Bilkent University

Department of Computer Engineering

Spring 2020 CS319 Project

1D - Slay the Spire

Project Design Report Iteration 1

Özge Yaşayan

Gülnihal Koruk

Fatih Karahan

Mehmet Bora Kurucu

Batuhan Özçömlekçi

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 tower while fighting enemies 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. While identifying the design goals, the classes interactions and MVC design patterns will be included.

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, 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. MVC design pattern will also help for this purpose. 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 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

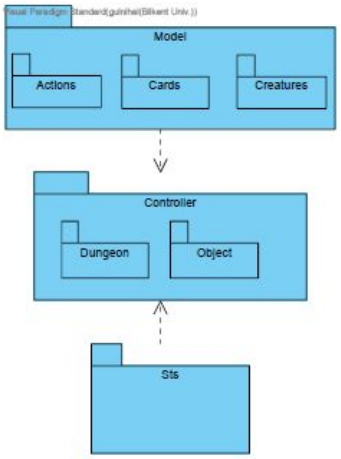

Figure 1: UML representation of system decomposition

In this part, subsystem decomposition will be written, and the detailed versions will be clarified in the following section parts. For having maintainable system, the system should be extensible, reusable and modifiable. Decomposing the system as stated will provide these. Thus, we use MVC design pattern. This pattern will make the connection between our subsystems. In the pattern, we have 3 layer; Model layer, Sts layer and Controller layer, and these fit MVC design; Model, View, Controller, respectively. Controller takes input from the user and then it will contact Model that store the data. With the connection of Controller, Model updates the View that shows the screen to the user and these data flows continue. In our game, by the buttons in the main menu, the controller takes the input from the user then the corresponding screen comes with connection of classes of Sts, view layer. We have 3 layer that decomposed according to their functionalities and so the classes are also placed

in layer according to their functionality. Further information about the relationship between classes of packages will be in the following sections.

2.2. Hardware/Software Mapping

For the game storage, we will use text files. 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. For hardware configuration, the game will not need a keyboard, user interaction will be provided by mouse pointer. 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 text file. Some data members such as setting preferences, that made by the user from the options selection in the main menu, will be stored in a modifiable text file so that they can be updated preferably during the game. Besides, some of the text files that store the map step process for the player will be in the non modifiable text 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 not keep the user's private information anyway. 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.

3. Subsystem Services

3.1 Controller and Model Subsystem

Model Subsystem includes classes that control the game logic and game flow. It interacts with the UI subsystem to draw the necessary information to the user screen. Model subsystem can be divided into two categories: Controllers and Game Models. While it would have been much better to divide the game flow from the Models subsystem, due to the nature of the game they are very entwined, and dividing would only create unnecessary overhead.

3.1.1. Controllers category

This category includes the Actions class, as well as the various child classes of (it saysAbstractRoom. Action Class acts on the information that came from the Game Models, like an Attack action or a Add Power Action. This class makes sure that every type of action that a player can do is standardized. Child classes of Abstract Room, as well as the main Game class controls how the game flows. They define the structure of the game, from the turn system in the fights to ascending to a different room.

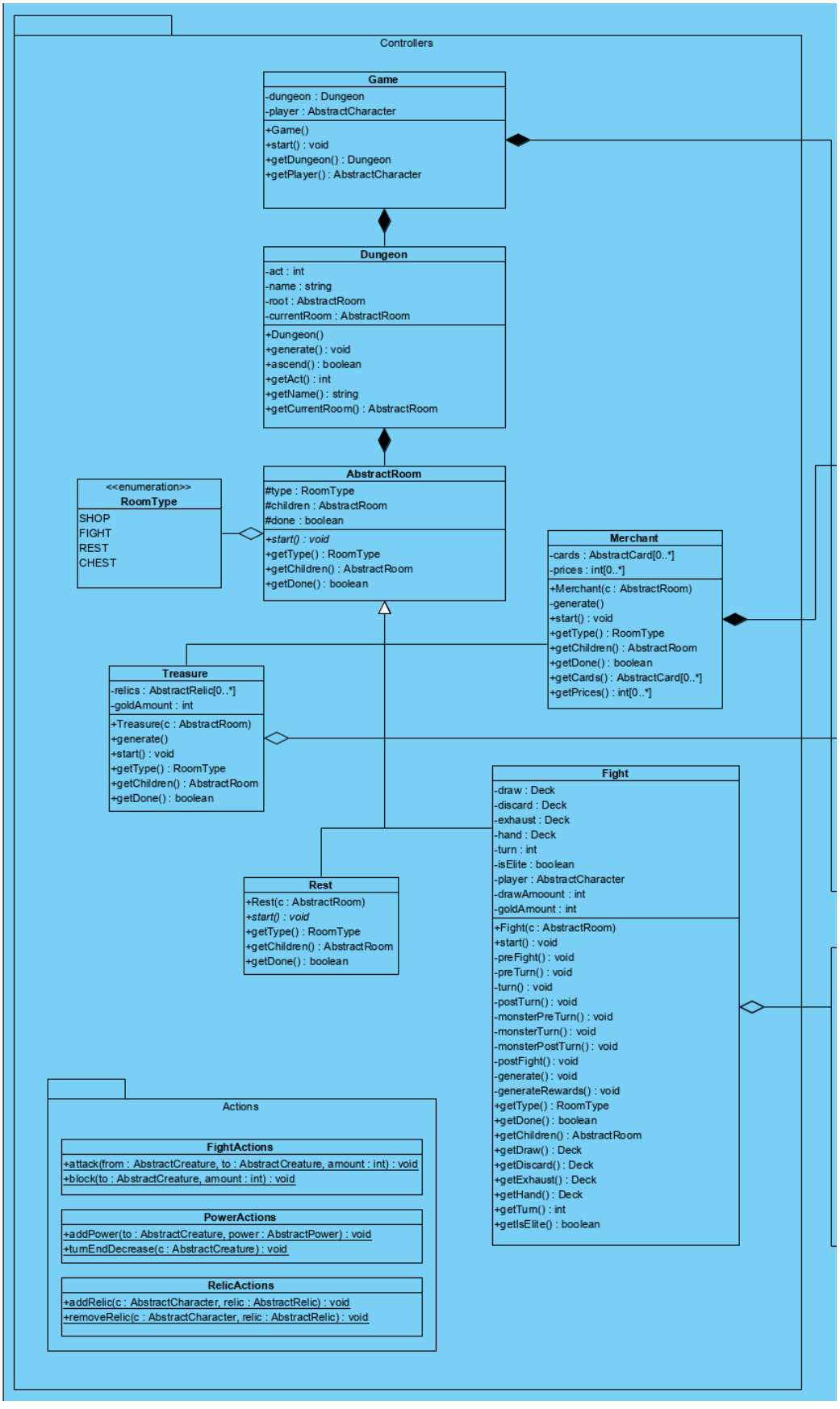

Figure 2: Diagram of Controllers

3.1.2. Game Models category

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

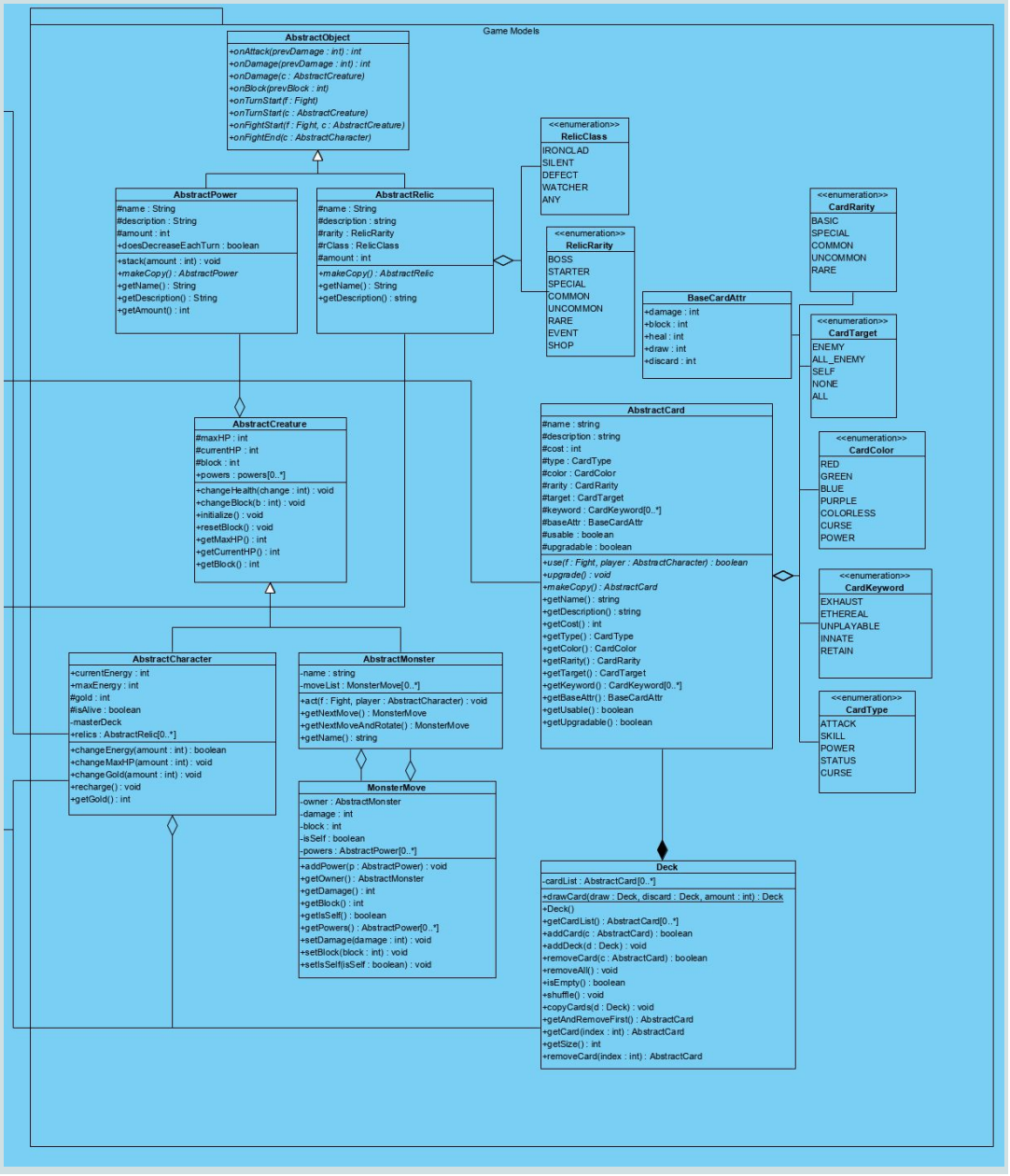

Figure 3: Diagram of Game Models

A higher resolution image can be found at https://cdn.discordapp.com/attachments/676834662814842932/700722207206277140/Subsystem.png

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 scene is called and initialize() d. While the game is played,the scene is constantly update() d, then draw()n again.Currently, there are two scenes in the game which are menu scene initialized in Main class and fight scene constructed by FightScene class.

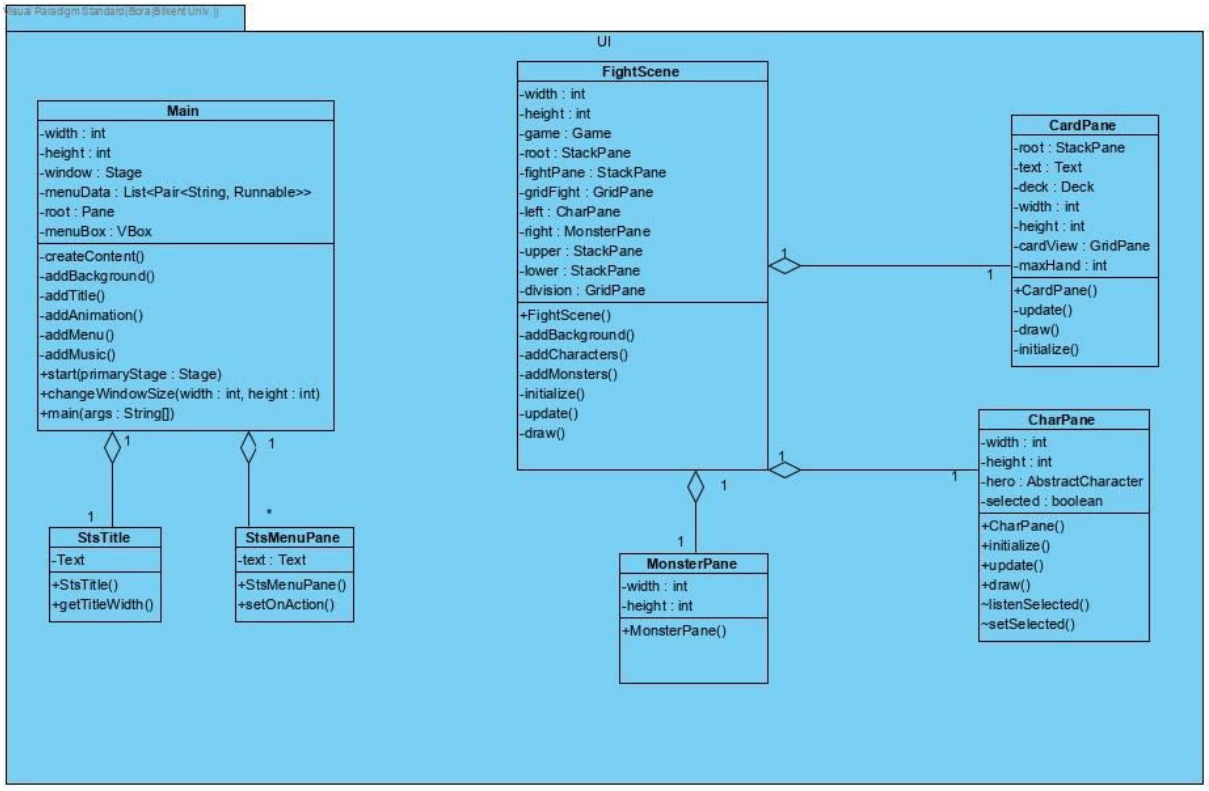

Figure 4: UML representation of UI

4. Low-Level Design

4.1. Trade-offs

4.1.1. Portability vs. Efficiency

Since we chose Java Language to implement our game, portability is more important than the efficiency. This programming language also requires Java Virtual Machine that affects

the performance because there will be a virtual environment through the game and operating system. However, since we don’t use so complicated graphics in our game, the effect will be minimum.

4.1.2. Performance vs Memory

In the game, our priority is performance. We focus on the screens in the game and we provide the user a good experience on changing the screen. For instance, in a map screen, the user will be able to go back to main menu and select the button options instantly. Thus, we create multiple screen classes such as FightScene, StsMenuPane etc. rather than drawing the screen on the same frame. These results come up with an occupation more space in the memory. Therefore, for the better performance, we sacrifice memory space.

4.1.3. User Friendliness vs. Functionality

One of our focuses is a good user experience. We make the game simple, clear and understandable. Thus, the game will not include complex functionality or to 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

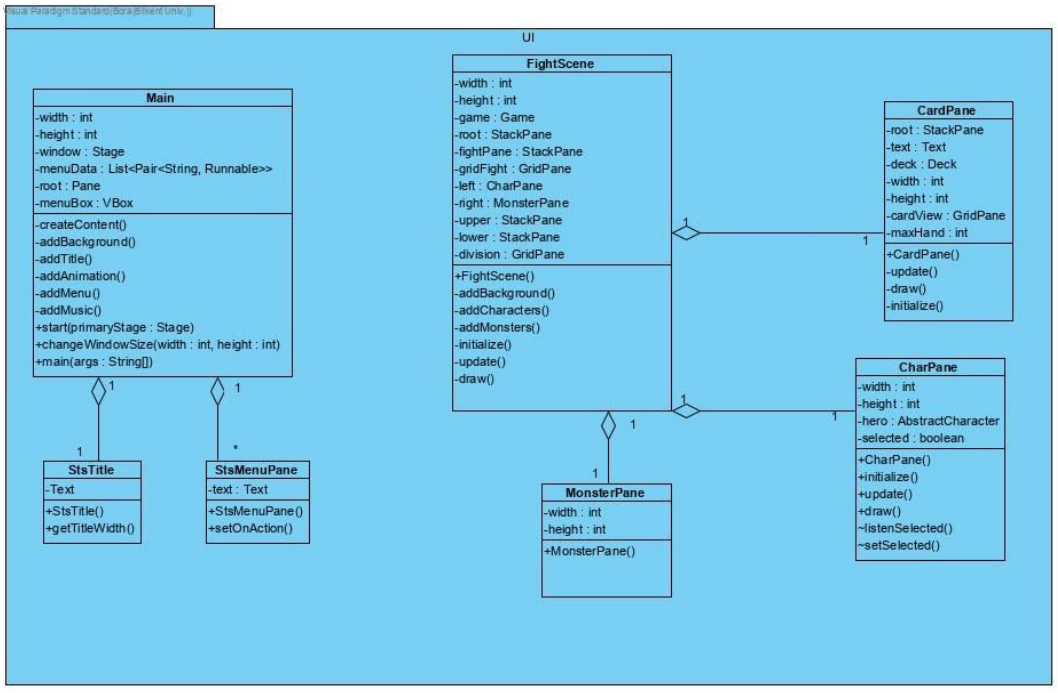

Figure 5: Object diagram of UI

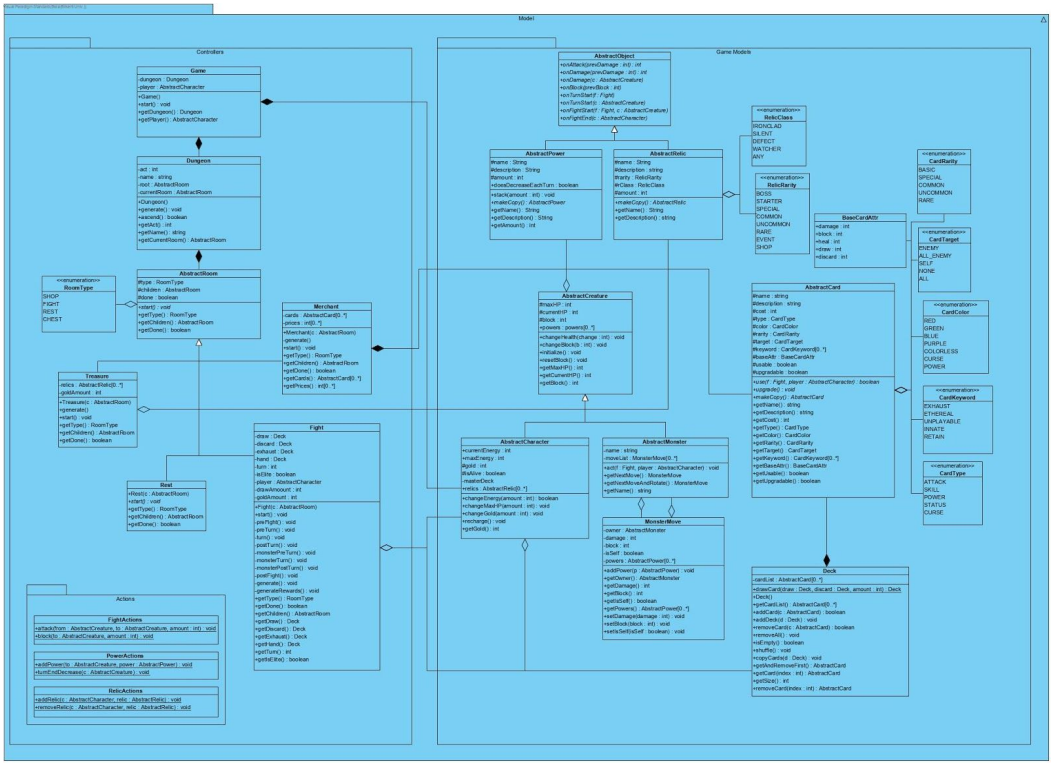

Figure 6: Object diagram of models

4.3. Packages

4.3.1. Models Package

Models 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).

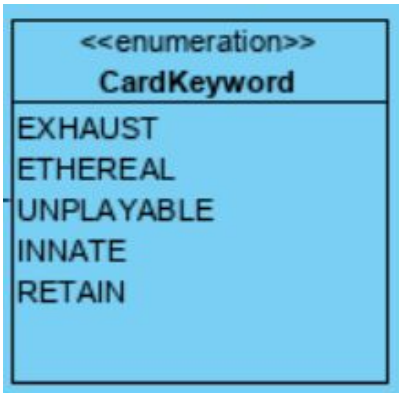

Figure 8: Class diagram of CardKeyword

4.3.1.3. CardRarity class

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

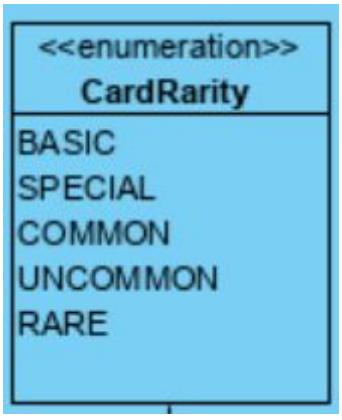

Figure 9: Class diagram of CardRarity

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).

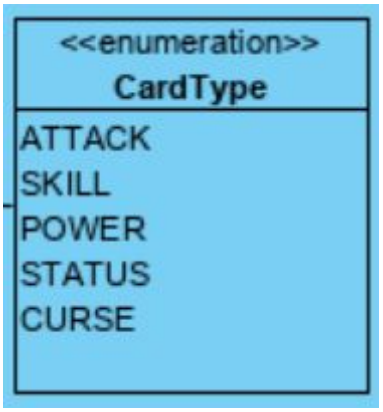

Figure 11: Class diagram of CardType

4.3.1.6. BaseCardAttributes class

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

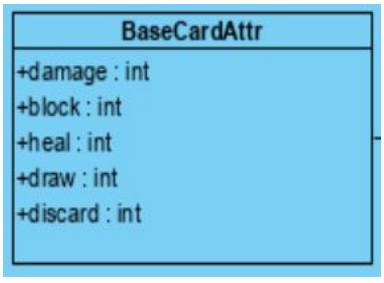

Figure 12: Class diagram of BaseCardAttr

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. AbstractCard 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.

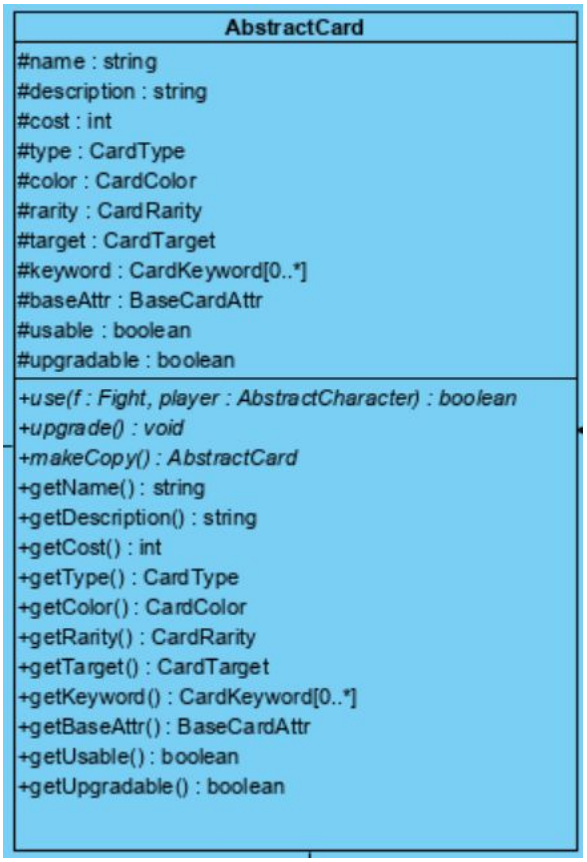

Figure 13: Class diagram of AbstractCard

Attributes:

- private String name:
  The name of the card.
- private String description:
  The description of the card.
- private int cost:
  The energy amount that it costs to use the card.
- private CardType type:
  The type of the card.
- private CardColor color:
  The color of the card.
- private CardRarity rarity:
  The rarity of the card.
- private CardTarget target:
  The target of the card.
- private CardKeyword keyword:
  The keyword of the card.
- private BaseCardAttributes baseAttr:
  Basic attributes of the card.
- private boolean usable:
  Indication of whether the card is usable or not.
- private boolean upgradable:
  Indication of whether the card is upgradable or not.

Methods:

- public abstract boolean use(Fight f, AbstractCharacter player):
  Has the set of instructions that will be executed when the card is played.
- public abstract void upgrade():
  Upgrades the card.
- public abstract AbstractCard 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.

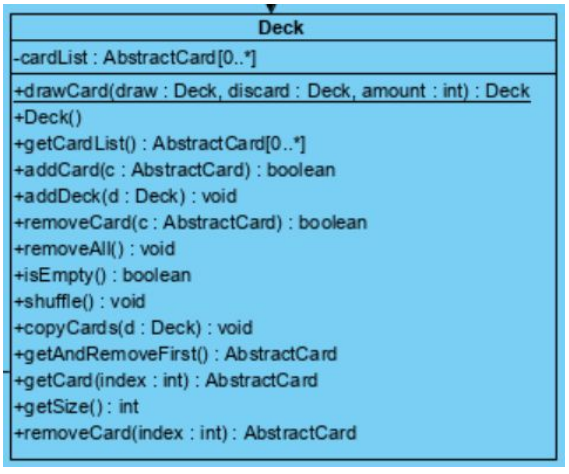

Figure 14: Class diagram of Deck

Attributes:

- private ArrayList<AbstractCard> cardList:
  A list of AbstractCard class instances.

Methods:

- public Deck():
  Constructor of the class
- public ArrayList<AbstractCard> getCardList():
  Returns the card list.
- public boolean addCard(AbstractCard 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(AbstractCard 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 AbstractCard getAndRemoveFirst():
  Gets the first card and then removes it.
- public AbstractCard getCard(int index):
  Gets a card based on a specified index.
- public int getSize():
  Returns the size of the deck.
- public AbstractCard 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. AbstractCreature 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.

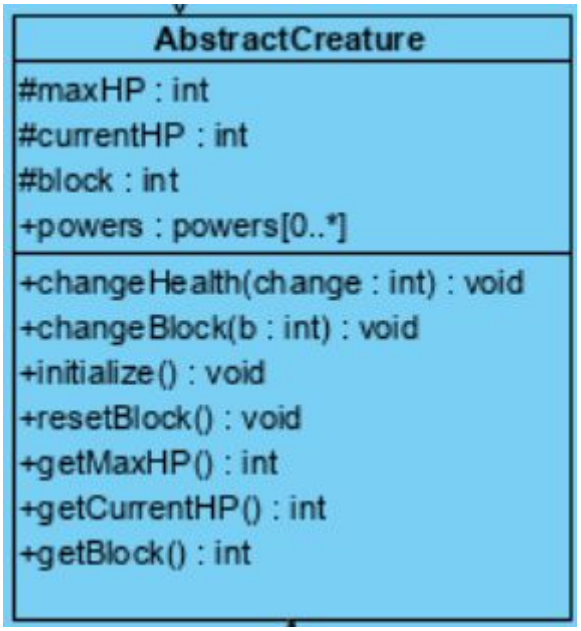

Figure 15: Class diagram of AbstractCreature

Attributes:

- private int maxHP:
  Maximum HP the creature can have.
- private int currentHP:
  Current HP of the creature.
- private int block:
  Block amount that the creature has.
- public ArrayList<AbstractPower> 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. AbstractCharacter class

This class inherits from the AbstractCreature 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.

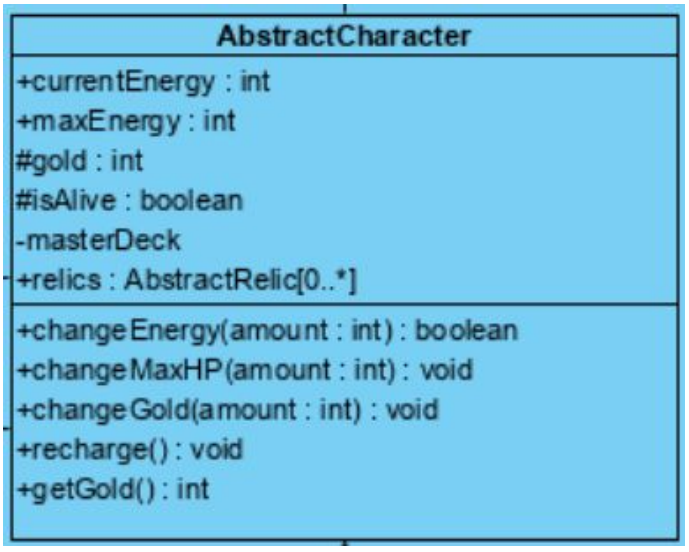

Figure 16: Class diagram of AbstractCharacter

Attributes:

- public int currentEnergy:
  Current energy of the character.
- public 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.
- public Deck masterDeck:
  Master deck of the character (all of the cards that the character has).
- public ArrayList<AbstractRelic> 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. AbstractMonster class

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

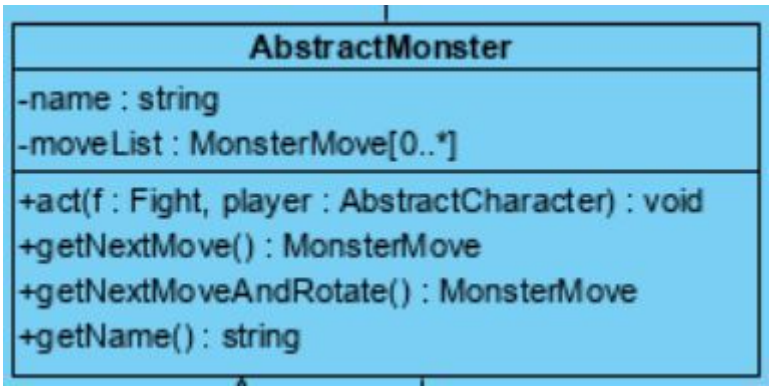

Figure 17: Class diagram of AbstractMonster

Attributes:

- private String name:
  Name of the monster.
- private ArrayList<MonsterMove> moveList:
  List of the moves that the monster is going to make.

Methods:

- public void act(Fight f, AbstractCharacter 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.

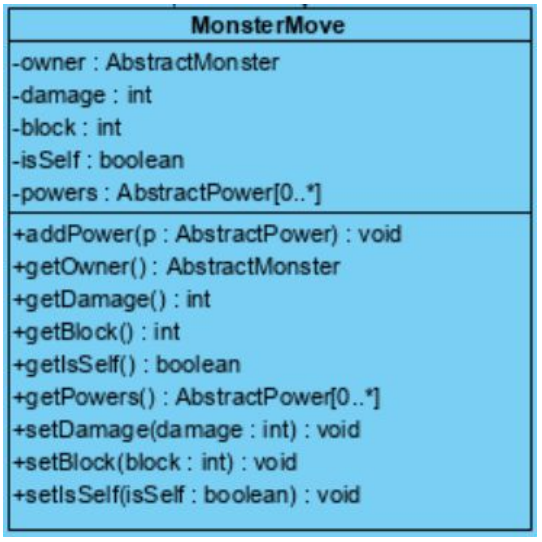

Figure 18: Class diagram of MonsterMove

Attributes:

- private AbstractMonster 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<AbstractPower> powers:
  The list of powers that are in effect.

Methods:

- public MonsterMove(AbstractMonster o):
  Constructor for MonsterMove. Initializes the properties.
- AbstractMonster getOwner():
  Returns the owner.
- boolean getIsSelf():
  Returns isSelf.
- public int getDamage:
  Returns the damage.
- public int getBlock():
  Returns the block.
- public ArrayList<AbstractPower> 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(AbstractPower p):
  Adds a power to the powers collection.

4.3.1.13. AbstractObject 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.

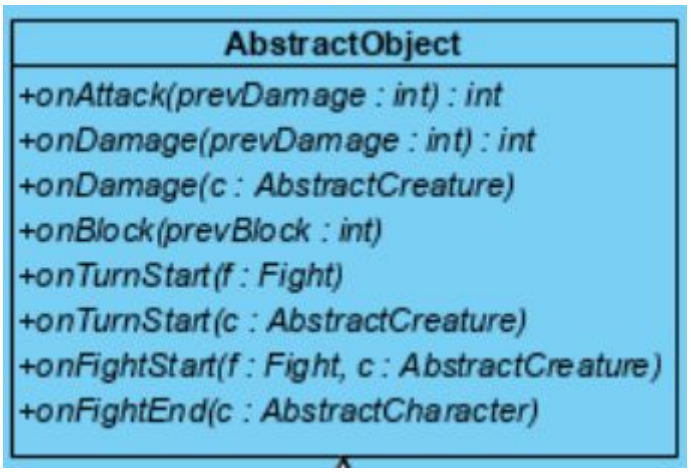

Figure 19: Class diagram of AbstractObject

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(AbstractCreature 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(AbstractCreature c):
  At the start of every turn, a set of instructions are executed and the creature gets affected.
- public void onFightStart(Fight f, AbstractCharacter c):
  At the start of every fight, a set of instructions are executed.
- public void onFightEnd(AbstractCharacter 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).

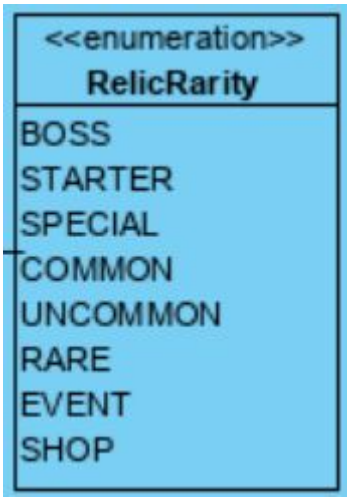

Figure 20: Class diagram of RelicRarity

4.3.1.15. RelicClass class

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

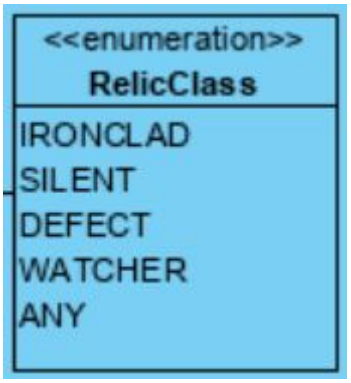

Figure 21: RelicClass

4.3.1.16. AbstractRelic class

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

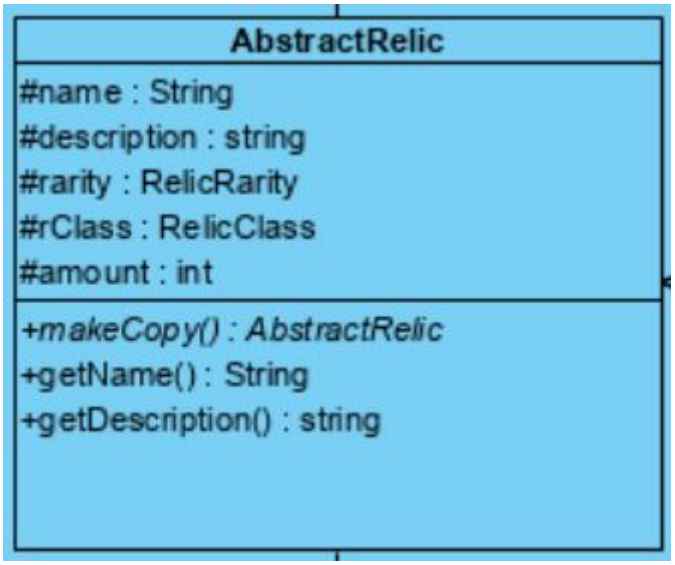

Figure 22: Class diagram of AbstractRelic

Attributes:

- private String name:
  Name of the relic.
- private String description:
  Description of the relic.
- private RelicRarity rarity:
  Rarity of the relic.
- private RelicClass rClass:
  Class of the relic.
- private int amount:
  The amount of X (block, damage…) that the relic will affect.

Methods:

- public abstract AbstractRelic 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.
- public abstract void affect(Fight f, AbstractCharacter player):
  Has the set of instructions that will be executed when a relic is in effect.

4.3.1.17. RelicActions class

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

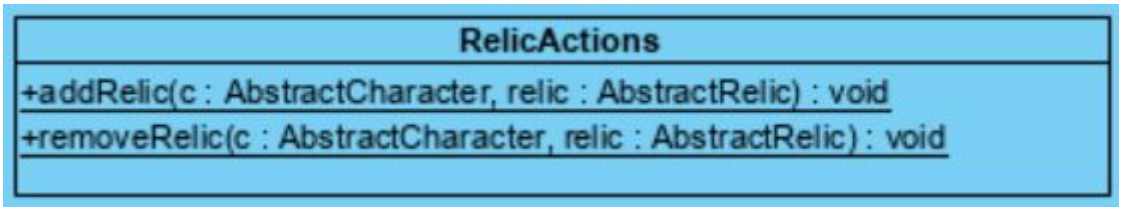

Figure 23: Class diagram of RelicActions

Methods:

- public static void addRelic(AbstractCharacter c, AbstractRelic relic):
  Adds the relic to the specified character's relic collection.
- public static void removeRelic(AbstractCharacter c, AbstractRelic relic):
  Removes the relic from the specified character's relic collection.

4.3.1.18. AbstractPower class

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

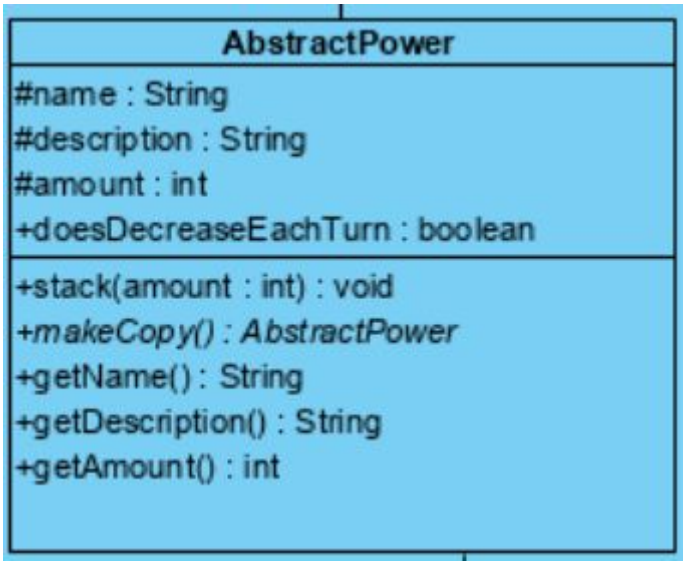

Figure 24: Class diagram of AbstractPower

Attributes:

- private String name:
  Name of the power.
- private String description:
  Description of the power.
- private 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 AbstractPower 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.

4.3.1.19. PowerActions class

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

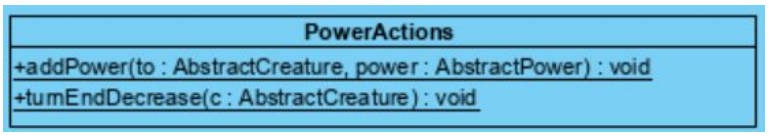

Figure 25: Class diagram of PowerActions

Methods:

- public static void addPower(AbstractCreature to, AbstractPower power):
  Adds the power to a specified creature.

- public static void turnEndDecrease(AbstractCreature c):
  Makes the power's effect decrease at the end of each turn.

4.3.1.20. RoomType class

This class contains the enumeration for the room types (shop, fight, rest, chest, event).

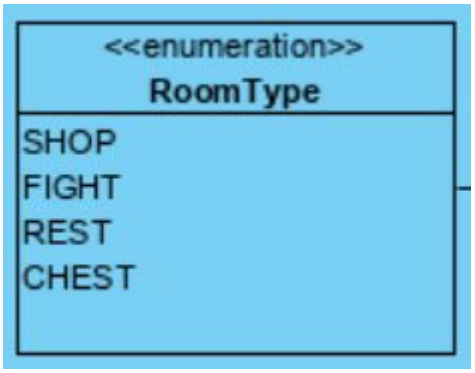

Figure 26: Class diagram of RoomType

4.3.1.21. AbstractRoom class

This class represents a room in the game map. It is the skeleton for all of the room classes. Every room class inherits from this class.

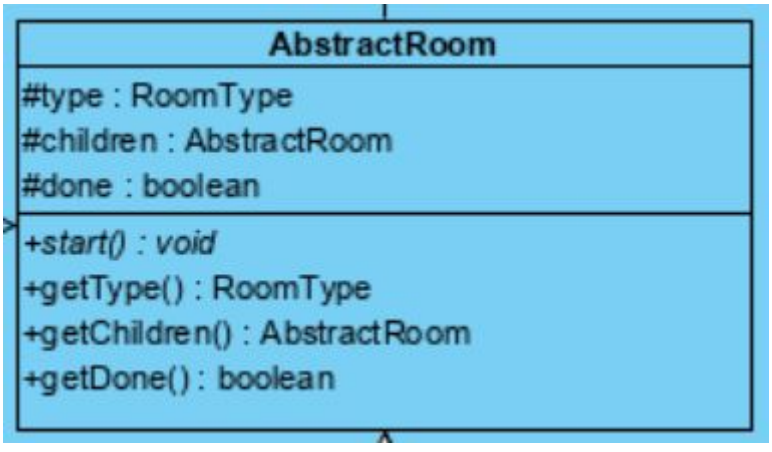

Figure 27: Class diagram of AbstractRoom

Attributes:

- private RoomType type:
  Type of the room.

- private AbstractRoom children:
  The room that comes after this room.
- private boolean done:
  Indication of whether the room was succesfully passed by the player.

Methods:

- public abstract void start():
  Starts the events of the room.
- public abstract RoomType getType():
  Returns the room type.
- public abstract AbstractRoom getChildren():
  Returns the “children” attribute.
- public abstract boolean getDone():
  Returns the “done” attribute.

4.3.1.22. Dungeon class

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

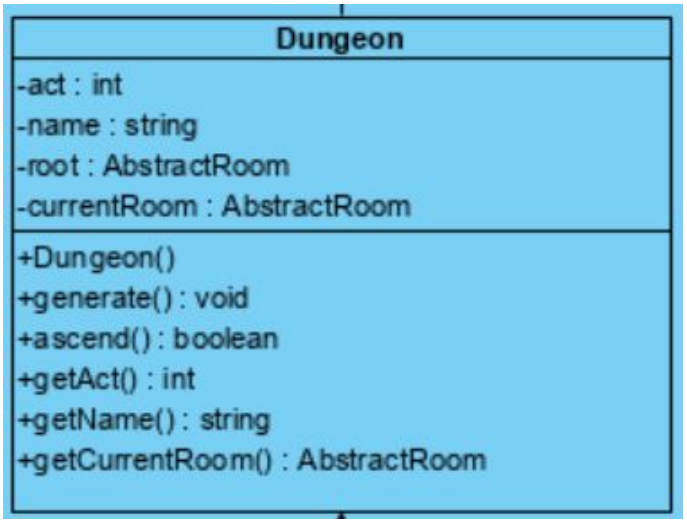

Figure 28: Class diagram of Dungeon

Attributes:

- private int act:
  Act number.

- private String name:
  Name of the dungeon.
- private AbstractRoom root:
  The first room of the dungeon.
- private AbstractRoom currentRoom:
  Current room that the player is playing in.

Methods:

- public Dungeon():
  Constructor of the dungeon. Initializes the act as 1 and the name as Exordium.
- public void generate():
  Generates the game map.
- public int getAct():
  Returns the act.
- public String getName():
  Returns the name.
- public AbstractRoom getCurrentRoom():
  Returns the current room.
- public boolean ascend():
  Proceeds to the next room.

4.3.1.23. FightActions class

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

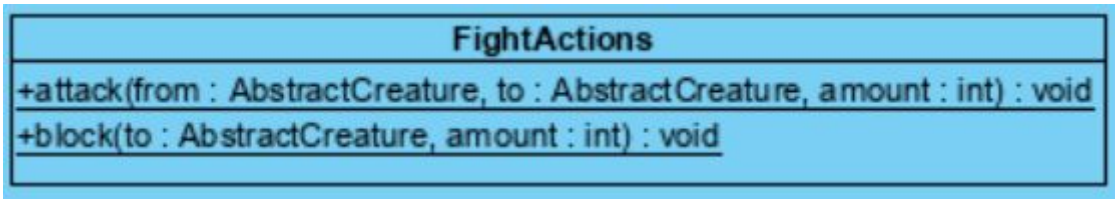

Figure 29: Class diagram of FightActions

Methods:

- public static void attack(AbstractCreature from, AbstractCreature to, int amount):
  Attacks the specified creature by a specified amount.

- public static void block(AbstractCreature to, int amount):
  Specified creature blocks the attack by a specified amount.

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. Scene class is the key element in the interface design. The UI switches from one Scene object to another to change the UI screen. Therefore the classes built are constructed with respect to this principle.

Currently, there are two scenes in the game which are menu scene initialized in Main class and fight scene constructed by FightScene class.

4.3.2.1. Main Class

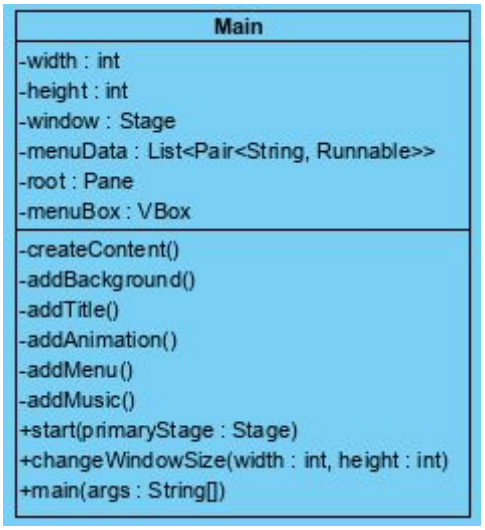

Figure 30: Main Class Diagram

Main class is the starter of the UI and extends Application class of the JavaFX libraries. It overrides the start method of Application class.

Attributes:

- private int width:
  Width of the initial stage and scene
- private int height:
  Height of the initial stage and scene.
- private Stage window:
  Keeps the current stage so that enables to perform operations on it.
- 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 FightScene and switches to that scene.
- private Pane root:
  Root pane where title of the game and the menu of the game is drawn on.
- 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 scene which should be a type of Parent. In this case, Parent is the root Pane that is added to the current scene. This method calls the relevant methods to add menu, title and background.
- private void addBackground():
  Adds an background image to the current scene.
- 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 scene.
- public void start(Stage primaryStage):
  Overriden method that starts the Java FX view.

- public void changeWindowSize(int width,int height):
  Changes the window size as the desired.
- public static void main(String[] args):
  Main method that is executed first. It calls the appropriate functions to initialize the stage.

4.3.2.2. FightScene Class

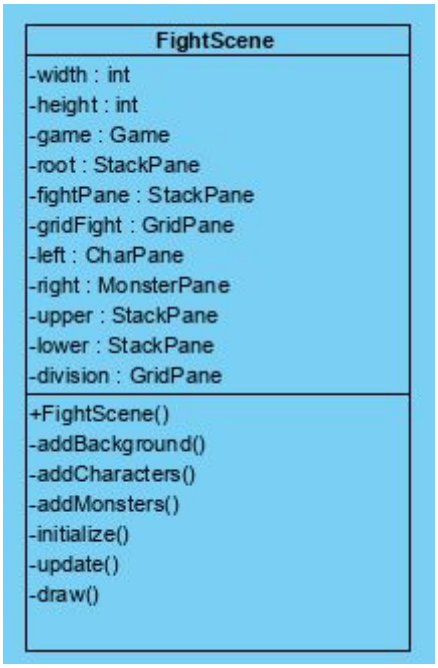

Figure 31: FightScene Class Diagram

Duty of the FightScene class is displaying a scene for the fights in the game. It is splited into three parts to give the player a better view of the game and an access to interactions with the game.

Attributes:

- private int width:
  Width of the FightScene scene.
- private int height:
  Height of the FightScene scene.
- private Game game:

Game object that manages the fight on the current scene. All game logic, mechanics and characters are obtained from this class.

- private StackPane root:
  Fundamental pane that includes other panes and images on it.
- private StackPane fightPane:
  It is the pane including fighting creatures and it resides in middle of the root pane.
- private GridPane gridFight:
  It is the grid that separates monster pane and character pane. It is inside the fightPane.
- private CharPane left:
  This pane corresponds to the middleleft of the fight scene and includes friendly characters. It is a type of StackPane that resides inside gridFight.
- private MonsterPane right:
  This pane corresponds to the middleright of the fight scene and includes enemy monsters. It is a type of StackPane that resides inside gridFight.
- private StackPane upper:
  Upper part of the fight scene. It includes a view of menu buttons, the information about the character and its items.
- private StackPane lower:
  Lower part of the fight scene. It includes a view of cards, piles and the amount of energy.
- private GridPane division:
  It is a grid to divide the screen into three parts. The three parts that are lower, upper and fightPane is included in this division pane.

Methods:

- public FightScene():
  Constructor of the class that calls its superclass constructor and the methods to initialize the current scene.
- private void addBackground():
  Adds an background image to the current fight scene.
- private void addCharacters():
  Adds the images of the characters to its appropriate pane.
- private void addMonsters():
  Adds the images of the monsters to its appropriate pane.
- private void initialize():

Initializes the attributes of the current pane.

- private void update():
  Updates the view.
- private void draw( ):
  Draws the view.

4.3.2.3. StsMenuPane Class

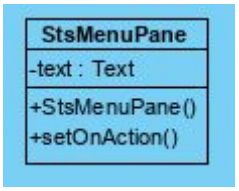

Figure 32: StsMenuPane Class Diagram

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

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.

4.3.2.4. StsTitle Class

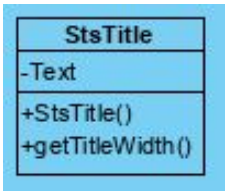

Figure 33: StsTitle Class Diagram

StsTitle class creates and shapes the title part of the menu screen.

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.

4.3.2.5. CardPane Class

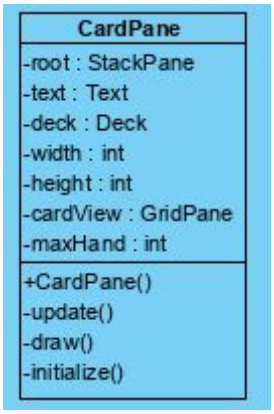

Figure 34: CardPane Class Diagram

CardPane class displays the information about cards, piles and it enables interaction with the cards.

Attributes:

- private StackPane root:
  Main pane that embodies all of the content in it.
- private Text text:
  It provides the necessary texts for the CardPane.
- private Deck deck:
  It is the deck of the character that is useful for game mechanics.

- private int width:
  Width of the current pane
- private int height:
  Height of the current pane.
- private GridPane cardView:
  This pane provides a view of cards in its grids.
- private int maxHand:
  Maximum number of cards that can be in the hand.

Methods:

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

4.3.2.6. CharPane Class

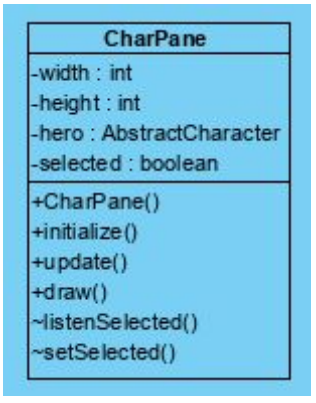

Figure 35: CharPane Class Diagram

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

Attributes:

- private int width:
  Width of the current pane
- private int height:

Height of the current pane

- private AbstractCharacter hero:
  This includes the attributes and operations of the friendly character that will be used in game mechanics. The character, hero is the character whose information is going to be displayed via UI.
- private boolean selected:
  Determines whether the character is selected or not.

Methods:

- public CharPane():
  Constructor of the class.
- public void update():
  Updates the view for CharPane.
- public void draw():
  Draws the view for CharPane.
- public void initialize():
  Initializes the view for CharPane.
- void listenSelected():
  Attaches a listener to the pane to listen the interactions with the character such as selecting it.
- void setSelected():
  Drops a shadow over the character to show that it is selected or vice versa.

4.3.2.7. MonsterPane Class

Figure 36: MonsterPane Class Diagram

MonsterPane class includes the view of the enemy monsters and enables to interact with them. This class is constructed partly.

Attributes:

- private int width:
  Width of the current pane
- private int height:
  Height of the current pane

Methods:

- public MonsterPane():
  Constructor of the class.

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 classes such as the fight room, the shop, 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. Fight class

This class represents the room in which the player has to fight monsters. It extends the AbstractRoom class and overrides the relevant methods.

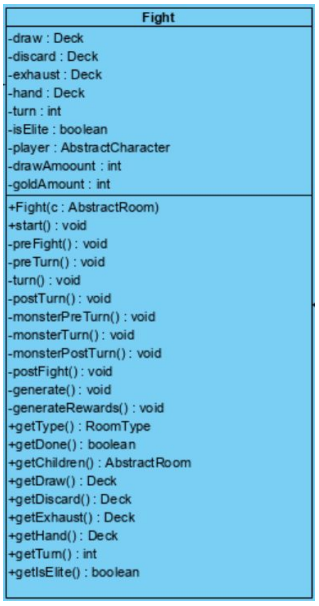

Figure 37: Class diagram of Fight

Attributes:

- private ArrayList<AbstractMonster> monsters:
  The monsters that are going to be involved in the fight.
- 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 isElite:
  Whether the monster is an elite.
- private AbstractCharacter 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.

Methods:

- public Fight(AbstractRoom c):
  Constructor of the class where c is the next room.
- 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.
- 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 AbstractRoom getChildren():
  Returns the room that comes after this room.
- public boolean getDone():
  Returns whether the room has been cleared.
- public ArrayList<AbstractMonster> getMonsters():
  Returns the monsters.
- 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.

4.3.3.2. Merchant class

This class represents the room in which the player can visit the shop. It extends the AbstractRoom class and overrides the relevant methods.

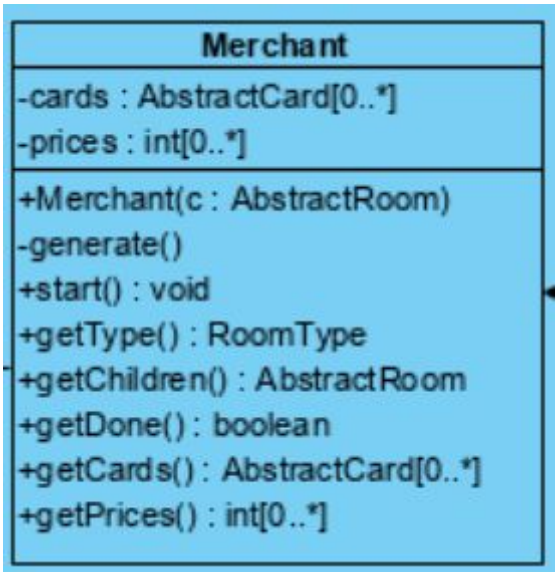

Figure 38: Class diagram of Merchant

Attributes:

- private ArrayList<AbstractCard> cards:
  Cards in the shop.
- private ArrayList<Integer> prices:
  Prices of the cards.

Methods:

- public Merchant(AbstractRoom c):
  Constructor of the class where c is the next room.
- private void generate():
  Generates the cards and the prices.
- public void start():
  Starts the flow of the events in the room.
- public RoomType getType():
  Returns the room type.
- public AbstractRoom getChildren():
  Returns the next room.
- public boolean getDone():
  Returns whether the room has been cleared.
- public ArrayList<AbstractCard> getCards():
  Returns the cards.
- public ArrayList<Integer> getPrices():
  Returns the prices.

4.3.3.3. Rest class

This class represents the room in which the player can rest. It extends the AbstractRoom class and overrides the relevant methods.

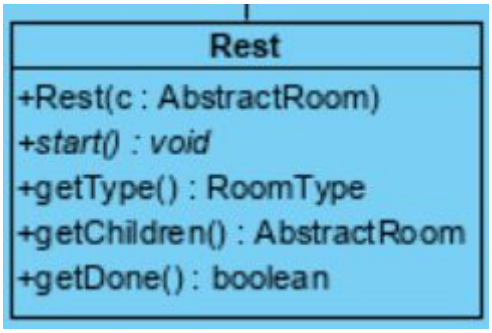

Figure 39: Class diagram of Rest

Methods:

- public Rest(AbstractRoom c):
  Constructor of the class where c is the next room.

- public void start():
  Starts the flow of the events in the room.
- public RoomType getType():
  Returns the room type.
- public AbstractRoom getChildren():
  Returns the next room.
- public boolean getDone():
  Returns whether the room has been cleared.

4.3.3.4. Treasure class

This class represents the room in which the player can get rewarded with prizes. It extends the AbstractRoom class and overrides the relevant methods.

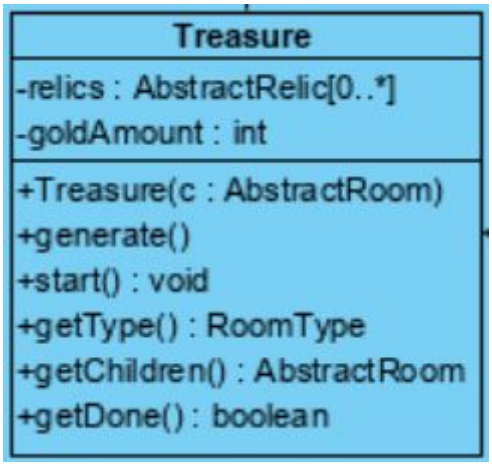

Figure 40: Class diagram of Treasure

Attributes:

- private ArrayList<AbstractRelic> relics:
  The list of relics that can be earned.
- private int goldAmount:
  The amount of gold that can be earned.

Methods:

- public Treasure(AbstractRoom c):
  Constructor of the class where c is the next room.

- private void generate():
  Generates the relics and the gold amount.
- public void start():
  Starts the event flow of the room.
- public RoomType getType():
  Returns the room type.
- public AbstractRoom getChildren():
  Returns the next room.
- public boolean getDone():
  Returns whether the room has been cleared.
- public ArrayList<AbstractRelic> getRelics():
  Returns the relics.
- public int getGoldAmount():
  Returns the amount of the gold prize.

4.3.3.5. 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.

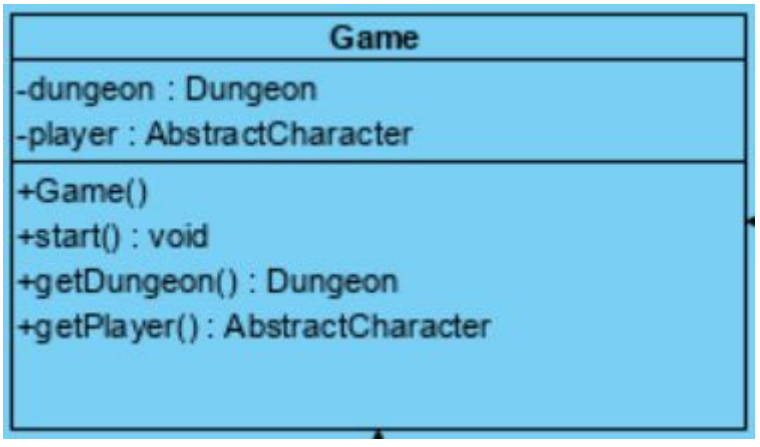

Figure 41: Class diagram of Game

Attributes:

- private Dungeon dungeon:
  Dungeon in the game.

- private AbstractCharacter player:
  Player (user) of the game.

Methods:

- public Game():
  Constructor of the class.
- public void start():
  Starts the flow of events from the first room after displaying the dungeon.
- public Dungeon getDungeon():
  Returns the dungeon.
- public AbstractCharacter getPlayer():
  Returns the player.

4.4. Class Interfaces

JavaFX library is used in implementation of class interfaces. Currently runnable method, gridpane and stackpane 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].