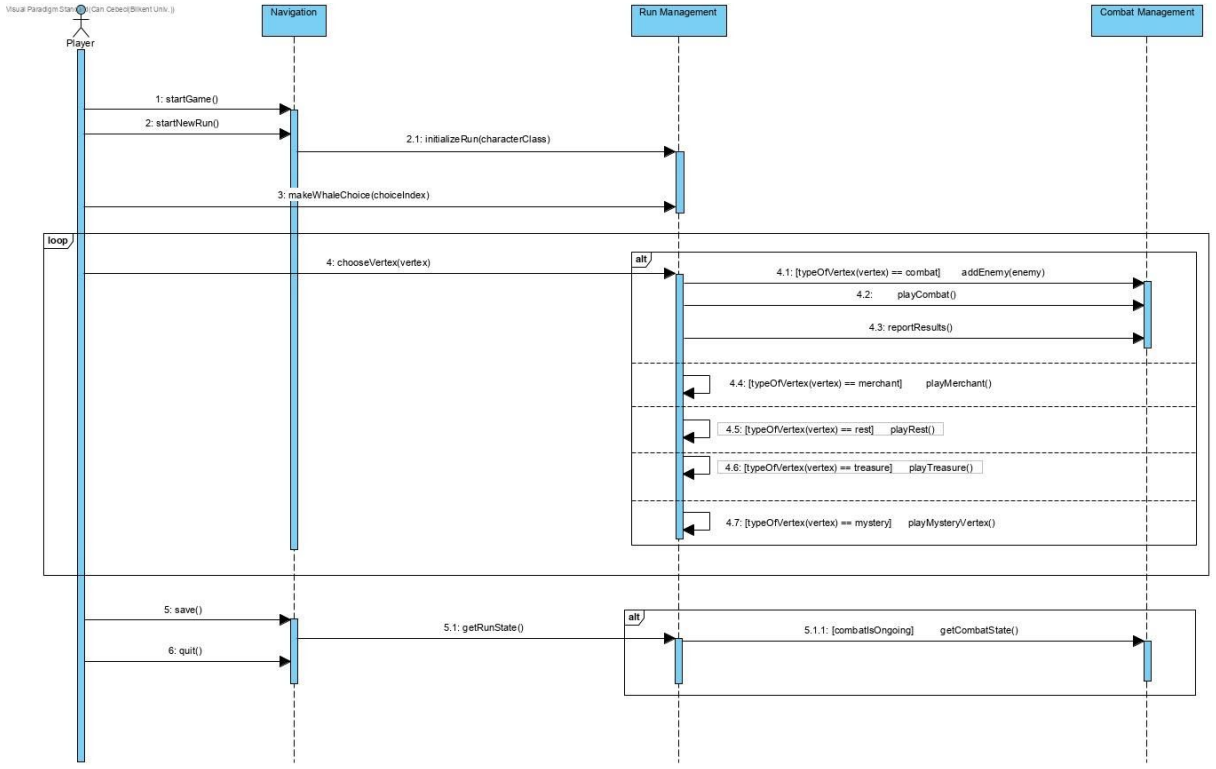

Figure 2: Sequence diagram depicting a generic instance of how the subsystems interact when the game is being played.

3. Hardware / Software Mapping:

The mapping is trivial in our design. The whole product is a single java application that runs on a single computer. The application runs on the Java Virtual Machine. So, any computer with JVM installed is a viable platform for an independent instance of the game to run.

4. Access Control:

The system only has one type of user, which is the player. Moreover, the game does not require any collective remote storage on a server or any maintenance by an authority. Thus, the player can access all stored data and all functionalities. While this does enable the player to modify their game data, this is not a concern of the design since security is not an essential design goal for the system.

5. Persistent Data Management:

The system needs to store persistent data to provide the two following facilities:

- Save and exit the game at any point and load the game when the application is launched again.
- Maintain a set of achievements, classes, cards and relics unlocked by the player across multiple runs.

The two sets of information are stored in separate files. The mechanism providing each facility is described in the following sections.

5.1 Saving a game:

To provide the first functionality, the system needs to save the state of a run. To save the state, the following information has to be stored:

- Contents of the deck
- The amount of gold the player has
- The relics the player has
- The amounts of maximum HP and current HP the player has
- The potions the player has and the maximum number of potions they can carry
- The map and the player's current position
- The current cost of using the card removal service
- If a combat is ongoing, the state of the combat, which includes
    - A set of enemies, each having
        - An amount of current HP and maximum HP.
        - A set of status effects (described in detail in section 1.4.4)
        - An intent (described in detail in section 1.4.5)
    - The amount of energy and maximum energy the player has.
    - Three piles of cards, which are
- The draw pile, which is the pile the player draws cards from.
- The hand, which are the cards that are currently playable.
- The discard pile, which is the pile that contains cards recently played or discarded.
- A set of status effects the player is afflicted by.

The system provides this functionality as follows:

1) The player presses the “Save and Quit” button, which invokes the save() method of the Navigation Subsystem.
2) The Navigation Subsystem calls the getRunState() method of the Run Management Subsystem.
3) If a combat is ongoing, the Run Management Subsystem calls the getCombatState() method of the Combat Management Subsystem.
4) All of the state is returned as a String to the Navigation Subsystem, which writes it all into a single save file.

5.2 Saving the achievements:

All of the achievements in the system, including the class/cars/relic unlocks are identified by an ID number unique to the achievement. A file stores the ID’s of all of the unlocked achievements. If the ID is present in the file, the achievement is unlocked. Otherwise, it is not.
This file is possibly updated at the end of each run. If there is any newly-unlocked achievement, it is written to the file by the Navigation Subsystem.

6. Subsystem Services

The game will be played offline with a desktop and MVC architecture is used. MVC architecture separates the application into three components: Model, View and Control

6.1 Navigation Subsystem

Navigation subsystem includes the front stage operations of the game. It is responsible for directing the user to the desired run subsystem with requested game options. Users may want to play a new game with creating a new character or resume their previous game. Users also may change the game settings and display achievements if they want to. If a user wants to exit the game navigation subsystem, save the data to a .txt file and close the program.

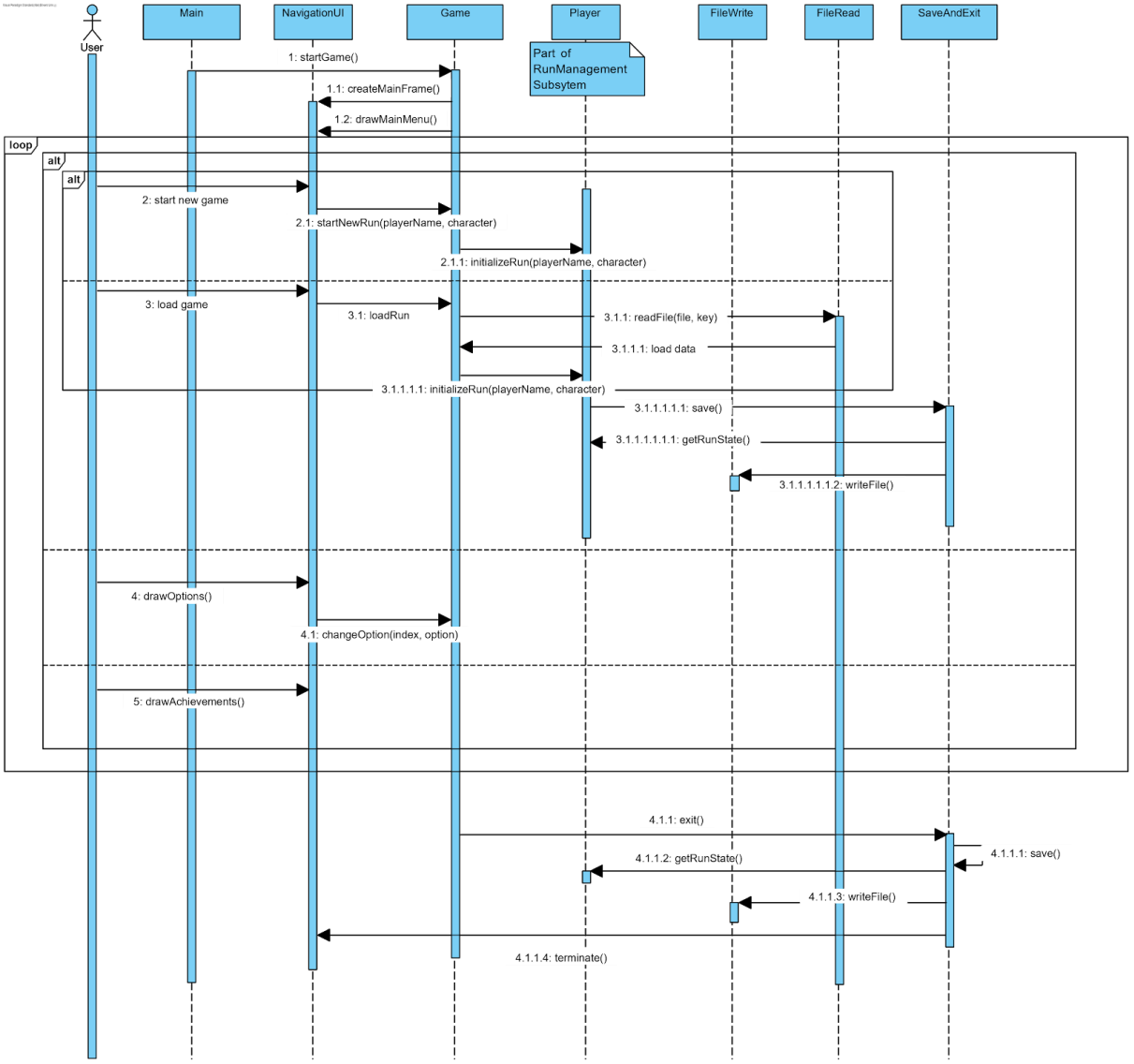

Figure 3: Sequence Diagram of the Navigation Subsystem, describes the flow of a typical execution of the program.

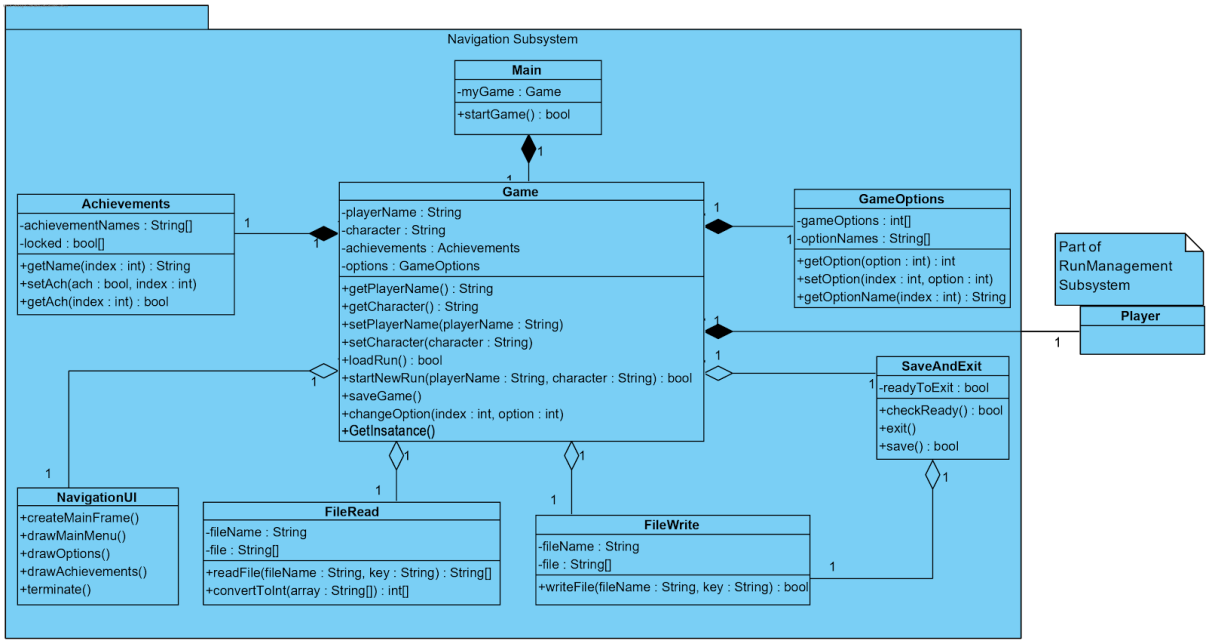

Figure 4: Class Diagram of the Navigation Subsystem.

6.1.1 Main Class:

The Main class starts the game.

Attributes:

- private Game myGame: Current game of the program.

Methods:

- public bool startGame(): Starts the game.

6.1.2 NavigationUI Class:

The NavigationUI class paints the game.

Methods:

- public bool createMainFrame(): Draws the main frame of the game.
- public bool drawMainMenu(): Draws the main menu.
- public bool drawOptions(): Draws the options menu.
- public bool drawAchievements(): Draws the achievements.
- public void terminate(): terminates the frame.

6.1.3 Game Class:

The Game class contains the player information, game achievements and game options. It may load the current run or start a new run.

This class is designed considering the singleton design pattern since at any instant, there will not be more than one game instance. Making Game a singleton object lets us access the existing Game instance in all of the classes with the getInstance() method rather than having to pass references to an arbitrary Game instance to almost all classes.

Attributes:

- private String playerName: Name of the player.
- private String character: Character of the player.
- private Achievements achievements: Achievements of the game.
- private GameOptions options: Options of the game.

Methods:

- public void setPlayerName(String name): Sets the player name.
- public String getPlayerName(): Gets the player name.
- public void setCharacter(String char): Sets the character of the player.
- public String getCharacter(): Gets the character of the player.
- public void loadRun(): Loads the current game and starts.
- public void saveGame(): Saves the current game.
- public void startNewRun(String playerName, String character): Starts a new game with specified name and character.
- public void changeOption(int index, int option): changes the desired option with desired value.

6.1.4 GameOptions Class:

The GameOptions class contains the game options in an integer array for both on/off and interval needed options.

Attributes:

- private int[] gameOptions: Log of game options in an int array.
- private String[] optionNames: Name of game options array.

Methods:

- public int getOption(int option): Gets the option in wanted index.
- public void setOption(int index, int option): Sets the option in wanted index.
- public String getOptionName(int index): Gets the specified option.

6.1.5 FileRead Class:

The FileRead class reads the desired file with a specified key to clear which specific data needed.

Attributes:

- private String fileName: Name of the file.
- private String[] file: String array of the file.

Methods:

- public String[] readFile(String fileName, String key): Read the wanted file and return String array.
- public int[] convertToInt(String[] array): Convert String array to int array.

6.1.6 FileWrite Class:

The FileWrite class writes the desired file with a specified key to clear which specific data needed.

Attributes:

- private String fileName: Name of the file.
- private String[] file: String array of the file.

Methods:

- public bool writeFile(String fileName, String key): Write the wanted file and return boolean whether it is successful or not.

6.1.7 Achievements Class:

The Achievements class contains the achievements logs in a string and boolean arrays to display the name of the achievement and to check if the achievement is locked or unlocked.

Attributes:

- private String[] achievementNames: Name of the achievements.
- private bool[] locked: Achievements are locked or unlocked.

Methods:

- public String getName(int index): Get the achievement name.
- public void setAch(bool ach, int index): Set the achievement lock info.
- public bool getAch(int index): Get the achievement lock info.

6.1.8 SaveAndExit Class:

The SaveAndExit class helps the game class to save and exit the game. It saves the current runs info, achievements and game options in a .txt file.

Attributes:

- private bool readyToExit: Is everything saved and ready to exit

Methods:

- public bool controlReady(): Control everything and return if it is safe to exit.
- public bool save(): Saves the current game and returns if it is successful or not.
- public void exit(): Close the program.

6.2 Run Management Subsystem

This subsystem makes the connection between Navigation and Combat subsystems by maintaining the game objects interact with each other. During the flow of the game, it updates the state of run. This subsystem is made up of several classes, which are the game's fundamental elements such as: Player, Map, Vertex, Deck, Card, Combat classes. This subsystem generates the vertex chosen and it implements the actions as viewing deck by controlling the general model of the game. Combat and it's connected classes will be explained in 6.3 Combat subsystem. After the 6.3 section is invoked, Run Management Subsystem updates parameters which are HP MaxHP, Deck ,Foes, Relics, Potions. This subsystem manages the game cycle by choosing the next vertex. What change occurs in those vertices: Merchant, Treasure and Rest, is updated generally in the game.

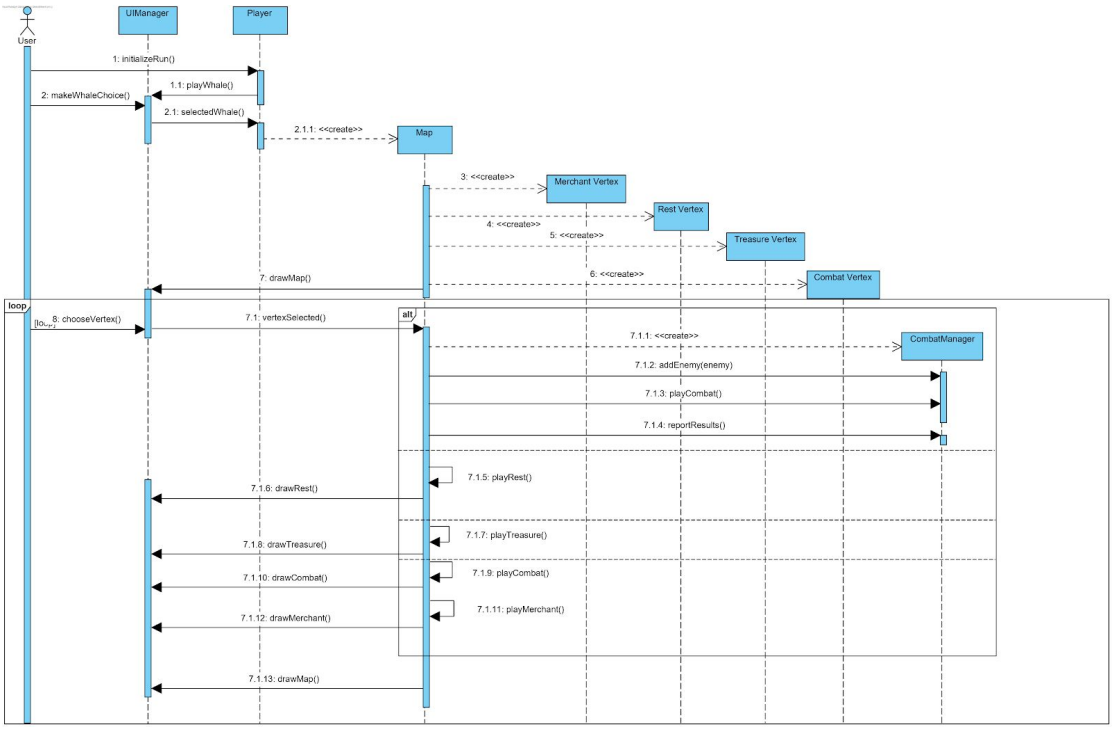

Figure 5: Sequence Diagram of the Run Management Subsystem

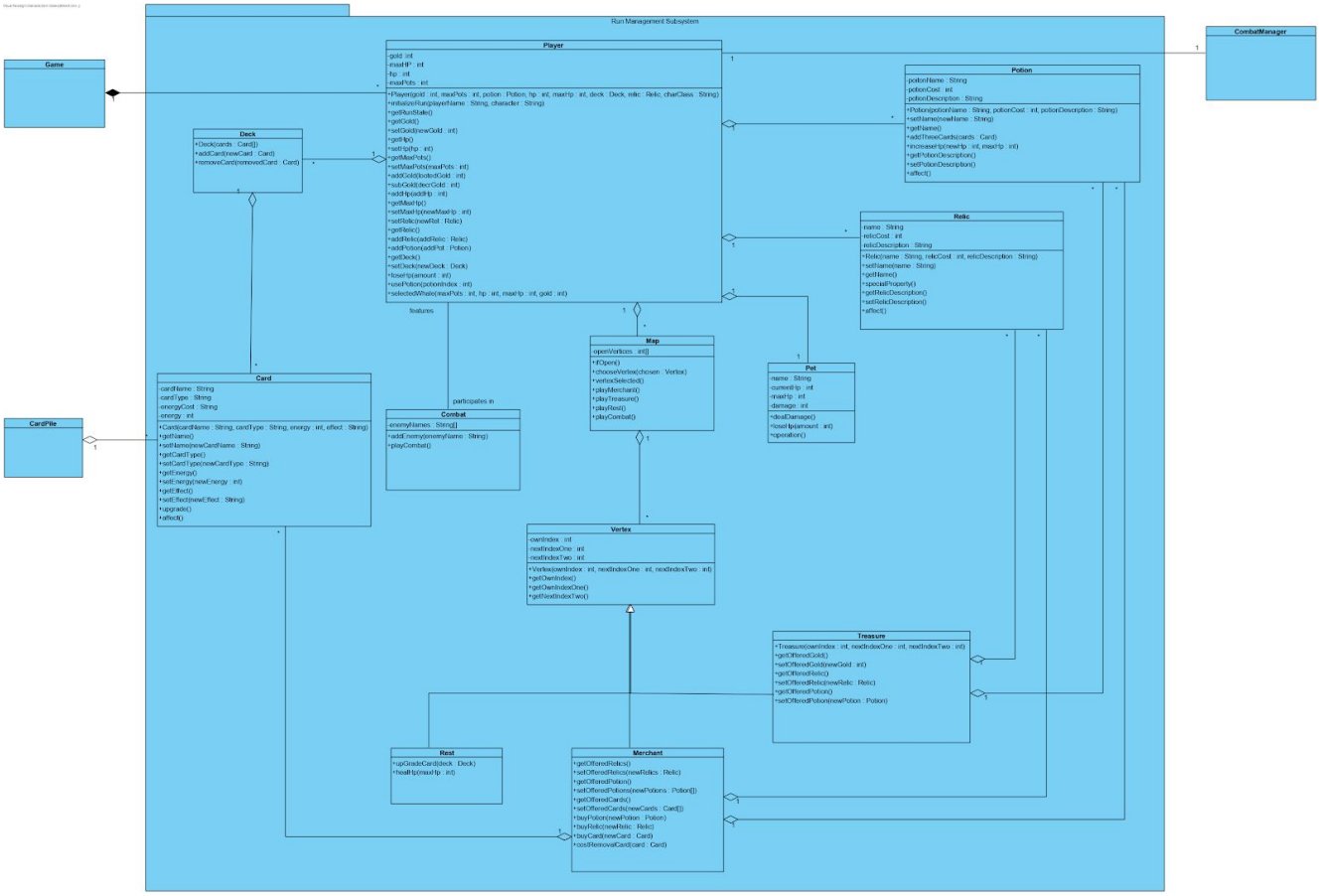

Figure 6: Class Diagram of the Run Management Subsystem

6.2.1 Player Class:

The Player class contains the properties of the player and its actions. This class has also Deck, Potion, Relic classes as attributes.

Attributes:

- private int maxPots : Holds the max pots of players.
- private int hp: Hp that player has
- private int maxHp: maxHP player has
- private int gold: Gold amount player has.

Methods:

- public Player(int gold, int maxPots,Potion potion, int hp, maxHp, Deck deck, Relic relic, String charClass): constructor of player.
- public bool initializeRun(String playerName, String character): Takes over from Navigation.
- public String getRunState(): Reports back the information of current Run State.
- public int getGold(): enables to get Gold
- public void setGold(int newGold) : enables to set Gold
- public void addGold(int lootedGold): adds gold to players gold amount
- public void subGold(int decrGold):subtracts used amount of gold
- public int getMaxPots(): enables to get maxPots
- public void setMaxPots(int newMaxPots): enables to set maxPots
- public int getHp(): enables to get HP
- public void setHp(int newHp): enables to set newHP
- public void addHp(int addHp): adds HP to player's HP
- public void loseHp(int amount): subtracts HP from the player's total.
- public int getMaxHp(): enables to get MaxHP
- public void setMaxHp(int newMaxHp): enables to set MaxHP
- public void setRelic(Relic newRel): enables to set Relic
- public Relic getRelic(): enables to get Relic
- public void addRelic(Relic addRel): adds Relic to player's Relic
- public void addPotion(Potion addPot): adds Potion to player's potion
- public Deck getDeck(): enables to get Deck
- public void setDeck(Deck newDeck): enables to set Deck
- public void usePotion(int potionIndex):Consumes potion and applies effects
- public void selectedWhale(int maxPots, int hp, int maxHp, int gold): Applies the attributes of chosen whale to the player.

6.2.2 Potion Class:

Potion class represents Potion Object in the game, besides its basic attributes, it has the affect() method that carries the in-game effect of potion.

Attributes:

- private String potionName: name of the potion
- private int potionCost: cost of potion
- private String potionDescription: description that shows potion properties

Methods:

- public Potion(String potionName, int potionCost, String potionDescription): constructor of Potion class
- public void setName(String newName): enables to set name to potion
- public String getName(): enables to get name of potion
- public void addThreeCards(Card []cards): adds three cards to deck
- public int increaseHp(int newHp, int maxHp): increases HP of the player
- public String getPotionDescription(): enables to get description of potion
- public void setPotionDescription(): enables to set description of potion
- public void affect(): other effects of potion will be determined later

6.2.3 Relic Class:

Relic class represents Relic Object in the game, besides its basic attributes, it has the affect() method that carries the in-game effect of relic.

Attributes:

- private String name: name of the relic
- private int relicCost: cost of relic
- private String relicDescription: description that shows relic properties

Methods:

- public Relic(String name, int relicCost, String relicDescription): constructor of Relic class
- public void setName(String name): enables to set name of relic
- public String getName(): enables to get name of relic
- public void specialProperty():
- public String getRelicDescription(): enables to get description of relic property description
- public void setRelicDescription(): enables to set description of potion property description
- public void affect():

6.2.4 Map Class:

The Map class contains the properties of map.

Attributes:

- private Vertex[] allVertices: an array of vertices that Map class totally has
- private int[] openVertices: shows the open vertices that the player can choose to continue

Methods:

- public boolean ifOpen(): shows if the vertex is open
- public void chooseVertex(Vertex chosen): vertex chosen by the player is conveyed.
- public String vertexSelected(): Takes information from UI and directs to chosen vertex play.
- public void playMerchant(): Keep the information between UI and Merchant activities.
- public void playTreasure(): Keep the information between UI and Treasure activities.
- public void playRest(): Keep the information between UI and Rest activities.
- public void playCombat(): Keep the information between UI and Combat activities.

6.2.5 Vertex Class:

The Vertex class is the general class of different types of vertices, since all vertices have different attributes we keep only our own and next vertex index.

Attributes:

- private int ownIndex: index of the vertex
- private int nextIndexOne: possible index one
- private int nextIndexTwo: possible index two

Methods:

- public Vertex(int ownIndex, int nextIndexOne, int nextIndexTwo): constructor of Vertex class
- public int getOwnIndex(): enables to get ownIndex
- public int getNextIndexOne(): enables to get nextIndexOne
- public int getNextIndexTwo(): enables to get nextIndexTwo

6.2.5.1 Treasure Subclass:

The Treasure class is inherited from the Vertex Class.

Attributes:

- private int offeredGold: offered golden by the class to the player
- private Relic offeredRelic: offered relic by the class to the player
- private Potion offeredPotion:offered potion by the class to the player

Methods:

- public Treasure(int ownIndex, int nextIndexOne, int nextIndexTwo): constructor of Treasure class
- public int getOfferedGold(): enables to get offered gold
- public void setOfferedGold(int newGold): enables to set offered gold
- public Relic getOfferedRelic(): enables to get offered relic
- public void setOfferedRelic(Relic newRelic): enables to set offered gold
- public Potion getOfferedPotion(): enables to get offered potion
- public void setOfferedPotion(Potion newPotion): enables to set offered potion

6.2.5.2 Merchant Subclass:

The Merchant class is inherited from the Vertex Class.

Attributes:

- private Relic[] offeredRelics: array of offered relics to player
- private Potion[] offeredPotions: array of offered potions to player
- private Card[] offeredCards: array of offered cards to player

Methods:

- public Relic[] getOfferedRelics(): enables to get offered relics
- public void setOfferedRelics(Relic[] newRelics): enables to set offered relics
- public Potion[] getOfferedPotions(): enables to get offered potions
- public void setOfferedPotions(Poiton[] newPotions): enables to set offered potions
- public Card[] getOfferedCards(): enables to get offered cards
- public void setOfferedCards(Card[] newCards): enables to set offered cards
- public void buyPotion(Potion newPotion): enables to buy new potion
- public void buyRelic(Relic newRelic): enables to buy new relic
- public void buyCard(Card newCard): enables to buy new card
- public void costRemovalCard(Card card): removes card from the deck in exchange of gold

6.2.5.3 Rest Subclass:

The Rest class is inherited from the Vertex Class.

Attributes:

Methods:

- public void upgradeCard(Card upgradeCard): upgrades a card
- public int healHP(int maxHP): increases HP of player

6.2.5.4 Combat Subclass:

The Combat class is inherited from the Vertex class. After preparing the combat arena, invoke the CombatManager.

Attributes:

- private String[] enemyNames: array of names of enemies.

Methods:

- public void addEnemy(String enemyName): Calls the method of the same name in CombatManager.
- public void playCombat(): Calls the method of the same name in CombatManager.

6.2.6 Deck Class:

The Deck class contains the properties and methods of the deck. As an attribute holds an array of cards.

Methods:

- public Deck(Card[] cards): constructor of Deck class
- public void addCard(Card newCard): adds new card to deck
- public void removeCard(Card removedCard): removes card from deck

6.2.7 Card Class:

The Card class contains the properties and methods of the card.

Attributes:

- private String cardName: name of the card
- private String cardType: type of the card
- private String effect: effect of the card
- private int energy: energy of the card

Methods:

- public Card(String cardName, String cardType, int energy, String effect): constructor of Card class
- public String getName(): enables to get name
- public void setName(String newCardName): enables to set name
- public String getCardType(): enables to get card type
- public void setCardType(String newCardType): enables to set card type
- public String getEnergy(): enables to get energy
- public void setEnergy(int newEnergy): enables to set energy
- public String getEffect(): enables to get effect of the card
- public void setEffect(String newEffect): enables to set effect of card
- public void upgrade(): upgrades the card
- public void affect():

6.2.8 UI Class:

Methods:

- drawWhale(): First screen after Run Management takes over the system, it shows whale options.
- drawMap(): After the whale screen, we draw a map and it places vertices according to Model Map class.
- drawAttribute(): In all RM screens there will be a frame that shows the player's attributes.
- showDeck(): Displays your deck in frame.
- drawMerchant(): Invoked when the player selects Merchant vertex, places potions, relics, cards and their prices. If any selected upgrade player attributes.
- drawRest(): Invoked when the player selects Rest vertex, shows rest and smith choices with the help of buttons.
- upgradedCard(Card upgradedCard): When the player selects a upgradeCard button, it passes information to Player by GUI elements.
- healedHP(int healedHP):When the player selects a healHP button, it passes information to Player by GUI elements.
- drawTreasure(): Invoked when the player selects Treasure vertex. Relic, Potion or Card is chosen from the buttons.
- cardSelected(Card cardSelected): When the player selects a card, passes information to Player by GUI elements.
- potionSelected(Potion potionSelected): When the player selects a potion, passes information to Player by GUI elements.
- relicSelected(Relic relicSelected): When the player selects a relic, passes information to Player by GUI elements.
- drawCombat(): Invoked when the player selects Combat vertex, places enemies and calls Combat Subsystem.
- showDescription(): Invoked when player drags mouse on to Potion, Relic or Card, and a small description frame will be shown to the player.

6.2.8 Pet Class:

Attributes:

- String name : the name of the pet.
- int currentHp: the current Hp. Initially equal to maxHp.
- int maxHp: the maximum amount of Hp the pet may have.
- int damage: the amount of damage the pet deals each turn.

Methods:

- public void dealDamage(): the pet deals damage equal to its damage attribute to a randomly chosen enemy.

- public void loseHp(int amount): the pet loses the specified amount of currentHp. Called when the pet is hit by an enemy.
- public void gainHp(int amount): currentHp is restored by the specified amount. If the amount after restoration is larger than maxHp, the amount becomes maxHP.

6.3 Combat Management Subsystem

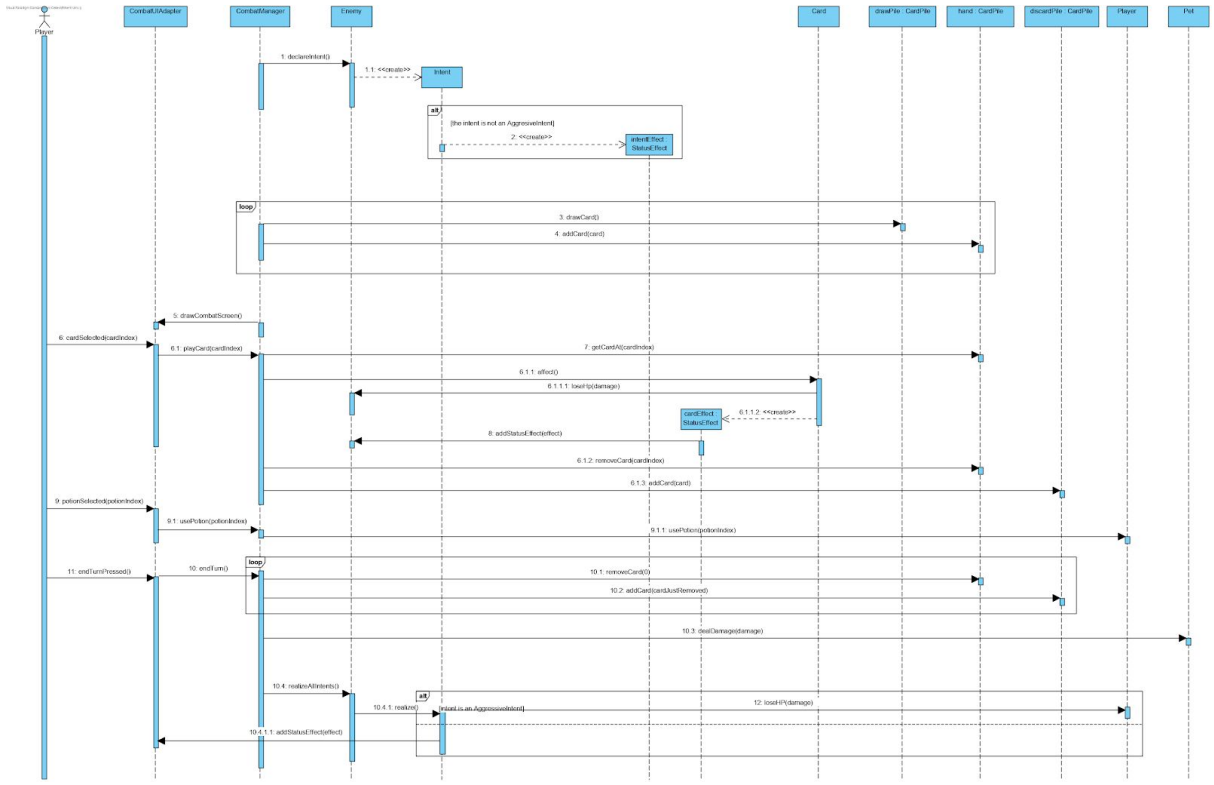

Figure 7 : Sequence Diagram of the Combat Management Subsystem, depicting the object interactions while a single typical turn is being played.

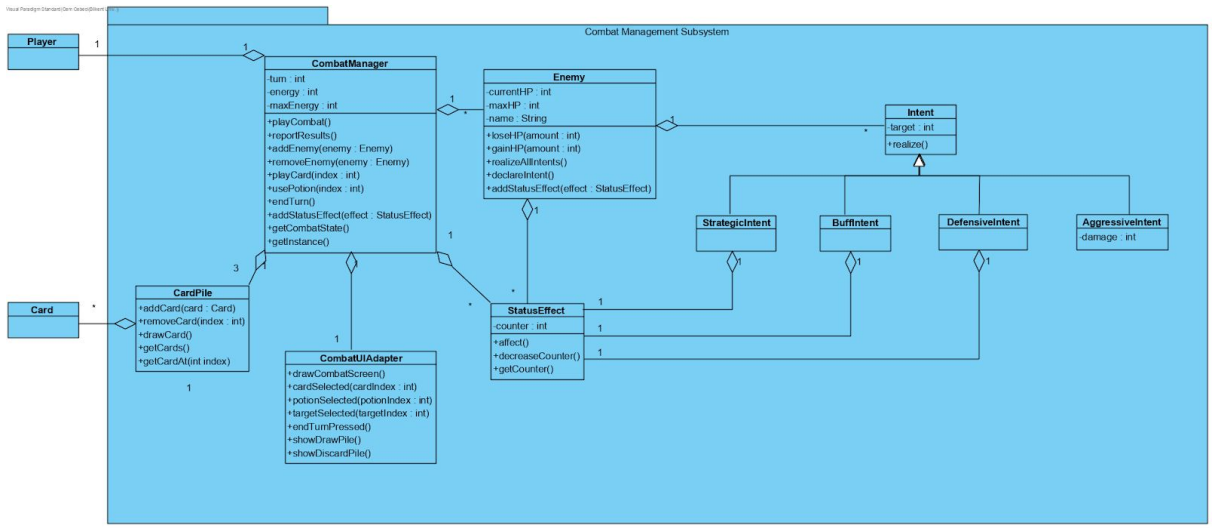

Figure 8: Class Diagram of Combat Management Subsystem

6.3.1 CombatManager Class:

The class that represents a specific combat instance. A CombatManager is created by the Run Management Subsystem when the player enters combat. Then, the RMS adds the enemies which the combat will include. Once this is done, the combat begins with the playCombat() method. As the combat proceeds, the CombatManager modifies the state of the combat according to user input passed to it by CombatUIAdapter.

This class is designed considering the singleton design pattern since at any instant, there will not be more than one combat instance. Making CombatManager a singleton object lets us access the existing CombatManager instance in all of the classes with the getInstance() method rather than having to pass references to an arbitrary CombatManager instance to all combat-related classes. Also, it ensures that the game can not have unstable states with multiple combat instances.

Attributes:

- private int turn: Keeps track of the number of turns completed since the beginning of the combat. Useful for cards or relics that have effects that depend on the turn.
- private int energy: The amount of energy the player has left for the turn.
- private int maxEnergy: The amount of energy the player gains each turn.

Methods:

- public void playCombat(): Initiates combat. The call returns once the combat is finished.
- public void reportResults(): After the combat ends, the Run Management Subsystem will call this method to learn the results of the combat (special flags for specific situations such as a bandit escaping with the player's gold etc.).
- public void addEnemy(Enemy enemy): Used by the Run Management Subsystem to add the enemies before initiating combat.
- public void removeEnemy(Enemy enemy): Called whenever an enemy loses all of their HP. Removes the enemy from the combat.
- public void playCard(int index): Called by the CombatUIAdapter when a card is clicked. Plays the card at the given index in the player's hand. Makes sure the card is discarded as well as realizing the card's effects.
- public void usePotion(int index): Similarly to playCard(index), called by the CombatUIAdapter when a potion is clicked. Uses the potion and realizes its effects.
- public void addStatusEffect(StatusEffect effect): adds a status effect to the list of status effect the player is afflicted by during the combat instance
- public void endTurn(): Called by CombatUIAdapter, ends the turn by discarding all of the cards in the hand, restoring all energy and realizing all of the enemy intents.
- public CombatManager getInstance(): returns the singleton instance of CombatManager.

- public String getCombatState(): called by the Navigation Subsystem when the game is meant to be saved. Returns a String describing the current state of the combat, which is then written to the save file.

6.3.2 Enemy Class:

The class that represents a single enemy in combat, the enemies should be added to CombatManager before the playCombat() method is called and they are removed once the enemy's HP reaches zero. An enemy may have more than one Intent.

Attributes:

- int currentHP: The enemy's current HP, the amount of damage they can take without dying.

- int maxHP: The maximum amount of currentHP an enemy can have, healing effects that increase currentHP beyond maximumHP set currentHP to maximumHP.

- String name: The name of the enemy, important for the UI and determining the set of intents the enemy can declare.

Methods:

- public void loseHP(int amount): The enemy loses HP equal to the specified amount, dies if the HP reaches 0.
- public void gainHP(int amount): The enemy gains HP equal to the specified amount.
- public void addStatusEffect(StatusEffect effect): The enemy adds the effect to the list of StatusEffects it is affected by.
- public void declareIntent(): The enemy chooses and declares their intent(s) for the turn. The intents they can choose depend on their name, the intents are instances of the Intent class.
- public void realizeAllIntents(): The enemy realizes (acts on) all of their intents.

6.3.3 Intent Class:

Represents an action an enemy intends to take. The class Intent is an abstract class, an intent is either an AgressiveIntent, DefensiveIntent, StrategicIntent or a BuffIntent, if an enemy intends to do a combination of these, the enemy has multiple Intents.

Attributes:

- int target: an index into the list of enemies that specifies the target of the intent. At the end of the turn, if the enemy survives, the buff / block status effect specified by the intent is added to the enemy at the target index. Index -1 is used to represent the player as a target.

Methods:

- public void realize(): realizes the intent, this may include dealing / healing damage, applying status effects. The effect of this method polymorphically depends on the subclass.

6.3.3.1 StrategicIntent Subclass:

Intents that apply negative status effects to the player. Each StrategicIntent has an associated StatusEffect.

6.3.3.2 BuffIntent Subclass:

Intents that apply positive status effects to the enemies. Each BuffIntent also has an associated StatusEffect.

6.3.3.2 DefensiveIntent Subclass:

Intents that increase the enemy's block. Block is a status effect in our design and the amount of block the enemy gains is the counter of the corresponding StatusEffect.

6.3.3.2 AggressiveIntent Subclass:

Intents that deal damage to the player.

Attributes:

- int damage: The amount of damage the player will be dealt.

6.3.8 StatusEffect Class:

A status effect that affects an enemy or the player. The enemies keep track of the StatusEffects they are affected by. The CombatManager keeps track of the StatusEffects that affect the player.

Attributes:

- int counter: StatusEffects have counters for various purposes, typically, the counter represents the effect's duration or intensity.

Methods:

- public void affect(): Implements the effect of the StatusEffect, affect() is an abstract method, the implementation depends on the subclass.
- public void decreaseCounter(): Decreases the StatusEffect's counter.
- public int getCounter(): Returns the counter.

6.3.9 CardPile Class:

A card pile is an ordered collection of Card objects. The CombatManager has three card piles: the draw pile, the discard pile and the hand.

Methods:

- public void addCard(Card card): Adds a card to the collection.
- public void removeCard(int index): Removes the card at the specified index. Returns the removed card.
- public Card drawCard(): Returns the top card and removes it from the collection.
- public Card getCardAt(index): Returns the Card at the specified index.
- public Card[] getCards(): Returns a list of all the cards in the collection. Useful for displaying the contents of a pile.

6.3.10 CombatUI Adapter Class:

This class handles the user interface, it contains the methods that will be called by the GUI elements and it controls the graphical output elements when something needs to be displayed.

Methods:

- public void drawCombatScreen(): Invoked whenever the display needs to change.
- public void cardSelected(int cardIndex): Invoked by the GUI elements when the player selects a card. Passes the information to the CombatManager.
- public void potionSelected(int potionIndex): Invoked by the GUI elements when the player selects a potion. Passes the information to the CombatManager.
- public void targetSelected(int targetIndex): Invoked by the GUI elements when the player selects a target for a card or a potion. Passes the information to the CombatManager.
- public void endTurnPressed(): Invoked by the GUI elements when the player presses the end turn button. Passes the information to CombatManager.
- public void showDrawPile(): Displays the cards in the draw pile to the player.
- public void showDiscardPile(): Displays the card in the discard pile to the player.