Bilkent University

Department of Computer Engineering

Spring 2020 CS319 Project

1D - Slay the Spire

Project Design Report Iteration 2

Özge Yaşayan

Gülnihal Koruk

Fatih Karahan

Mehmet Bora Kurucu

Batuhan Özçömlekçi

Table of Contents

1. Introduction

1.1. Purpose of the Systems

Slay the Spire is a single player roguelike deck building game. The purpose of the game is move to upwards on a map while fighting monsters using the cards in their deck. Player will try to create a better deck while continue to move on. Additionally, there will be some innovations in order to make the game more enjoyable. These are, an animal companion (pet) to assist the main character in the fights and playing the game with several characters. For another additional features, in the main menu there will be view compendium option that will give the full visibility for the cards and relics in the card library and relic collection sections.

1.2. Design Goals

In these sections the maintenance and end-user criteria will be mentioned. We will clarify the non-functional requirements that is stated in our analysis report.

1.2.1. End User Criteria

1.2.1.1. Usability

For the simplicity and understandability of the game the UI design will be simple. Since the game is based on the creating a better card deck for the fights in the FightRooms, so the only information needed for the user will be card information. To be able to access this information, there will be a view compendium button on the main menu that covers all info about cards and relics. Since the player can learn all info needed with this button, there will be no other screen such as how to play. Moreover, we keep the UI design in a basic so that player can understand the game easily and play.

1.2.1.2. Performance

Since we will avoid the complex graphics and simulation, the game will not require a higher performance for the PC. For the better user experience, response time is important, so the player actions will be result within at most 1.5 second. User can open the game with 2 clicks and the button will be activated in 1 click.

1.2.2. Maintenance Criteria

1.2.2.1. Extensibility

We have designed the game in such way that modifying and changing the features and functionalities is easy. For instance, we design our package and classes so that adding new character or a card will require only creating new classes for these and the rest of the classes will not require many modifications but just connections. Moreover, the written class diagram will make easy to understand the relations between classes so that it serves the purpose of extensibility.

1.2.2.2. Modifiability

For being able to modify the game without changing lots of classes, we keep the independency of the classes so that any modification in one class will have a minimum effect on others. Thus, in our design, the connection between the subsystems is created weakly intentionally. That is, modifying or adding new features to the game will be simple.

1.2.2.3. Reusability

As mentioned in extensibility part, our class design keeps the purpose of independency. Thus, in our game system, the classes are created by their own purposes, so they are reusable. That is, model, view and controller maintain the reusability since they are organized by their class purposes.

1.2.2.4. Portability

The game will be portable since the software will be written using Java. That means the language will be the cross-platform. Since its JVM ensure that the platform will be independent, our game will be runnable in all operating system, so many users can run the game.

2. Software Architecture

2.1. Subsystem Decomposition

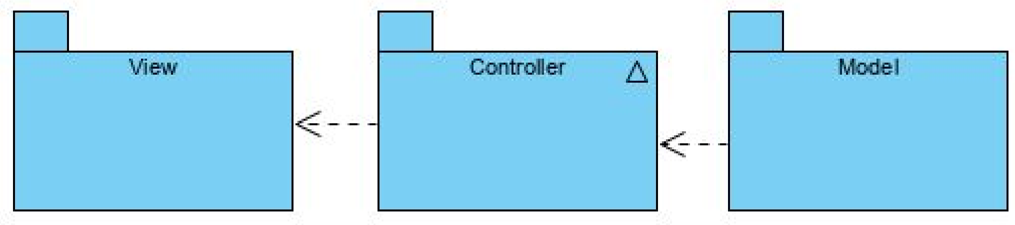


Figure 1: UML representation of system decomposition

A system should be divided into multiple subsystems. This decomposition and decoupling of different systems is necessary for the maintainability of the system. Decomposing different parts also increases the reusability of the said system. Our system consists of three different and distinct subsystems: Model, Controller and UI. Each of these systems also divide into multiple sub-subsystems in themselves. These three subsystems was chosen to follow the MVC: Model-View-Controller design pattern. The Controller subsystems is the main driver of the program: it is the bridge between the View and the Game Model. It's job is to move the game forward, and send the inputs from the View subsystem to the Model subsystem so it can be processed and responded accordingly. The Model subsystem is the Game Model: it consists of classes that makes up the game itself, and carries most of the game data. Controller class progresses through the game according to the information in the Model subsystem. View subsystem is the subsystem that handles the UI part of our program. Its job is to get the input from the user and send it to the Controller subsystem. After Controller subsystem processes it accordingly, View subsystem display the incoming response to the screen so user can continue. Further details about this part will be discussed in their own section.

2.2. Hardware/Software Mapping

The game does not require a network connection or database to operate because the data will be small size such as images, user setting preferences and player account. The game will be implemented in Java. Therefore, the user's computer operating system needs to support Java Runtime Environment. For the UI part, animations and graphical components will be implemented by using JAVAFX libraries. Thus, "Slay the Spire" will require at least Java Development Kit 11. We have preferred JAVAFX because it provides rich set of graphics and it has more concise binding relationships than traditional event listeners. As a result, the game will run in all platform with necessary software installed.

2.3. Persistent Data Management

Slay the Spire does not require a database system. As stated before, the game data will be stored in a binary file. Some data members such as setting preferences, that are made by the user from the options selection in the main menu, will be stored in a modifiable binary file so that they can be updated preferably during the game. Besides, some of the binary files that store the map step process for the player will be in the non modifiable binary file. For the images we will use .png and .jpeg format and for the game music we will use .mp3 and .waw format preferably.

2.4. Access Control and Security

As stated in 2.2., our game does not require network connection or database server, so there will not be any security issue. The game will have only one user type, so we don't design different access levels for host, admin etc. That is, there will not be access controls or security measurements.

2.5. Boundary Conditions

Slay the Spire will be in a .jar file. Thus, if the computer has at least Java Development Kit 11 and it’s operating system support Java Runtime Environment, the game will be run and played with double clicks. The game will be terminated by two ways. One is that the user can terminate it by clicking close button of window frame. The game has also an “EXIT” button on the main menu and user can terminate the game by one click on this button. In the case of instant crash because of the any performance or design issue, system function may not work, and the current data might be lost.

2.6. Design Patterns

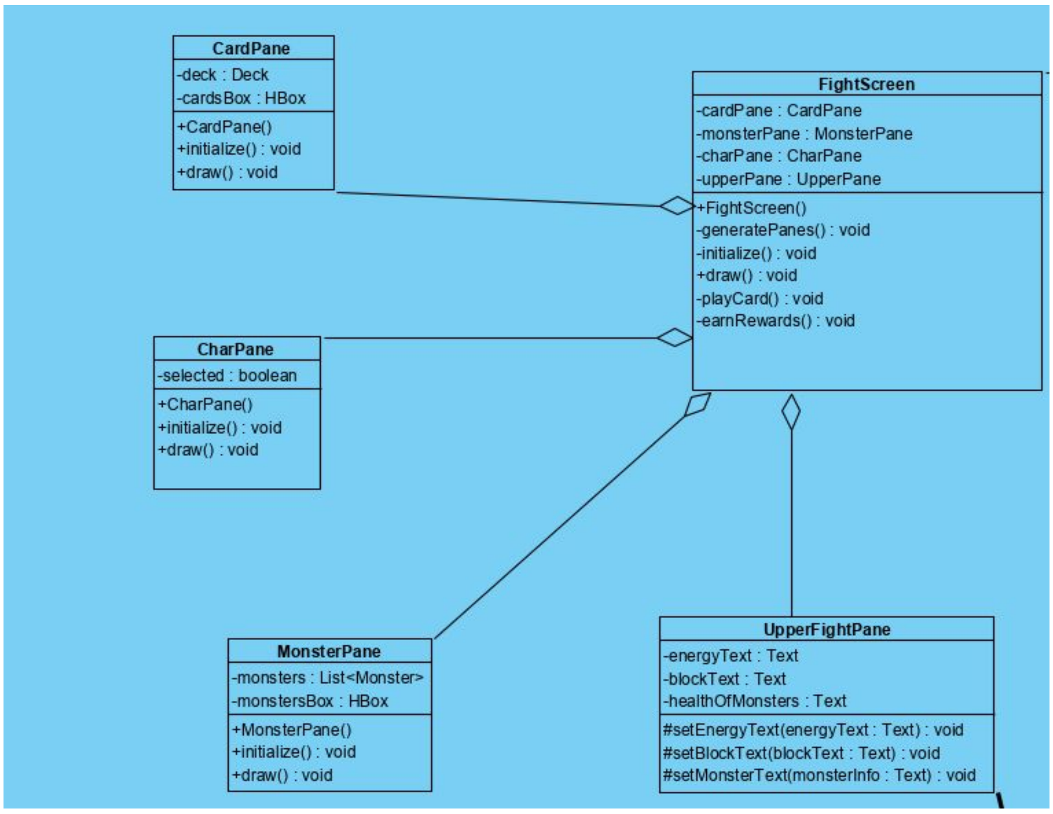


Figure 2: FightScreen

Façade design pattern is used in the development FightScreen, TreasureScreen, MerchantScreen, RestScreen. These Screens are composed of multiple interfaces, but only single higher-level interface is provided to user to reduce complexity, e.g. FightScreen includes CharPane, CardPane ,MonsterPane, UpperFightPane. However, the user does not need to know about any of these as they will only see a single simple interface. For instance, whenever the user plays a card, setHealthText(healthText:Text) function of UpperFightPane will automatically will be called without user noticing it.

State design pattern is used in FightRoomController class. It manages the turns during the fight. For instance, after PREFIGHT and PRETURN, FightRoomController waits in state TURN until user makes a move. The “waiting” is possible thanks to the state design pattern. By using the state design pattern, an user input based cyclic fight is done in FightRoom.

Composite design pattern is used in Room class, as FightRoom and other room types extend that class and these classes have a child property that is also a Room (child is the Room object that comes after a certain room).

3. Subsystem Services

3.1 Controller and Model Subsystem

Model and Controller subsystems include classes that control the game logic and game flow. They interact with the UI subsystem to draw the necessary information to the user screen. Higher resolution of the final object diagram can be found via this link.

3.1.1. Controller subsystem

This subsystem includes the Actions package, as well as the various manager and controller classes that controls the flow of the game. Actions package is the package that contains various actions that happen frequently during the course of the game, like attacking something, getting a power or a relic. MapManager class is the class that manages all of the Rooms that are part of the game, as well as the switching between them. RoomController and its subclasses are the classes that controls what is going to happen in each room, and they each has a counterpart that holds the data of the room in the Model subsystem.

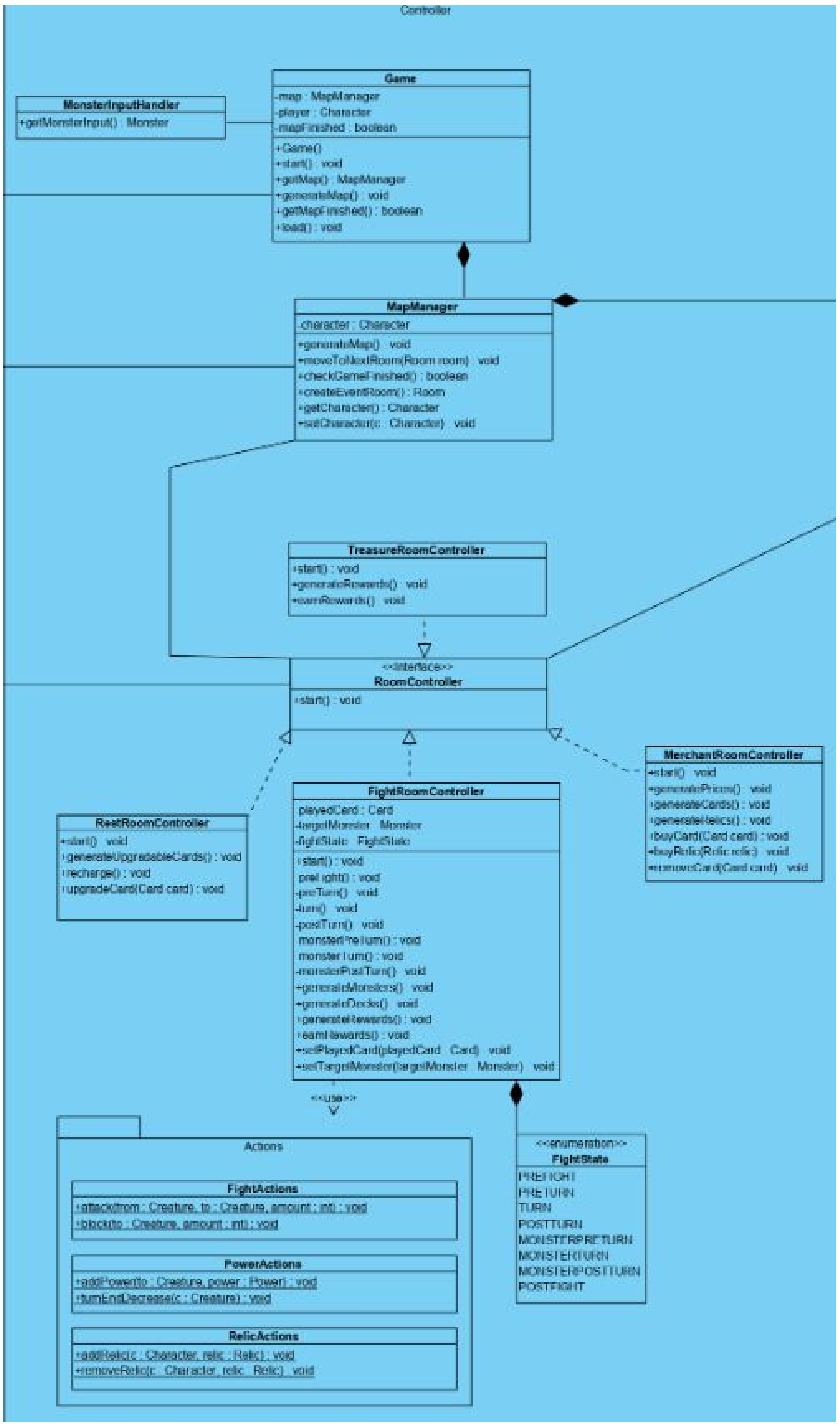


Figure 3: Diagram of Controller classes

3.1.2. Model subsystem

This subsystem includes classes that carry information, and uses the actions defined in the Actions class as necessary. For example, a child of Card can have an AttackAction in its use method to attack, use another action to draw a card. The monsters in each Fight is defined in this category, as well as Relics, Characters, Cards and Powers.

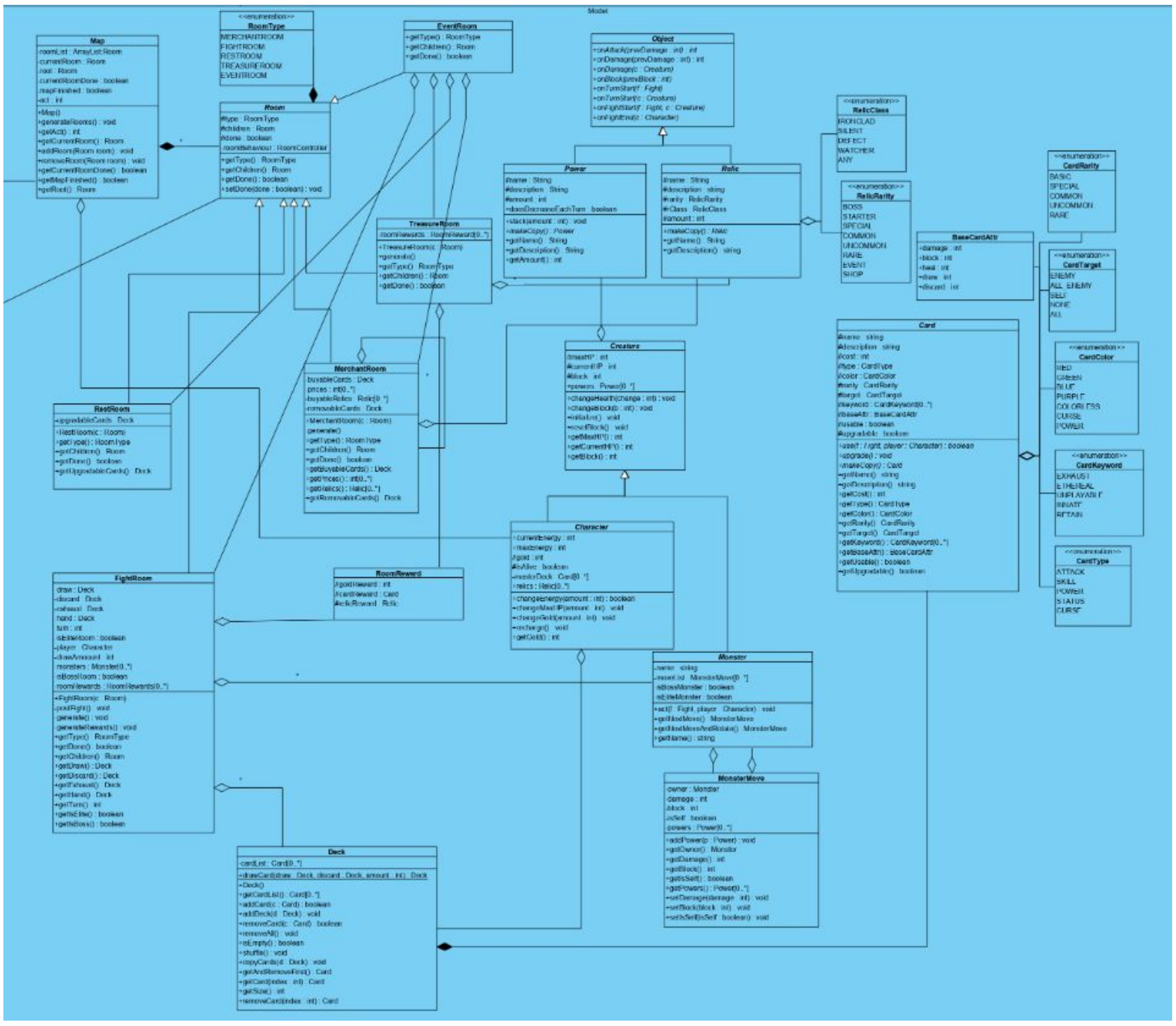


Figure 4: Diagram of Game Models

3.2. UI Subsystem

JavaFX library elements is used in the construction of UI components.Controller class decides the stage of the game depending on the stage,the corresponding Screen is called and initialized. While the game is played,the Screen is constantly updated, then drawn again.Currently, there are two Screens in the game which are menu Screen initialized in Main class and fight Screen constructed by FightScreen class.

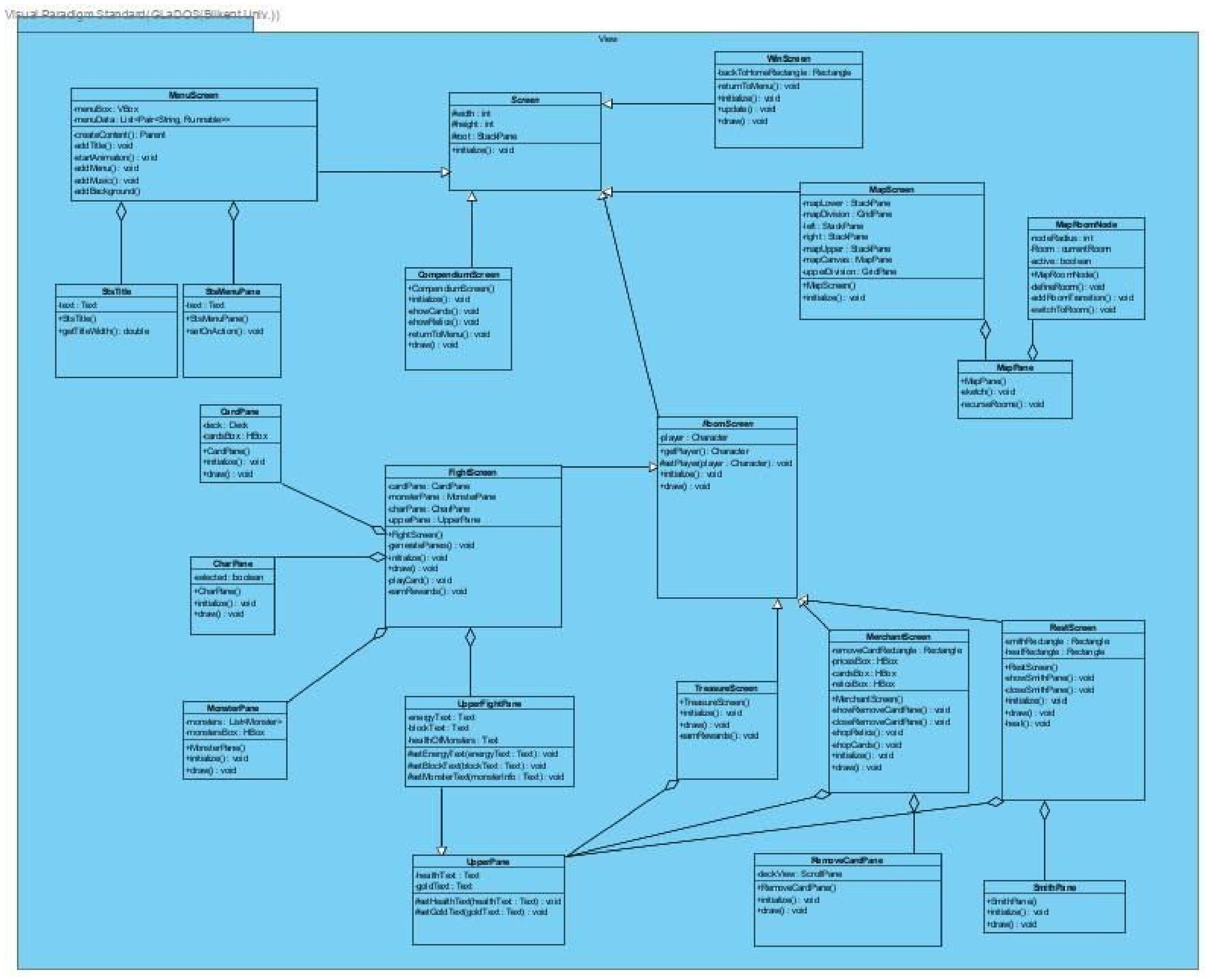


Figure 5: Diagram of UI

4. Low-Level Design

4.1. Trade-offs

4.1.1. Portability vs. Efficiency

Since Java is a portable language that is supported by various platforms, we chose Java Language to implement our game. In addition to this, Java language also requires Java Virtual Machine that affects the efficiency because there will be a virtual environment through the game and operating system. Corollary, portability outweighs the efficiency. However, since we don’t use complicated graphics in our game, the cost of efficiency will be minimum.

4.1.2. Performance vs Memory

The game performance is also important. We focus on the screens and panes in the game and we provide the user a good experience on changing view of screens and panes. For instance, in a fight screen, instead of creating the scene from scratch, we draw the components (attributes) of the scene and redraw them (hence the scene itself).In other words, creating new screens and panes is slower than redrawing (hence updating) the components of the same one. These results come up with an occupation more space in the memory. Therefore, for a better performance, we sacrifice the extra memory space for new attributes.

4.1.3. User Friendliness vs. Functionality

One of our focuses is a good user experience. Since the game should attract the player, we make the game simple, clear and understandable. Thus, the game will not include complex functionality or too many options. For instance, we don’t include the keyboard option so that the user will not effort to learn how to play the game with keyboard, rather than that, user can enjoy the simple game with mouse, and everything will be clear.

4.2 Final Object Design

Diagram itself can be found on this link.

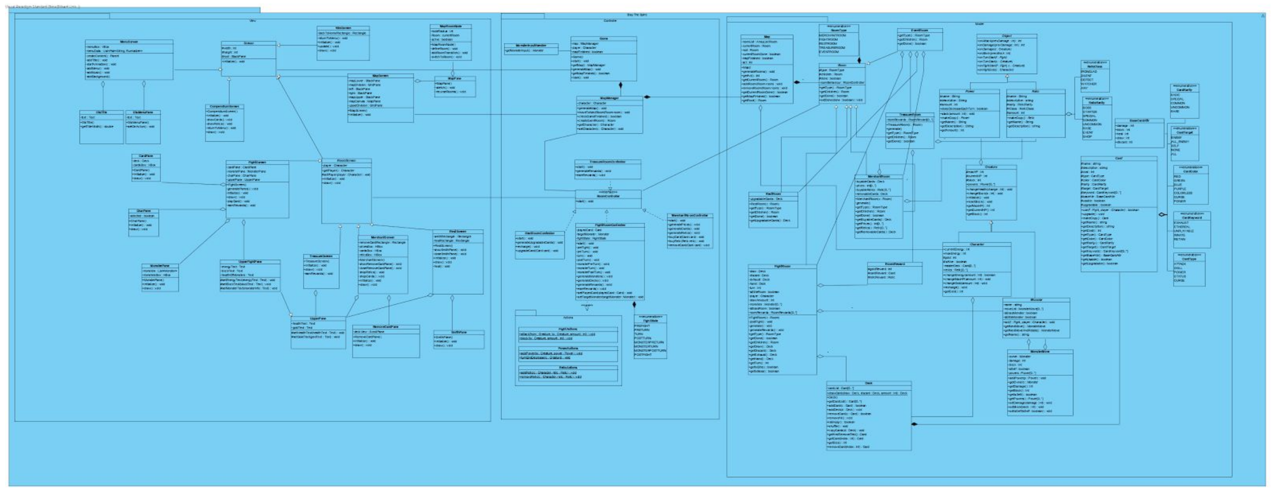


Figure 6: Final Object Diagram

4.3. Packages

4.3.1. Model Package

Model package contains the information about the implementation of the classes which are needed for the logical structure of the game. All of the classes contain crucial information about the game mechanics, such as cards, monsters, relics, powers etc. When combined with the other classes, the game will be able to function as a whole.

4.3.1.1. CardColor class

This class contains the enumeration for card colors (red, green, blue, purple, colorless and curse).

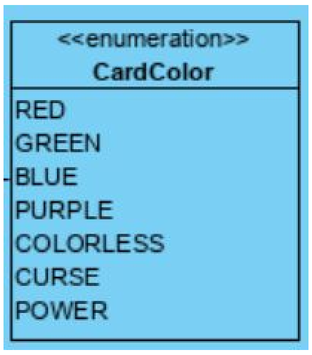


Figure 7: Class diagram of CardColor

4.3.1.2. CardKeyword class

This class contains the enumeration for card keywords (exhaust, ethereal, unplayable, innate and retain).

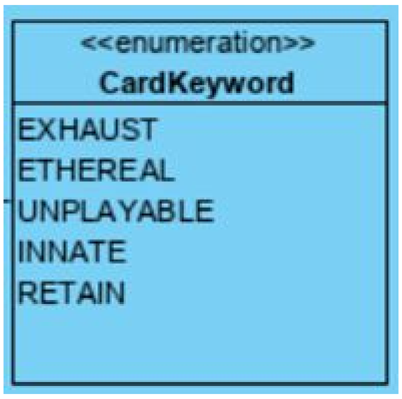


Figure 8: Class diagram of CardKeyword

4.3.1.3. CardRarity class

This class contains the enumeration for card rarities (basic, special, common, uncommon and rare).

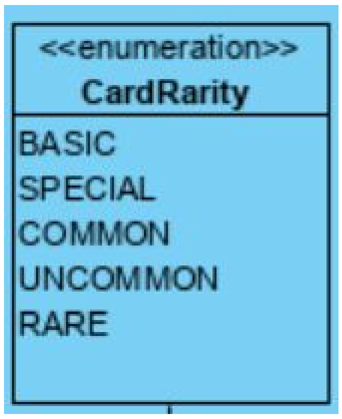


Figure 9: Class diagram of CardRarity

4.3.1.4. CardTarget class

This class contains the enumeration about who the card's target is:

- ENEMY: The card targets just 1 enemy.
- ALL_ENEMY: The card targets all of the enemies.
- SELF: The card targets the player.
- NONE: The card doesn't have a target.
- ALL: The card targets everybody.

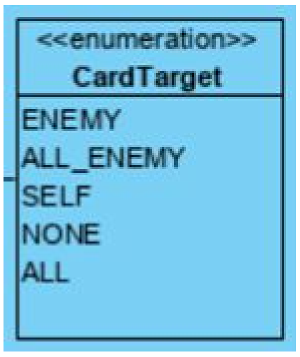


Figure 10: Class diagram of CardTarget

4.3.1.5. CardType class

This class contains the enumeration for the card types (attack, skill, power, status and curse).

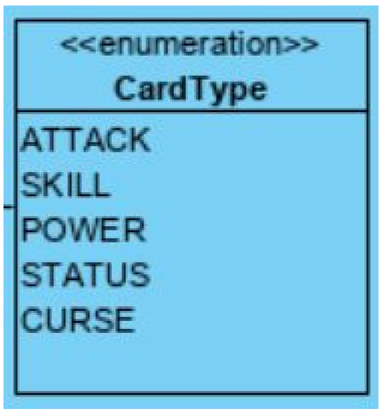


Figure 11: Class diagram of CardType

4.3.1.6. BaseCardAttributes class

This class contains the information about a card’s general specifications.

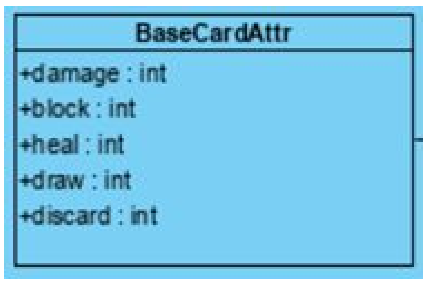


Figure 12: Class diagram of BaseCardAttr

Attributes:

- public int damage:

  How much damage the card will do.

- public int block:

  How much block the card will grant.

- public int heal:

  How much HP the card will heal.

- public int draw:

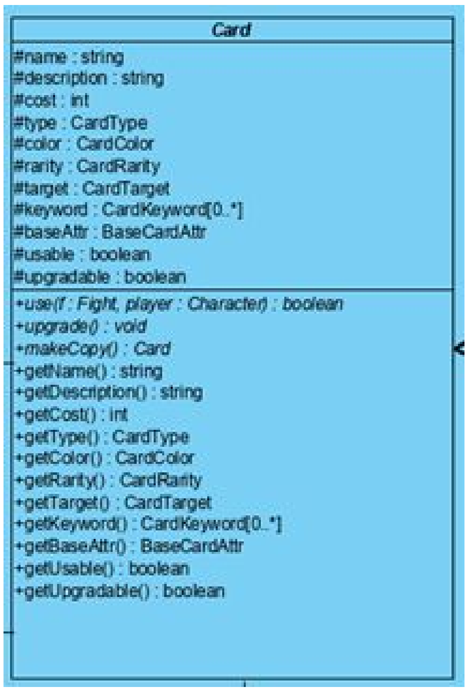
  How many cards the player will draw.

- public int discard:

  How many cards the player will discard.

4.3.1.7. Card class

This class is the skeleton of all cards. Every card that is implemented in the game inherits from this class. It contains the attributes of the card and its methods.



```
Card
#name : string
#description : string
#cost : int
#type : CardType
#color : CardColor
#rarity : CardRarity
#target : CardTarget
#keyword : CardKeyword[0..*]
#baseAttr : BaseCardAttr
#usable : boolean
#upgradable : boolean
+use(f : Fight, player : Character) : boolean
+upgrade() : void
+makeCopy() : Card
+getName() : string
+getDescription() : string
+getCost() : int
+getType() : CardType
+getColor() : CardColor
+getRarity() : CardRarity
+getTarget() : CardTarget
+getKeyword() : CardKeyword[0..*]
+getBaseAttr() : BaseCardAttr
+getUsable() : boolean
+getUpgradable() : boolean
```

Figure 13: Class diagram of Card

Attributes:

- protected String name:

  The name of the card.

- protected String description:

  The description of the card.

- protected int cost:

  The energy amount that it costs to use the card.

- protected CardType type:

  The type of the card.

- protected CardColor color:

  The color of the card.

- protected CardRarity rarity:

  The rarity of the card.

- protected CardTarget target:

  The target of the card.

- protected CardKeyword keyword:

  The keyword of the card.

- protected BaseCardAttributes baseAttr:

  Basic attributes of the card.

- protected boolean usable:

  Indication of whether the card is usable or not.

- protected boolean upgradable:

  Indication of whether the card is upgradable or not.

Methods:

- public abstract boolean use(Fight f, Character player):

  Has the set of instructions that will be executed when the card is played.

- public abstract void upgrade():

  Upgrades the card.

- public abstract Card makeCopy():

  Returns a shallow copy of the card.

- public String getName():

  Returns the name of the card.

- public CardColor getColor():

  Returns the color of the card.

- public CardType getType():

  Returns the type of the card.

- public CardRarity getRarity():

  Returns the rarity of the card.

- public int getCost():

  Returns the energy cost of the card.

- public CardTarget getTarget():

  Returns the target of the card.

- public CardKeyword getKeyword():

  Returns the keyword of the card.

- public BaseCardAttributes getBaseAttr():

  Returns the basic attributes of the card.

- public boolean isUsable():

  Returns whether the card is usable.

- public boolean isUpgradable():

  Returns whether the card is upgradable.

- public String getDescription():

  Returns the description of the card.

4.3.1.8. Deck class

This class contains a collection of cards and is essentially a card deck. A deck can be of different types. Functionalities such as drawing cards, adding and removing cards to a deck etc. are implemented in this class.

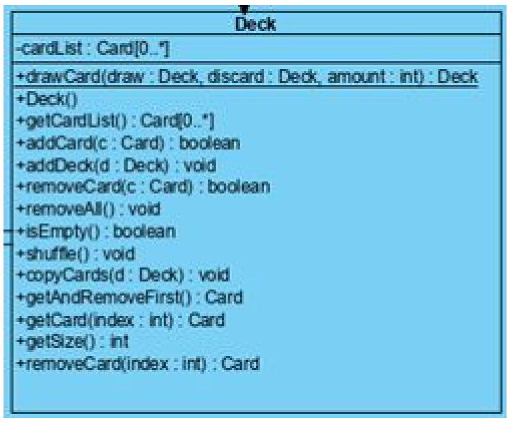

Deck
-cardList : Card[0..*]
+drawCard(draw : Deck, discard : Deck, amount : int) : Deck
+Deck()
+getCardList() : Card[0..*]
+addCard(c : Card) : boolean
+addDeck(d : Deck) : void
+removeCard(c : Card) : boolean
+removeAll() : void
+isEmpty() : boolean
+shuffle() : void
+copyCards(d : Deck) : void
+getAndRemoveFirst() : Card
+getCard(index : int) : Card
+getSize() : int
+removeCard(index : int) : Card

Figure 14: Class diagram of Deck

Attributes:

- private ArrayList<Card> cardList:

  A list of Card class instances.

Methods:

- public Deck():

  Constructor of the class.

- public ArrayList<Card> getCardList():

  Returns the card list.

- public boolean addCard(Card c):

  Adds a certain card to the deck.

- public void addDeck(Deck d):

  Adds a deck of cards to the existing deck.

- public boolean removeCard(Card c):

  Removes a certain card from the deck.

- public void removeAll():

  Removes every card from the deck.

- public boolean isEmpty():

  Returns whether the deck is empty.

- public void shuffle():

  Shuffles the deck.

- public void copyCards(Deck d):

  Copies the cards into another deck.

- public Card getAndRemoveFirst():

  Gets the first card and then removes it.

- public Card getCard(int index):

  Gets a card based on a specified index.

- public int getSize():

  Returns the size of the deck.

- public Card removeCard(int card):

  Removes a card based on a specified index.

- public static Deck drawCard(Deck draw, Deck discard, int amount):

  Draws a specific amount of cards from the draw pile, and if needed, shuffles cards from the discard pile into the draw pile.

4.3.1.9. Creature class

This class is the skeleton for all creatures in the game i.e. monsters and characters (heroes). Every creature inherits from this class. It contains the information about the HP, powers and blocks of the creatures. Creature HP's and blocks can be manipulated here.

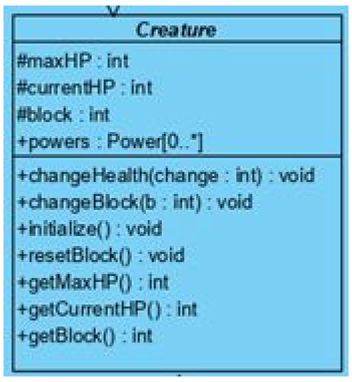


Figure 15: Class diagram of Creature

Attributes:

- protected int maxHP:

  Maximum HP the creature can have.

- protected int currentHP:

  Current HP of the creature.

- protected int block:

  Block amount that the creature has.

- public ArrayList<Power> powers:

  A list of the powers of the creature.

Methods:

- public void changeHealth(int change):

  Changes the creature’s HP by a specified amount.

- public void changeBlock(int b):

  Changes the creature’s block by a specified amount.

- public void initialize():

  Initializes the max HP and the current HP values to 100.

- public void resetBlock():

  Resets the block to 0.

- public int getCurrentHP():

  Returns the current HP.

- public int getMaxHP():

  Returns the max HP.

- public int getBlock():

  Returns the block.

4.3.1.10. Character class

This class inherits from the Creature class and is the skeleton for all of the characters. It contains the information and functionality that monsters don’t have, but the characters do, such as having relics, energy and gold.

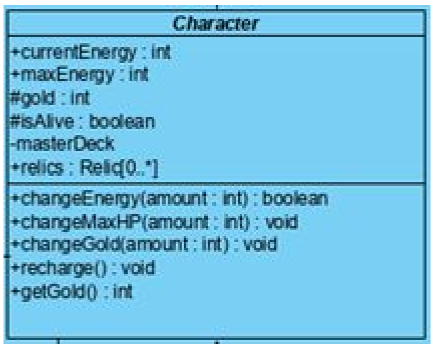


Figure 16: Class diagram of Character

Attributes:

- private int currentEnergy:

  Current energy of the character.

- private int maxEnergy:

  Maximum energy the character can have.

- protected int gold:

  The amount of gold the character possesses.

- protected boolean isAlive:

  Indication of whether the character is alive.

- private Deck masterDeck:

  Master deck of the character (all of the cards that the character has).

- public ArrayList<Relic> relics:

  A list of relics the character has obtained.

Methods:

- public boolean changeEnergy(int usage):

  Changes the energy of the character by a specified amount.

- public void changeMaxHP(int hp):

  Changes the max HP of the character by a specified amount.

- public void changeGold(int amount):

  Changes the gold amount of the character by a specified amount.

- public void recharge():

  Sets the current HP to (maxHP * 3) / 10).

- public int getGold():

  Returns the amount of gold the character has.

4.3.1.11. Monster class

This class is the base class for all monsters. It extends the Creature class. Every monster inherits from this class.

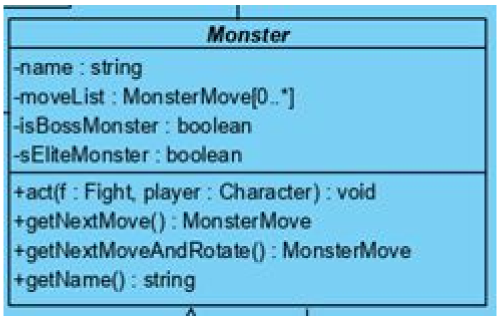

Monster
-name : string
-moveList : MonsterMove[0..*]
-isBossMonster : boolean
-sEliteMonster : boolean
+act(f : Fight, player : Character) : void
+getNextMove() : MonsterMove
+getNextMoveAndRotate() : MonsterMove
+getName() : string

Figure 17: Class diagram of Monster

Attributes:

- private String name:

  Name of the monster.

- private ArrayList<MonsterMove> moveList:

  List of the moves that the monster is going to make.

- private isBossMonster:

  Whether the monster is a boss.

- private isEliteMonster:

  Whether the monster is an elite.

Methods:

- public void act(Fight f, Character player):

  Makes the move to the player.

- public MonsterMove getNextMove():

  Returns the first move of the list.

- public MonsterMove getNextMoveAndRotate():

  Returns the first move of the list and adds it to the end of the list.

- public String getName():

  Returns the name of the monster.

4.3.1.12. MonsterMove class

This class represents the move of a monster. The move has an owner, and the specified amounts of damage/block. It also has a target. Moves get affected by powers as well. Therefore, this information is all kept inside this class.

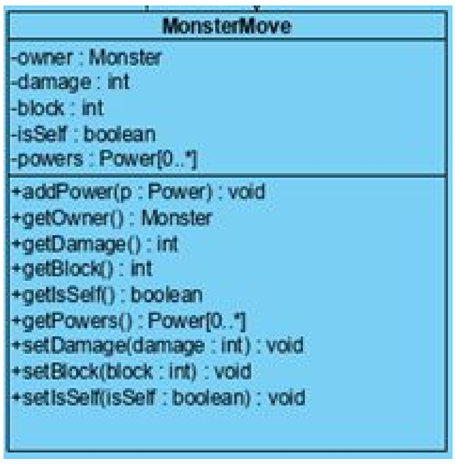


Figure 18: Class diagram of MonsterMove

Attributes:

- private Monster owner:

  The owner of the move.

- private int damage:

  The amount of damage the move is going to make.

- private int block:

  The amount of block the move is going to provide.

- private boolean isSelf:

  Whether the move affects the monster itself.

- private ArrayList<Power> powers:

  The list of powers that are in effect.

Methods:

- public MonsterMove(Monster o):

  Constructor for MonsterMove. Initializes the properties.

- Monster getOwner():

  Returns the owner.

- boolean getIsSelf():

  Returns isSelf.

- public int getDamage:

  Returns the damage.

- public int getBlock():

  Returns the block.

- public ArrayList<Power> getPowers():

  Returns the powers.

- void setDamage(int damage):

  Changes the damage value.

- void setBlock(int block):

  Changes the block value.

- public void setSelf(boolean self):

  Changes the isSelf value.

- public void addPower(Power p):

  Adds a power to the powers collection.

4.3.1.13. Object class

This class is the skeleton for all objects in the game such as powers and relics. Every object in the game inherits from this class. It contains the functionalities that objects have. Objects override the needed methods.

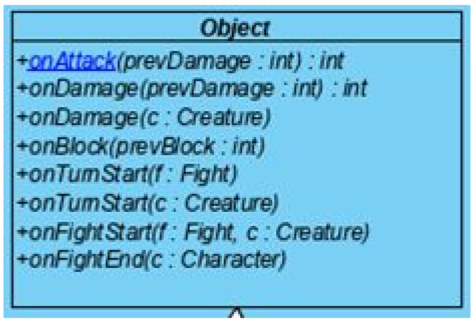

Object
+onAttack(prevDamage : int) : int
+onDamage(prevDamage : int) : int
+onDamage(c : Creature)
+onBlock(prevBlock : int)
+onTurnStart(f : Fight)
+onTurnStart(c : Creature)
+onFightStart(f : Fight, c : Creature)
+onFightEnd(c : Character)

Figure 19: Class diagram of Object

Methods:

- public int onAttack(int prevDamage):

  When the creature attacks, the damage they make changes by a specified amount.

- public int onDamage(int prevDamage):

  When the creature is damaged, the damage they get changes by a specified amount.

- public void onDamage(Creature c):

  When the creature is damaged, it will get affected in some way.

- public int onBlock(int prevBlock):

  When the creature blocks an attack, the block changes by a specified amount.

- public void onTurnStart(Fight f):

  At the start of every turn of a fight, a set of instructions are executed.

- public void onTurnStart(Creature c):

  At the start of every turn, a set of instructions are executed and the creature gets affected.

- public void onFightStart(Fight f, Character c):

  At the start of every fight, a set of instructions are executed.

- public void onFightEnd(Character c):

  At the end of every fight, a set of instructions are executed.

4.3.1.14. RelicRarity class

This class contains the enumeration for the relic rarities (boss, starter, special, common, uncommon, event, shop, rare).

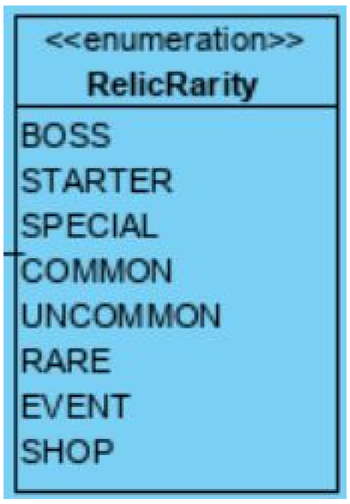


Figure 20: Class diagram of RelicRarity

4.3.1.15. RelicClass class

This class contains the enumeration for the relic classes (Ironclad, Watcher, Defect, Silent, Any).

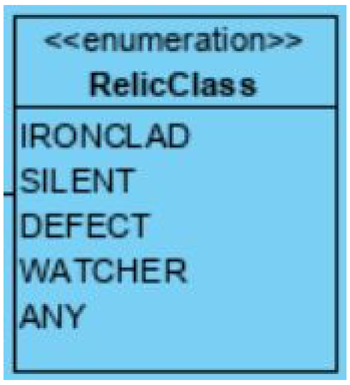


Figure 21: RelicClass

4.3.1.16. Relic class

This class extends the Object class. It is the skeleton for all of the relic classes.

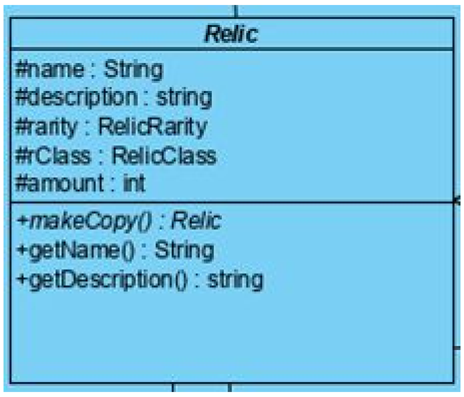


Figure 22: Class diagram of Relic

Attributes:

- protected String name:

  Name of the relic.

- protected String description:

  Description of the relic.

- protected RelicRarity rarity:

  Rarity of the relic.

- protected RelicClass rClass:

  Class of the relic.

- protected int amount:

  The amount of X (block, damage…) that the relic will affect.

Methods:

- public abstract Relic makeCopy():

  Makes a shallow copy of the relic.

- public String getName():

  Returns the name of the relic.

- public String getDescription():

  Returns the description of the relic.

4.3.1.17. Power class

This class inherits from the Object class. It is the skeleton for all of the power classes.

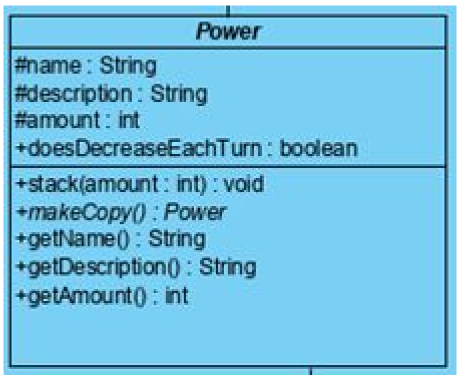


Figure 23: Class diagram of Power

Attributes:

- protected String name:

  Name of the power.

- protected String description:

  Description of the power.

- protected int amount:

  The amount of X (block, damage…) that the power will affect.

- public boolean doesDecreaseEachTurn:

  Indication of whether the effect of the power decreases each turn.

Methods:

- public void stack(int a):

  Stacks the power by a specified amount (makes it more effective).

- public abstract Power makeCopy():

  Makes a shallow copy of the power.

- public String getName():

  Returns the name of the power.

- public String getDescription():

  Returns the description of the power.

- public int getAmount():

  Returns the “amount” attribute.

4.3.1.18. RoomType class

This class contains the enumeration for the room types (merchant, fight, rest, treasure, event).

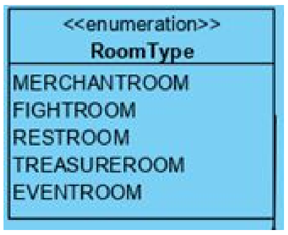


Figure 24: Class diagram of RoomType

4.3.1.19. Room class

This class represents a room in the game map. It is the skeleton for all of the room classes. It has a RoomController property that regulates the events in the room and the mechanics of the room. Every room class inherits from this class.

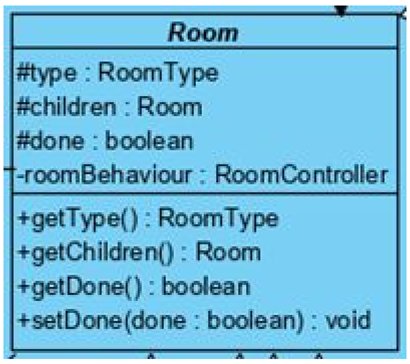


Figure 25: Class diagram of Room

Attributes:

- protected RoomType type:

  Type of the room.

- protected Room children:

  The room that comes after this room.

- protected boolean done:

  Indication of whether the room was succesfully passed by the player.

- private RoomController roomBehaviour:

  Controls the flow of events in the room.

Methods:

- public abstract RoomType getType():

  Returns the room type.

- public abstract Room getChildren():

  Returns the “children” attribute.

- public abstract boolean getDone():

  Returns the “done” attribute.

- public void setDone(boolean done):

  Sets the done attribute to the given Boolean.

4.3.1.20. RoomReward class

This class represents the rewards that can be earned from a room.

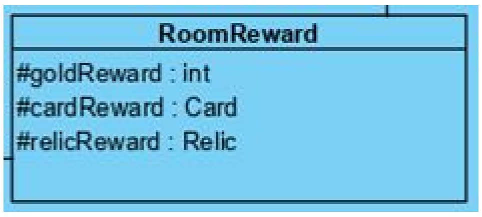


Figure 26: Class diagram of RoomReward

Attributes:

- protected int goldReward:

  The amount of gold that will be rewarded.

- protected Card cardReward:

  The card that will be rewarded.

- protected Relic relicReward:

  The relic that will be rewarded.

4.3.1.21. EventRoom class

This class represents the room in which the player will encounter an event. It extends the Room class and overrides the relevant methods.

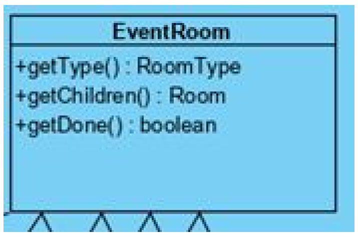


Figure 27: Class diagram of EventRoom

Methods:

- public RoomType getType():

  Returns the room type.

- public Room getChildren():

  Returns the room that comes after this room.

- public boolean getDone():

  Returns whether the room has been cleared.

4.3.1.22. FightRoom class

This class represents the room in which the player has to fight monsters. It extends the Room class and overrides the relevant methods. It does not directly control the combat mechanism.

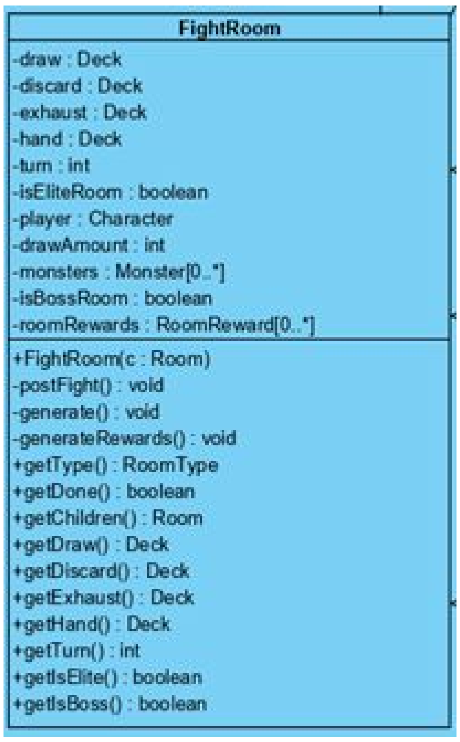


Figure 28: Class diagram of FightRoom

Attributes:

- private Deck draw:

  The draw pile.

- private Deck discard:

  The discard pile.

- private Deck exhaust:

  The exhaust pile.

- private Deck hand:

  Hand of the player.

- private int turn:

  Turn number.

- private boolean isEliteRoom:

  Whether the monster is an elite.

- private boolean isBossRoom:

  Whether the monster is a boss.

- private Character player:

  The player (user).

- private int drawAmount:

  Amount of cards that will be drawn.

- private int goldAmount:

  Amount of gold that will be earned after the fight.

- private ArrayList<Monster> monsters:

  Monsters in the room.

- private ArrayList<RoomReward> roomRewards:

  Rewards that will be won if the player kills the monsters.

Methods:

- public FightRoom(Room c):

  Constructor of the class where c is the next room.

- private void postFight():

  Events that happen after the fight.

- private void generate():

  The function to generate monsters.

- private void generateRewards():

  The function to generate rewards.

- public RoomType getType():

  Returns the room type.

- public Room getChildren():

  Returns the room that comes after this room.

- public boolean getDone():

  Returns whether the room has been cleared.

- public Deck getDraw():

  Returns the draw pile.

- public Deck getDiscard():

  Returns the discard pile.

- public Deck getExhaust():

  Returns the exhaust pile.

- public Deck getHand():

  Returns the hand of the player.

- public boolean getIsElite():

  Returns whether the monster is elite.

- public boolean getIsBoss():

  Returns whether the monster is a boss.

4.3.1.23. MerchantRoom class

This class represents the room in which the player can visit the merchant. It extends the Room class and overrides the relevant methods. It does not directly control the events.

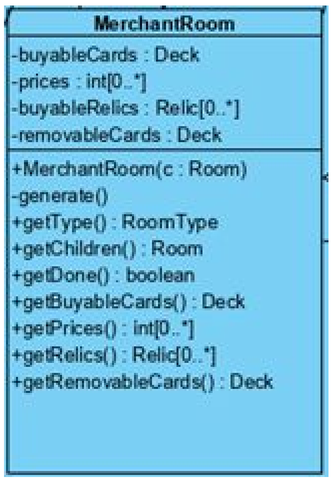


Figure 29: Class diagram of MerchantRoom

Attributes:

- private Deck buyableCards:

  Cards that can be bought.

- private ArrayList<Integer> prices:

  Prices of the items.

- private ArrayList<Relic> buyableRelics:

  Relics that can be bought.

- private Deck removableCards:

  Cards that can be removed from the player’s deck.

Methods:

- public MerchantRoom(Room c):

  Constructor of the class where c is the next room.

- private void generate():

  Generates the cards and the prices.

- public RoomType getType():

  Returns the room type.

- public Room getChildren():

  Returns the next room.

- public boolean getDone():

  Returns whether the room has been cleared.

- public Deck getBuyableCards():

  Returns the cards that can be bought.

- public ArrayList<Integer> getPrices():

  Returns the prices.

- public ArrayList<Relic> getRelics():

  Returns the relics that can be bought.

- public Deck getRemovableCards():

  Returns the removable cards.

4.3.1.24. RestRoom class

This class represents the room in which the player can rest. It extends the Room class and overrides the relevant methods. It does not directly control the flow of events in the room.

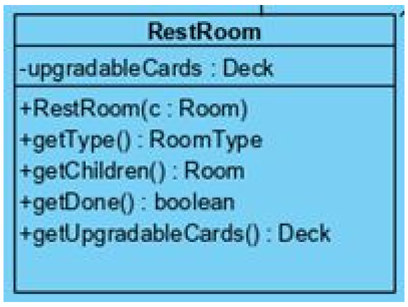


Figure 30: Class diagram of RestRoom

Attributes:

- private Deck upgradableCards:

  Cards that the user can upgrade.

Methods:

- public RestRoom(Room c):

  Constructor of the class where c is the next room.

- public RoomType getType():

  Returns the room type.

- public Room getChildren():

  Returns the next room.

- public boolean getDone():

  Returns whether the room has been cleared.

- public Deck getUpgradableCards():

  Returns the upgradable cards.

4.3.1.25. TreasureRoom class

This class represents the room in which the player can get rewarded with prizes. It extends the Room class and overrides the relevant methods. It does not directly control the flow of events in the room.

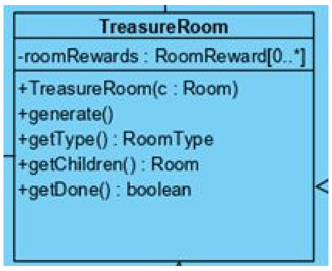


Figure 31: Class diagram of TreasureRoom

Attributes:

- private RoomReward roomRewards:

  Rewards that can be earned from the room.

Methods:

- public TreasureRoom(Room c):

  Constructor of the class where c is the next room.

- public void generate():

  Generates the relics and the gold amount.

- public RoomType getType():

  Returns the room type.
- public Room getChildren():

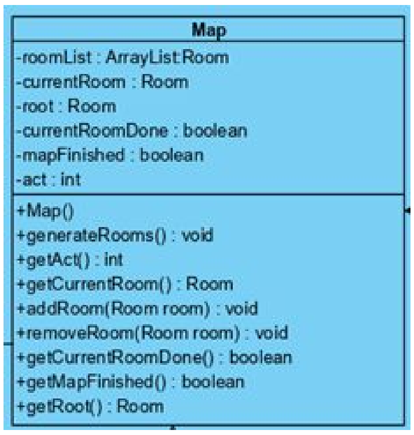
  Returns the next room.

- public boolean getDone():

  Returns whether the room has been cleared.

4.3.1.26. Map class

This class represents the map of the game. The current room, the act, the map layout are all controlled in this class.



```
Map
-roomList : ArrayList:Room
-currentRoom : Room
-root : Room
-currentRoomDone : boolean
-mapFinished : boolean
-act : int
+Map()
+generateRooms() : void
+getAct() : int
+getCurrentRoom() : Room
+addRoom(Room room) : void
+removeRoom(Room room) : void
+getCurrentRoomDone() : boolean
+getMapFinished() : boolean
+getRoot() : Room
```

Figure 32: Class diagram of Map

Attributes:

- private ArrayList<Room> roomList:

  List of the rooms in the map.

- private Room root:

  The first room of the map.

- private Room currentRoom:

  Current room that the player is playing in.

- private boolean currentRoomDone:

  Whether the current room is cleared.

- private boolean mapFinished:

  Whether the map is all cleared.

- private int act:

  Act number.

Methods:

- public Map():

  Constructor of the map. Initializes the act as 1.

- public void generateRooms():

  Generates the rooms.

- public int getAct():

  Returns the act.

- public Room getCurrentRoom():

  Returns the current room.

- public void addRoom(Room room):

  Adds the room to the room list.

- public void removeRoom(Room room):

  Removes the room from the room list.

- public boolean getCurrentRoomDone():

  Returns whether the current room is done.

- public boolean getMapFinished ():

  Returns whether the map is all finished.

- public Room getRoot():

  Returns the root room.

4.3.2. UI (User Interface) Package

The UI components of the game is constructed by Java FX library elements and utilities. The Main class of UI starts the menu screen by initializing a Stage in Java FX and forms a basis for the application, in other words extends Application class coming from Java FX libraries. Screen class is the key element in the interface design. The UI switches from one Screen object to another to change the UI screen. Therefore the classes built are constructed with respect to this principle.

Currently, there are two Screens in the game which are menu Screen initialized in Main class and fight Screen constructed by FightScreen class.

4.3.2.1. Screen Class

Screen class is a building block for the screens in the game. It extends Scene class of the JavaFX library.

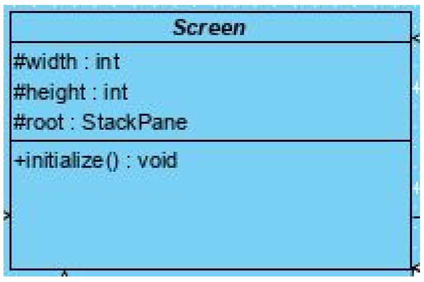


Figure 33: Screen Class Diagram

Attributes:

- StackPane root:

  Root pane where the game view is drawn on.
- int width:

  Width of the Screen
- int height:

  Height of the Screen.

Methods:

- private void initialize():

  Initializes the attributes of the current pane.

4.3.2.2. MenuScreen Class

MenuScreen class is the starter of the UI and extends Screen class.

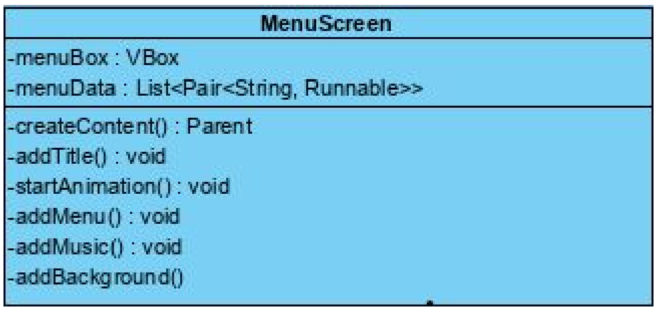


Figure 34: MenuScreen Class Diagram

Attributes:

- private List<Pair<String,Runnable>> menuData:

  Keeps the menu choices as a list of string and runnable data. Runnable data corresponding to one menu choice runs the proper action on click. For example, when menu button to fight is clicked, it creates a new FightScreen and switches to that Screen.
- private VBox menuBox:

  List of vertically aligned boxes. They form a basis for the menu.

Methods:

- private Parent createContent():
  Creates the content for the current Screen which should be a type of Parent. In this case, Parent is the root Pane that is added to the current Screen. This method calls the relevant methods to add menu, title and background.
- private void addBackground():
  Adds an background image to the current Screen.
- private void addTitle():
  Adds a title to the current pane by creating a new StsTitle.
- private void addAnimation():
  Adds an opening animation to the items of menuBox.
- private void addMenu():
  Adds a menu to the current pane by creating a new StsMenuPane.
- private void addMusic():
  Adds a music to the current Screen.

4.3.2.3. FightScreen Class

Duty of the FightScreen class is displaying a Screen for the fights in the game. It is split into four parts to give the player a better view of the game and an access to interactions with the game. It is a RoomScreen.

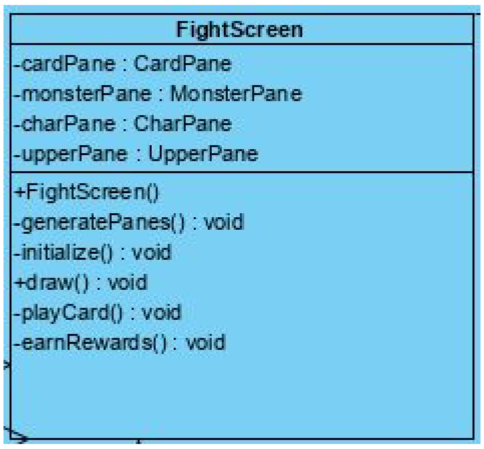


Figure 35: FightScreen Class Diagram

Attributes:

- private CharPane charPane:
  This pane corresponds to the middleleft of the fight Screen and includes friendly characters. It is a type of StackPane.
- private MonsterPane monsterPane:
  This pane corresponds to the middleright of the fight Screen and includes enemy monsters. It is a type of StackPane.
- private UpperPane upperPane:
  Upper part of the fight Screen. It includes a view of menu buttons, the information about the character and its items.
- private CardPane cardPane:
  This pane displays the cards, decks and the interactions for them.

Methods:

- public FightScreen():
  Constructor of the class that calls its superclass constructor and the methods to initialize the current Screen.
- private void generatePanes():
  Generates the panes inside this Screen.
- private void playCards():
  Adds interactions to play the cards in the Screen.
- private void earnRewards():
  Adds interactions to earn the rewards in the Screen.
- private void initialize():
  Initializes the attributes of the current pane.
- private void draw( ):
  Draws the view.

4.3.2.4. StsMenuPane Class

StsMenuPane class creates and shapes the menu part of the MenuScreen. For each of the menu data (menu choice), StsMenuPane is created and called.

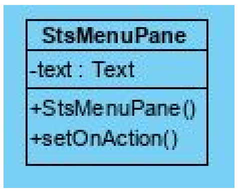


Figure 36: StsMenuPane Class Diagram

Attributes:

- private Text text:

  The text area for each of the menu screen buttons.

Methods:

- public StsMenuPane():

  Constructor of the class.
- public void setOnAction():

  This method attaches an action to each button created by StsMenuPane.

4.3.2.5. StsTitle Class

StsTitle class creates and shapes the title part of the MenuScreen.

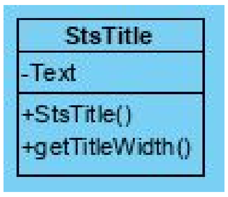


Figure 37: StsTitle Class Diagram

Attributes:

- private Text text:

  It includes the title of the game which is Slay the Spire.

Methods:

- public StsTitle():

  Constructor of the class.

- public void getTitleWidth():
  Returns the width of the title for alignment purposes.

4.3.2.6. CardPane Class

CardPane is a StackPane and it displays the information about cards, piles and it enables interaction with the cards.

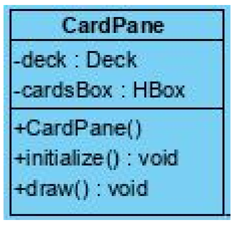


Figure 38: CardPane Class Diagram

Attributes:

- private Deck deck:
  It is the deck of the character that is useful for game mechanics.
- private cardsBox HBox:
  This horizontal box contains the view for each card in the play.

Methods:

- public CardPane():
  Constructor of the class.
- public void draw():
  Draws the view for CardPane.
- public void initialize():
  Initializes the view for CardPane.

4.3.2.7. CharPane Class

CharPane class is a StackPane and includes the view of friendly characters.

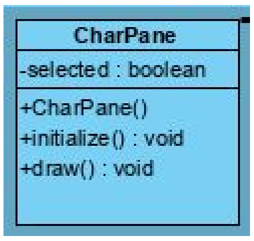


Figure 39: CharPane Class Diagram

Attributes:

- private boolean selected:

  Determines whether the character is selected or not.

Methods:

- public CharPane():

  Constructor of the class.
- public void draw():

  Draws the view for CharPane.
- public void initialize():

  Initializes the view for CharPane.

4.3.2.8. MonsterPane Class

MonsterPane class includes the view of the enemy monsters and enables to interact with them.

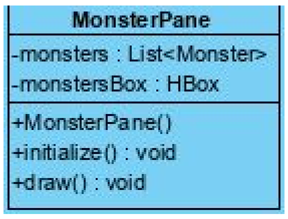


Figure 40: MonsterPane Class Diagram

Attributes:

- private List<Monster> monsters:

  List of the monsters in the pane
- private HBox monsterBox:

Horizontal boxes for the view of the monsters in the pane.

Methods:

- public MonsterPane():

  Constructor of the class.
- public void draw():

  Draws the view for MonsterPane.
- public void initialize():

  Initializes the view for MonsterPane.

4.3.2.9. RoomScreen Class

RoomScreen is an abstract class for the all the rooms in the game. It extends Screen class.

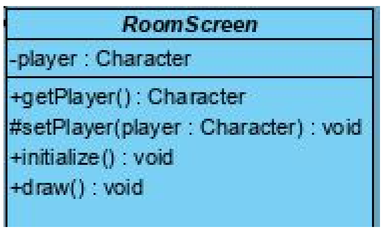


Figure 41: RoomScreen Class Diagram

Attributes:

- private Character player:

  Chosen character of the run.

Methods:

- public void getPlayer():

  Returns the player in the Screen.
- protected void setPlayer():

  Changes the character in the Screen.
- public void draw():

  Draws the view for RoomScreen.
- public void initialize():

  Initializes the view for RoomScreen.

4.3.2.10. UpperFightPane Class

This class creates the upper pane of the fight screen.

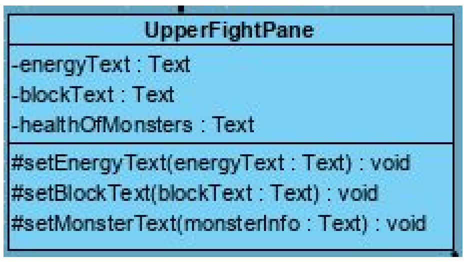

UpperFightPane
-energyText : Text
-blockText : Text
-healthOfMonsters : Text
#setEnergyText(energyText : Text) : void
#setBlockText(blockText : Text) : void
#setMonsterText(monsterInfo : Text) : void

Figure 42: UpperFightPane Class Diagram

Attributes:

- private Text energyText:

  The display for the amount of energy.
- private Text blockText:

  The display for the amount of block.
- private Text healthOfMonsters:

  THe display for the info about monsters

Methods:

- protected void setEnergyText():

  Changes the display for the amount of energy.
- protected void setBlockText():

  Changes the display for the amount of block.
- protected void setMonsterText():

  Changes the display for the monster info.

4.3.2.11. UpperPane Class

The pane in the upside of every room. It displays the info related to the character.

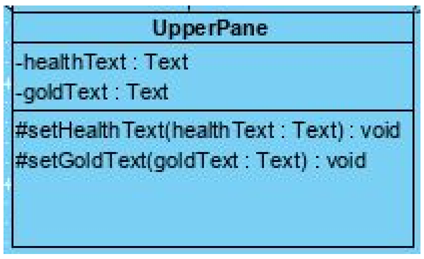

UpperPane
-healthText : Text
-goldText : Text
#setHealthText(healthText : Text) : void
#setGoldText(goldText : Text) : void

Figure 43: UpperPane Class Diagram

Attributes:

- private Text healthText:

  The display for the amount of health.
- private Text goldText:

  The display for the amount of gold.

Methods:

- protected void setHealthText():

  Changes the display for the amount of energy.
- protected void setGoldText():

  Changes the display for the amount of block.

4.3.2.12. TreasureScreen Class

The display screen for the treasure room.

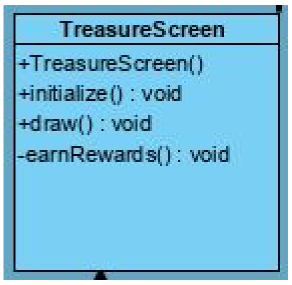


Figure 44: UpperPane Class Diagram

Methods:

- public TreasureScreen():

  Constructor of the TreasureScreen.
- public void draw():

  Draws the view for TreasureScreen.
- public void initialize():

  Initializes the view for TreasureScreen.
- private void earnRewards():

  This function opens the treasure and claims the treasures.

4.3.2.13. MerchantScreen Class

The display screen for the merchant room.

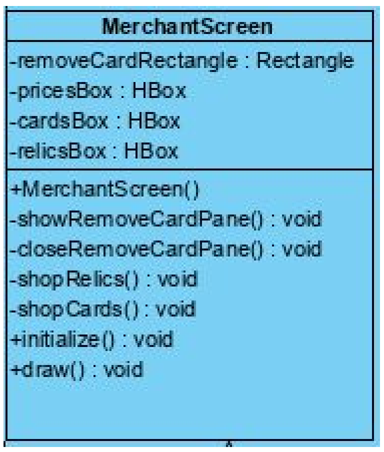


Figure 45: MerchantScreen Class Diagram

Attributes:

- private Rectangle removeCardRectangle:

  The display for the card removal part of the shop.
- private HBox pricesBox:

  The display for the prices in the shop.
- private HBox cardsBox:

  The display for the cards in the shop
- private HBox relicsBox:

  The display for the relics in the shop

Methods:

- public MerchantScreen():
  Constructor of the MerchantScreen.
- private void showRemoveCardPane():
  Shows the pane for card removal.
- private void closeRemoveCardPane():
  Initializes the view for MerchantScreen.
- private void shopRelics():
  This function enables to buy relics in the shop.
- private void shopCards():
  This function enables to buy cards in the shop.
- public void draw():
  Draws the view for MerchantScreen.
- public void initialize():
  Initializes the view for MerchantScreen.

4.3.2.14. RestScreen Class

The display screen for the rest room.

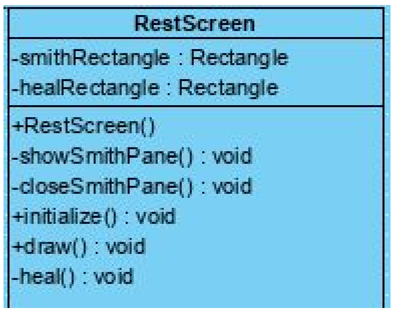


Figure 46: RestScreen Class Diagram

Attributes:

- private Rectangle smithRectangle:
  The rectangular display for the button to open blacksmith.
- private Rectangle healRectangle:
  The rectangular display for the button to heal character.

Methods:

- public RestScreen():

  Constructor of the RestScreen.
- public void draw():

  Draws the view for RestScreen.
- public void initialize():

  Initializes the view for RestScreen.
- private void showSmithPane():

  This method opens the SmithPane
- private void showSmithPane():

  This method closes the SmithPane
- private void heal():

  Heals the character.

4.3.2.15. SmithPane Class

The display pane for the blacksmith in RestScreen.

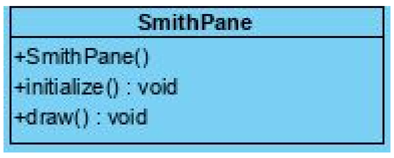


Figure 47: SmithPane Class Diagram

Methods:

- public SmithPane():

  Constructor of the SmithPane.
- public void draw():

  Draws the view for SmithPane.
- public void initialize():

  Initializes the view for SmithPane.

4.3.2.16. RemoveCardPane Class

The display pane for the card removal option in merchant.

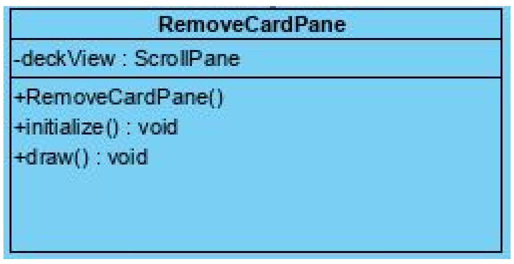


Figure 48: RemoveCardPane Class Diagram

Attributes:

- private ScrollPane deckView:

  The view for the cards in the character's current deck

Methods:

- public RemoveCardPane():

  Constructor of the RemoveCardPane.
- public void draw():

  Draws the view for RemoveCardPane.
- public void initialize():

  Initializes the view for RemoveCardPane.

4.3.2.17. MapScreen Class

The display screen for randomly generated map of the game.

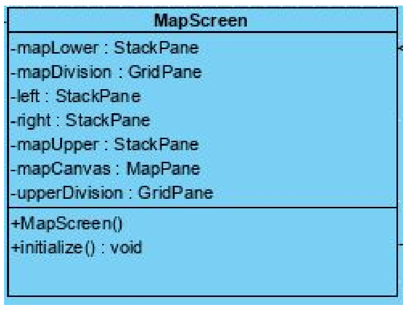


Figure 49: MapScreen Class Diagram

Attributes:

- private Rectangle mapLower:

  Pane for showing the lower part of the map screen.
- private GridPane mapDivision:

  Pane dividing the lower part of the map screen.
- private StackPane left:

  Pane that refers to the left of the mapDivision GridPane
- private StackPane right:

  Pane that refers to the right of the mapDivision GridPane. It contains map legend.
- private StackPane mapUpper:

  Pane that refers to the upper part of the map screen.
- private MapPane mapCanvas:

  Pane that contains the actual map.
- private GridPane upperDivision:

  Pane separating the lower and upper part of the map screen.

Methods:

- public MapScreen():

  Constructor of the MapScreen.
- public void draw():

  Draws the view for MapScreen.
- public void initialize():

  Initializes the view for MapScreen.

4.3.2.18. MapPane Class

The pane that contains the map of the game.

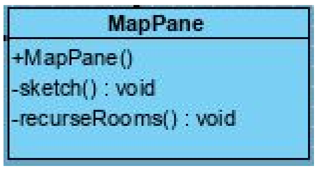


Figure 50: MapPane Class Diagram

Methods:

- public MapPane():

  Constructor of the MapPane.
- private void sketch():

  Draws the view for MapPane.
- private void recurseRooms():

  An auxiliary function for sketching the map.

4.3.2.19. MapRoomNode Class

This class is helpful in displaying the room nodes of the map. It extends Circle class of JavaFX.

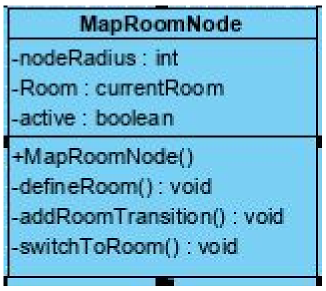


Figure 51: MapRoomNode Class Diagram

Attributes:

- private int nodeRadius:

  The radius of the circle displayed
- private Room currentRoom:

  The room data that the node contains
- private boolean active:

  The boolean showing checking the room’s availability.

Methods:

- public MapRoomNode():

  Constructor of the MapRoomNode.
- private void defineRoom():

  Defines the type of the room that will be switched.

- private void addRoomTransition():

  Adds a transition to the room that the node points out.
- private void switchToRoom():

  The screen switches to the currentRoom.

4.3.2.20. CompendiumScreen Class

This class is useful displaying the screen of compendium objects such as relics and cards.

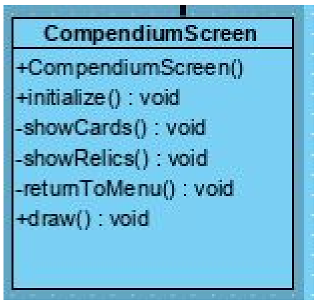


Figure 52: CompendiumScreen Class Diagram

Methods:

- public CompendiumScreen():

  Constructor of the CompendiumScreen.
- private void returnToMenu():

  Method to return to the main menu.
- private void showCards():

  Method to show the cards in the compendium
- private void showRelics():

  Method to show the relics in the compendium
- public void draw():

  Draws the view for CompendiumScreen.
- public void initialize():

  Initializes the view for CompendiumScreen.

4.3.2.21. WinScreen Class

This class displays the win screen of the game.

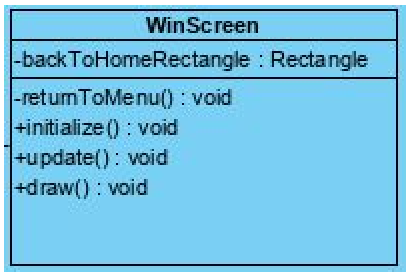


Figure 53: WinScreen Class Diagram

Attributes:

- private Rectangle backToHomeRectangle:

  The rectangular display for the button to return menu screen.

Methods:

- public WinScreen():

  Constructor of the WinScreen.
- private void returnToMenu():

  Method to return to the menu of the game.
- public void draw():

  Draws the view for WinScreen.
- public void initialize():

  Initializes the view for WinScreen.

4.3.3. Controller Package

Classes in this package all interact with the end user in various ways and they are the only classes that the users can directly interact with. Multiple room controller classes such as controllers for the fight room, the merchant, and rest sites are included in this package as well as a class called Game that controls the flow of the game, based on the input received from the user's end.

4.3.3.1. RoomController class

This class is the interface that controls the behavior of the rooms i.e. the flow of events and the mechanics of a room. Every room controller class extends this class.

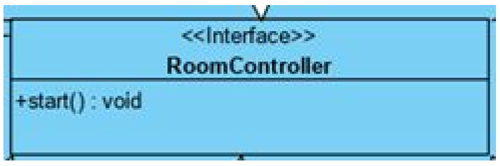


Figure 54: Class diagram of RoomController

Methods:

- public void start():

  Starts the flow of events in the room.

4.3.3.2. FightState class

This class contains the enumeration for fight states.

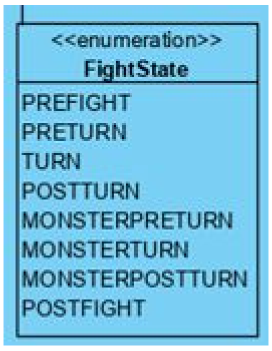


Figure 55: Class diagram of FightState

4.3.3.3. FightRoomController class

This class extends the RoomController class and it controls the fight mechanisms by making use of the states: pre fight, pre turn, turn, post turn, monster pre turn, monster turn and monster post turn.

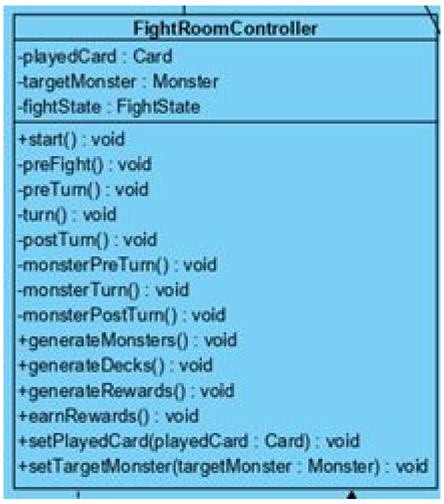


Figure 56: Class diagram of FightRoomController

Attributes:

- private Card playedCard:

  The card played by the player.

- private Monster targetMonster:

  The monster targeted by the player.

- private FightState fightState:

  Current state of the fight.

Methods:

- public void start():

  Starts the event flow of the room.

- private void preFight():

  Events that happen before the fight.

- private void preTurn():

  Events that happen before the turn of the player.

- private void turn():

  Events that happen during the turn of the player.

- private void postTurn():

  Events that happen after the turn of the player.

- private void monsterPreTurn():

  Events that happen before the turn of the monster.

- private void monsterTurn():

  Events that happen during the turn of the monster.

- private void monsterPostTurn():

  Events that happen after the turn of the monster.

- public void generateMonsters():

  Generates the monsters in the room.

- public void generateDecks():

  Generates the decks in the room.

- public void generateRewards():

  Generates the rewards in the room.

- public void earnRewards():

  Gives the rewards to the player.

- public void setPlayedCard(Card playedCard):

  Sets the played card to the specified card.

- public void setTargetMonster(Monster targetMonster):

  Sets the targeted monster to the specified monster.

4.3.3.4. MerchantRoomController class

This class extends the RoomController class and controls the mechanisms of the merchant rooms.

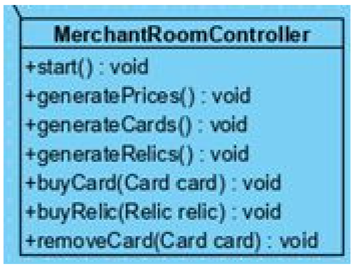


Figure 57: Class diagram of MerchantRoomController

Methods:

- public void start():

  Starts the flow of events in the room.

- public void generatePrices():

  Generates the prices of the items.

- public void generateCards():

  Generates the cards that can be bought.

- public void generateRelics():

  Generates the relics that can be bought.

- public void buyCard(Card card):

  Buys the card for the player.

- public void buyRelic(Relic relic):

  Buys the relic for the player.

- public void removeCard(Card card):

  Removes the card from the player's master deck.

4.3.3.5. TreasureRoomController class

This class extends the RoomController class and controls the mechanisms of the treasure rooms.

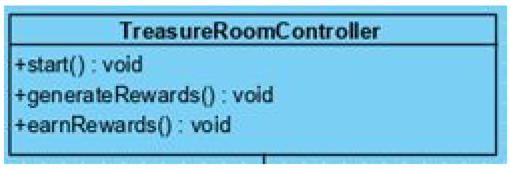


Figure 58: Class diagram of TreasureRoomController

Methods:

- public void start():

  Starts the flow of events in the room.

- public void generateRewards():

  Generates the rewards of the room.

- public void earnRewards():

  Rewards the player.

4.3.3.6. RestRoomController class

This class extends the RoomController class and controls the mechanisms of the rest rooms.

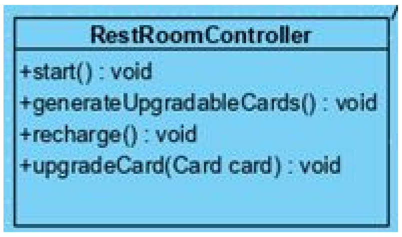

RestRoomController
+start() : void
+generateUpgradableCards() : void
+recharge() : void
+upgradeCard(Card card) : void

Figure 59: Class diagram of RestRoomController

Methods:

- public void start():

  Starts the flow of events in the room.

- public void generateUpgradableCards():

  Generates the cards that can be upgraded.

- public void recharge():

  Heals the player’s HP.

- public void upgradableCard(Card card):

  Upgrades the card.

4.3.3.7. MonsterInputHandler class

This class handles the input that comes from the user when they are choosing which monster to attack, and communicates this input to the relevant classes that are controlling the fight mechanism.

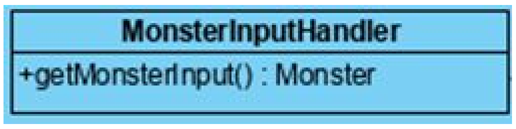


Figure 60: Class diagram of MonsterInputHandler

Methods:

- private Monster getMonsterInput():

  Returns the monster that was selected by the user.

4.3.3.8. MapManager class

This class manages the map and controls the events such as proceeding to the next room.

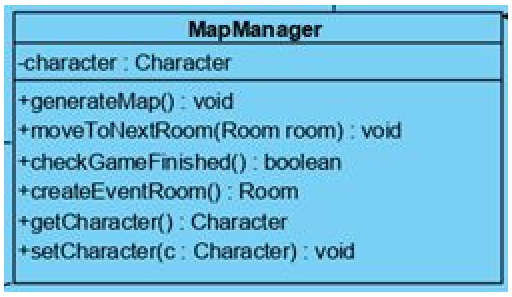

MapManager
-character : Character
+generateMap() : void
+moveToNextRoom(Room room) : void
+checkGameFinished() : boolean
+createEventRoom() : Room
+getCharacter() : Character
+setCharacter(c : Character) : void

Figure 61: Class diagram of MapManager

Attributes:

- private Character character:

  The character that the user chose to play as.

Methods:

- public void generateMap():

  Generates the map layout.

- public void moveToTheNextRoom(Room room):

  Proceeds to the specified room.

- public boolean checkGameFinished():

  Returns whether the game has been finished.

- public Room createEventRoom():

  Creates an event room.

- public Character getCharacter():

  Returns the character.

- public void setCharacter(Character c):

  Sets the character to the specified character.

4.3.3.9. Game class

This class controls the flow of events for the whole game and interacts with the user through the UI. It has a static instance in the Main class of the application.

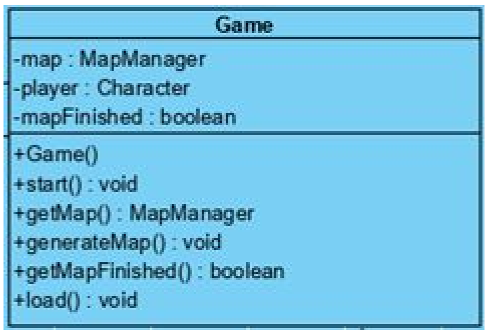


Figure 62: Class diagram of Game

Attributes:

- private MapManager map:

  Map of the game.

- private Character player:

  Player (user) of the game.

- private boolean mapFinished:

  Whether the map is finished.

Methods:

- public Game():

  Constructor of the class.

- public void start():

  Starts the flow of events from the first room after displaying the map.

- public MapManager getMap():

  Returns the map.

- public void generateMap():

  Calls the generateMap method of MapManager.

- public boolean getMapFinished():

  Returns whether the map has been finished.

- public void load():

  Loads the progession of the user in the game.

4.3.3.10. Actions Package

4.3.3.10.1. RelicActions class

This class contains the functionality about relics such as adding a relic to a player's relic collection.

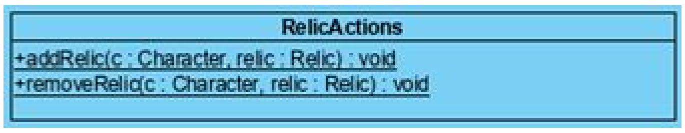

RelicActions
+addRelic(c : Character, relic : Relic) : void
+removeRelic(c : Character, relic : Relic) : void

Figure 63: Class diagram of RelicActions

Methods:

- public static void addRelic(Character c, Relic relic):

  Adds the relic to the specified character’s relic collection.

- public static void removeRelic(Character c, Relic relic):

  Removes the relic from the specified character’s relic collection.

4.3.3.10.2. PowerActions class

This class contains the functionality about powers such as adding a power to a creature’s power list.

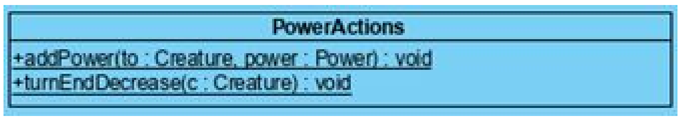

PowerActions
+addPower(to : Creature, power : Power) : void
+turnEndDecrease(c : Creature) : void

Figure 64: Class diagram of PowerActions

Methods:

- public static void addPower(Creature to, Power power):

  Adds the power to a specified creature.

- public static void turnEndDecrease(Creature c):

  Makes the power’s effect decrease at the end of each turn.

4.3.3.10.3. FightActions class

This class contains the functionality about fights such as attacking a creature and blocking incoming attacks.

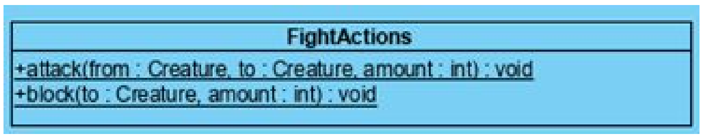

FightActions
+attack(from : Creature, to : Creature, amount : int) : void
+block(to : Creature, amount : int) : void

Figure 65: Class diagram of FightActions

Methods:

- public static void attack(Creature from, Creature to, int amount):

  Attacks the specified creature by a specified amount.

- public static void block(Creature to, int amount):

  Specified creature blocks the attack by a specified amount.

4.4. Class Interfaces

JavaFX library is used in implementation of class interfaces. Mainly, Runnable method, GridPane, StackPane, Scene, Rectangle, HBox, VBox of JavaFX are used.

5. References

[1]“Slay the Spire Wiki,” Fandom . [Online]. Available: https://slay-the-spire.fandom.com/wiki/Slay_the_Spire_Wiki. [Accessed: 04-Mar-2020].
[2]“Java SE 11 Downloads,” *Java SE Development Kit 11- - Downloads*. [Online]. Available: https://www.oracle.com/java/technologies/javase-jdk11-downloads.html. [Accessed: 17-Apr-2020].