Bilkent University

Department of Computer Engineering

CS 319 Course Project

Group 1C-SS

Design Report

- Alperen Öziş
- Arda Göktoğan
- Cemal Gündüz
- Eren Şenoğlu
- Duygu Nur Yaldız
- Yavuz Faruk Bakman

Supervisor: Eray Tüzün

Contents

1. Introduction

1.1. Purpose of the System

The purpose of this project is to make the extended version of the game Slay the Spire. Slay the Spire is a single-player, turn-based strategy game. The purpose of this game is to complete all the challenges on the map. The player starts the game from the beginning of the map with the chosen character. The map consists of different locations, which are rooms of hostile enemies, resting areas, treasure rooms, unknown rooms or merchants. The player has a deck of cards that can be extended via the merchants, treasures or fights. The deck is used during the turn-based fights. In each turn, several cards are randomly distributed to the player's hand from the deck. The game is over when the player slays the boss at the final room of the map or when the player dies.

Our main concerns in this system's design is reliability, performance, cost, maintainability and modifiability. We use a layered architecture in our design.

1.2. Design Goals

Reliability:

Providing a game with very little likelihood of encountering a bug is one of our main design goals. Throughout the implementation process we are aiming to test for possible bugs by different members of team, separately. For the possible bugs that might be overlooked, we have the possibility of examining and experiencing different cases of our game and had the chance of correcting bugs that we encountered. The fact that our project is a small it is possible to discover and fix bugs by exhaustion. After applying reliability tests, our prior aim is to reach reliability magnitude of 0.95 at least.

Performance:

Although we are designing one of the most basic games, it should provide the user a smooth gaming experience. Thus, we will implement the game in a way that it runs fast and correctly without minimum system requirements. Java is a fast language so it will help us a lot to achieve this goal.

Cost:

We expect our system to be low cost in terms of development time. Since we have limited time for design and implementation of the project, we don't want to have complex design that can not be handled. Also since we have no budget, we don't want to use any software, database or other components that are not free.

Maintainability:

We want our design to be maintainable for new card, relic and potion designs. It should be easy to design your own card with its rules and add to our game. This will provide ability of extension to our game and it will not be boring after a long time of playing if our software is maintainable.

Modifiability:

We want our design to be modifiable. It means that if we need to change a functionality, or add a new one it should be easy to adapt it. If our game is modifiable, making these things will not require to change the current design significantly and will not be much challenging.

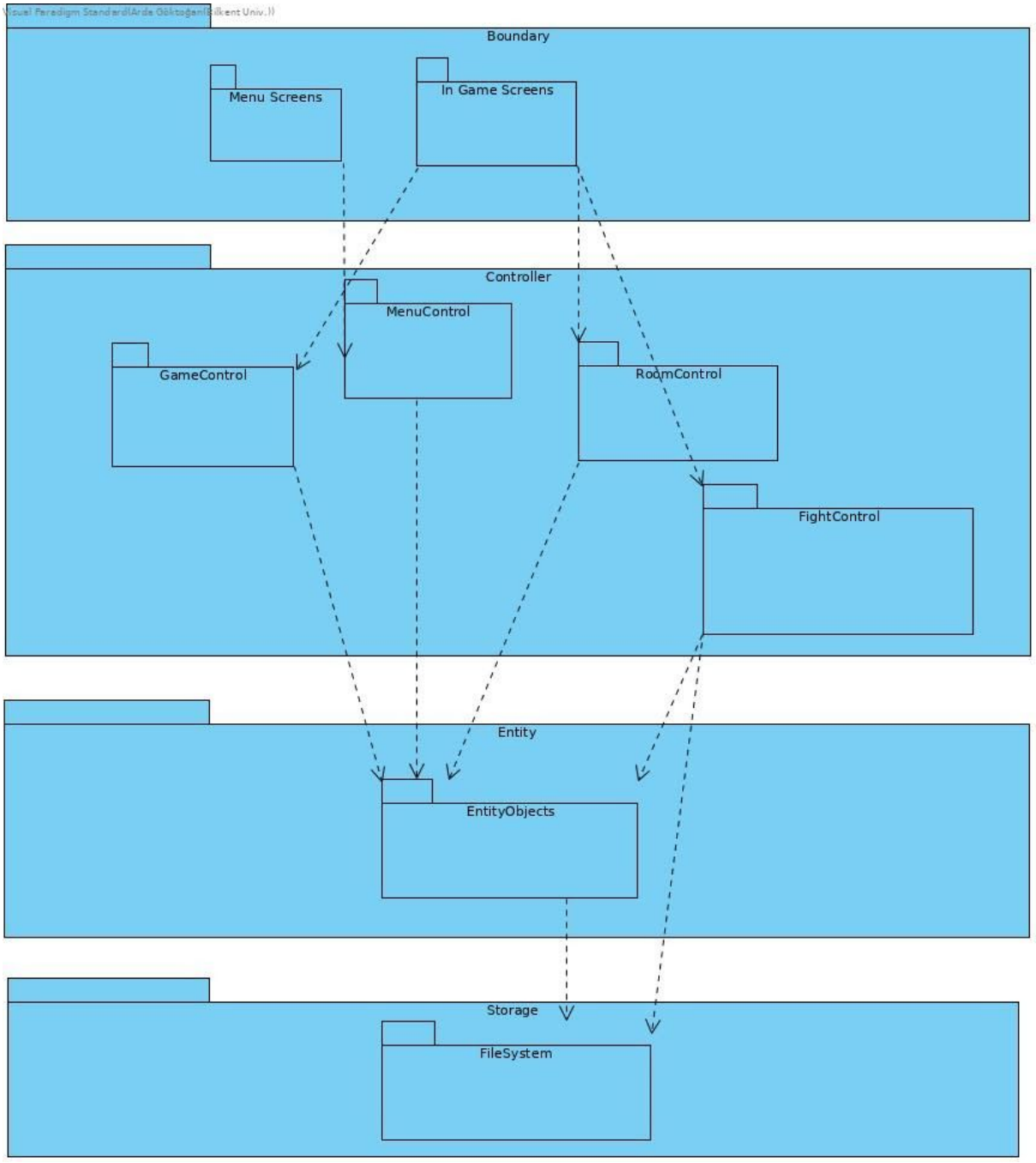
Portability:

We are implementing our game using Java Programming language, this means that Java's JVM will provide us a chance to implement our game in different portals since it is executable on any machine that supports it.

User Friendliness: The system provides a very simple user-interface that every player can understand what is going on. Every components in the game such as cards, relics, potions etc. has explanation, if the player clicks the component, it learns what is the component, how it works etc. Also, the game mechanics does not require any complicated stuff. The player play its movement by easily applying a sequence of simple events. These events are very intuitive such that the player even don't need to think about it. Also, the settings of the game can be easily changed by the user therefore the user can rearrange its name in the game or the window size and therefore it creates a better game experience.

2. High-Level Software Architecture

2.1. System Decomposition



In our system architecture, we used 4 layer architecture. In the first later We have storage classes to communicate with file system to store persistent data to initialize entity objects. Also we have card rules in the file system that are available for the usage of FightControl subsystem. In the second system we have entity objects. In the third layer, we have the business logic. Most of the functionality to play game like traveling around map and actions in the rooms are handled. It consist of 4 subsystems. MenuControl is responsible for menu options like saving game or loading game, starting new game, or changing the user.

GameControl is responsible for managing map, traveling of character in the map and directing to the right room when the character wants to advance in the map during the game. RoomControl is responsible for the actions that happens in the Rooms. FightControl is a separate subsystem from RoomControl because complicated events are possible during the fights. To be able to handle all possible interactions of cards,relics and potions; FightControl subsystem will be useful.

2.2. Hardware Software Mapping

We will implement our game using Java 8 and JavaFX for visuality concerns. This requires user's computer to have a support for java applications which means the game will be available in popular operating systems like Windows, Linux, OSx etc. In our internal design we will use JavaScript(ECMAScript2018) for handling the card effects and we will use JSON to store our save files in the database.

User will need a mouse to play the game: choosing cards, drinking potions, buying items etc. Keyboard is optional for changing save's name. And a sound system is highly recommended for the enhancement of the game experience since we will provide sound effects and music.

2.3. Persistent Data Management

Our game will be offline game so we don't need online database to store our game data. We will store our data locally by using ".json" files. We are planning to store game settings, game saves, characters with starting cards and relic special for that character, other relics, other cards, enemies, events that separated according to acts, and lastly leaderboard information. We can use two different file systems to separate these data by functionality. One is to store game saves and settings which is specific to user, and the other is to store general game data that mentioned above.

2.4. Access Control and Security

Since our game is not an online game, it will not connect to the internet. Also, our game does not require any personal information, access control and security is not a major concern for us.

2.5. Boundary Conditions

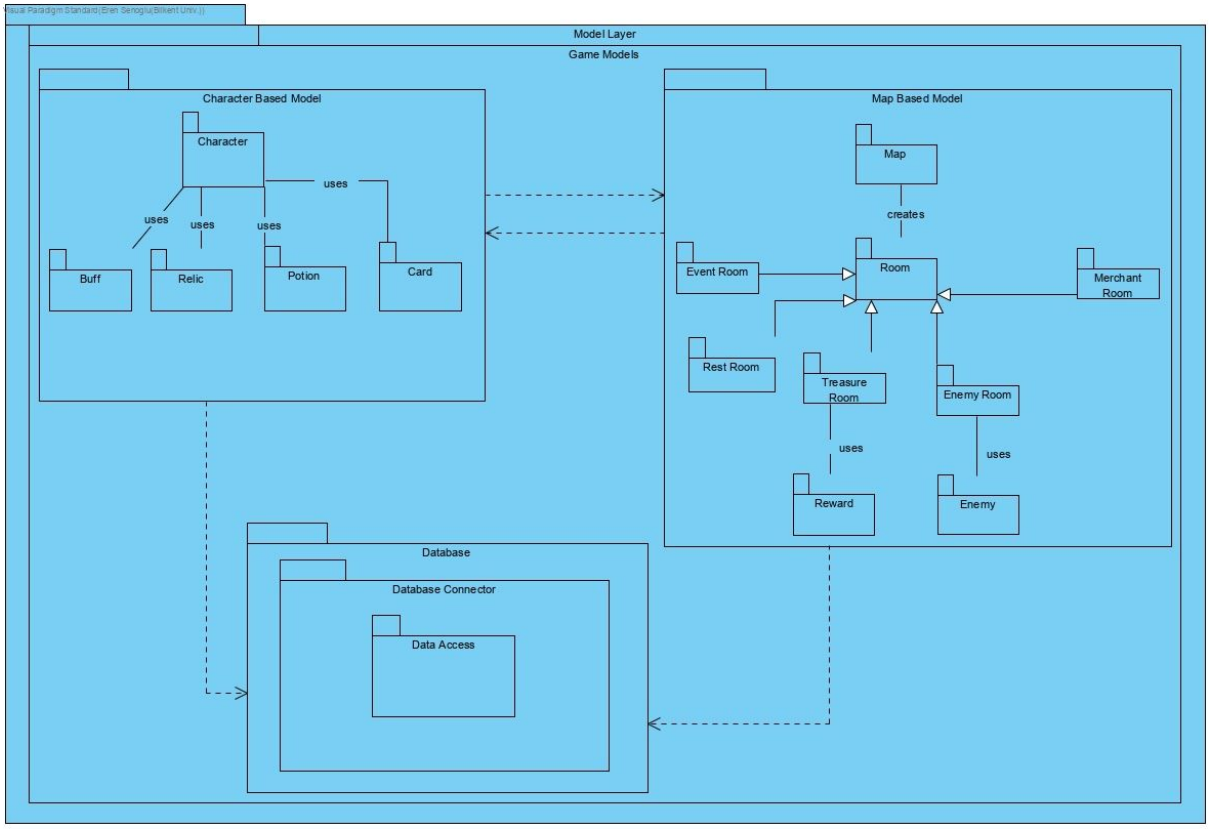
Initialization:

The start of the program is done through the jar file. The initialization of the saved game is done by the main menu buttons. If the player wants to continue with the saved game, the program get contacted with the database and load necessary game documents. Also, there are lots of cards, relics etc. in the. These objects in the game are initialized when the player starts a new game or load a new game. The properties of the objects are done through the JSON files. Also, the specific objects belonging to the rooms are initialized when the player enters the room in the game.

Termination:
In the game, the objects are deleted after the player exits the current game. Also, specific objects belong to a room are terminated when the player is done with the room. When the program is closed by the user, the system connects with database and automatically save the current situation of the game. Also, in each checkpoint which are the end of the each room, the database is updated to save the current situation of the game. To close the system, the user can directly click the cross button on the top or use exit button in the main menu.
Errors:
When there is an error about the initialization of the objects through JSON files the system directly kick the player from the game ask the player to re-enter the game.
If there is a problem about the loading saved game from the database, the program also asks the player to re-click the load saved game.
The errors of the game will be saved to a log file to solve them further.

3. Subsystem Services

3.1. Game Model Subsystem Services



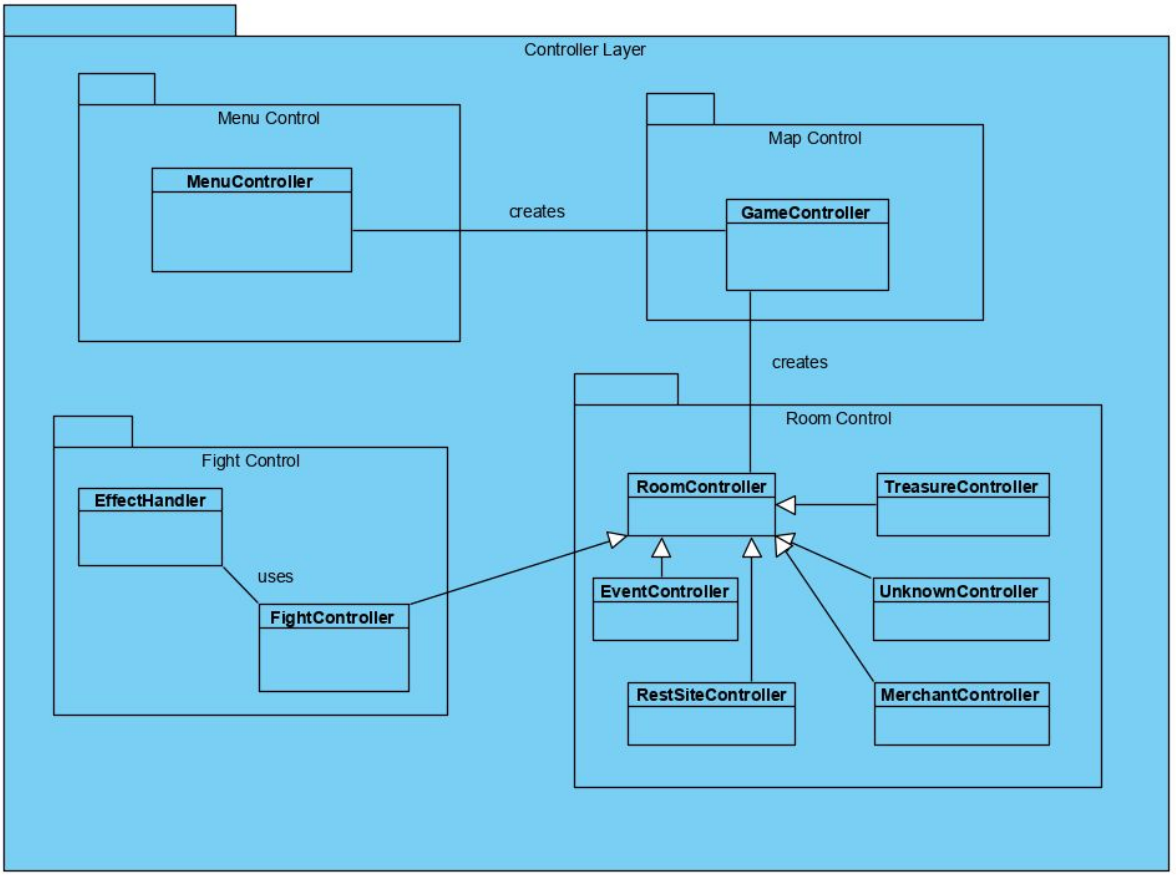
We can examine our model subsystem in two parts.

First main part is character and elements that is associated with character. Character is a core element for our game as the games is based on character's changing cards, relics,

potions. Relic, Potion, Pet, Card, and even Buff can be seen as supportive classes for character class' attributes.

Other crucial part in model subsystem is the Map and Rooms. Map includes different rooms in it and we divide these rooms into different classes according to functionality and name. Unknown room can also be mentioned as it is a special room. Because it will be handled by Event class as unknown room is randomly generated and also can be turned into other room models by events.

3.2. Controller Layer Subsystems

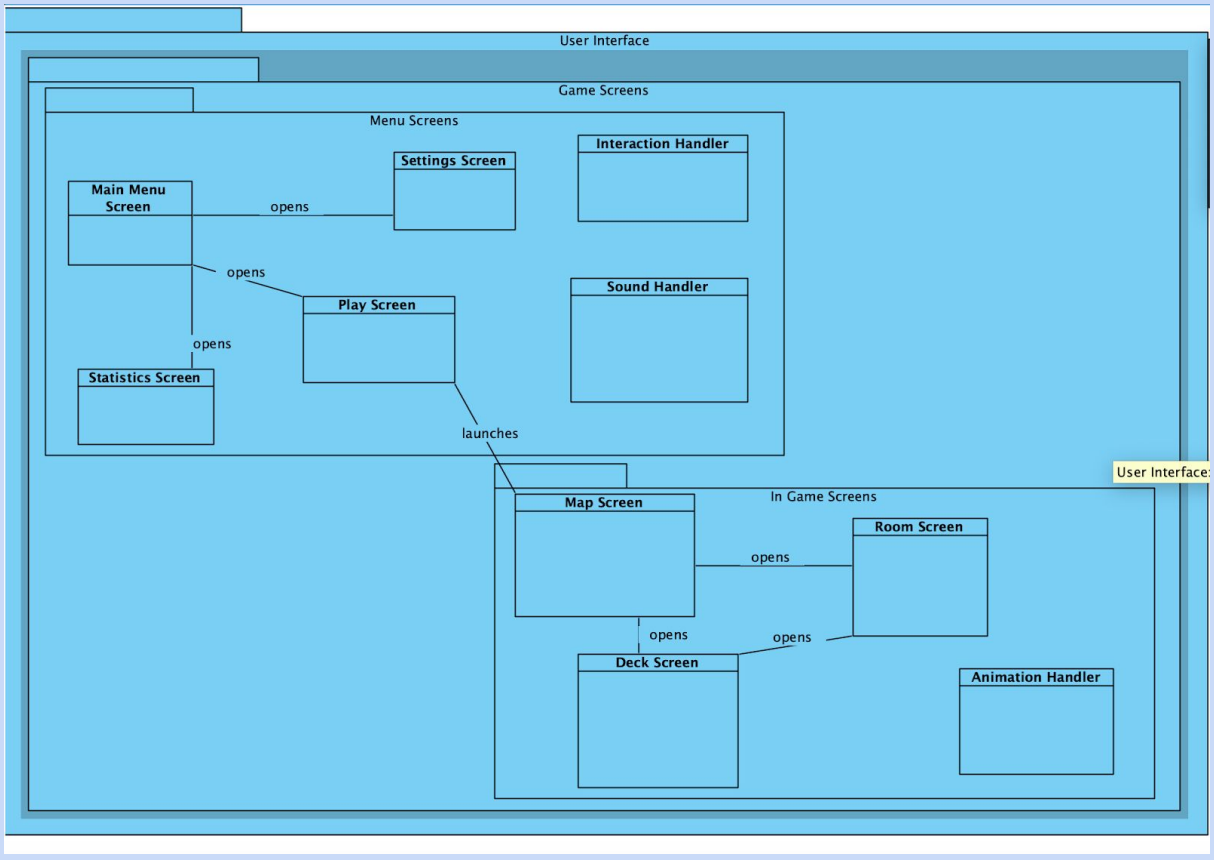


Controller layer consists of 4 subsystems. Menu Control subsystem is mainly for operations that are done in the menu of the game. When the player chooses to play game Map Control subsystem is activated. This subsystem is responsible for the operations done in the map page f the game, such as selecting the next play room.

Room Control subsystem is to handle the different room operations. When a room is chosen from the map, Room Control subsystem is activated. The operations of Treasure Room, Merchant Room, Unknown Room, Rest Site and Event Rooms are handled in this subsystem.

Fight Control subsystem is actually a kind of a part of Room Control, however, since it is an important and big part of the game, we put it in a different subsystem. Another reason that we consider Fight Control as a separate subsystem is that it communicates with the database. Fight Control subsystem is mainly responsible for the operations done during a combat, such as playing a card, potion or enemies playing.

3.3. View Subsystem



User Interface subsystem is responsible for creating screens according to user input. The game screens can be divided into 2 different subsystems which have basically fundamental different components. The menu screens subsystem handles the screens that happens in where the real game does not start yet. These screens are settings screen, main menu screen, statistics screen. The Interaction handler changes the displayed screen and sound handler is responsible for the background music. From the play screen, the user can launch the In Game Screens subsystem. This subsystem deals with the user interface components that exist in where real game works. The 3 fundamental part of this system are Map, Room and Deck Screen which are 3 main parts of the game screens. The animation handler is responsible for the animation events it these 3 rooms.

4. Low Level Design

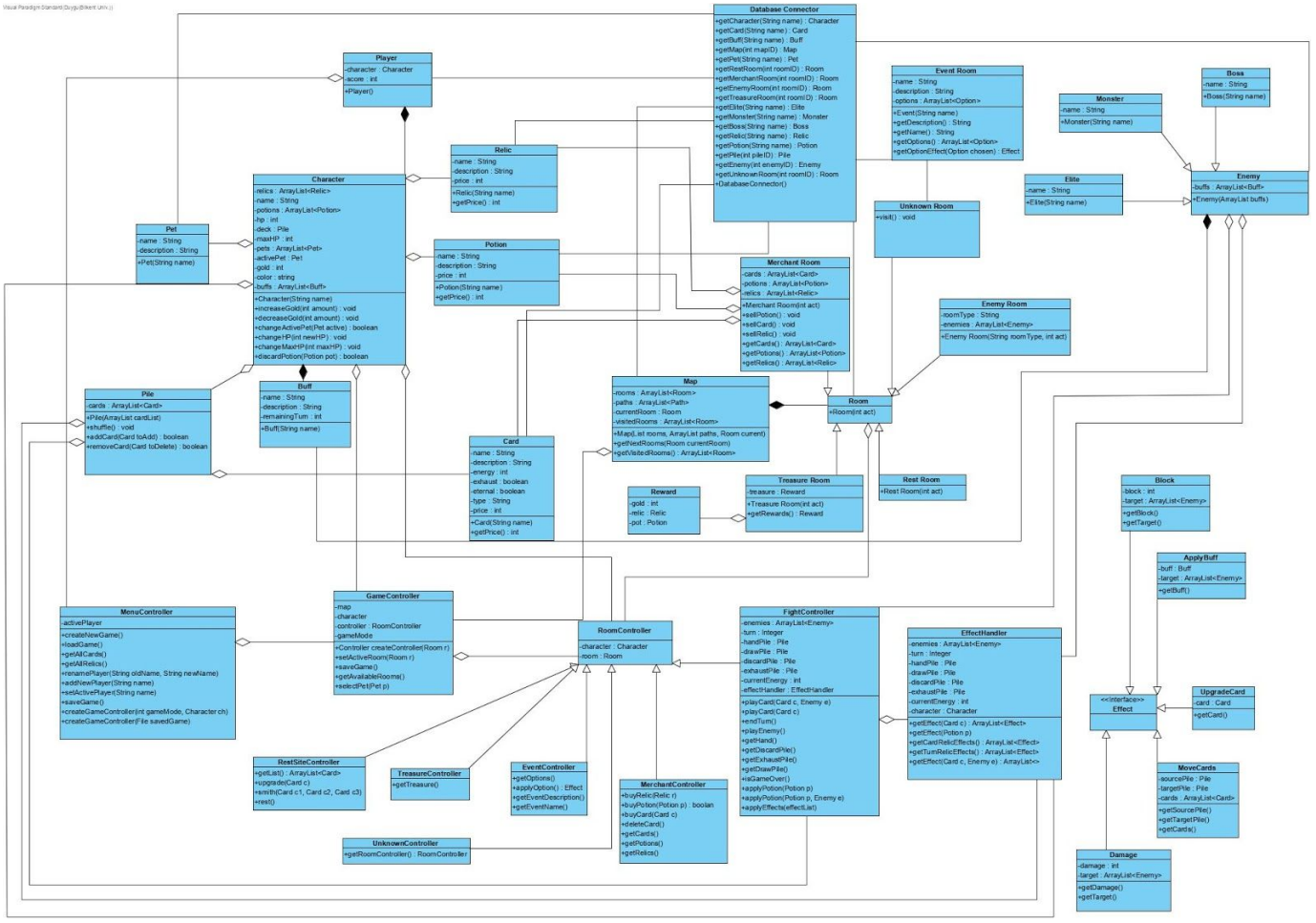
4.1. Object Design Trade-offs

Efficiency v. Portability
We choose Portability because Java can run any of architectures and system if required JRE is installed.

Functionality vs. Cost

Since we have limited time until the deadline of the project, we want to keep functions as simple as possible. Core logic of the game will be designed but extra features that are in analysis report can be postponed for the second iteration. Therefore we can reduce the time that is required for software design and implementation. comptress command line

4.2. Final object design



4.3. Packages

We will use JavaFX to construct the GUI of our game. The list of some packages that we will use and their simple explanations is below:

- javafx.scene : Provides the core set of base classes for the JavaFX Scene Graph API
- javafx.animation : Provides the set of classes for ease of use transition based animations
- javafx.application : Provides the application life-cycle classes
- javafx.fxml : Contains classes for loading an object hierarchy from markup.
- javafx.stage : Provides the top-level container classes for JavaFX content
- javafx.event : Provides basic framework for FX events, their delivery and handling.

We will use Javascript language while handling card effects. So we will use javax.script package.

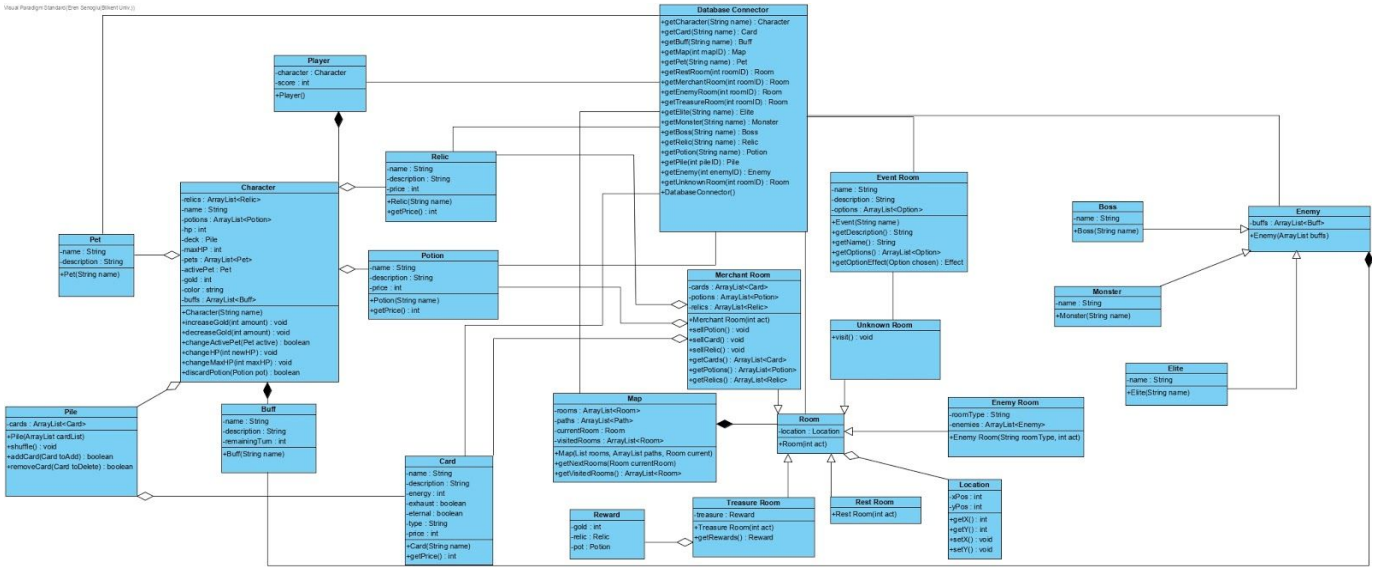
4.4. Class Interfaces

4.4.1. Database Connection Classes

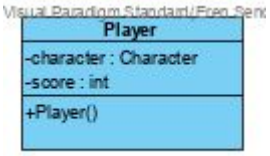
Database Connector

Gets all required information by interacting with .json files. Each method to take specific object's attribute.

4.4.2. Entity Classes



Player



Attributes:
-character : Character
Holds which character is player playing with.
-score : int
Holds the score of player.

Character

Attributes:
-relics : ArrayList<Relic>
List of relics that character has.

-name : String
The name of the character.

-potions : ArrayList<Potion>
List of potions that character has.

-hp : int
Current hit point of character.

-deck : Pile
List of cards that character has which is called Pile.

-**maxHP : int**

Maximum hit point of character.

-pets : ArrayList<pet>
List of pets character has.

-activePet : Pet
The currently active pet for character.

-gold : int
The amount of gold character has.

-color : string
Color indicator that used for character cards' and energy orb's color.

-buffs : ArrayList<buff>
List of currently active buffs on the character.

Methods:
+Character() :
Constructor method.

+increaseGold(int amount) : void
Increases the gold amount of player by given parameter.

+decreaseGold(int amount) : void
Decreases the gold amount of player by given parameter.

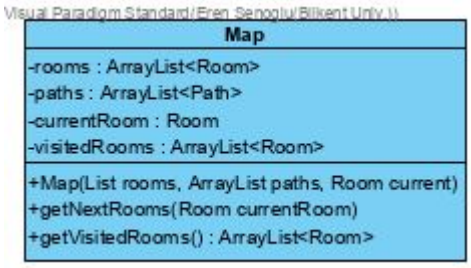
+changeActivePet(Pet active) : boolean
To change the active pet of ccharacter.

+changeHP(int newHP) : void
Change the curent hp of character to parameter.

+changeMaxHP(int maxHP) : void
Set the max HP of character to parameter value

+discardPotiont(Potion pot) : bool
Discard the current potion that character has.

Map



Attributes:
-rooms : ArrayList<Room>
List of all kind of rooms on the map.

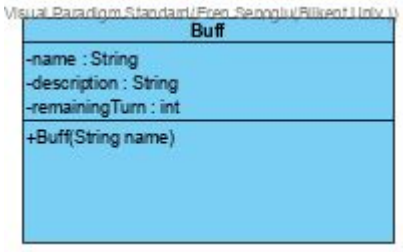
-paths : ArrayList<Path>
List of all possible paths player can follow.

-currentRoom : Room
Room that player currently in.

Methods:
+getNextRooms : ArrayList<Room>
Returns the rooms that are along the path which is not visited yet.

+getVisitedRooms : ArrayList<Room>
Returns the visited rooms on the path.

Buff

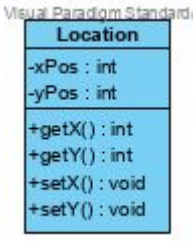


Attributes:
-name : String
Name of the buff.

-description : String
Understandable description of buff's effects.

-remainingTurn : int
Remaining turn to buff to become inactive or disappear completely.

Location



Attributes:
xPos :
Indicates the x position.

yPos:
Indicates the y position.

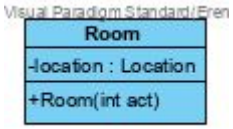
Methods:
+get(X) : int
Get the x position.

+get(Y) : int
Get the y position.

+set(X) : int
Set the x position.

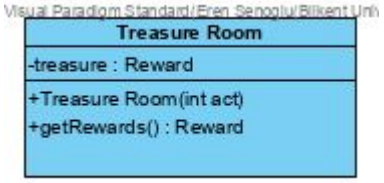
+set(Y) : int
Set the y position.

Room



Attributes:
-location :
indicates where the room is by x,y coordinates

Treasure Room



Attributes:

treasure:
Holds the rewards from treasure.

Methods:
+getRewards(): ArrayList<Reward>
Returns the rewards of treasure.

RestRoom

Merchant Room

Attributes:
-name : String

-cards : ArrayList<Card>
List of cards that is selled in merchant room.

-potions : ArrayList<Potion>
List of potions that is selled in merchant room.

-relics : ArrayList<Relic>
List of relics that is selled in merchant room.

Methods:
+sellPotion(): void
To sell specific potion in the merchant room.

+sellCard(): void
To sell specific card in the merchant room.

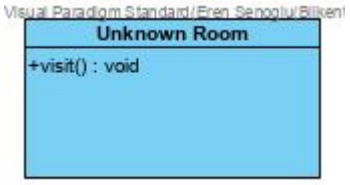
+sellRelic(): void
To sell specific relic in the merchant room.

+getCards(): void
Returns the cards that are sold in the merchant room.

+getPotions():void
Returns the potions that are sold in the merchant room.

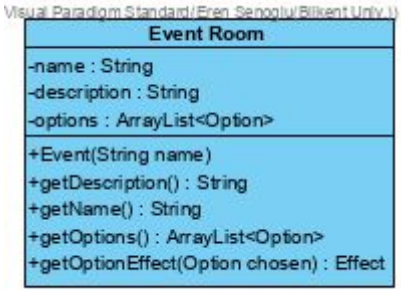
+getRelics(): void
Returns the relics that are sold in the merchant room.

Unknown Room



Methods:
+visit(): void
When unknown room is visited this method will start event process or room change process.

Event Room



Attributes:
-name : String
Name of the event.
-description : String
Understandable detailed information of event.
-options : ArrayList<Option>
Choices that player can choose to determine the effect of event.

Methods:
+getName(): string
Returns the name of the event.

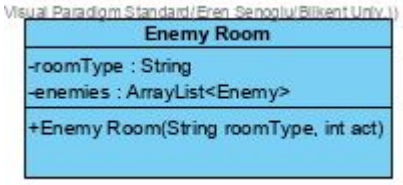
+getDescription(): string
Returns the description of the event.

+getOptions(): ArrayList<Option>

Returns the options of the event.

+getOptionEffect(Option chosen): Effect
Returns the effect of the event cause from option that is choosed.

Enemy Room



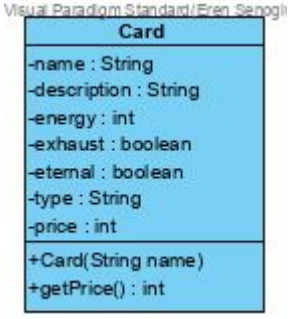
Attributes:
-roomType : String
Stores the type of the room

-enemies : ArrayList<Enemy>
Stores the list of enemies in the room.

Enemy

Attributes:
-buffs : ArrayList<Buff>
List of buffs that enemy has currently.

Card



Attributes:
-name : String
Name of the card.

-description : String
Detailed information of how cards work and what is its effect.

-energy : int
Indicator that how much energy is the card cost.

-exhaust : boolean
Indicator that whether is card can playable after one usage or not.

-ethereal : boolean
Indicator that whether card will turned automatically exhausted if the card is in player's hand at the end of the turn.

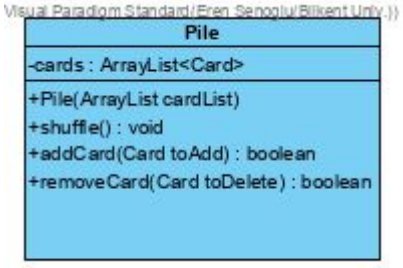
-type : String
Type of the card. A card can be attack, skill, power, status or curse type.

-price : int
The amount that the card will be sold in merchant rooms.

Methods:

getPrice() : int
Returns the price of the card.

Pile



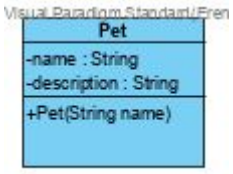
Attributes:
-cards : ArrayList<Card>
Pile holds the array list of cards.

Methods:
+shuffle(): void
Shuffles the player's card pile.

+addCard(Card toAdd): boolean
Adds a card to player's pile. Returns true if add operation is successful.

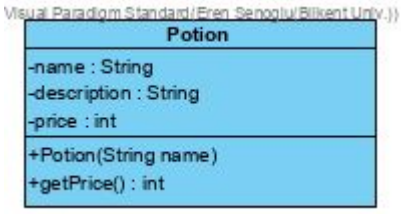
+removeCard(Card toDelete): boolean
Removes a card from player's pile. Returns true if remove operation is successful.

Pet



-name : String
Name of the pet.
-description: String
Description of the pet with the information of pet's attack pattern and specialities.

Potion



Attributes:
-name : String
Name of the potion.
-description: String
Description of the potion including effects on character and it's duration.
-price : int
The amount that the potion will be sold in merchant rooms.

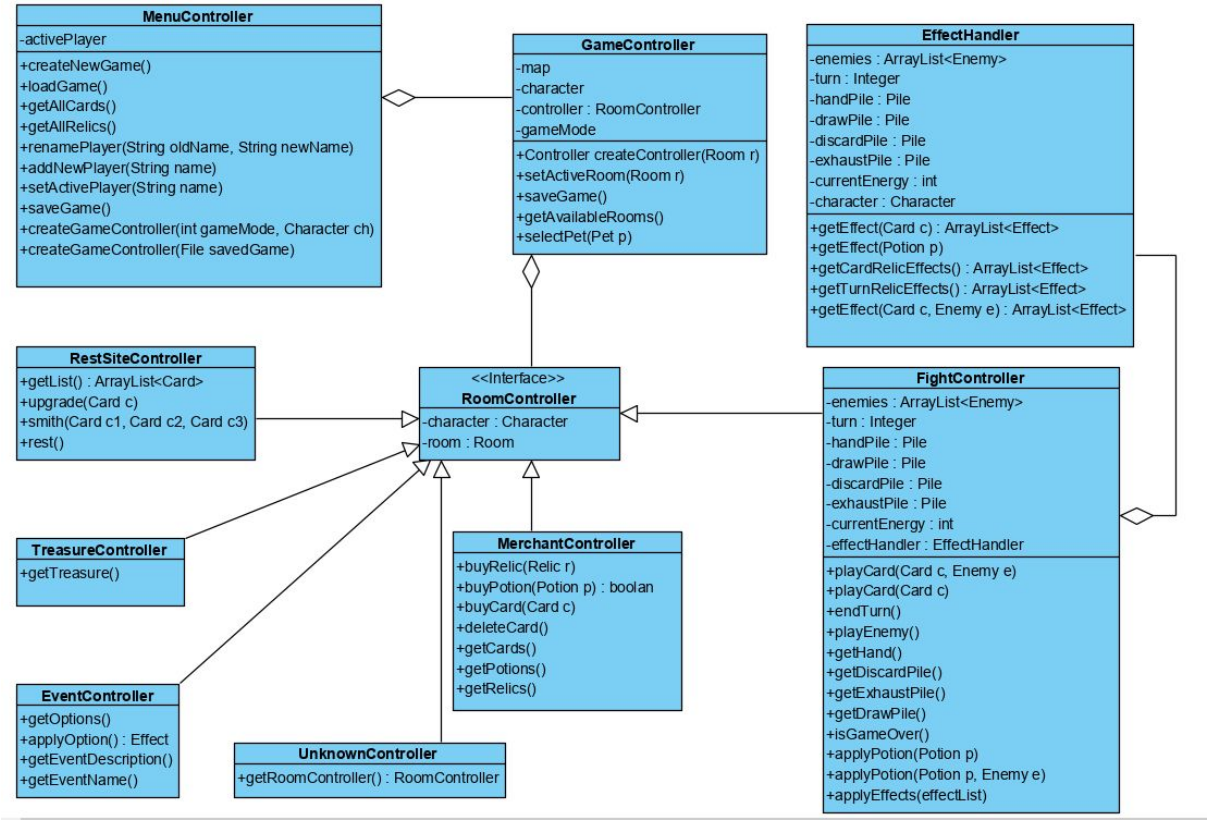
Methods:
+getPrice(): int
Returns the value of price.

Relic

Attributes:
-name : String
Name of the relic.
-description: String
Description of the relic including effects on character and it's duration.
-price : int
The amount that the relic will be sold in merchant rooms.

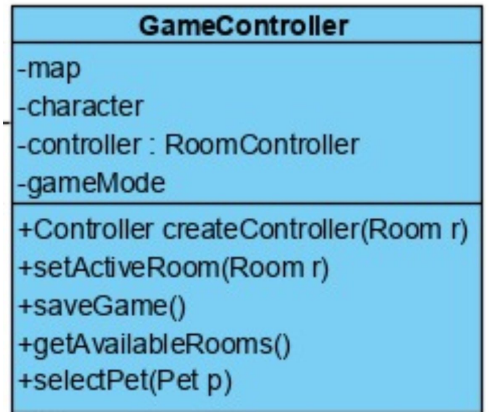
Methods:
+getPrice(): int
Returns the value of price.

4.4.3. Controller Classes



This figure shows the interaction between the controller layer classes.

4.4.3.1. Game Control Classes



GameController
-map: Map of the game

-character: Character that current user is playing.

-controller: This controller is responsible when player enters any room.

-gameMode: game mode keeps easy,medium or difficulty of game. It is used when the rooms are created.

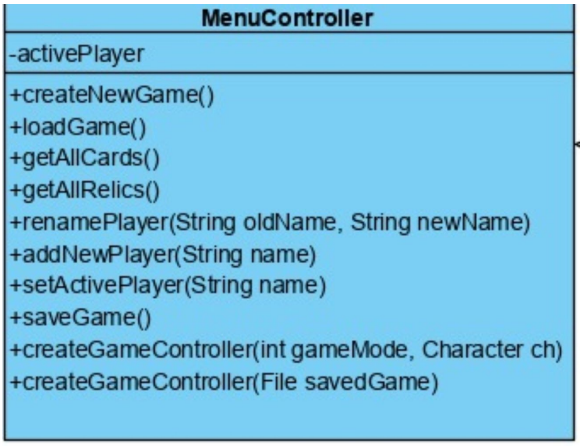
+createController(): Controller
this method is called whenever any player visits room and returned controller will be used for that room.

+setActiveRoom(Room r): void
this will set the current room in the map.

+saveGame(): void
This method is called if the user saves the game. It will save the game to file including current map situation, cards, relics and potions.

+getAvailibleRooms(): ArrayList<Room>
This will be called to get rooms that current room access to.

4.4.3.2. Menu Control Classes



MenuController
-activePlayer: Player
Active player of the current game

+createNewGame()
This option creates a new game, a new map.

+loadGame()
This function loads a saved game.

+getAllCards(): Pile
This function returns the all available cards in the game.

+getAllRelics():ArrayList<Relic>
This function returns the all available relics in the game.

+renamePlayer(String oldName, String newName)
This function is for editing an existing player profile.

+addNewPlayer(String name)
This function is for adding a new player profile to the game

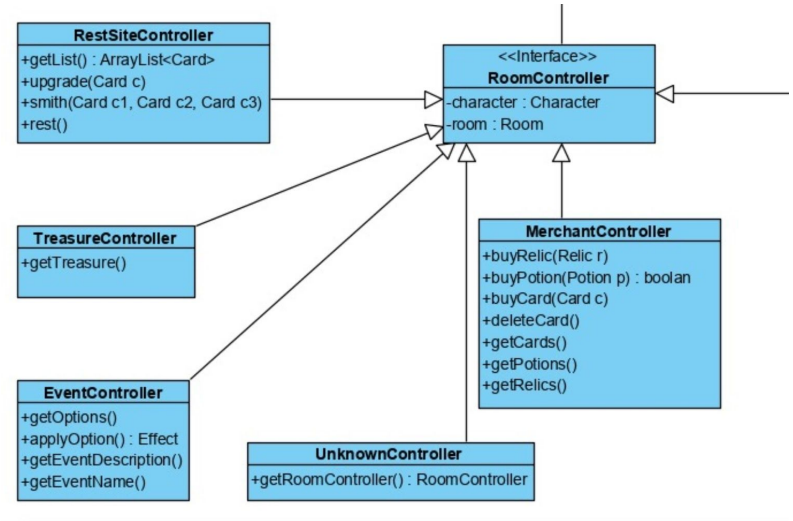
+setActivePlayer(String name)
It sets the active player of the game

+saveGame()
this function saves the game to a file.

+createGameController(int gameMode, Character ch): GameController
This function is called in the createNewGame() method. it takes the game mode and chosen character as parameters and creates and returns a GameController object.

+createGameController(File savedGame): GameController
This method is called in the loadGame() function. It takes the file of the chosen loaded game and creates and returns a GameController object with the information read from the file.

4.4.3.3. Room Management Classes



Room Controller
-character: this is reference to current character.
-room: this is reference to current room.

MerchantController

+buyRelic(Relic r): reduces the gold of the character and adds given relic to character's relics.
+buyPotion(Potion p):reduces the gold of the character and adds given potion to character's potions.
+buyCard(Card c): void
Reduces the gold of the character and adds given card to character's pile.
+deleteCard(Card c):void
Reduces the gold of the character and removes given card from character's pile.
+getCards():Pile
returns the cards for sale.
+getPotions():ArrayList<Potion>
returns the potions for sale.
+getRelics():ArrayList<Relic>
returns the relics for sale.

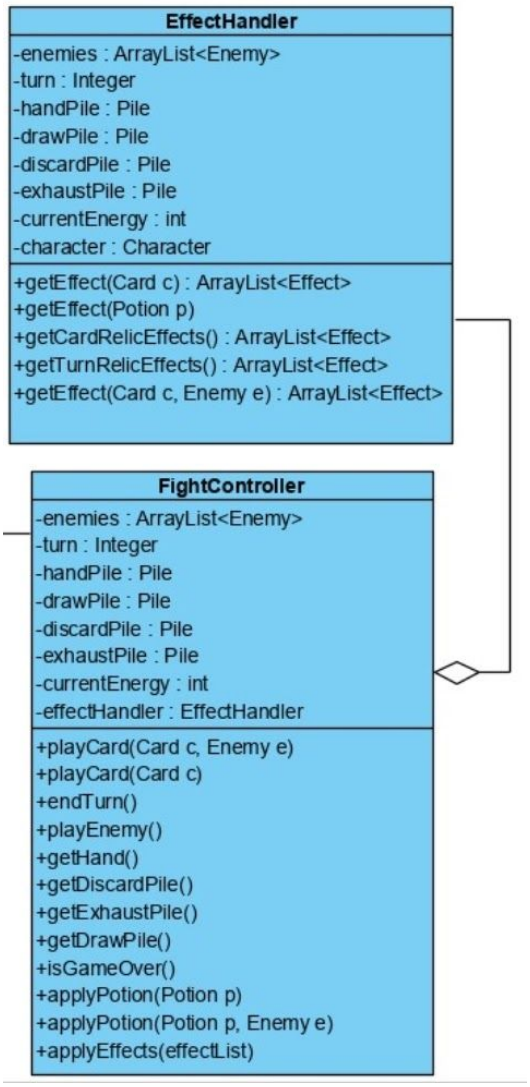
Unknown Controller
+getRoomController():Controller
This will return the controller that the unkonwn room turns into it.

Rest Site Controller
+getList(): This will return the cards of the character
+upgrade(Card c): This will be called if the user wants to upgrade card.
+smith(Card c1,Card c2,Card c3): This method takes 3 cards as input, removes them from character's deck and adds a single stronger card to character's pile.
+rest(): This function heals the character.

TreasureController
+getTreasure(): This will return Reward object that character finds from the treasure.

EventController:
This class will be used if unknown room turns into a event room.
+getOptions(): Returns the options of the event.
+applyOption(Option o):This function will apply the chosen option effect to player.
+getEventDescription():Returns the event description.
+getEventName():Returns the event name

4.4.3.4. Fight Management Classes



EffectHandler

This class will responsible for creating effect objects. For example if character deals damage to enemy with card, getEffect(Card c) will return the effect list that should be happen in the game with considering buffs and debuffs of character and enemies, relics and other cards of the character.

-enemies: ArrayList<Enemy>

this is a reference to the current enemy list

-turn: Integer

this is a reference to the turn variable

-handPile: Pile

This is a reference to the current hand pile.

-drawPile: Pile

This is a reference to the current draw pile.

-discardPile: Pile
This is a reference to the current discard pile.

-exhaustPile: Pile
This is a reference to current exhaust pile.

-currentEnergy: Integer
This is a reference to the current energy of the character.

-character: Character
This is a reference to current character.

+getEffect(Card c): ArrayList<Effect>
This function gets the card and returns the Effect objects related to that card.

+getEffect(Potion p): ArrayList<Effect>
This function gets the potion and returns the related Effect object of that potion.

+getCardRelicEffects(): ArrayList<Effect>
This function return all the relic Effect objects related to cards or a specific card.

+getTurnRelicEffects(): ArrayList<Effect>
Return the all relic Effect objects that are related to turns.

+getEffect(Card c, Enemy e): ArrayList<Effect>
This function gets the card and the target enemy and returns the Effect objects related to that cardand that enemy.

FightController
-enemies: ArrayList<Enemy>
this is a reference to the current enemy list

-turn: Integer
this is for holding the turn.

-handPile: Pile
The pile that the player can play in this turn.

-drawPile: Pile
the pile that the hand pile will be filled with the next turn.

-discardPile: Pile
The pile that played cards and the non-played card in the hand pile goes.

-exhaustPile: Pile

The pile that exhausted cards are collected.

-currentEnergy: Integer
This is the current energy of the character.

-character: Character
This is a reference to current character.

+playCard(Card c, Enemy e)
plays the chosen card, and makes its effects to target enemy.

+playCard(Card c)
Plays the chosen card, and makes its effects.

+endTurn()
this is called when the player's turn is over. playEnemy() function is called for each enemy.

+playEnemy()
Applies enemy's effects.

+getHand():Pile
Returns hand pile of character.

+getDiscardtPile():Pile
Returns discard pile of character.

+getExhaustPile():Pile
Returns exhaust pile of character.

+getDrawPile():Pile
Returns draw pile of character.

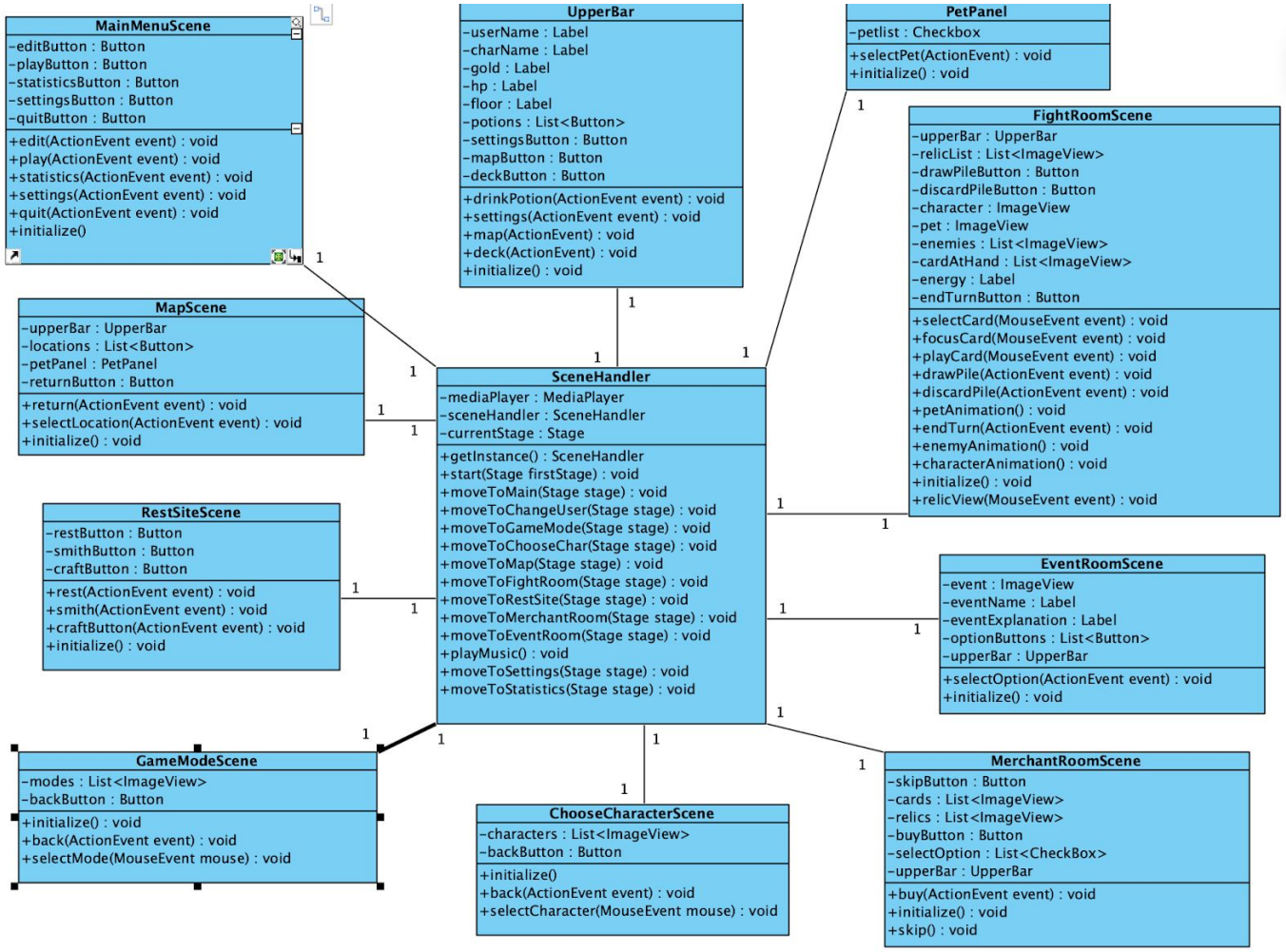
+isGameOver():bool
Returns true if current fight is over. Returns false otherwise.

+applyPotion(Potion p):void
This will apply the given potion.

+applyPotion(Potion p, Enemy e):void
This will apply the given potion effect to enemy target.

+applyEffects(ArrayList<Effect>):void
This will apply the given effects to proper targets.

4.4.4. Interface Classes



SceneHandler:
This class basically change the screen by moveTo methods. It has the Singleton design pattern to provide just one SceneHandler instance for the client-side system as can be observed through the sceneHandler attribute and getInstance function. The sound effects and the audio description features are also provided with this class. It has the mediaPlayer attribute and playMusic function for the sound effects. Also, the object holds the current screen by the current stage attribute.

MainMenuScene:
This class handles the main menu screen. There are buttons in the main menu that are used for moving other game screens which are Game mode screen, settings screen, edit name screen, statistics screen. Also, by the quit button, the player can terminate the program.

UpperBar:
This upper bar provides a short-cut tool to reach the basic stuffs in the game. It appers different scene of the game similarly. Therefore, we create this class individually and put the objects where it appears. It basically shows the username, character name, hp, current floor, gold by labels. There is also relics, deck and map buttons. When user clicks on of them it

gets detailed information about it. Also, there is settings button where user can directly access the setting page

GameModeScene:

This class handles the game mode selection screen. In this screen, there are list of image views where each of them corresponds a specific game mode. There is also a back button which change the screen that user come from.

FightRoomScene:

This class handles the fight room screen and its animations. It has all the elements which are buttons, images, labels etc. that the fight room needs. It handles the all results of the button by corresponding function of the buttons. It animates the pet, character and enemy by separate methods. These are not connected any button directly but that methods can be called by the result of the sequence of the events.

MapScene:

This class handles map screen. it has a list of buttons which are locations, only some of them will be clickable according to the user's current location and this operation will be done via selectLocation() function. it contains the upper bar which explained above and the pet panel which will be explained below. it also has a return button to turn back to the main menu.

PetPanel:

This class handles the pet panel in the Map scene. User's petList is displayed here and he will be able to choose which one he's bringing in to the fight via checkBoxes.

ChooseCharacterScene:

This class handles the character selection screen when starting to a new game. The characters which are avalaible to user are displayed. The user will choose one of these character by clicking on them with the help of selectCharacter() function. He will also be to return to the main menu.

EventRoomScene:

This class handles the Event room screen. The event name, image, explanation and the options of this event are displayed. The user will select the best option by clicking on it with the help of selectOption() function.

MerchantRoomScene:

This class handles the Merchant room screen. The items on sale are displayed. User can select whichever items he wants to buy by clicking on the checkboxes below the items, and purchase them by clicking on buy button. It is also possible to just skip by clicking on the skip button.

RestSiteScene:

This class handles the Rest site screen. Basically user has 3 options: rest, smith or craft. He will be able to do one them by clicking on one of these buttons.