  
  
  
CS 319 - Object-Oriented Software Engineering  
Design Report  
  
  
Project Name: Survival in Bilkent

Group 2-M

Pelin Elbin Günay - 21402149

Kübra Nur Güzel - 21400946

Alper Şahıstan - 21501207

Semih Teker– 21300964

Table of Contents

[1. Introduction 3](#_Toc496390006)

[1.1. Purpose of the system 3](#_Toc496390007)

[1.2 Design Goals 3](#_Toc496390008)

[End User Criteria: 3](#_Toc496390009)

[Maintenance Criteria: 3](#_Toc496390010)

[Performance Criteria: 3](#_Toc496390011)

[Trade-Offs: 4](#_Toc496390012)

[1.3 Definitions, acronyms, and abbreviations 4](#_Toc496390013)

[2. Software Architecture 5](#_Toc496390014)

[2.1 Subsystem Decomposition 5](#_Toc496390015)

[2.2 Hardware / Software Mapping 7](#_Toc496390016)

[2.3 Persistent Data Management 8](#_Toc496390017)

[2.4 Boundary Condition 8](#_Toc496390018)

[3. Subsystem Services 8](#_Toc496390019)

[Detailed Object Design 8](#_Toc496390020)

[Title Menu Class 9](#_Toc496390021)

[Pause Menu Class 10](#_Toc496390022)

[3.4 Level Manager Subsystem 15](#_Toc496390024)

[3.5 Game Logic Subsystem 19](#_Toc496390025)

[GameMaster Class 20](#_Toc496390026)

[InputManager Class 25](#_Toc496390027)

[InGameGUIMaster Class 25](#_Toc496390029)

[3.6 Game Screen Elements Subsystem 27](#_Toc496390030)

[GameUIScreen Class 27](#_Toc496390031)

[UIObject Class 27](#_Toc496390032)

[Icon Class 28](#_Toc496390033)

[PowerUpStatus Class 28](#_Toc496390034)

[IconXAmount Class 29](#_Toc496390035)

[Layer Class 29](#_Toc496390036)

[Multibar Class 30](#_Toc496390037)

[3.7 Game Entities Package 31](#_Toc496390038)

# **1. Introduction**

## **Purpose of the system**

Survival in Bilkent is a top to down 2D shooter game. The purpose of the game is to entertain the player. In order to design more enjoyable game, we created a variety of enemies such as bugs, assignments, quizzes, labs, midterms, and finals. These are the obstacles that the players shoot. Additionally, we designed our game that includes 4 levels. These levels’ difficulty increases from first to the last level. Because of that, the game is more enjoyable and challenging.

## **1.2 Design Goals**

### End User Criteria:

**User-Friendliness:** The game can be played easily. In other words, the mechanics are generic and self-explanatory that seen in many 2-D shooter games. Additionally, we create an understandable user interface.

### Maintenance Criteria:

**Extensibility:** Reusability and extendibility are crucial for software projects. Especially, we will add new upgrade items to the project. To addition, Survival in Bilkent can be modified and re-used in another project easily.

**Portability:** Our game will work in different software environments and we will implement our project in Java because it has JVM that provides an opportunity to work correctly.

### Performance Criteria:

**Game Performance**: The game should move instantly. Average FPS should be greater than 30. Otherwise, the game cannot be playable and enjoyable.

### Trade-Offs:

**Performance vs. Memory**: Performance is important for our games. We want our game will run quickly. In order to move our game smoothly, we should increase the memory space to give better game experience.

**Efficiency – Reusability:** We indicated that our game will be reusable. We will try to write the code as much reusable we can. However, if the game’s efficiency is affected in a bad way, we will try to write the code more efficiently rather than reusability.

## **1.3 Definitions, acronyms, and abbreviations**

MVC: Model View Controller

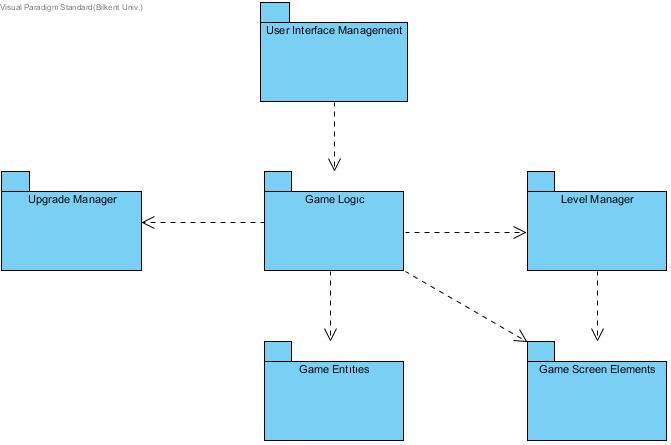
JDK: JavaDevelopment Kit

JVM: Java Virtual Machine

FPS: frames per second

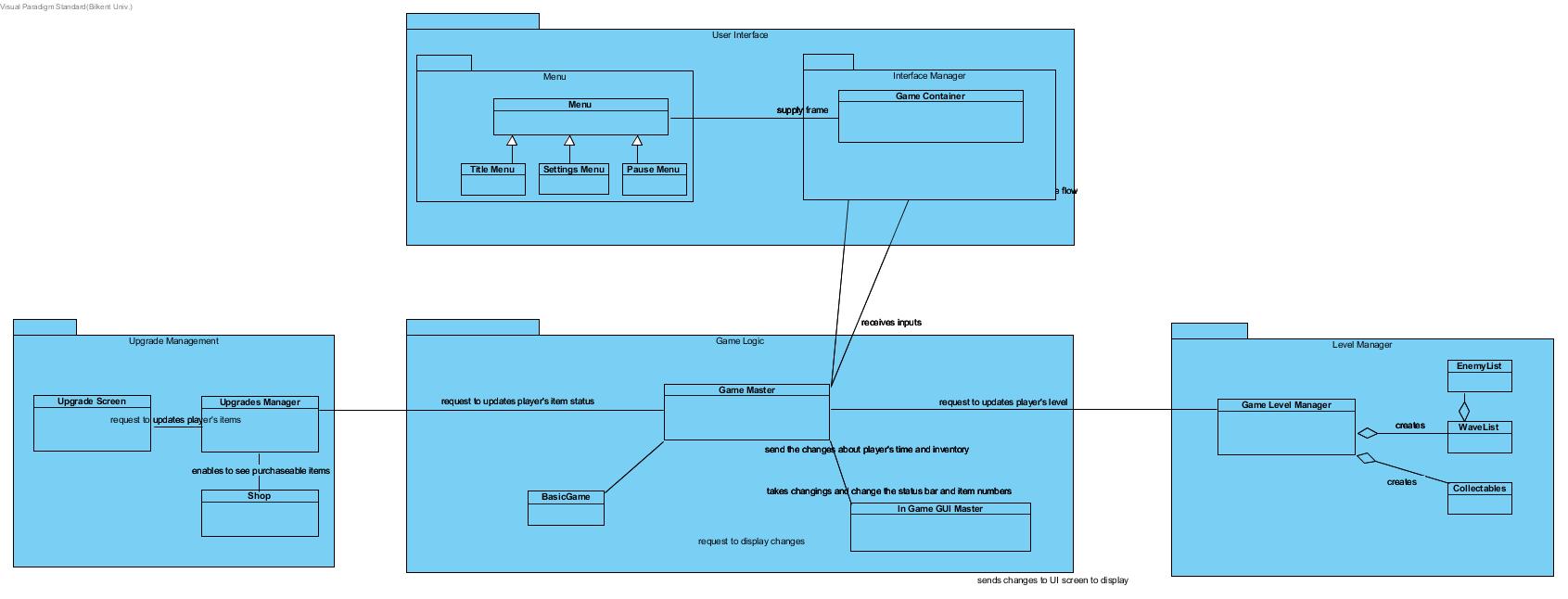
# **2. Software Architecture**

## **Subsystem Decomposition**

We choose to construct a three-tier architectural decomposition for our projects system because it is the most suitable option for this game. Subsystem decomposition includes User Interface Management, Game Logic, Upgrade Manager, Level Manager, Game Entities, and Game Screen Elements.

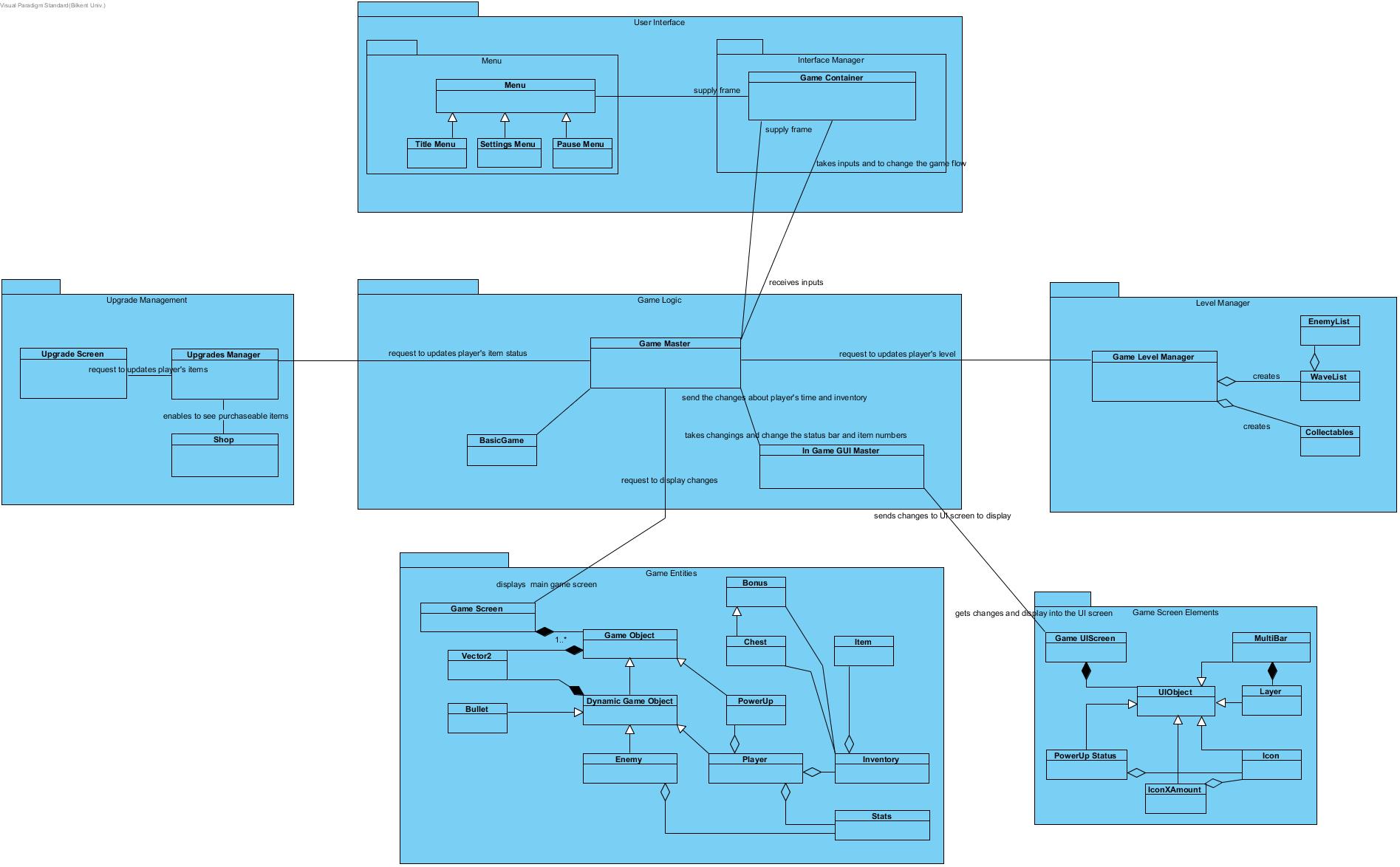
Subsystem Decomposition

In the first layer, presentation layer, we have classes responsible for presenting the interface to the user. Those classes will create the bridge between the user and the game system. In the User Interface Subsystem, there will be classes called Menu and Screen Manager, where Menu is the first interaction the user will come across. After the choices user makes, these options will be passed to the Game Logic subsystem.



In the logic layer which is the second layer of the subsystem, choices that user made in the first layer will be evaluated in the Game Logic subsystem. For example, if the user chooses the “Start Game” option, it will be constructed by the Game Masterclass. Also in this layer, Game Logic is in full association with Upgrade Manager and Level Manager subsystem. Level Manager is responsible for creating the layouts of the levels, then passing the created level to the Game Logic. This subsystem will update the information that passed to the Game Logic continuously due to the results of game objects.

Relation Between Layer 1 and Layer 2



In the data tier, the third layer of the decomposition, utility classes will operate game entities and screen elements. Game Entities package contains the classes that are responsible for storing relevant information about objects in the game. Game Screen Elements, on the other hand, handles the In-Game User Interface that changes according to the actions occurring in the Game.

Relation Between Layer 2 and Layer 3

## **Hardware / Software Mapping**

Our game will require the Java Runtime Environment to be executed since it will be developed in Java. A keyboard and a mouse are required to play the game. In terms of graphical requirements of the program, we are planning to use Slick 2D graphics library. Average computer standards will be enough to handle the game.

## **Persistent Data Management**

Game data will be stored in the user’s local hard drive. Our game does not require any database system since the data that is used in the game need to be accessed in real time. Thus, all the necessary files and data will load onto the memory.

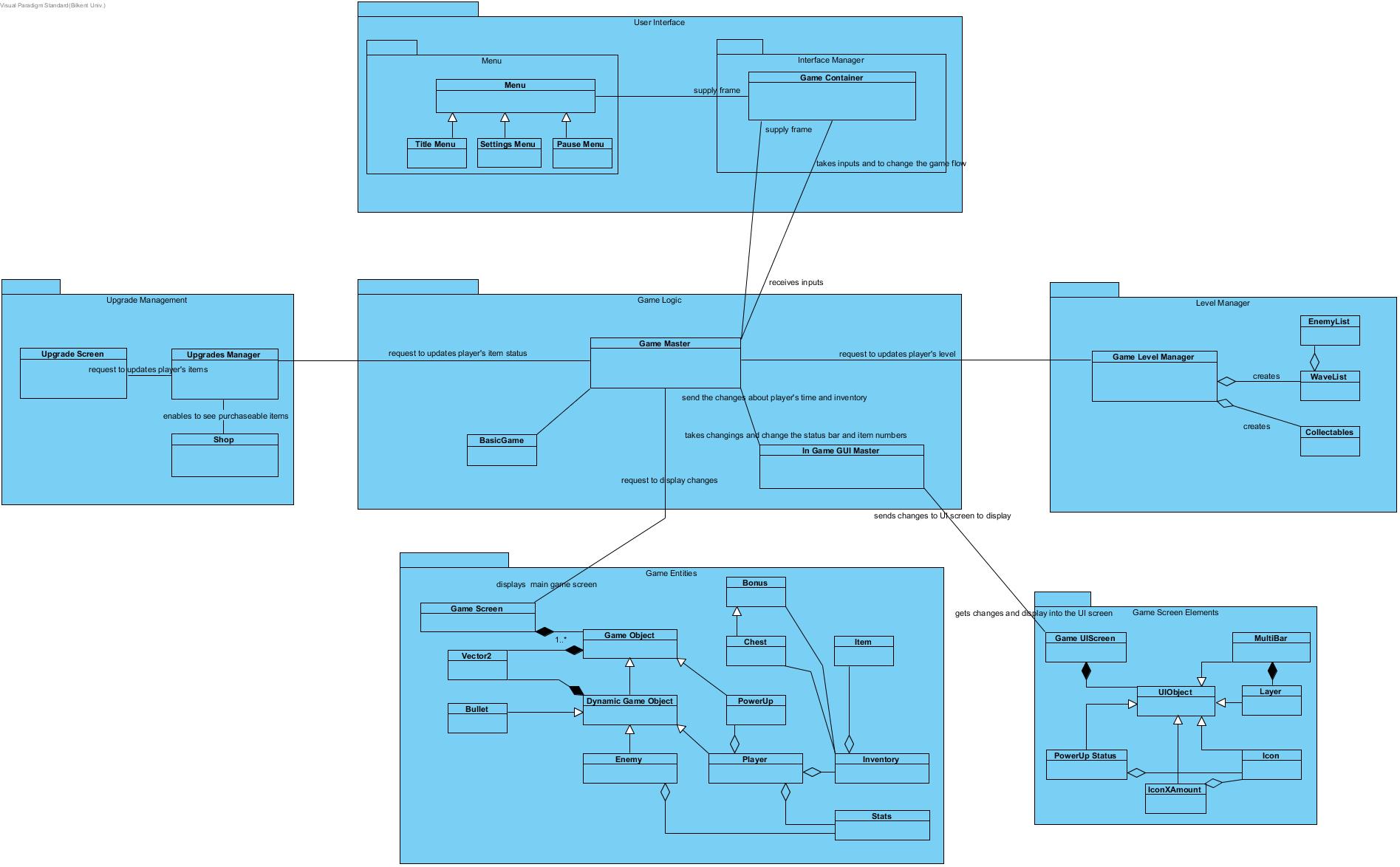
## **Boundary Condition**

The game will give an error if the file is corrupted and will delete its content. The game will return to the main menu if all lives of the player are gone. The game has a finite number of levels so if the player will be able to complete the game will return to the main menu again.

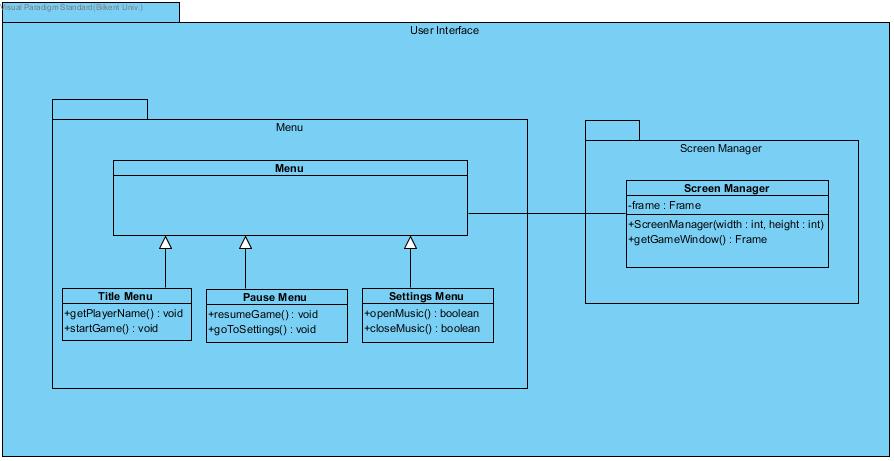
# **Subsystem Services**

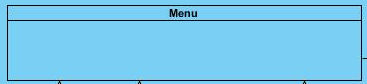
## **Detailed Object Design**

The overall class diagram will provide a better understanding of the subsystem and classes inside the packages. With the help of this diagram, it will be easier to comprehend the design of our project.



## **User Interface Subsystem**



**Menu Class  
,**

This class organizes Title Menu Class, Pause Menu Class, and Settings Menu Class. Each individual menu has the same contents with the “Menu” class, so having a parent class for other menus makes hierarchical system easier to understand.

### **Title Menu Class**

****

A simple title screen to start the game and let the player enter his/her name.

***Methods:*public void setPlayerName(String name):** This method gets the player’s name or nick-name. It takes a parameter ‘name’ and passes it to the player object.**public void startGame():** This method starts the game and opens the main game screen. It does not return anything.

### **Pause Menu Class**

### A simple pause menu that can call settings or resumes the game.

### ***Methods:***

### **public void resumeGame():** Calls the Game master’s resumeGame() function of GameMaster. **public void goToSettings():** This method displays the setting menu to the user. It does not return anything.

**Setting Menu Class**

****

Settings menu allows user to open of close the music sound.

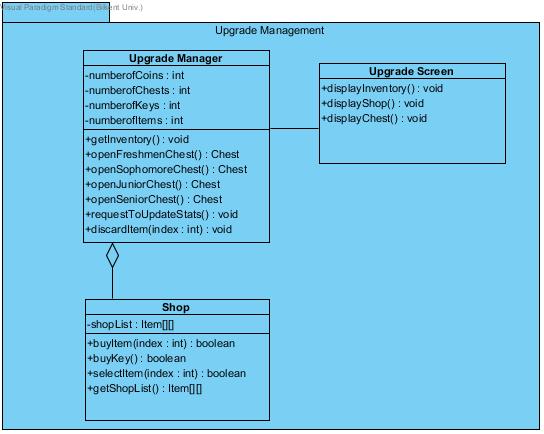
***Attributes:***

**private Boolean playingMusic:** An indicator variable to hold if the game music is currently played or not. ***Methods:*   
public boolean toggleMusic():** This method starts playing the game music if it’s turned off or stops playing the music if its turned on. Returns true if music starts returns false if it stops. Sets the playingMusic accordingly.

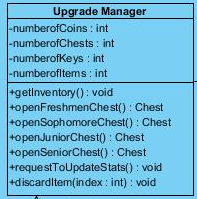
**Game Container Class**Library provided class that provides frames and and input adapters.

**3.3 Upgrade Management Subsystem**

Upgrade Management Subsystem is one of the major subsystems in the design process. This subsystem includes Upgrade Manager, Upgrade Screen, and Shop classes. These classes have following functions;

* Upgrade Manager class handles the situation of player’s items.
* Upgrade Screen class displays the changes into the main screen.
* Shop class handles the purchase and removal of the items

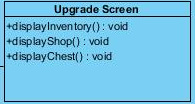
**Upgrade Manager Class**

****

Upgrade Manager class is for the game upgrade system. After each level, user enters an upgrade phase and this class manages that phase by holding values such as credits, number of chests, keys and items player has. Opening chests, stat upgrading and inventory management is done by this class as well. ***Attributes:*   
private int credits**: It keeps the coin number that user has.  
**private intnumberOfChests:** It keeps the chest number that user has.  
**private intnumberOfKeys**: It keeps the key number that user has.

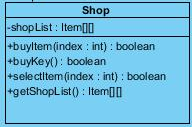
***Methods:*public Inventory getInventory():** This method gets player’s inventory object.**public Item openFreshmenChest():**This method opens freshmen type of chest. To open freshmen chest, it requires 1 key. It has 90% chance of giving a standard tier item, 7% chance of giving a rare tier item and 3% chance of giving an ultra-rare tier item. Therefore, according to these possibilities, this method fetches the item and returns it. Also decrements the key and the chest amounts from player’s inventory accordingly.**public Item openSophomoreChest():**This method opens a sophomore type of chest. To open the sophomore chest, it requires 2 keys. It has 50% chance of giving a standard tier item, 30% chance of giving a rare tier item and 20% chance of giving an ultra-rare tier item. Therefore, according to these possibilities, this method gives the item and returns it. Also decrements the key and the chest amounts from player’s inventory accordingly.**public Item openJuniorChest():**This method opens a junior type of chest. To open the junior chest, it requires 3 keys. It has 30% chance of giving a standard tier item, 35% chance of giving a rare tier item, 25% chance of giving an ultra-rare tier item and 10% chance of giving a “hacker” tier item. Therefore, according to these possibilities, this method gives the item and returns it. Also decrements the key and the chest amounts from player’s inventory accordingly.**public Item openSeniorChest():**This method opens ajunior type of chest. To open thejunior chest, it requires 3 key. It has 18% chance of giving a standard tier item, 25% chance of giving a rare tier item, 32% chance of giving an ultra-rare tier item and 25% chance of giving a “hacker” tier item. Therefore, according to these possibilities, this method gives the item and returns it. Also decrements the key and the chest amounts from player’s inventory accordingly.**public void requestToUpdateStats():**This method sends an update request to game master in order to update the player’s stats or game flow. It does not return anything.**public boolean discardItem(int index):**This method removes the item that is given in the index of the player’s inventory and returns true if removal is successful or false if it fails.

**Upgrade Screen Class**

****

This class is basically for displaying the upgrade phase to the player. Player should be able to see his/her inventory, the shop and chests to be able to tell what to buy or discard or which chest to open or not. ***Methods:*public void displayInventory():**This method displays the inventory of the player on the screen by iterating over the list of items, number of chests and keys. It does not return anything.**public void displayShop():** This method displays the shop on the screen in order to show the purchasable items from the shop by iterating over the shop’s item list. It does not return anything.

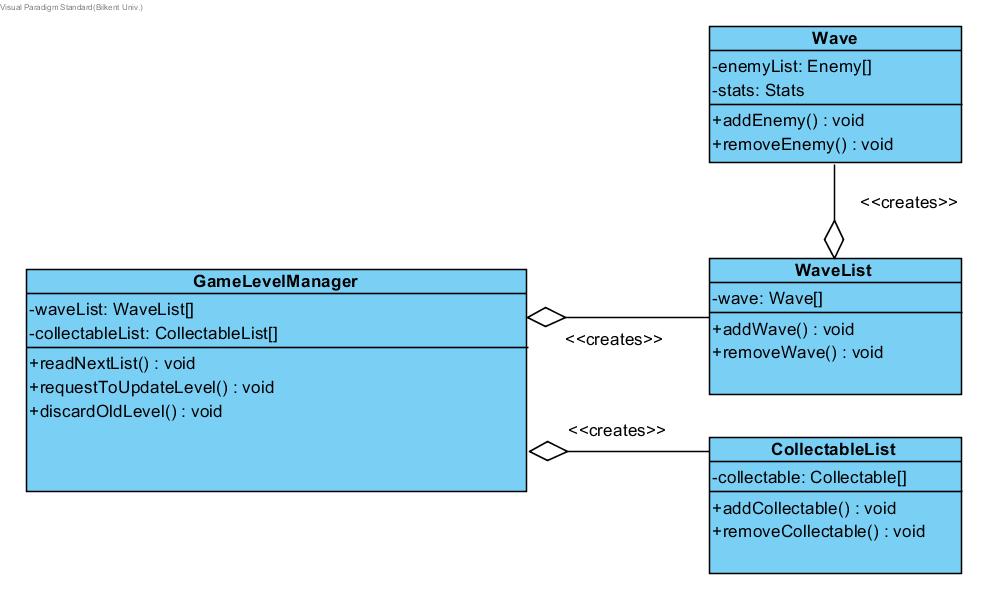
**Shop Class**

****

Shop class has an attribute, a shop list which is an array of items that are sold in the shop. With this class, player will be able to do the operations that should be done in a shop system; buy, select or get the item list of the shop. ***Attributes:*   
private Item[] shopList:** It keeps the shop items into the one-dimensional array.

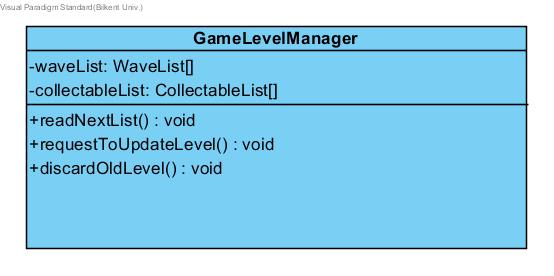
***Methods:*public boolean buyItem(index: item**): This method buys the item and it returns true if the item is successfully bought or false if the item is not bought.**public boolean buyKey():**This method buys the key in order to open a chest and it returns true if the key is successfully bought or false if the key is not bought.**public Item[] getShopList():**This method gets the shop item’s list and returns them.

## **3.4 Level Manager Subsystem**



Level Manager subsystem is used to create each level that player encounters. Every level has different components and elements which are being read from a text file and integrated into levels. Each class of Level Manager subsystem will be explained in detail in this section.

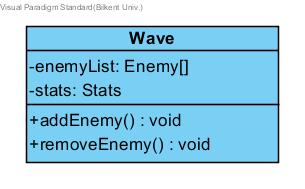
**GameLevelManager Class**

****

Reads the level’s or game specification from a text file then constructs a waveList that contains waves which contains lists of enemies. ***Attributes:*private WaveList[] waveList:** it is an instance of WaveList class and used for connecting waveList class with the GameLevelManager. **private CollectableList[] collectableList:** it is an instance of CollectableList class and used for connecting CollectableList class with the GameLevelManager.

***Methods:*public void readNextList():** This method is used for reading from a text file to get the lists of objects. It used Java’s FileReader method.**public void requestToUpdateLevel():**This method is for requesting to update the level from the GameMaster class in the GameLogic subsystem.**public void discardOldLevel():** This method discards the level when the user completes of fails to complete a level.

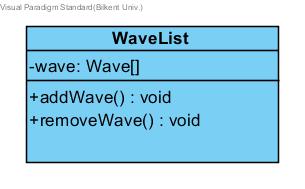
**Wave Class**



Holds a list of enemies that is specified by GameLevelManager. Waves hold an array of Enemy objects and their stats. Enemy objects can be added or removed from waves.  
***Attributes:***   
**private Enemy[] enemyList:** This attribute is a list of Enemy objects.  
**private Stats stats:** This attribute is used to hold the stats of each enemy object.

***Methods:*public void addEnemy():** Adds an enemy object to the list.**public void removeEnemy():** Removes an enemy object from the list.

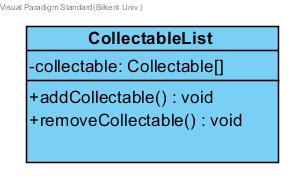
**WaveList Class**



Holds list of Waves, which was the previous class explained.  
***Attributes:***   
**private Wave[] wave:** This attribute is a list of Wave objects.

***Methods:*public void addWave():** Adds an Enemy object to the list.**public void removeWave():** Removes an Enemy object from the list.

**CollectableList Class**

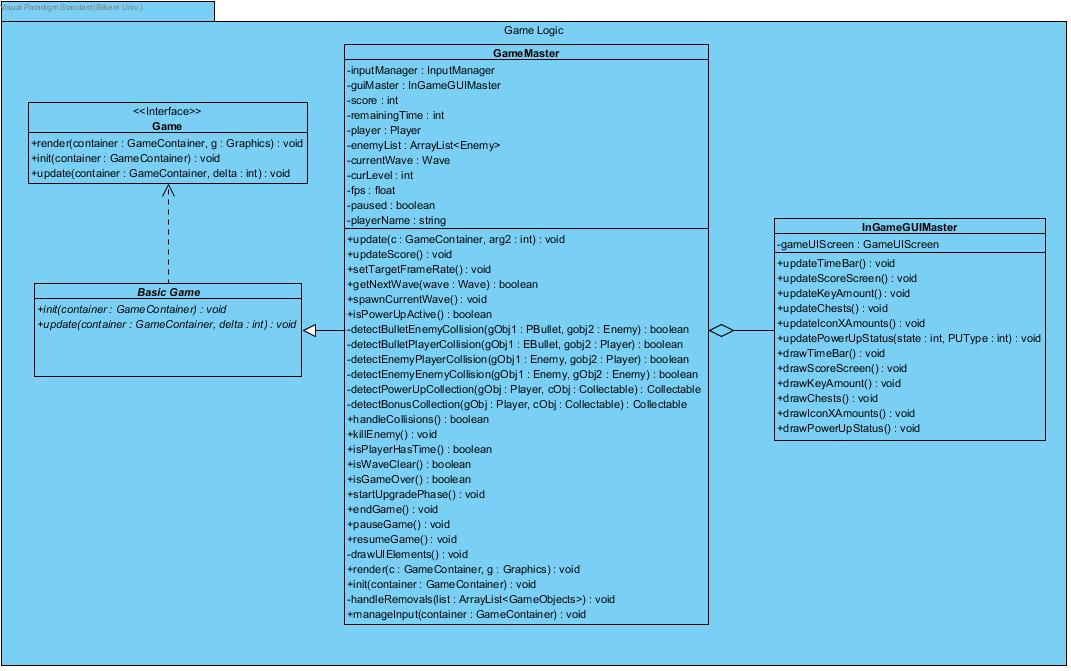


Holds the collectables that will be instantiated during game play by GameMaster. ***Attributes:*private Collectable[] collectibles:** This attribute is a list of Collectable objects.

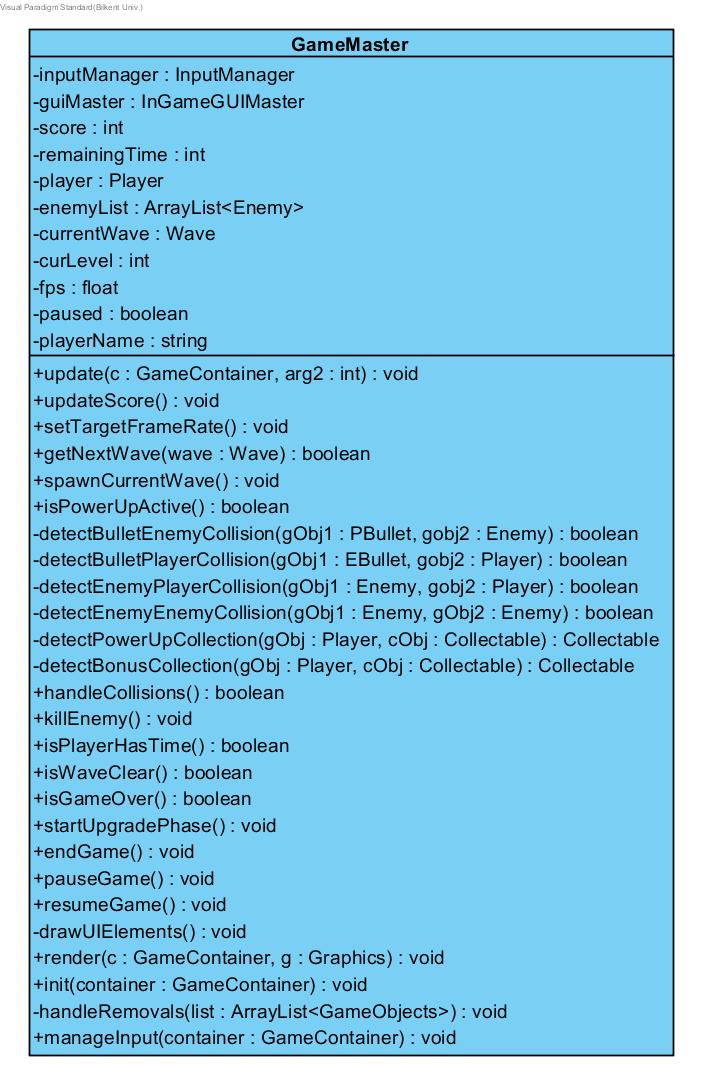
***Methods:*public void addCollectable():** Adds a collectible object to the list.**public void removeCollectable(int: index):** Removes a collectible object from the list in the given index.

**Public void createCollectableList():** creates and assigns the list of Collectables via random selection. The resulting list is assigned to collectables.

## **Game Logic Subsystem**



In the Game Logic subsystem, our controller objects are grouped together to manage the actual game dynamics and game logic. We have GameMaster, InGameGUIMaster and BasicGame. These classes are explained in detail, in this section. The base idea is to have a **centralised control**.

**GameMaster Class**

GameMaster class is the control object of the Game Logic subsystem, it performs the proper operations according to the requests that came from User Interface subsystem, and also this class runs the game in a loop. This class implements **centralized control**. (that’s why it has so many methods.)

***Attributes:*private InputManager inputManager:** this attribute is used for detecting user actions in the game.**private InGameGUIMaster guiMaster:** this attribute is used for updating graphical user interface on the game screen.**private int score:** it is used for player’s score to represent the success in the game.**private int remainingTime:** it is used for determining the game time representing both real-time and player’s health.**private Player player:** this attribute initializes a player object to use in the game.**private ArrayList<Enemy>enemyLists:** this attribute initialize enemy list taking from the level subsystem.**private Wave currentWave:** this attribute initialize a wave of the enemy by taking the enemy list when the player killed them**private int curLevel:** this attribute holds the which level is called.**private float fps:** this attribute takes the frame per second to determine the movements of game objects.**private boolean pause**: this attribute is used for whether the game is paused or not.

**private String playerName:** holds the name that system acquired from the player in the title screen.

***Methods:***

**public void update(c: GameContainer, arg2, int):** runs a loop in which all system components is updated continuously until the breakpoints (such as pause, game over, or finish game). Checks if the wave is cleared, using isWaveClear method and calls getNextWave and spawnCurrentWave accordingly.

**public void updateScore():** updates the score of the player if it successfully kills an enemy.**public void setTargetFrameRate():** holds the average frame per second until the 30.**public Boolean getNextWave(wave:Wave):** calls the new wave of enemies when all enemies in the previous wave are killed by the player.**public void spawnCurrentWave() :** if getNextWave is true, this method creates current wave enemies.**public Boolean isPowerUpActive():**checks whether the player has activated any power-ups or not.**public Boolean detectBulletEnemyCollision(gObj1:PBullet, gobj2:Enemy):**detects the Player Bullet and Enemy collision, in this type collision enemy takes damage and its health decreases. After the collision, bullet disappears.**public Boolean detectBulletPlayerCollision(gObj1:EBullet, gobj2:Player):**detects the Enemy Bullet and Player collision, in this type collision player takes damage and its health decreases. After the collision, bullet disappears.**public Boolean detectEnemyPlayerCollision(gObj1: Enemy, gobj2:Player):**detects the Enemy and Player collision, in this type collision both enemy and player takes damage and their health decreases.  
**public Boolean detectEnemyEnemyCollision(gObj1: Enemy, gObj2:Enemy):**detects the Enemy and Enemy collision, in this type collision both enemies hits and removes each other without any damages.**public Collectable detectPowerUpCollection(gObj : Player, cObj : Collectable) :** detects the Player and Power-ups collision. After the collision, theplayer takes the power up and isPowerUpActiveis returned true. And also, the power up disappears on the game arena then it goes to the power-up box on the corner of the game screen to be used later in the game.**public Collectable detectBonusCollection(gObj : Player, cObj : Collectable) :** detects the Player and collectable item(key, chest or coin) collisions. After the collision player takes the items and one of these InGameGUIMsster methods (updateKeyAmount, updateChests, updateIconXAmounts) is called and this item is added player’s inventory. And also, the item disappears on the game arena.**public Boolean handleCollision():** checks any collision, if there is any call the detectCollision methods and return true, else return false.

**public void killEnemy():** changes the death-flag to true if the enemy has no life.**public Boolean isPlayerHasTime():**checks whether the player has the time or not. If the time is over then returns false, otherwise, returns true.

**public boolean isWaveClear():** checks if the current wave is cleared by the player.

**public boolean isGameOver():** checks the player has enough time (if isPlayerHasTime is true) or not. This method returns true if isPlayerHasTime is false, otherwise, returns false.

**public void startUpgradePhase():**before each level, this method is calledand upgrade screen appears.

**public void endGame():** exits the game when the player presses the “exit the game” button.

**public void pauseGame():** pauses the game and call the pause screen when the player presses the pause button.

**public void resumeGame():** resumes the game and closes pause screen.

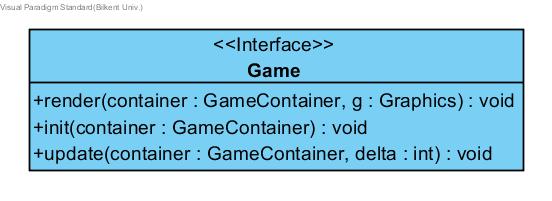
**public void drawUIElements():**gives the command to draw UI elements to inGameGUIMaster

**public void render (c:GameContainer, g: Graphics):** draws game objects (player, enemies, bullets, collectables) while iterating over the certain game object lists.

**public**  **void init(container: GameContainer):** constructs the player, initializes enemyList, bulletList, collectableList and sets current wave using getNextWave and spawnCurrentWave methods.**public void handleRemovals(list: ArrayList<GameObjects>):** checks death-flags of all enemies and removes them if the flags are true.

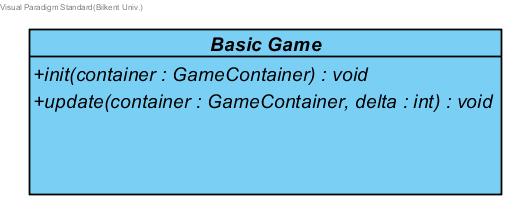
**public void manageInput(container:GameContainer):**

**Game Interface**

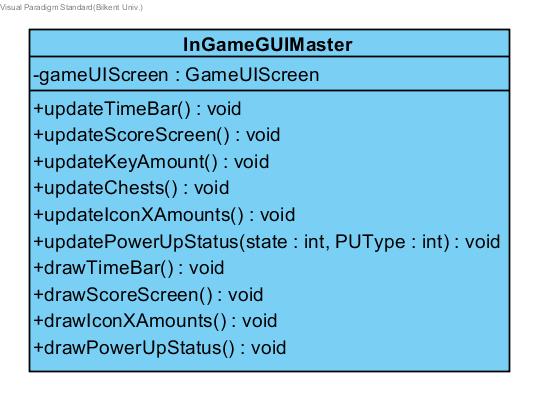
******

A library provided interface. The main game interface that should be implemented by any game being developed using the container system. There will be some utility type sub-classes as development continues. [[1]](#footnote-1)

**BasicGame Class**

  
A library provided class. BasicGame class is a basic implementation of a game to take out the boring bits.[[2]](#footnote-2)

**InGameGUIMaster Class**

***  
Attributes:*private GameUIScreengameUIScreen:** User Interface screen for the in-game GUI. It allows the connection between the Game Screen Elements package’s GameUIScreen class and Game Logic package’s InGameGUIMaster.

***Methods:*public void updateTimeBar():** Updates the time bar of the player depending on the events occurring in the Game Master.**public void updateScoreScreen():** Updates the score screen in the game depending on the player’s score collection taken from the Game Master.**public void updateKeyAmount():** Updates a number of keys in the game depending on the Level Manager’s Collectables class.**public void updateChests():** Updates the chests in the game depending on the Level Manager’s Collectables class.**public void updateIconXAmounts():** Updates the number of Items (Number X “Icon of Object”) in the user’s inventory.**public void updatePowerUpStatus(state:int, PUType:int):**Updates the current PowerUp icon on the screen by taking its states and types.

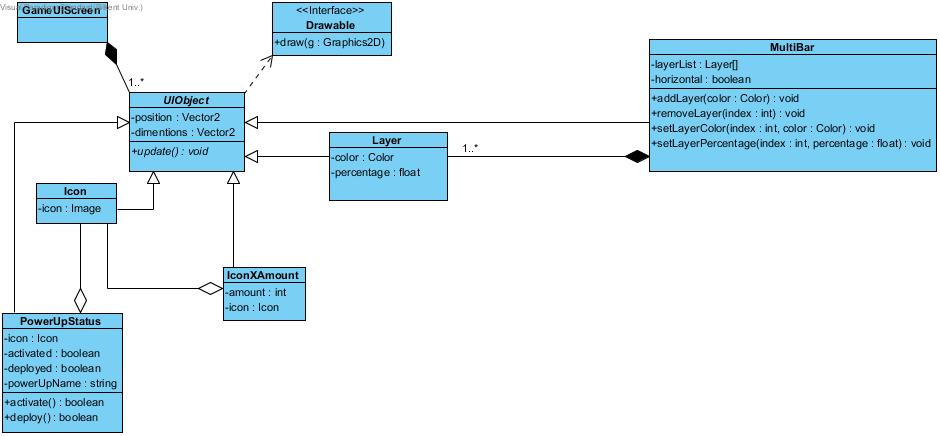
**public void drawTimeBar():** draws the time bar using graphics.

**public void drawScoreScreen():** draws the score screen using graphics.

**public void drawIconXAmounts():** draws the number of items (number X) in the user’s inventory.

**public void drawPowerUpStatus():** draws the powerup status on the right bottom corner of the screen.

## **Game Screen Elements Subsystem**

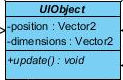


Game Screen Elements subsystem is used for creating in game user interface elements and updating them accordingly if necessary.

**GameUIScreen Class**

****It holds the list of UI Objects.

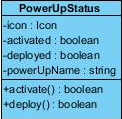
**UIObject Class**

**  
*Attributes:*  
private Vector2 position**: It keeps the position of the objects. **private Vector2 dimensions**: It keeps the size of the objects.  
***Methods:*public void update()**:

**Icon Class**

**  
*Attributes:*  
private Image icon**: It holds an image of the icons.

**PowerUpStatus Class**

**  
*Attributes:*  
private Icon icon**: It holds an icon of the power-ups. **private boolean activated**: It holds whether the power-up is activated or not. **private boolean deployed**: It holds whether the power-up is activated or not. **private string powerUpName**: It holds power up’s name.

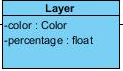
***Methods:*public booleanactivate():**This method activates the power up and it returns whether the power-up is activated or not.**public boolean deploy():**This method deploys the power up and it returns whether the power-up is deployed or not.

**IconXAmount Class**

****

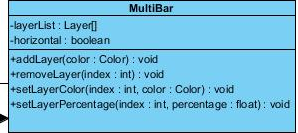
IconXAmount is the class that is used to display icon and the amount of a specific inventory statictic. That is, it shows how many of one specific item is in the possession of the player. It is another user interface element of the game. ***Attributes:*  
private Icon icon:** It holds an icon the items. **private int amount:** It holds an amount of the items.

**Layer Class**

****

Layer class is for the multiBar component of the game. This is simply a rectangular graphical box that scales itself according to percentage.  ***Attributes:*  
private Color color:** It holds a color of the layer. **private float percentage:** It holds a percentage of the layer.

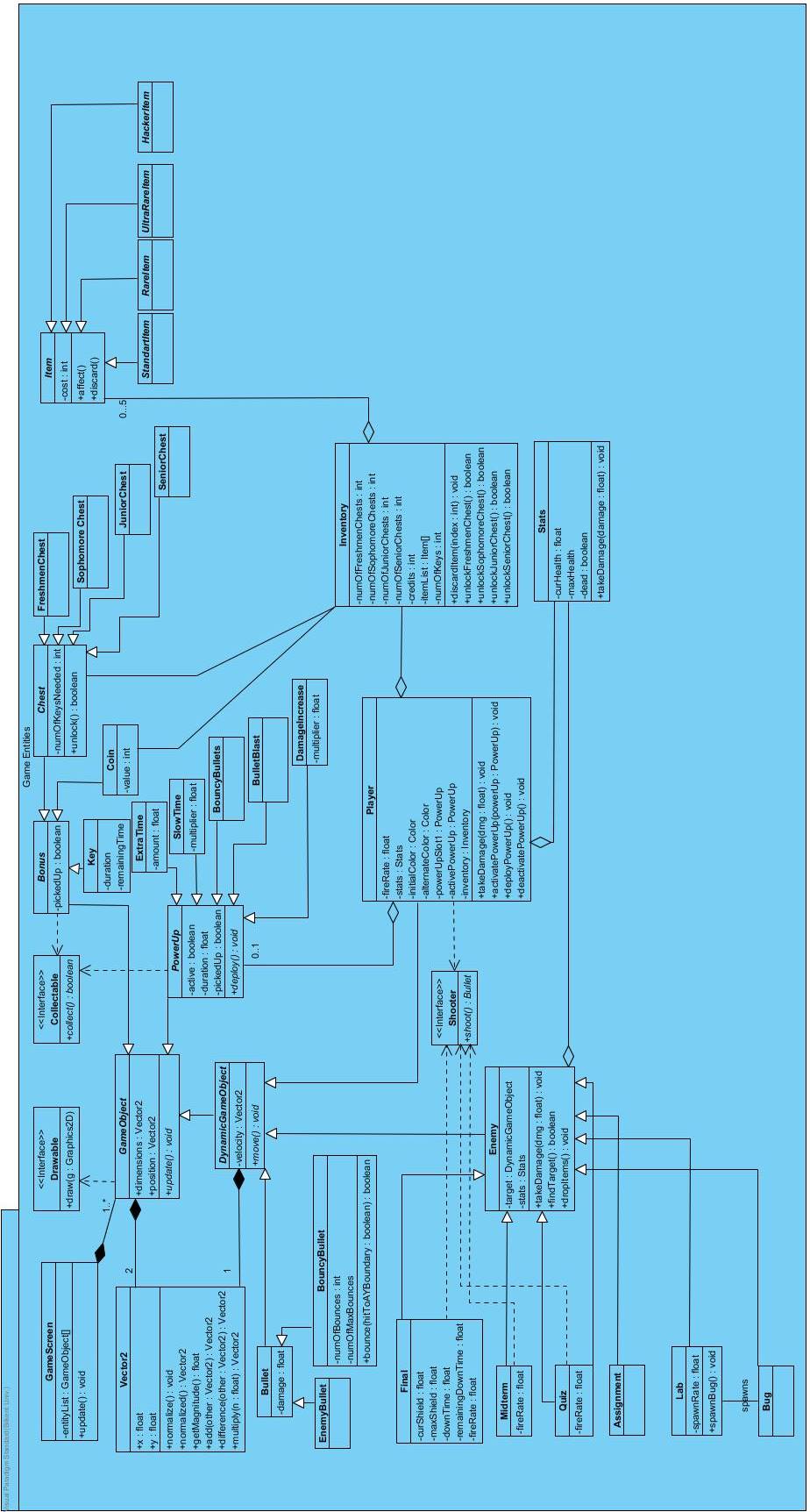
**Multibar Class**

****

Multibar is a multi-purpose, multi-layered bar that displays health like statistics of an enemy or the player. Usually used for linearly decreasing or increasing statistics.  ***Attributes:*  
private Layer[] layerList:**It holds a list of the layer. **private boolean horizontal**: It holds the direction of the layer.

***Methods:*public void addLayer(color:Color):** This method adds the new layer and it does not return anything.**public void removeLayer(Index:int):** This method removes the existing layer and it does not return anything.**public void setLayerColor(Index:int, color:Color**): This method sets the color of the existing layer and it does not return anything.**public float setLayerPercentage(Index:int, percentage: float):** This method sets the percentage of the existing layer and it returns the percentage.

## **Game Entities Package**



**GameScreen Class** ***Attributes:***  
**private ArrayList<GameObject>entityList:** Holds the entity(GameObjects) objects of the game.  
***Methods:*public void update():** calls the update method of every entity object.

**Vector2f\***

A library provided representation of two-dimensional vector[[3]](#footnote-3)

**Drawable (interface)** ***Methods:*public void draw():**overridden by gameObjects a simple draw method to visualize objects.

**GameObject Class (abstract)** ***Attributes:*public Vector2 dimensions:** rectangle sizes of a GameObject.**public Vector2 position: the**center position of a GameObject.

***Methods:***  
**public void update():** updates the information accordingly to the internal commands.

**DynamicGameObject (Abstract)**  
***Attributes:***  
**private Vector2 velocity:** holds 2D velocity of a moving object.

***Methods:*private void move():** updates the location by velocity.

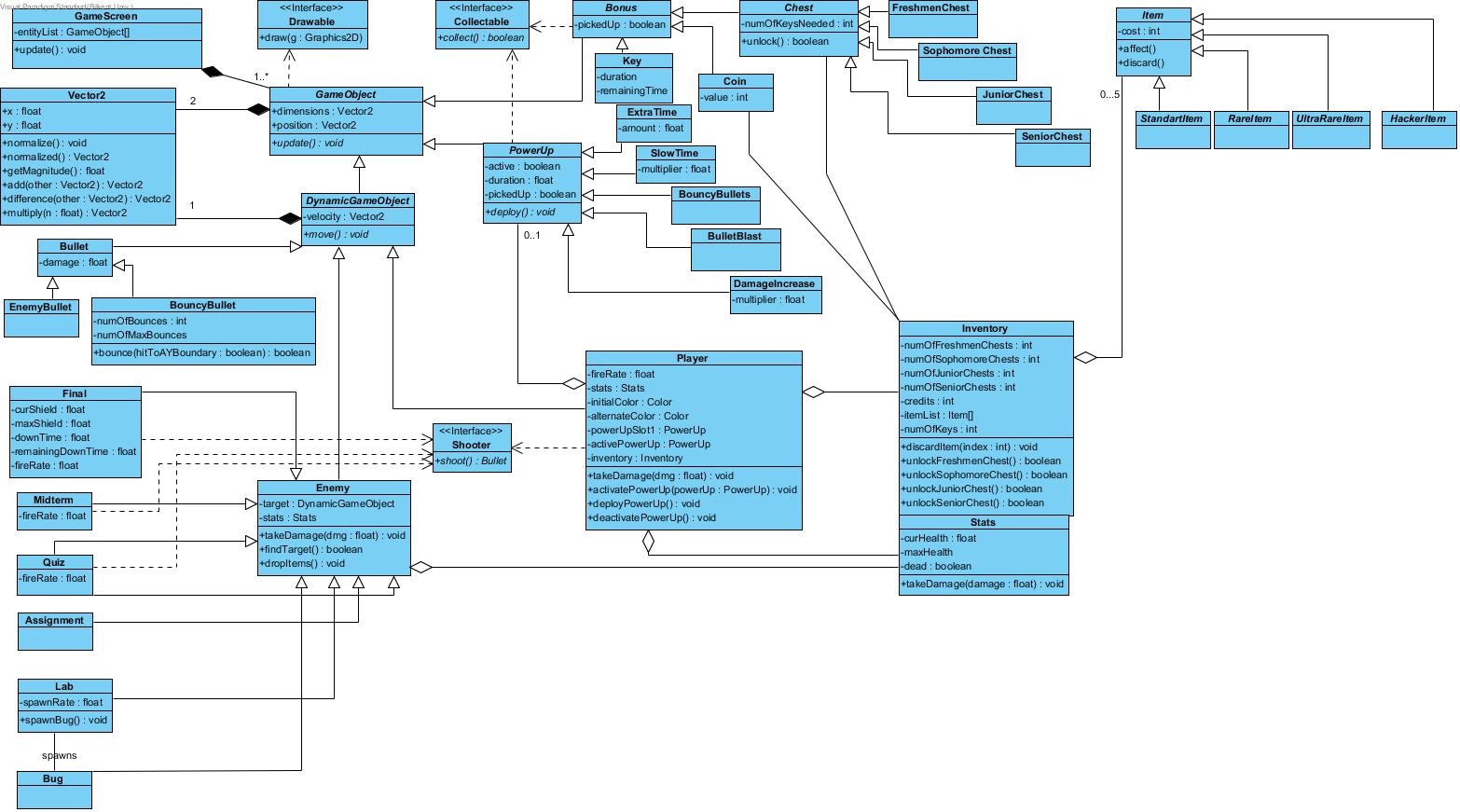
**Stats Class**

**  
*Attributes:*  
private float curHealth:  
private float maxHealth:**These are all values for health stats. **private booleandead():** flag value for removal from the gameScreen.

***Methods:*  
public void takeDamage(float damage):** decrements the curHealth by damage. Checks dead flag true if curHealt<0.

**Enemy Class  
*Attributes:*  
private DynamicGameObject target:** the target of the enemy which will be attacked and possibly damaged. **private Stats stats:** stats objects for the enemy.

***Methods:*  
public void takeDamage(float dmg):** the amount of Damage that will be applied to Enemy if enemy is killed it will be marked for removal.(calls stats.takeDamage(dmg)) **public booleanfindTarget():** Enemy seeks target. **public void dropItems():** Enemy drops items before getting removed because of death.

**Inventory Class**

***Attributes:*  
private intnumOfFreshmenChests:** the number of chests of type ‘freshmen’ which are collected and not opened. **private intnumOfSophomoreChests:** the number of chests of type ‘sophomore’ which are collected and not opened. **private intnumOfJuniorChests:** the number of chests of type ‘junior’ which are collected and not opened. **private intnumOfSeniorChests:** the number of chests of type ‘senior’ which are collected and not opened. **private intcredits:** number of coins collected. **private Item[] itemList:** 5 items that are owned by theplayer. **Private intnumOfKeys:** number of keys player has in the inventory

***Methods:*  
public void discardItem(intindex):** removes theitemat the given index. **public booleanFreshmenChest():**instantiates and opens a freshmenChest. **public booleanSophomoreChest():**instantiates and opens a SophomoreChest. **public booleanJuniorChest():**instantiates and opens a JuniorChest. **public booleanSeniorChest():** instantiates and opens a SeniorChest.

**Shooter(interface)**

***Methods:*  
public Bullet shoot():** An abstract shoot method for shooting player and enemies.

**Player Class  
 *Attributes:*  
private float fireRate:** To control the density of bullets. **private Stats stats:** players stats as a Stats object. **private ColorinitialColor:** color variable supplied for adrawn method to indicate player objects state changes. **private ColoralternateColor:** alternating color variable from the initial color **private PowerUp powerUpSlot1:** holds the primary and only(unless there is an item that specifies an extra slot) power up. **private PowerUpactivePowerUp:**holds a reference to an active power-up since there might be another one supplied by an item. **private Inventory inventory:** player’s inventory that holds items and num of chests keys credits…  
***Methods:*  
public void takeDamage(float dmg):** the amount of Damage that will be applied to Player if player is killed it will be marked for removal.(calls stats.takeDamage(dmg)) **public void activatePowerUp(PowerUppowerUp):** activates the powerup specified in the parameter. **public void deployPowerUp():** deploys the activated powerUp.(calls activePowerUp.deploy()). **public void deactivatePowerUp():** deactivates the active power up.

**Bullet Class**



***Attributes:*private float damage:** the amount of damage to be applied to the collided enemy.

**BouncyBullet Class**



***Attributes:*private intnumOfBounces:** counter for collisions with game Arena borders.**private intnumOfMaxBounces:** max number of bounces for a bullet.

***Methods:*public boolean bounce (booleanhitToAYBoundary):** First checks if numOfBounces<=macNumOfBounces then it inverts the bouncy bullet’s velocity’s x value if theparameter is false else it inverts y value.

**Final Class**

***Attributes:*private float curShield:** holds the current value of the shield.**private float maxShield:**holds the maximum value of the shield.**private float downTime:**holds the initial value to be counted down from before the shield is back up.**private float remainingDownTime:**holds the countdown time for shields to be back up.**private float fireRate:** To control the density of bullets.

**Midterm Class**



***Attributes:*private float fireRate:** To control the density of bullets.

**Quiz Class**



***Attributes:*private float fireRate:** To control the density of bullets.

**Assignment Class**



**Lab Class**

***Attributes:*private float spawnRate:** To control the density of “bug” objects spawned by thelab.

***Methods:*public spawnBug():** spawns bugs to attack player

**Bug Class**

**Chest Class**

**  
*Attributes:***Private intnumOfKeysNeed: It holds a number of the keys needed.

***Methods:***Public booleanunlock():This method unlocks the chest and it returns whether the chest is unlocked or not.

**Freshmen Chest Class**

****This class constructs the freshmen chest object.

**Sophomore Chest Class**

****This class constructs the sophomorechest object.

**Junior Chest Class**

****This class constructs the junior chest object.

**Senior Chest Class**

****This class constructs the senior chest object.

**Coin Class**

**  
*Attributes:***

**Private int value:** It holds a number of coins.

**Item Class**

****

***Attributes:***

**Private int cost**:It holds the cost of the item.

***Methods:***

**Public void affect():**It sends the effect to the game master.

**Public void discard():**It removes the item**.**

**Standard Item Class**

****This classconstructs the standard items

**Ultra Rare Item Class**

****This classconstructs the ultra-rare items

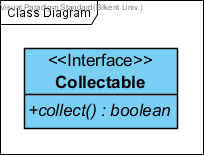
**Rare Item Class**

****This classconstructs the rare items

**Hacker Item Class**

****This classconstructs the hacker items.

**Collectable Interface**

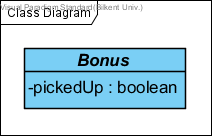


Collectable is an interface for all collectible items in the game. These are Bonus and PowerUps.

***Methods:***

**public Booleancollect():**is called when the player collides any Collectable items in the game.

**Bonus Class**

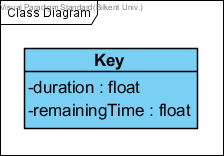


Bonus Class is a parent class and have three children classes are Chest, Coin, and Key.

***Attributes:***

**private booleanpickedUp:**is determine whether the Bonus item is picked up by player or not.

**Key Class**

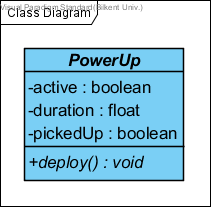


***Attributes:***

**private float duration:**it is specified the duration of existence of akey in the game.

**private float remainingTime:**it is specified the remaining time from the duration of thekey.

**PowerUp Class**



PoverUp Class is a parent class and has five children classes are ExtraTime, SlowTime, BouncyBullets, BulletBlast, and DamageIncrese.

***Attributes:***

**private boolean active:** it specifies whether the power-up is active or not.

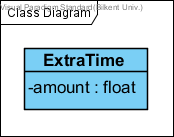
**private float duration:** it specifies the duration of existence of power-up in the game.

**private float pickedUp:** it specifies whether the power-up is picked-up by player or not

***Methods:***

**public void deploy():** this method deploys the power-up when the player collides power-up to use in the game.

**ExtraTime Class**

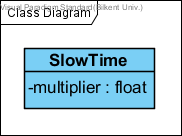


ExtraTime Class adds the extra time to player’s game-time

***Attributes:***

**private float amount:** it specifies the amount of time which is added to theplayer.

**SlowTime Class**

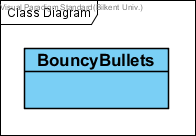


SlowTime Class slows the time for all game objects.

***Attributes:***

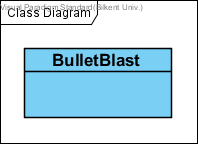
**private float multiplier:** it specifies the ratio of the slowing time.

**BouncyBullets Class**



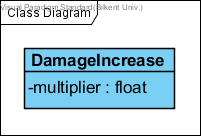
BouncyBullets Class replaces the bullets with the bouncy bullets which bounce off (3 times per bullet) from the borders of the game are instead of simply going out.

**BulletBlast Class**



BulletBlast Class sends out a circular group of bullets originating from the player.

**DamageIncrease Class**



DamageIncrease Class makes Player’s bullets harder for 5 seconds.

***Attributes:***

**private float multiplier:** it specifies the ratio of the increasing damage.

1. Slick2D Game Interface: <http://slick.ninjacave.com/javadoc/org/newdawn/slick/Game.html> [↑](#footnote-ref-1)
2. BasicGame class in Slick2D: <http://slick.ninjacave.com/javadoc/org/newdawn/slick/BasicGame.html> [↑](#footnote-ref-2)
3. In the feedback It was advised us to name our classes “**wisely**” although we switched libraries in between and this particular class is no more our implementation the name is roughly the same(It used to be Vector2 now Vector2f). We also would like to inform you that representing 2D points/Vectors in programming with the name “**Vector2**” is common convention.

   Here are some links to popular Game Engines and Java Game Libraries:

   Unity Engine **Vector2**: <https://docs.unity3d.com/ScriptReference/Vector2.html>

   Unreal Engine F**Vector2D**: <https://docs.unrealengine.com/latest/INT/API/Runtime/Core/Math/FVector2D/>

   Hero Engine(has only 3D vectors) **Vector**3: <http://hewiki.heroengine.com/wiki/Vector>

   Java Light Weight Game Library’s AI**Vector2D**: <https://javadoc.lwjgl.org/>

   Slick 2D’s **Vector2**f: <http://slick.ninjacave.com/javadoc/org/newdawn/slick/geom/Vector2f.html> (our library) [↑](#footnote-ref-3)