CS 319 - Object-Oriented Software Engineering

System Design Report

Animal Uprising

Group 3-A

Bora Ecer

Ata Gün Öğün

Albjon Gjuzi

Tanay Akgül

1. Introduction

1.1 Purpose of the System

Animal Uprising is a 2D strategy/adventure game which aims to provide a well-designed and enjoyable gameplay to entertain the players. The gameplay experience starts with an easy level which will work as a tutorial for the players, so that they can learn the game. However, in order to provide more satisfaction and pleasure the next levels of the game is designed to be more challenging, so that the game will have the player's attention, and increase the urge to play the game.

1.2 Design Goals

1.2.1 Efficiency

The main design goal of the system is efficiency, to achieve that, the system must be able to work in high performance. Since, a smooth gameplay is one of the most important feature which increases the player's urge to play the game, we are going to minimize the memory usage and the CPU usage. In order to achieve that, first, we are going to implement the code in the most efficient way possible. Also, we are going to design the system so that the workload of the objects is going to be nearly balanced.

1.2.2 Portability

Since portability is an important feature for a software to have various users from different platforms, we have decided to implement the game in Java, since it provides platform independent software which will make our system portable through various platforms.

1.2.3 Reliability

Our system will be reliable in terms of being consistent with the boundary conditions. The system should not respond with any unexpected results -like bugs, crashes- which are not specified in the boundary conditions. In order to provide that, we are going to test the system in all possible ways during and after the development stage. Also, the boundary conditions will be selected carefully and with caution so that there won't be a case with which puts the system in an unexpected situation. This will provide the system to foresee the possible fatal failures which will be dealt with.

1.2.4 Extensibility

In order to keep the interest of the players of Animal Uprising, the game requires to have new features and functionalities. For that the design of the game must be suitable to add further improvements and additions to the current system. In order to achieve this, object oriented architecture of our game must be designed in a way that each object should be able to operate with few dependencies. So that the modifications and further additions won't cause any bugs or crashes.

2. Software Architecture

2.1 Subsystem Decomposition

In this section, we will decompose the overall system into three different subsystems in order to use Model-View-Controller architectural style on our system. By doing that, we would like to accomplish creating a maintainable, efficient and flexible system.

Figure-1 is showing how the system is decomposed into three subsystems which are User-Interface Management, Game Control and Game Models.

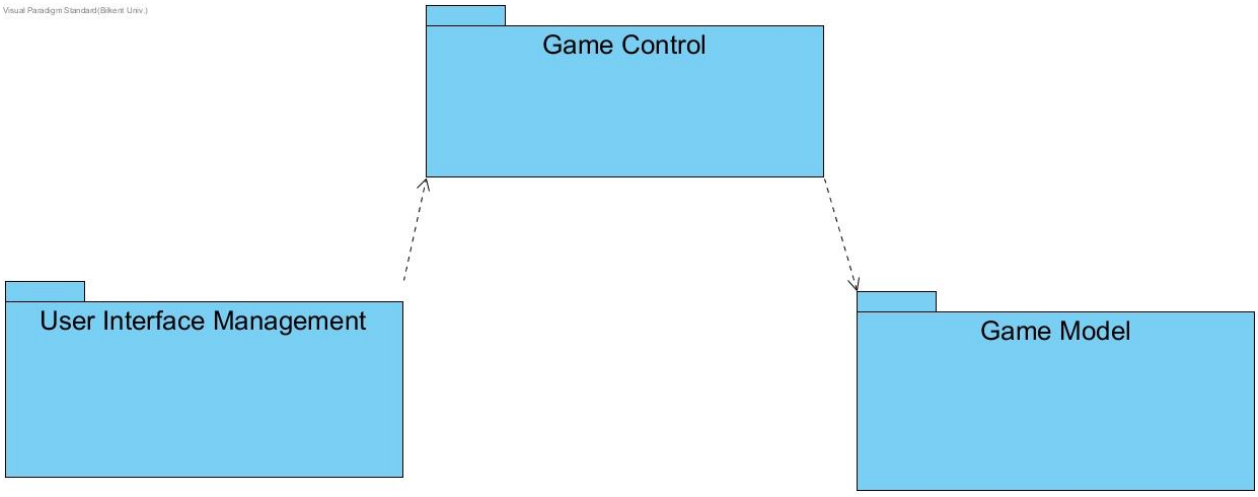


Figure-1 (System Decomposition)

All three systems are working on different tasks, and they communicate as given in the figure. User Interface Management includes the Menu package, and the Game Engine class, which will be responsible the construction of the game screen, according to the inputs from GameManager class which is in the Game Control Subsystem. Game Manager class will be responsible for handling the inputs, and making the decisions regarding to them, these decisions includes manipulating the user interface and the objects. Therefore, the GameManager class will be the façade class for the entire system.

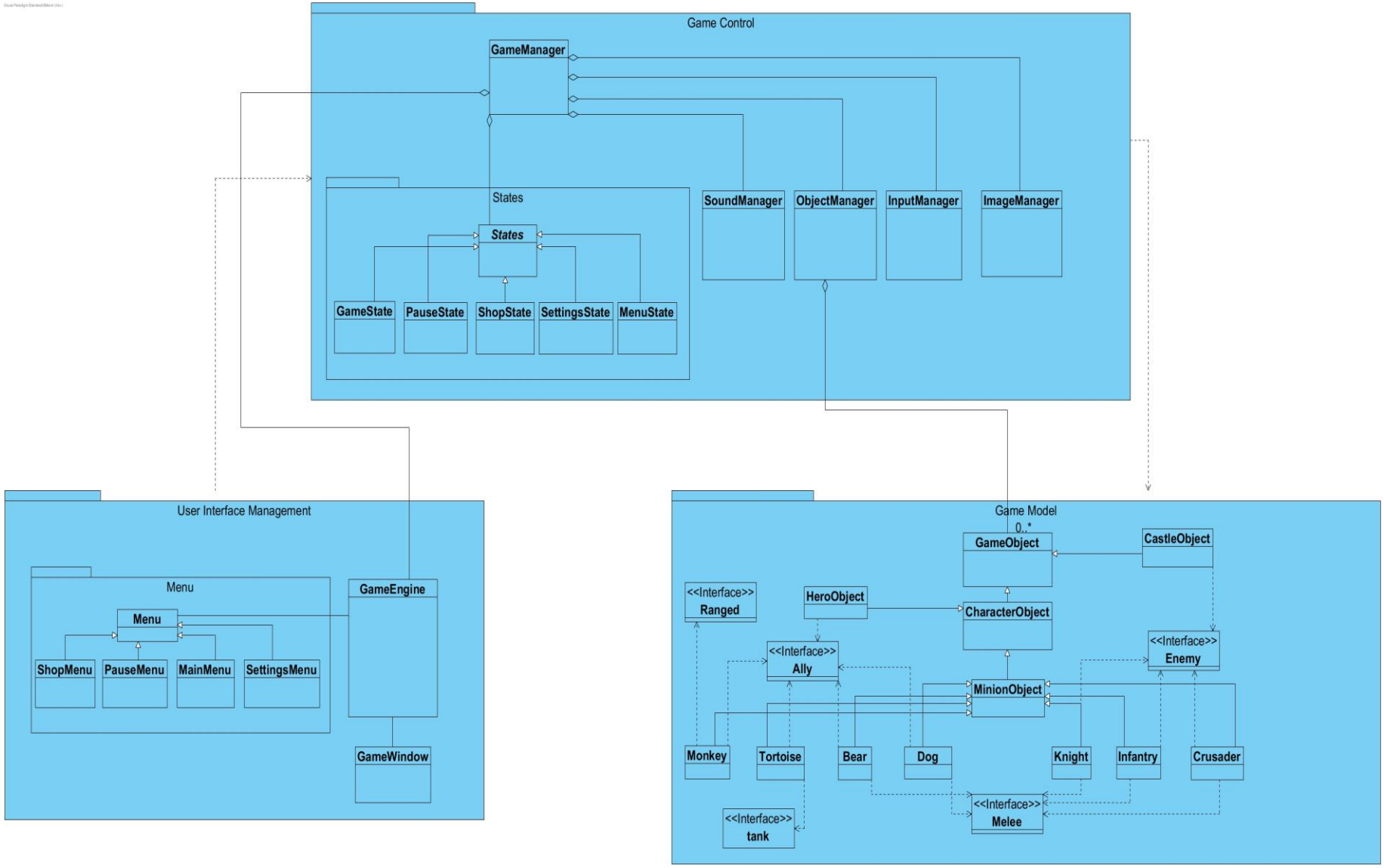


Figure-2 (Detailed System Decomposition)

Game Models Subsystem which has the game objects and their attributes. Because of this, the Game Control Subsystem, mainly ObjectManager class will be able to manipulate them.

2.2 Hardware/Software Mapping

Since our game will be implemented in Java, we will use the JDK 8. As for the hardware requirements, a keyboard and a mouse will be required so that the player can interact with the game. Since we are not planning to implement a complex system, a basic computer should be enough to run our game.

2.3 Persistent Data Management

Our system does not require a complex database for managing the data, therefore we are planning to store the game data in the client disk. So, whenever the system executes, we are going to load the files which are required for the system, to memory. Also, we are planning to store the images (gifs, and etc.) also the sound effects of the objects and the background music of the game.

2.4 Access Control and Security

Since the game does not require any network connection, therefore we will not implement any user authentication system. For software control, the GameControl subsystem will be able to access the files in order to assure the security of the data required for the game.

2.5 Boundary Conditions

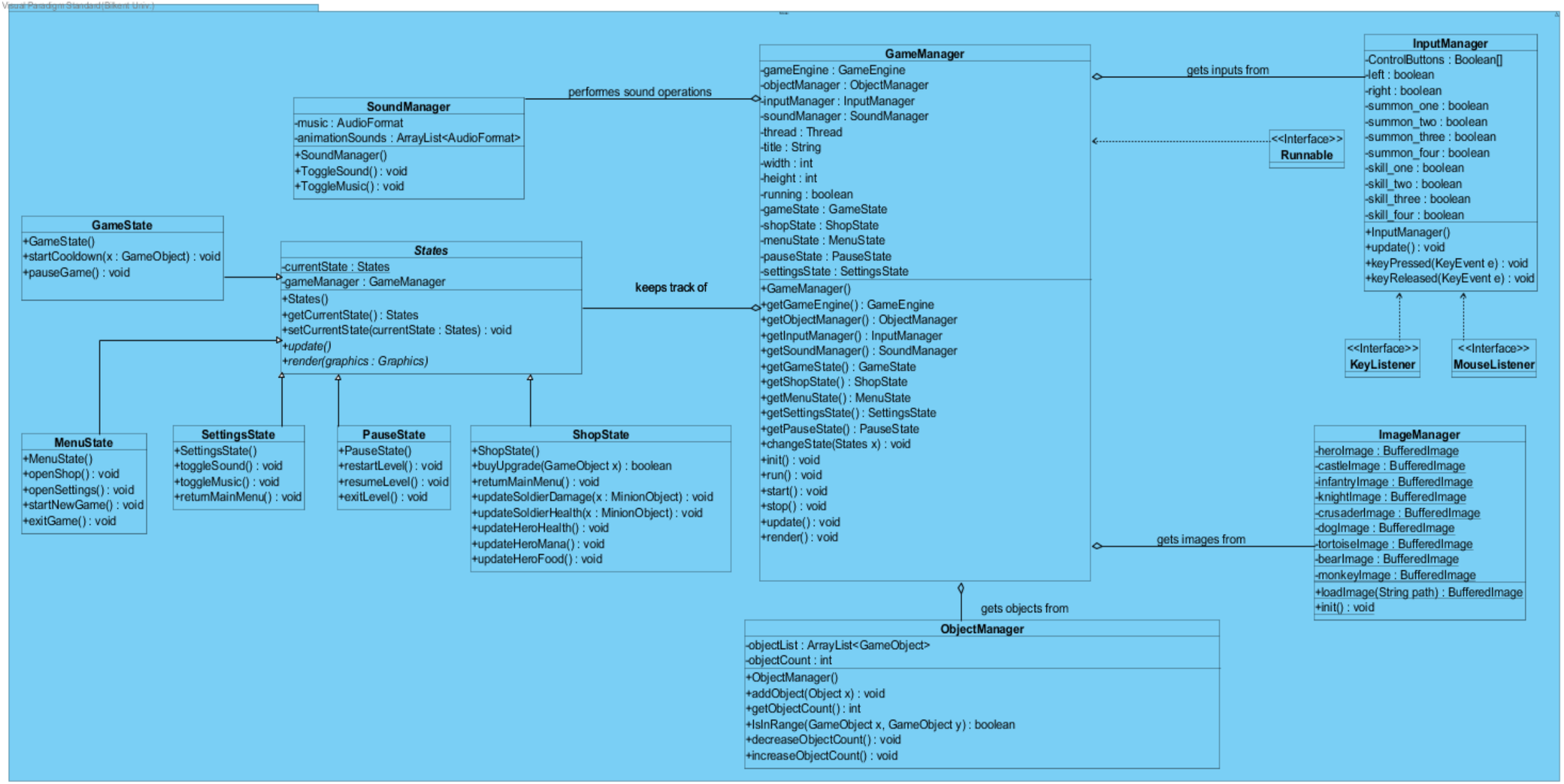
Initialization: The executable file of the game will be .jar file so it will not require an install. At startup time, the GameControl subsystem will be registered. The GameControl will access the data which is required for the user interface, and the sound effects.

Termination: The game can only be terminated from the main and pause menus, which will have a "Quit Game" button, and it will require user input. When the game is terminated, the Game Control subsystem will automatically inform the other subsystems to terminate as well.

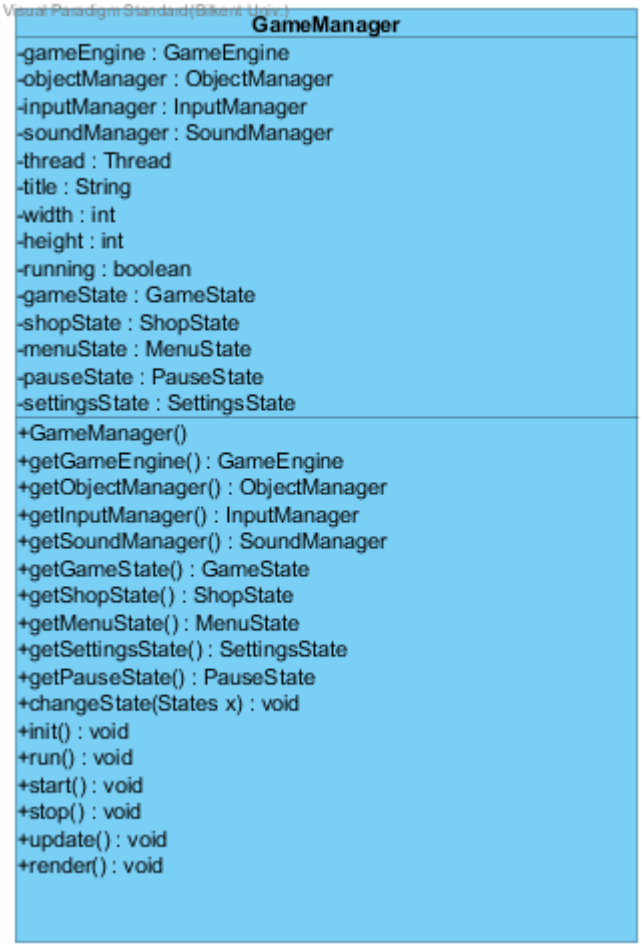
Failure: During a communication failure, the system will check the files or the data required are corrupted or not, if it is the system will erase the contents of these files.

3. Subsystem Services

3.1 Game Control Subsystem



3.1.1 GameManager Class



Attributes:

private gameEngine: This attribute will be an instance of GameEngine class which is definitely needed to make the necessary updates on the screen.

private objectManager: This attribute will be an instance of ObjectManager class in order to keep and update every game object that will be displayed.

private inputManager: This attribute will be an instance of InputManager class that is needed to make the user play the game, otherwise he would not move and click.

private soundManager: This attribute will be an instance of SoundManger class that is needed to control the sound effects during the execution time of the program.

private imageManager: imageManager is an instance of ImageManager class, which is used to load the images that will be rendered into the screen.

private thread: this attribute provides a concurrent execution, which is required for the game loop.

private title: a string variable to keep and pass the title to gameEngine.

private width: an integer variable to keep the width of the game screen and pass it to gameEngine.

private height: an integer variable to keep the height of the game screen and pass it to gameEngine.

private running: a Boolean attribute to check whether the game loop is running or not.

private gameState: an instance of GameState whose methods will be used while the player plays the game.

private menuState: an instance of menuState whose methods will be used while the player is using the main menu.

private shopState: an instance of ShopState whose methods will be used while the player is using the shop.

private pauseState: an instance of PauseState whose methods will be used when the program needs to render the pause menu.

Constructor:

Public GameManager(): initializes instances of GameManager.

Methods:

Public GameEngine getGameEngine(): getGameEngine method will get the GameEngine object needed for the app to function.

Public ObjectManager getObjectManager():getObjectManager method will be used to obtain the ObjectManager that controls all the objects in the game.

Public InputManager getInputManager(): getInputManager method will be used to get the input controls that the user can use to play the game.

Public ImageManager getImageManager(): getImageManager method will be used to get the images from the imageManager instance that the user can use to play the game.

Public SoundManager getSoundManager(): getSoundManager method will be used to obtain the sound settings.

Public void changeState(States x): it changes the currentState of the game, this method will only work when another state requests it.

Public void init(): It initializes the manager objects, gameEngine object, states objects and finally sets the currentState, which is the menuState.

Public void run(): run method is an unimplemented method provided by Runnable interface. Which is automatically called when the start() method is used. Run() method first calls init() method to initiate the objects required, then it starts the game loop, which does not end until the game ends. In every

iteration of the game loop, it calls update() and render() methods. Once the game ends, the execution of the run method ends with the call of the stop() method.

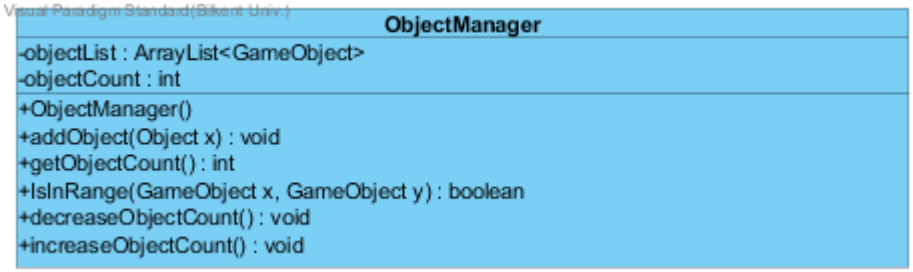
Public void start(): start() method, initiates the thread if the game is not running, in order to check that the method uses the running Boolean flag.

Public void stop(): this method calls the join method of the thread if the game is still running when its called, so similarly, it checks the Boolean flag, running.

Public void render(): render method initiates the bufferStrategy of the gameEngine canvas, if its not already created. Then it first clears the canvas of the view class which represents the current state, and calls the render method of the currentState.

Public void update(): this method is to update the objects which will be drawn into the screen. Therefore, since in different states of the game it will work differently, it will just call the update() method of the currentState.

3.1.2 ObjectManager Class



ObjectManager is also one of the important classes of our project since every object that will be displayed throughout the whole execution time of the program will be controlled and stored the ObjectList this class will have. It will basically function as an ArrayList of GameObject's.

Constructor:

public ObjectManager(): It initializes instances of **ObjectManager** object.

***Attributes*:**

private objectList: This attribute will be an ArrayList of GameObjects. GameObject is everything that will be displayed in the screen, the castle, the enemies, the hero or its allies.

Private objectCount: an integer variable to keep the amount of objects inside the objectList.

***Methods*:**

public void addObject(GameObject x): This method will be used to add gameObjects to the ArrayList mentioned above.

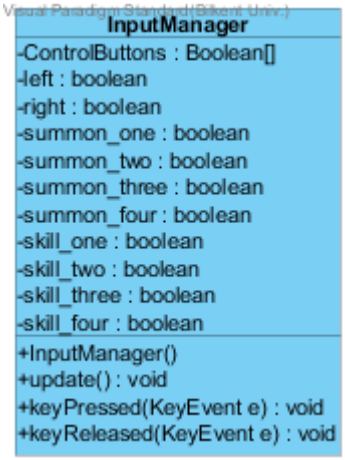
Public int getObjectCount(): returns the amount of objects in the objectList, which currently needs to be rendered.

Public void decreaseObjectCount(): decreases the object count. This method is called only when a object dies or needs to be destroyed.

Public void increaseObjectCount(): increases the object count, when addObject method called.

public boolean isInRange(GameObject x, GameObject y): The method will be used to check the location of each object. This Boolean method will be helpful for the update methods above.

3.1.3 InputManager Class



Contsructor:

public InputManager(): It initializes instances of **InputManager** object.

***Attributes*:**

private ControlButtons: This is a Boolean array in order to keep track of which button is pressed.

Private left, right, summon_one, summon_two, summon_three, skill_one, skill_two, skill_three, skill_four: these boolean variables are to check specific keys assigned in order to play the game.

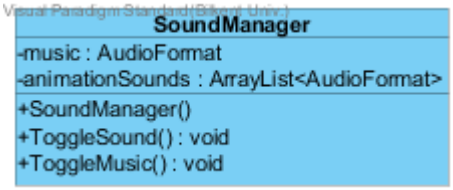
***Methods*:**

Private keyPressed(KeyEvent e): this method is called whenever a button is pressed and it sets true to the ControlButtons element which represents that specific key.

Private keyReleased(KeyEvent e): this method is called whenever a button is released and it sets false to the ControlButtons element which represents that specific key.

Private void update(): this method updates the current values of the left, right, summon_one, summon_two, summon_three, skill_one, skill_two, skill_three, skill_four from the ControlButtons array.

3.1.4 SoundManager Class



Constructor:

Public SoundManager() : initiates SoundManager instances.

Attributes:

Private music: an audio format instance for the theme music of the game.

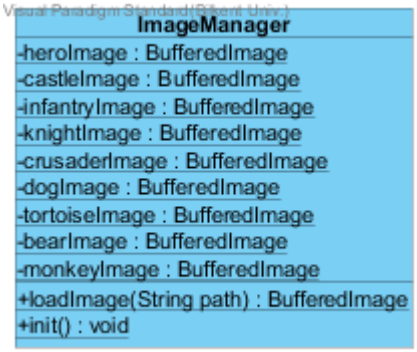
Private animationSounds: an instance of Audio Format arraylist which keeps the sound animations of the game objects.

Methods:

public void toggleSound(): This method enables the animation sounds of the game.

public void toggleMusic(): This method sets the music of the game.

3.1.5 ImageManager Class



Attributes:

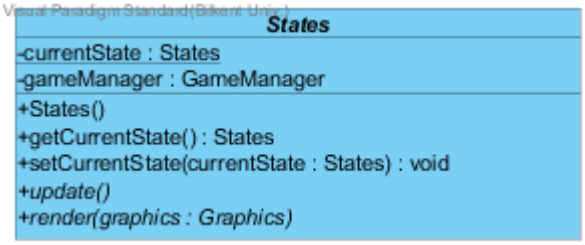
Public static heroImage, castleImage, InfantryImage, knightImage, crusaderImage, dogImage, tortoiseImage, bearImage, monkeyImage: these are static BufferedImage instances of the game objects, which will be rendered.

Methods:

Public static BufferedImage loadImage(String path): is a static method which loads the image from the given directory.

Public static void init(): loads all of the images to heroImage, castleImage, InfantryImage, knightImage, crusaderImage, dogImage, tortoiseImage, bearImage, monkeyImage**.**

3.1.6 States Class



Constructor:

Public States(): initiates the States instances.

Attributes:

private static currentState: is States instance, which represents the current state of the game.

Protected gameManager: is the game manager object created in the main method. Since the initialization of GameManager instances uses Singleton Pattern to provide only one GameManager instance created.

Methods:

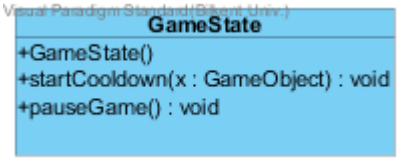
Public States getCurrentState(): returns the currentState object.

Public void setCurrentState(States state): changes the current state.

Public abstract void update(): abstract method, since it differs from state to state they will be defined in the child classes

Public abstract void render(graphics g): abstract methods since it differs from state to state they will be defined in the child classes.

3.1.7 GameState Class



Constructor:

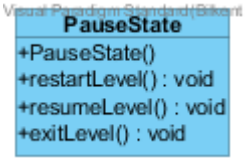
Public GameState(): initiates the GameState instances.

Methods:

Public void startCooldown(GameObject x): starts the cooldown of a given GameObject type.

Public void pauseGame(): requests to change the currentState to the pauseState.

3.1.8 PauseState Class



Constructor:

Public PauseState(): initiates the PauseState instances.

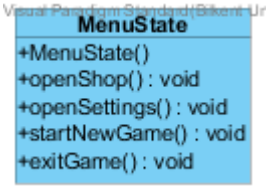
Methods:

Public void restartLevel(): resets the progress done in that level, and requests to change the currentState to gameState.

Public void resumeLevel(): resumes the game, and requests to change the currentState to gameState.

Public void exitLevel(): resets all of the level progresses and requests to change the currentState to menuState.

3.1.9 MenuState Class



Constructor:

Public MenuState(): initiates the MenuState instances.

Methods:

Public void openShop(): calls the changeState() method to change the currentState to shopState

Public void openSettings(): calls the changeSettings() method to change the currentState to settingsState.

Public void startNewGame(): calls the changeState() method to change the currentState to gameState

Public void exitGame(): calls the stop() method of the gameManager object.

3.1.10 ShopState Class

Constructor:

Public ShopState(): initiates the ShopState instances.

Methods:

Public void returnMainMenu():changes the currentState to menuState.

Public boolean buyUpgrade(GameObject x): buys upgrade for the given GameObject type.

Public void updateSoldierDamage(MinionObject x): updates the damage output of the given MinionObject type if the buyUpgrade method is done successfully for that MinionObject

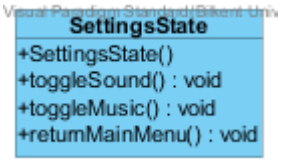
Public void updateSoldierHealth(MinionObject x): updates the health amount of the given MinionObject type if the buyUpgrade method is done successfully for that MinionObject

Public void updateHeroDamage(): updates the damage output of the HeroObject if the buyUpgrade method is done successfully for the HeroObject.

Public void updateHeroMana (): updates the mana amount of the HeroObject if the buyUpgrade method is done successfully for the HeroObject.

Public void updateHeroFood (): updates the food amount of the HeroObject if the buyUpgrade method is done successfully for the HeroObject.

3.1.11 SettingsState Class



Constructor:

Public SettingsState(): initiates the SettingsState instances.

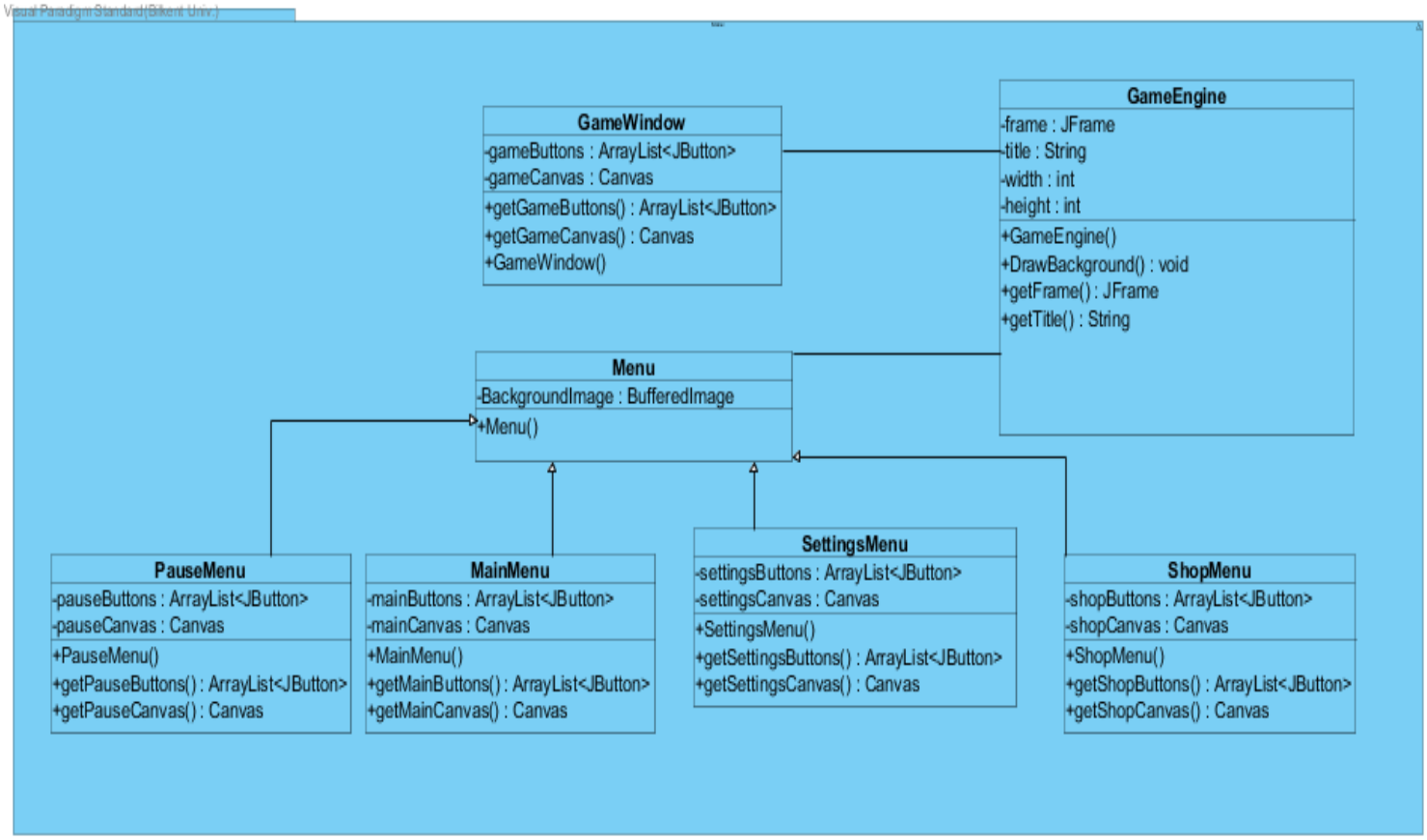
Methods:

Public void toggleSound(): requests to toggle the sound animations through gameManager.

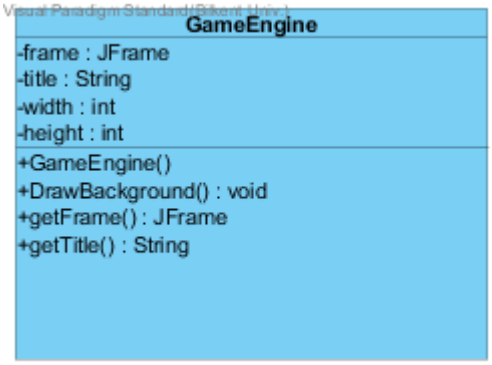
Public void toggleMusic(): requests to toggle the music through gameManager.

Public void returnMainManu(): calls the changeState() method to change the currentState to menuState.

3.2 UI Management Subsystem



3.2.1 GameEngine Class



- Every game needs its engine and so does ours. This class will provide everything needed to play the game in the correct manner and it will connect the other main subclasses. This class is updates GameManager class which then takes care of the other things.

Constructor:

public GameEngine(): It initializes instances of **GameEngine** object.

***Attributes*:**

Private frame: an instance of Jframe variable, the canvas object of the Menus or GameWindow classes will be added into it depending on the currentState of the game. In other words, if the currentState is GameState, then the canvas of the GameWindow will be active on the frame, else if the currentState is menuState, then the canvas of the mainMenu will be active on the frame.

Private title: String variable for the title of window.

Private width: integer variable for the width of the screen.

Private height: integer variable for the height of the screen

***Methods*:**

Public Jframe getFrame(): returns the frame object.

Public String getTitle(): returns the title.

public void DrawBackground(): This method is just going to draw the background based on what stage the game is.

3.3.2 Menu Class

Constructors:

public Menu(): Initializes background image.

Attributes:

private bufferedImage backgroundImage: This attribute sets background image of the menu.

3.2.3 ShopMenu Class

Constructors:

public ShopMenu(): Initializes shopButtons and shopCanvas and does the required settings.

Attributes:

private ArrayList<JButton > shopButtons: This arraylist of JButtons will keep the buttons needed in the shop menu.

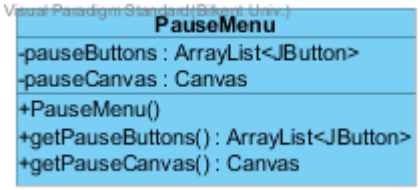
Private Canvas shopCanvas: a canvas attribute on which the objects and shapes needed on the shop menu will be rendered.

Methods:

public ArrayList<JButton > getShopButtons(): returns the shopButtons ArrayList.

public getShopCanvas(): returns the shopCanvas.

3.2.4 PauseMenu Class



Constructors:

public PauseMenu(): Initializes pauseButtons and pauseCanvas and does the required settings.

Attributes:

private ArrayList<JButton > pauseButtons: This arraylist of JButtons will keep the buttons needed in the pause menu.

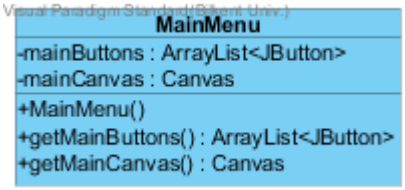
Private Canvas pauseCanvas: a canvas attribute on which everything the user sees on the pause menu will be rendered.

Methods:

public ArrayList<JButton > getPauseButtons(): returns the puaseButtons ArrayList.

public getPauseCanvas(): returns the pauseCanvas.

3.2.5 MainMenu Class



Constructors:

public MainMenu(): Initializes mainButtons and mainCanvas and does the required settings.

Attributes:

private ArrayList<JButton > mainButtons: This arraylist of JButtons will keep the buttons needed in the main menu.

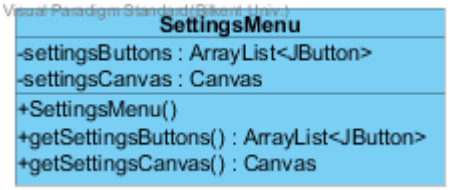
Private Canvas mainCanvas: a canvas attribute on which the objects and shapes needed on the main menu will be rendered.

Methods:

public ArrayList<JButton > getMainButtons(): returns the mainButtons ArrayList.

public getMainCanvas(): returns the mainCanvas.

3.2.6 SettingsMenu Class



Constructors:

public SettingsMenu(): Initializes settingsButtons and settingsCanvas and does the required settings.

Attributes:

private ArrayList<JButton > settingsButtons: This arraylist of JButtons will keep the buttons needed in the settings menu.

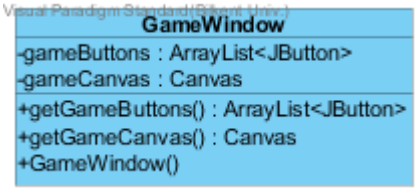
Private Canvas settingsCanvas: a canvas attribute on which the objects and shapes needed on the settings menu will be rendered.

Methods:

public ArrayList<JButton > getSettingsButtons(): returns the settingsButtons ArrayList.

public getSettingsCanvas(): returns the settingsCanvas.

3.2.7 GameWindow Class



Constructor:

Public GameWindow(): initiates the GameWindow instances, whose settings for view elements are done.

Attributes:

Private gameButtons: an instance of ArrayList which keeps the JButton objects. These buttons will be the soldier summoning, skills, and movement and pauseGame buttons.

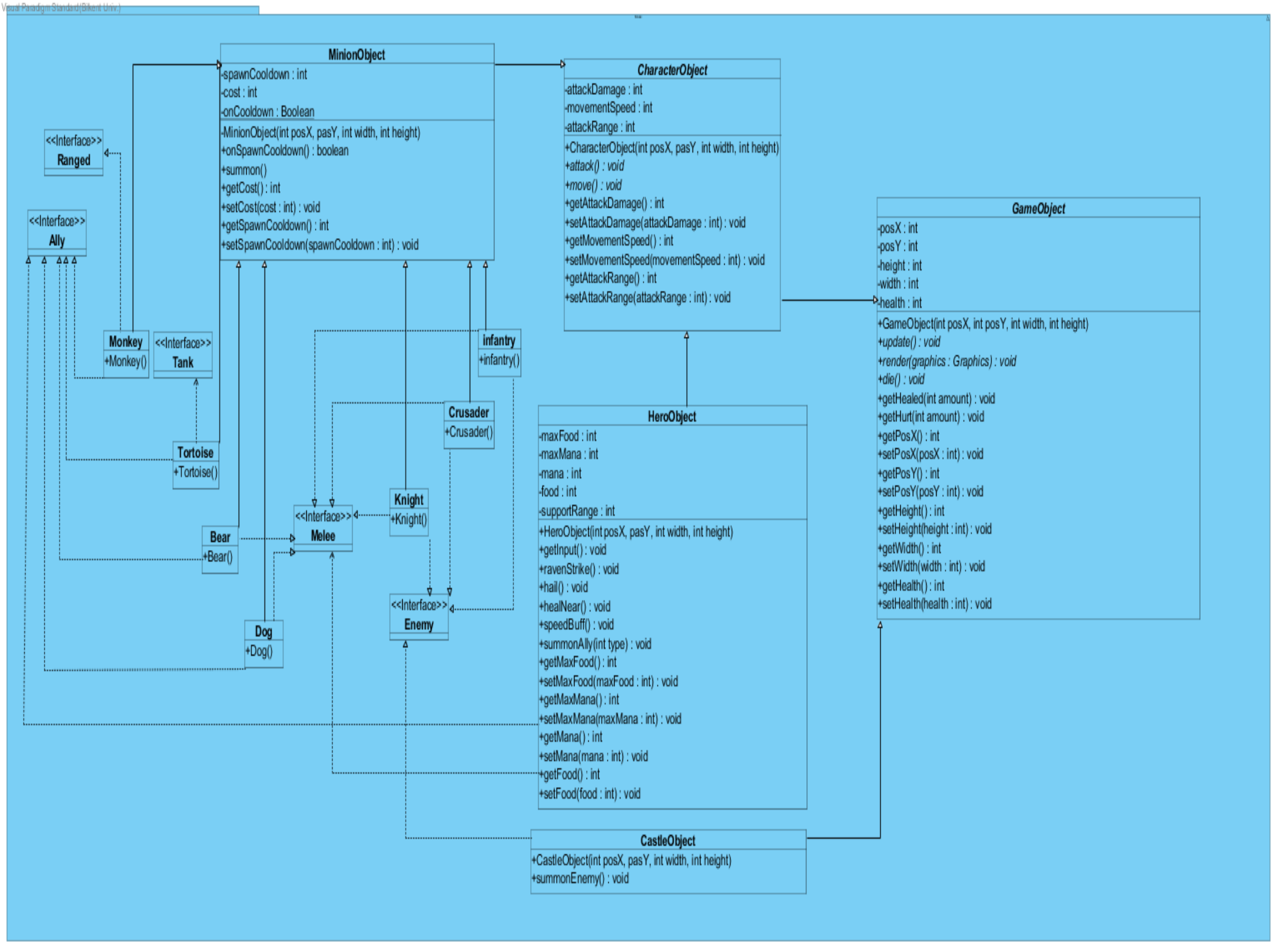
Private gameCanvas: an attribute which is an instance of Canvas which the gameButtons and the GameObjects will be rendered on.

Methods:

public ArrayList<JButton > getGameButtons(): returns the gameButtons ArrayList.

public getGameCanvas(): returns the gameCanvas.

3.3 Game Model Subsystem

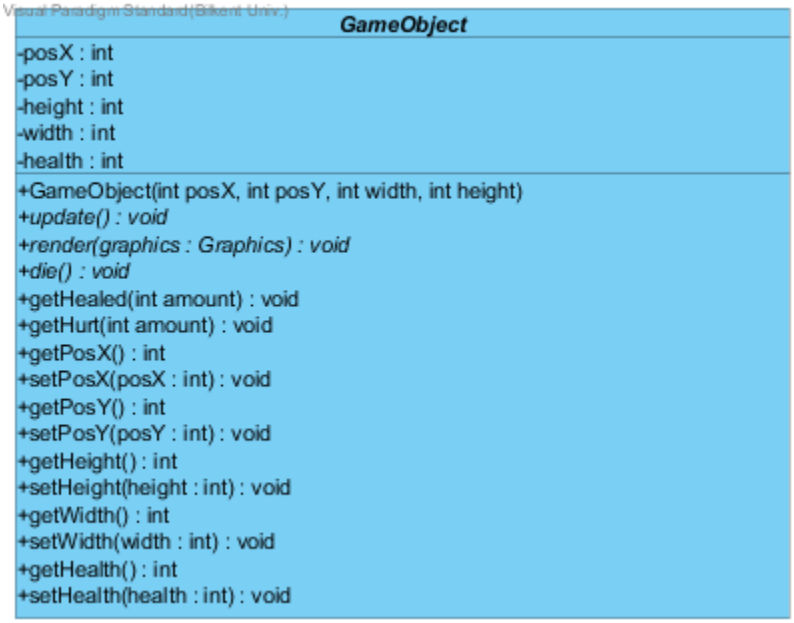


"Game Model Subsystem" consists of the objects which will be rendered to the screen during the gameState. There are three main game objects in our game, which are the following: HeroObject (Player), CastleObject (Enemy Base) and MinionObjects (both ally and enemy bots). Since the CastleObject cannot move and attack, while the HeroObject and MinionObjects can, we decided to make Hero and Minion objects subclasses of CharacterObject class, which is an abstract class.

MinionObject has also seven subclasses, which are the following: Monkey, Tortoise, Bear, Dog, Knight, Crusader, Infantry. The first four of them are the allied soldiers which only the HeroObject can summon them so Monkey, Tortoise, Bear and Dog implements Ally interface along with the HeroObject. Similarly, Knight, Crusader and Infantry are enemy soldiers which can be summoned by the CastleObject only so they implement Enemy interface along with the CastleObject.

Also, two of the MinionObjects have different types of roles in the game. These two are: Monkey and Tortoise. Monkey is a ranged damage dealer, so it attacks from a distance so it implements Ranged interface whereas Tortoise is a tank so it does not deal any damage to the enemy minions, but it absorbs their damages, protecting the other allies, so it implements Tank interface. The remaining of the MinionObjects and HeroObject are attacking in melee distance, so they implement Melee interface. Notice that the CastleObject does not implement any one of Ranged, Tank, Melee since it does not have any attack type. It is just a stationary object.

3.3.1 GameObject Class



"GameObject" is an abstract class that is the parent of all game objects including HeroObject, all MinionObjects and CastleObject.

Contsructor:

Public GameObject(int posX, int posY, int width, int height): initiates the GameObject.

***Attributes*:**

Private posX: int variable that keeps position of the object in x-coordinate.

Private posY: int variable that keeps position of the object in y-coordinate.

Private height: int variable that keeps height of the object image.

Private width: int variable that keeps width of the object image.

Private health: int variable that keeps the health of the object.

Methods:

Public abstract void update(): abstract method to update the object, this method can be different for each method so they are needed to be implemented inside the child classes.

Public abstract void render(): abstract method to render the object, this method can be different for each object, since images are different for each object, so they are needed to be implemented inside the child classes.

Public abstract void die(): another abstract method, it basically destroys the object but since the death animations of each object are different, this method should be implemented in child classes.

Public void getHealed(int amount): method to increase the health of the object.

Public void getHurt(int amount): method to decrease the health of the object.

Public int getPosX(): returns position of the object in X-coordinate.

Public void setPosX(int posX): sets the X-position to the given value.

Public int getPosY(): returns position of the object in Y-coordinate.

Public void setPosY(int posY): sets the X-position to the given value.

Public int getHeight(): returns the height of the image of the object.

Public void setHeight(int height): sets the height of the object image to given value.

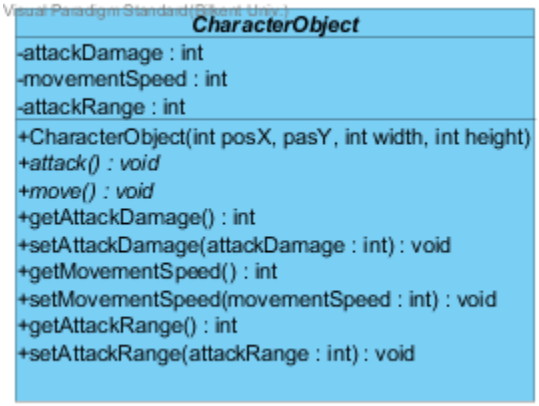
Public int getWidth(): returns the width of the image of the object.

Public void setWidth(int width): sets the width of the object image to given value.

Public int getHealth(): returns the current health of the object.

Public void setHealth(int health): sets the health of the object to given value.

3.3.2 CharacterObject Class



"**CharacterObject**" is also an abstract class that is the parent of MinionObject and HeroObject.

Contsructor:

Public CharacterObject(int posX, int posY, int width, int height): initiates the CharacterObject with given parameters.

***Attributes*:**

Private attackDamage: integer variable to keep the damage output of the object

Private movementSpeed: integer variable to keep the movement speed of the object

Private attackRange: integer variable to keep the attack range of the object.

***Methods*:**

Public abstract void attack(): this method is abstract because of the attacking animations of the objects are different from each other. So, the child classes must implement this method.

Public abstract void move(): move() method changes the location of the objects. The reason for the abstraction is the following: First of all, their movement animations are different. Also, Allied Minion objects move from posX = 0 to CastleObject's posX location, which is the opposite edge of the screen whereas the Enemy Minion objects move from CastleObject's posX location to posX = 0. So, it can be said that both Allied and Enemy Minion objects start moving to their destination automatically when they are summoned and each of them are able to move towards only one direction. However, the movement of the HeroObject is much more complicated than that. First of all, it requires user input from the keyboard so it is not done automatically, and HeroObject can be able to stop, move backward and move forward. So, because of all these, the move method must be implemented in the child classes of CharacterObject.

Public int getAttackDamage(): returns attackDamage of the object.

Public void setAttackDamage (int attackDamage): sets the attackDamage to the given value.

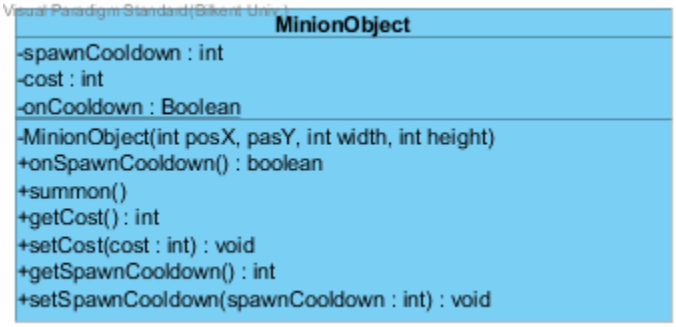
Public int getMovementSpeed (): returns movementSpeed of the object.

Public void setMovementSpeed (int posX): sets the movementSpeed to the given value.

Public int getAttackRange(): returns attackRange of the object.

Public void setAttackRange (int attackRange): sets the attackRange to the given value.

3.3.3 MinionObject Class



MinionObject Class is yet another class that has the common attributes of all the minion type objects. They have a private constructor and they are intended to be constructed only by the "**summon**()" operation given the conditions are met.

Contsructor:

Private MinionObject(int posX, int posY, int width, int height): initiates the MinionObject with given parameters.

***Attributes*:**

Private Static onCooldown: Static Boolean variable to check whether the minion is in cooldown, and summoning it is not possible for a while.

Private cost: integer variable to keep the movement speed of the object

Private spawnCooldown: integer attribute which has the number of seconds that the specific minion type is in cooldown.

***Methods*:**

Public static void summon(): this method is basically creating a new minion object to the objectList in the objectManager and sets the cooldown. Each minion object overrides this method since all of them have different healths.

Public int getCost(): returns cost of the object.

Public void setCost (int cost): sets the cost to the given value.

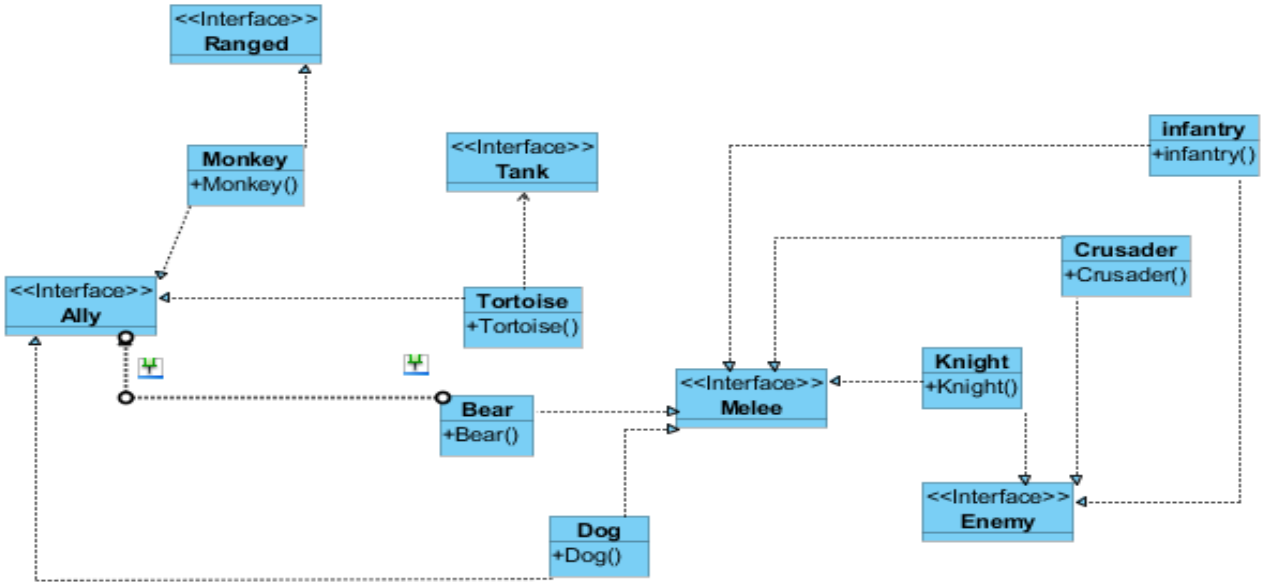
Public int getSpawnCooldown(): returns spawnCooldown of the object.

Public void setSpawnCooldown(int spawnCooldown): sets the spawnCooldown to the given value.

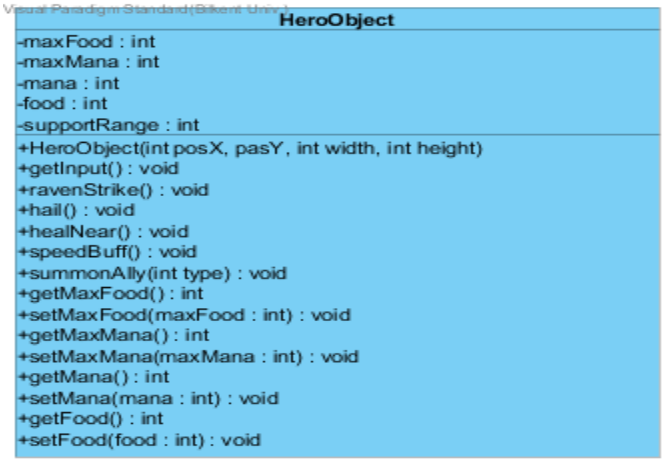
Public Boolean onSpawnCooldown: returns if the minion type object is in cooldown or not. In another words, returns onCooldown.

3.3.4 Minion Subclasses and Interfaces

Since the Subclasses of the MinionObject do only have the constructors and the overridden methods, they are presented like the following.



3.3.5 HeroObject Class



Contsructor:

Public HeroObject(int posX, int posY, int width, int height): Initiates the HeroObject. It applies Singleton Pattern in order to make sure there are only one HeroObject in the level.

***Attributes*:**

Private maxFood: integer variable to keep the upper limit of the food amount that the player can have. This can be upgraded through the upgradeHeroFood() method of ShopState.

Private maxMana: integer variable to keep the upper limit of the mana amount that the player can have. This can be upgraded through the upgradeHeroMana() method of ShopState.

Private food: integer attribute for the food that the player currently has. The value is incremented with each call of update() method of HeroObject, and decreased with every minion that the player summons, if the player has enough food to summon the ally.

Private mana: integer attribute for the mana that the player currently has. The value of it is incremented with each update() method, and decreased when the player casts a spell, if the player has enough mana to cast the spell in the first place.

Private supportRange: this integer attribute is used to determine which objects are able to be supported by the helpful spells of the HeroObject. -speedBuff() and healNear() methods-

***Methods*:**

Public getInput(): This method is the most important method of the HeroObject, it gets the Boolean variables, left, right, summon_one, summon_two, summon_three, summon_four, skill_one, skill_two, skill_three, skill_four from the inputManager and decides which of the following methods are going to be executed.

Public void ravenStrike(): is the first offensive skill of the HeroObject. It decreases the health of the first enemy -including castle object- if they are in the attackRange of the HeroObject

Public void hail(): is the last offensive skill of the HeroObject, functionally, it is similar to the ravenStrike() but this time it hits to an area of enemy objects rather than hitting first enemy object that comes in its path.

Public void healNear(): heals the ally minion type objects which are in the supportRange and the HeroObject itself.

Public void speedBuff(): increases the movement speed of the ally minion type objects which are in the supportRange, and increases the movement speed of HeroObject itself.

Public void summonAlly(int type) : this method calls the summon() method of the allied minion typed objects, with respect to the type integer, which is the input taken from the getInput() method.

Public int getMaxFood(): returns maxFood of the object.

Public void setMaxFood (int maxFood): sets the maxFood to the given value.

Public int getMaxMana(): returns maxMana of the object.

Public void setMaxMana (int maxMana): sets the maxMana to the given value.

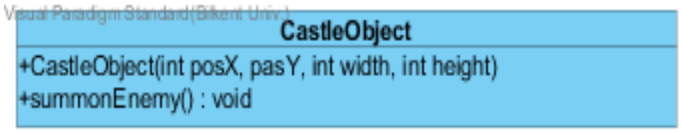
Public int getFood(): returns food of the object.

Public void setFood (int food): sets the food to the given value.

Public int getMana(): returns mana of the object.

Public void setMana (int mana): sets the mana to the given value.

3.3.6 CastleObject Class



Contsructor:

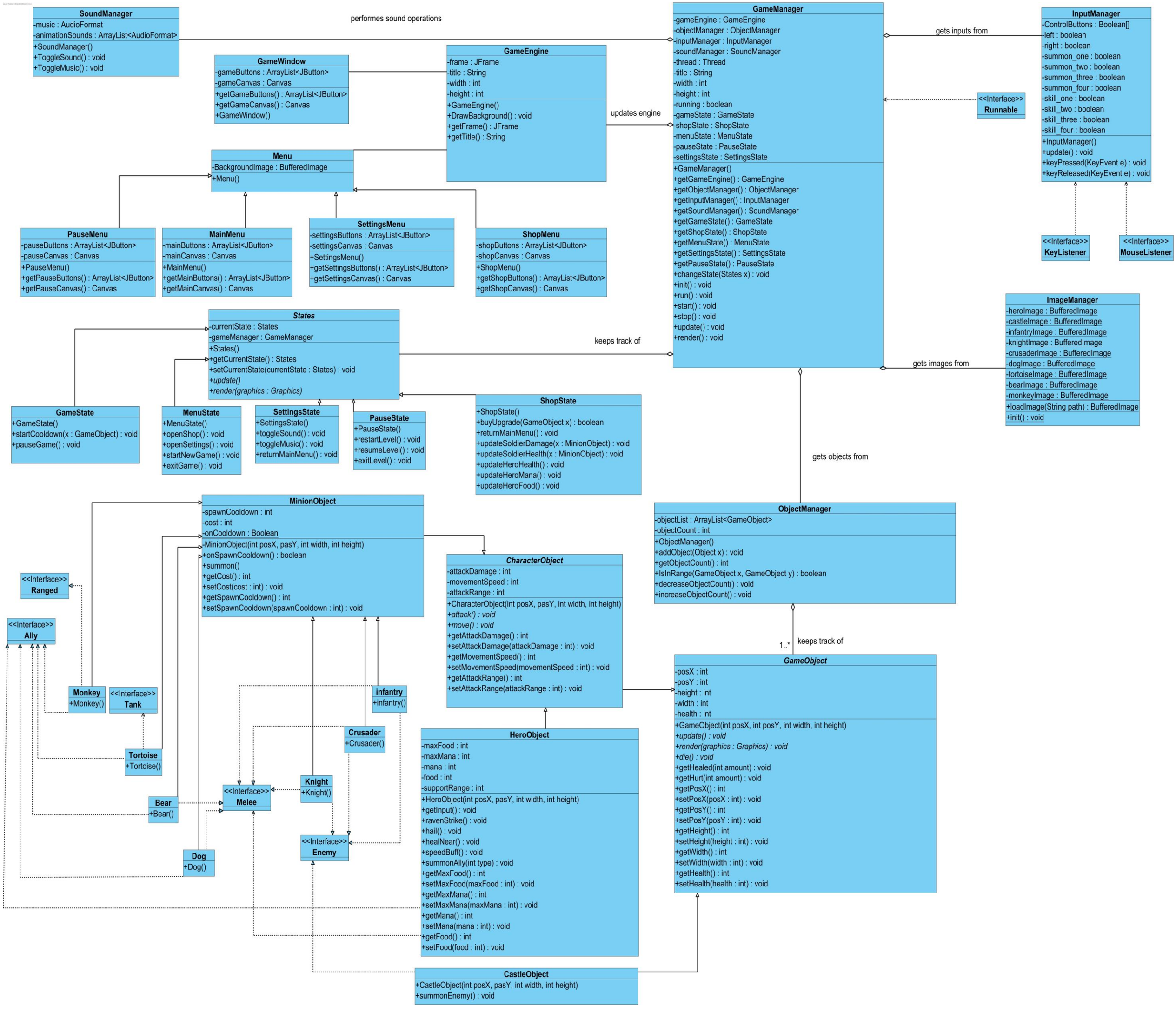
Public CastleObject(int posX, int posY, int width, int height): Initiates the CastleObject. Just like the HeroObject, the constructor applies Singleton Pattern in order to make sure there are only one CastleObject in the level.

***Methods*:**

Public void summonEnemy() : method which calls the summon() method of one of the enemy minion typed objects.

4. Low-Level Design

4.1 Final Object Design



4.2 Object design trade-offs

- **Functionality vs Usability**

  Our game focuses on usability more than functionality since our main goal is to entertain the users with a basic game. So, instead of developing a complex system, focused on developing a basic system which is easier to use.

- **Efficiency vs Portability:**

  Portability is very important for a game to reach wider range of users. Since we use Java, which offers a platform independent program, we satisfy the portability feature, however, since java is less efficient compared to the other languages, in this process we sacrifice the efficiency.

- **Cost vs Reusability:**

  We did not focus on reusability while designing our system because we are not planning to use the existing classes of our game in different projects. Therefore, the classes are designed to do their task only for this system which will make our system less complex and prioritizing the cost.

4.3 Packages

4.3.1 java.util

Util package contains the ArrayList which is used to store, update and control various objects with different types. For example in the ObjectManager class, the objectList is an ArrayList, which is essential to the gameState.

4.3.2 java.awt

Java.awt provides graphics and images to paint and it contains classes to create Graphical User Interface. It keeps visual components like buttons, menus and frames.

4.3.3 java.awt.event

The java.awt.event package contains the Listeners and Events interfaces such as KeyEvent or KeyListener, which handles interaction between GUI and user and it is used for event handling.

4.3.4 java.awt.image

This package provides the BufferedImage which is allowed us to store our images.

4.3.5 javax.imageio

This package provides the ImageIO which reads an Image from a directory and writes it into a BufferedImage.

4.3.6 java.io

Java.io package provides IOException which makes the System.exit(1) call if an exception occur during the ImageIO's read process.

4.3.7 javax.swing

This package provides important view elements such as JFrame and JButton.

4.4 Class Interfaces

4.4.1 MouseListener

MouseListener is an interface provided by java.awt.event package, which is useful if MouseEvent is going to be used, which is to say if user uses the mouse to interact with a GUI component. It requires to override mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased functions.

4.4.2 KeyListener

KeyListener is also provided by java.awt.event package, which is used in a similar to the MouseListener but used to track if any KeyEvent is going to be used, in other words the user interacts with the GUI using a keyboard. It requires to override keyPressed, keyReleased, keyTyped methods.

4.4.3 Runnable

Implementation of Runnable interface is used to provide classes whose instances are needed to be executed by a thread. Which is to say, it is important for the classes which requires to execute code while they are active. The interface requires to override run() method.

5. Improvement Summary

Considering the previous iteration, our system design had a lot of changes. The one of most important one of these changes must be the addition of the states objects. The reason for this addition was the necessity of using different update() and render() methods in different stages of the game. We came up with this idea from the observer pattern, however it is not directly observer pattern. This idea of adding states allowed us to connect the menu classes to the rest of the system. The menu classes were not something we added in this iteration but they were something which did not know how to use them with our current system. Also, we did increase the number of menu classes and added a new class called GameWindow to the UI Management subsystem, which is similar to the menu classes but it is for the game itself. Also, we did add a new class called ImageManager class to the GameControl Subsystem, which allowed us to use images for the objects. As for the Game Model Subsystem, we merged the StationaryObject with the CastleObject in order to have high coherence. Also, we did use the Singleton Pattern in certain object which we had to make sure that there are only one of them exists in the system. Lastly, we did add various new attributes and methods to almost all of the classes and removed some in order to make progress with the final product of the game.

6. Glossary & References

6.1 Definitions, acronyms and abbreviations:

JDK: [1] Java Development Kit

MVC: [2] Model-View-Controller

6.2 References

[1]: http://www.oracle.com/technetwork/java/javase/overview/index.html

[2]: *Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010.*