 Bilkent University

Object Oriented Software Engineering Project

CS 319 Project: Civilizational Wars

Design Report

Project members:

• Fuad Aghazada \_ 21503691

• Seyfullah Yamanoğlu \_ 21400697

• Bayram Muradov \_ 21503664

• Berk Erzin \_ 21201516

Supervisor: Uğur Doğrusöz

Progress: December 2, 2017

**Contents**

**1. Introduction\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1**

**1.1. Purpose System\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3**

**1.2 Design Goals\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_4**

**1.3 Design Tradeoffs\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 5**

**2. Software Architecture\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_6**

**2.1 Subsystem Decomposition\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_7**

**2.2 Hardware/Software mapping\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 8**

**2.3 Persistent Data Management\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_8**

**2.4 Access Control & Security\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_9**

**2.5 Boundary Conditions\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_10**

**3 Subsystem Services\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_10**

**4 Low-level design\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_12**

**4.1 Primary Class Description\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_12**

**4.2 Final object design\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_30**

**5 Improvements summary\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_32**

**6 Glossary & References\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 33**

Design Report

CS 319 Project: Civilizational Wars

# **1.** **Introduction**

By writing this report as a group we aimed to convey some general and detailed information about the analysis of this game, specifically, functional, non-functional requirements, and system model diagram.

# 1.1. Purpose of the system

This game will be an adventure game, in which there will be different maps to play in order to make the game interesting to the players. Game interface will be easy to adapt for the players within all groups of ages. The game controls will not only be easy to use as default, but also will be modifiable in the settings of the game. In terms of the interests of the players, the game will have 3 different difficulty types: easy, medium, hard - which could help the player to test his/her gaming skills in different types of difficulties. Another purpose behind the designing this kind of game is also experience the players, especially the ones with same generation with us, the nostalgic Atari games like Contra (in terms of map and fighting).

1.2. Design Goals

**Efficiency -** Every player would like to play the games, which are quite responsive and do not have so many performance issues, which are irritating. Considering this fact, we will make our game sufficiently efficient in order to be able to obtain the attraction of the players. So as to achieve this goal, we will try to handle our limited memory properly so that it is minimally used, and minimize the execution time of the main operations, which play the most crucial roles on the proper flowing of the game. For example, in order to check the collisions between the different game objects, we need to create a class, in which we will have some functions, control all the computations done efficiently, otherwise, it could lead to some bugs prevent the game flowing properly.

**User-friendliness -** In order not to make the player confused when he/she launches the game, we need to design the User Interface easy to control and use. In terms of the comfort of player, some settings, like activating music, changing difficulty modes, adjusting controls and so on will be available. However, comfort in the controls and the interface does mean the game be extremely easy that the player will be bored while playing the game.

**Adaptability -** So as to be able to run the game in different platforms, it could be a proper decision to implement the project in Java programming language, which is also advantageous in terms of the object-oriented approach. The only thing is need to launch the game in a platform is to install proper JRE [1] - Java Runtime Environment, which is compatible with almost all widespread operating systems. That is why we decided to choose Java among the languages like C++, C# and other languages which can provide us some different advantages in terms of performance and security.

**Modifiability** - The very first release of the game will contain only 3 different levels which is not sufficiently enough for a player to entertain, since it will finish soon in terms of the time. This is the reason why we may need to add new maps, characters and levels to the game. In order to achieve this, with the advantages of the object-oriented approach, our design and implementation needs to be modified and new features and game states should be added So our project should also have modifiability as a design goal.

**1.3 Design Tradeoffs**

* **Memory vs Programmability**: In Enemy and Character classes, we could have declared an enumeration to decide the type of the enemy or character, and we could process its data and other functionalities inside the if or switch blocks, it firstly seemed very well working and easy to implement. However, after some test, we decided to change it to basic abstraction with character and enemy classes are parent classes but the types of the enemy and character classes extends the parent class. In the first choice we could allocate less memory but now it is more flexible, reliable and programmable.
* **Programmability, Flexibility, Maintainability and Usability**: In the animation of game objects we’ve come up with 2 different types of implementation. In the first one: the whole game object will consist of only one texture and when any animation is played, the whole texture will be animated. This means, all the parts of the game object will be concrete and no weapon class will be needed. Of course, it would be very well looking, but we could have no flexibility at all. In the second choice, we could divide the game objects into the pieces that each piece manages itself, which is more natural and appropriate for OOP. Although the animations would be drastic, we chose the second type of implementation. It provided us with more flexible, maintainable, usable implementation to manage easily, but also we sacrificed the programmability costs in terms of time. After the first iteration, we had discovered that the second choice also let us implement multiplayer in a better way to let players have different weapons and different textures.
* **Manageability**: We were changing the panel through the ScreenManager and we sometimes need to access this class instance from lower level classes. It is not available in OOP, so in Java. We tried to find a solution to this problem and came up with 2 different choices. 1. We can create a PanelChanger interface that has a method to change the current, existing panel and can update the screen manager, so one more interface to let ScreenManager listens the PanelChangers. It is exactly the Observer design pattern. 2. We can make ScreenManager class singleton and by getting the instance in the lower level classes we can change the current panel, which is exactly the Singleton design pattern. We chose second option because we think that it is more appropriate for performance and manageability. At first Observer design pattern looks better but in testing we decided to use Singleton design pattern to manage and perform better.

**2.** **Software Architecture**

2.1. Subsystem Decomposition



Figure 1. Subsystem decomposition

We made up our mind to use 3-tier architecture as an architectural style, because it will not be too complex game system and by using this style, we will be able to make changes on the interface while keeping the core of the system and models the same. Using this architectural style, we can divide our system to mainly 3 subsystems as described in Figure 1: GUI, in which there will be rendering system for the game states (menu, pause, settings and etc.), GameManagement, will contain the game logic, input, data managements, GameObjects, will keep the game entities with their properties. More details for the subsystem are described in Figure 2 below.

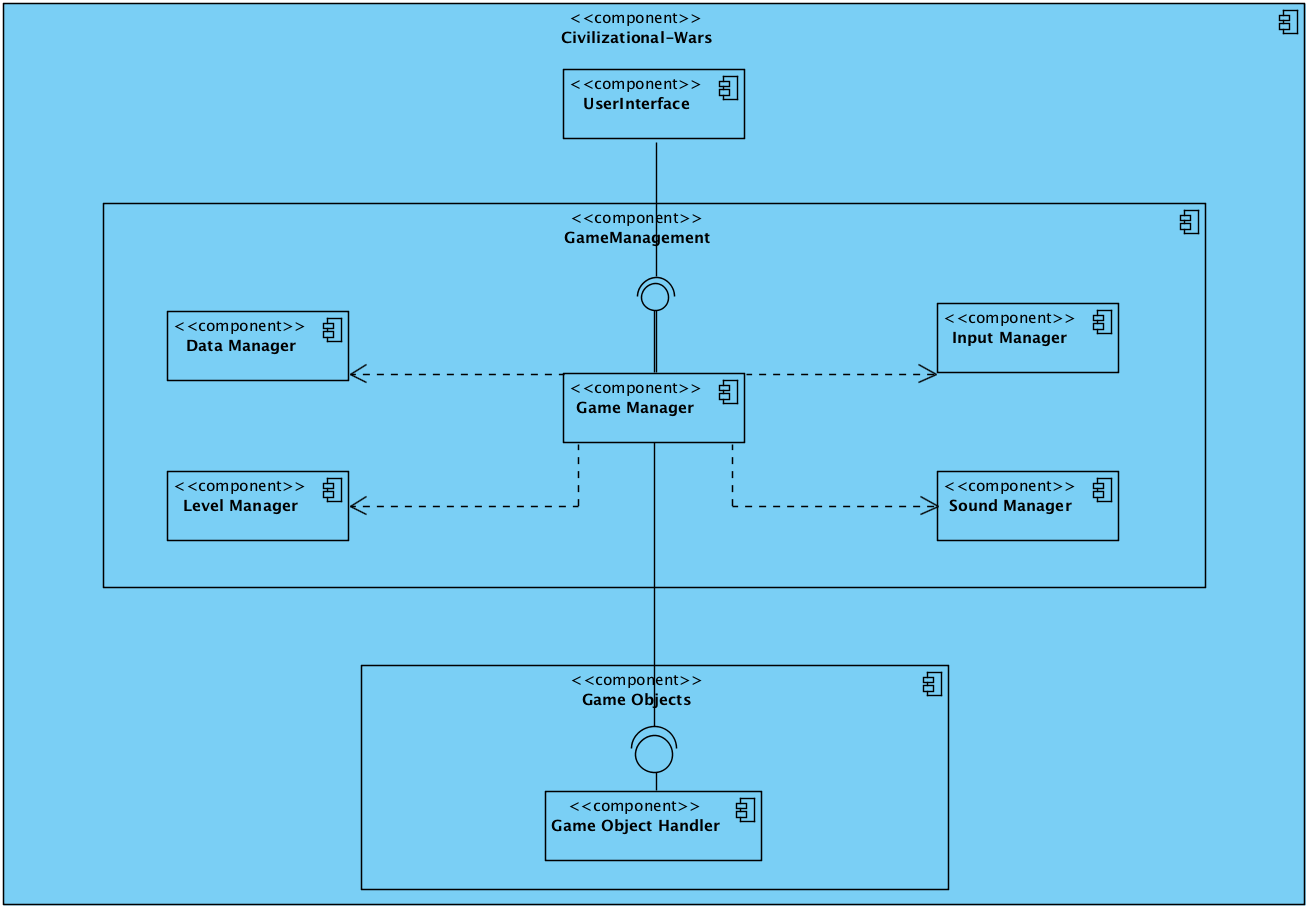


Figure 2. Some details for the subsystems

2.2. Hardware/Software Mapping

As hardware requirements, our game will need a keyboard and mouse in order to create an interaction between the user and the game. The player will use keys from the keyboard to move and fight, and mouse for browsing the game menus. In case of multiplayer, second player will also use the same keyboard with different control keys. In order to run the game, the machine in which the game is launched and played doesn’t need to have extremely high requirements rather than nowadays’ standards since the game that has been developed in Java programming language only requires to have Java Runtime environment to be executed.

2.3. Persistent Data Management

Since our game will have features for saving and loading the game, we will need to save the current time data for game. Our game isn’t going to use a database since the game resources need to be accessible in real time. As a benefit this allows users to change the game by editing the accessible resources. Data for the game will not be very large and will not be concurrently used by different users - the game will be installed to each machine with the necessary files in order to properly run the program. That is the reason why, we decided to use a simple file system so that it will save the data for the game objects, maps and the levels in proper formats to some files, so we can access saved data by loading the game from those files.

2.4. Access Control & Security

Since our game will not be using any database it will not have a feature of login authentication, it will use a simple file system to store the necessary data for the game. Nevertheless, we will need proper access modifiers for the data of the game objects so that it will not affect the security of data flow of the program. Additionally, we need to protect the file, where the saved game data is stored. In order, not to be modified manually by the players, we will need to encrypt the game data in a way that the data is not obviously readable by the users.

Object data serialization would also be another solution for this security problem in a way that, when the game is saved, the necessary data, which will be saved, could be represented as sequence of bytes, which contains every information about the object - object type, data type and etc. After writing the serialized data into the file, when the game is loaded it will get the data by deserializing it. In Java Virtual Machine, the data can be serialized in a platform, and can be deserialized in another platform, which again makes the Java programming language much more adaptable.

2.5. Boundary Conditions

When the game is launched, necessary data will be initialized and will be waiting for the user input. When the gameplay starts, when the game is loaded, it will initialize the game object with the obtained data coming from the file. However, if there is some format problems in the file, which have been created by the modifications of the players, the program will not allow the user to load the game by throwing an exception. If the data that has been saved during the gameplay is corrupted system will not allow user to access that data and will delete the corrupted data. When all the levels have been passed, the player will be congratulated and brought to the main menu to create a chance for trying the levels in different difficulty modes.

**3.** **Subsystem services**

User Interface subsystem

Graphics of the game will be visualized in User Interface subsystem. This subsystem will represent the View part of the MVC design pattern.

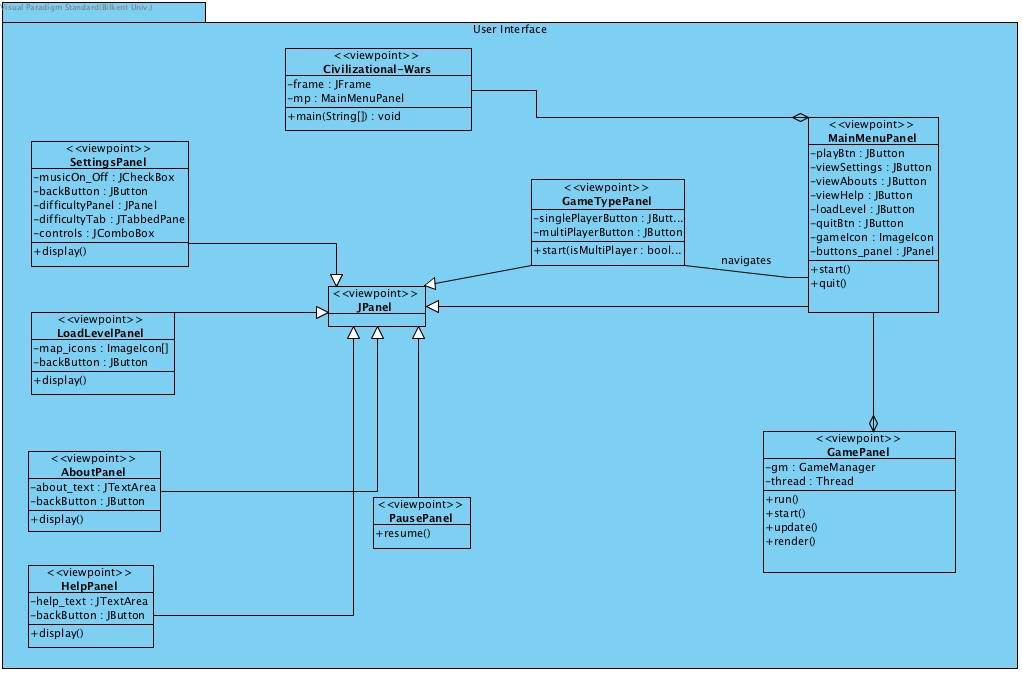


Figure 3. User Interface subsystem

GameManagement subsystem

Inputs and game data will be managed in this subsystem. This subsystem will represent the Controller part of the MVC design pattern.

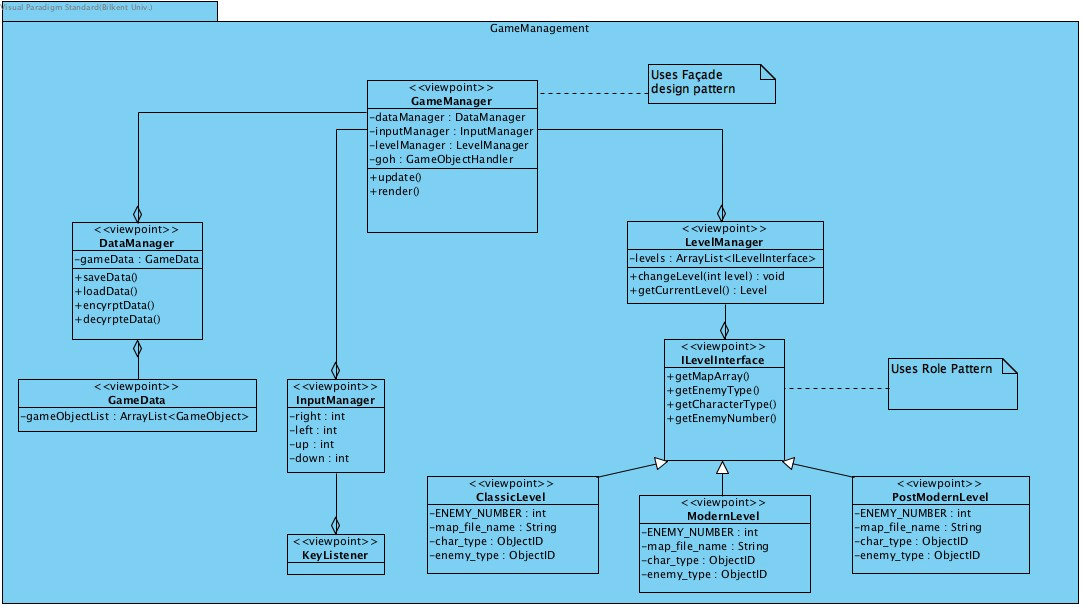
\

Figure 4. Game management subsystem

GameObjects subsystem

Object of the game will be designed in this subsystem. This subsystem will represent Model part of the MVC design pattern.

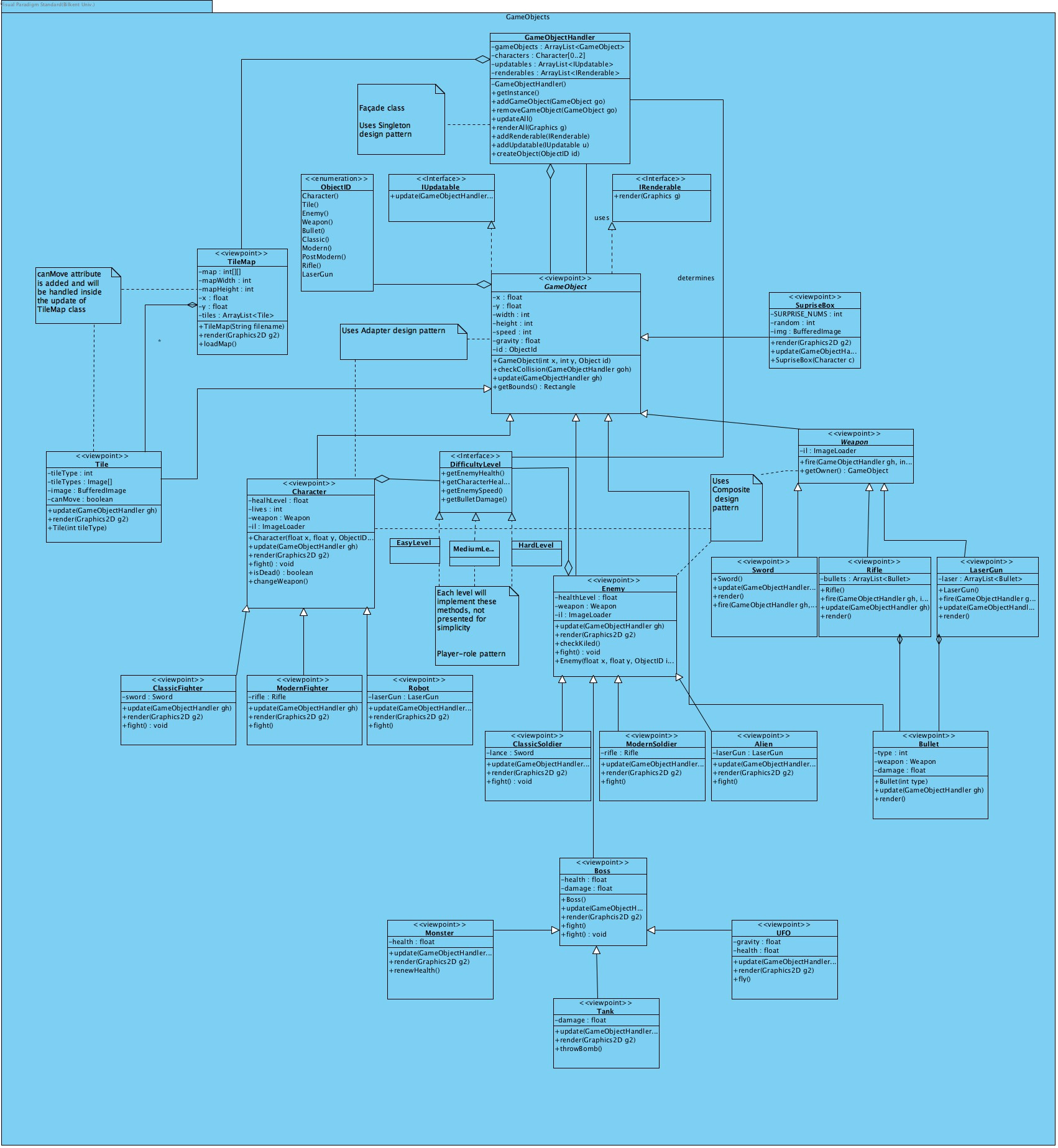


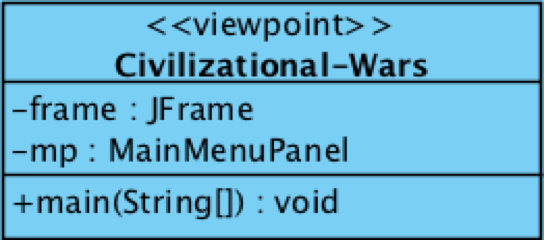
Figure 5. Game objects subsystem

**4.** **Low-level Design**

4.1. Detailed Class descriptions

User interface subsystem

**Civilizational-Wars Class**

****

**Attributes**

· *private JFrame frame*: Creates and customizes the frame of the game.

· *private MainMenuPanel*: Denotes and contains the properties of the main menu of the game.

**Operations**

· *public void main (String [] args)*: Main method for running the game.

**MainMenuPanel Class**

****

**Attributes**

· *private JButton playBtn*: Button component for the play button of the menu.

· *private JButton viewSettings***:** Button component for the viewing the settings of the game.

· *private JButton loadLevel:*Button component for the loading functionality of the game.

· *private JButton viewAbout:*Button component for the viewing about section of the game.

· *private JButton viewHelp*: Button component for the viewing help section of the game.

· *private JButton quitBtn*: Button component for terminating the game.

· *private JPanel buttons\_panel*: Panel component as a container for the buttons.

· *private JLabel icon\_container***:** Label component for keeping the image icon.

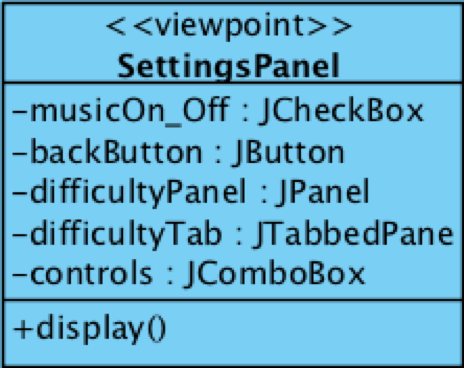
· *private ImageIcon gameIcon***:** Image icon of the game.

**Note\*:** each component has its own listener implemented inline style in the code.

**Operations**

· *public MainMenuPanel ()*: Constructs the main menu with its components and containers.

**SettingsPanel Class**

****

**Attributes**

· *private JCheckBox music\_on\_off*: Check box component managing the on/off mode of the game music.

· *private JCheckBox [] difficultyLevels*: Check box component managing the difficulty of the game.

· *private JComboBox [] controls:* Combo box component managing the difficulty of the game.

· *private JButton backButton***:** Button component for the coming back to the main menu of the game.

· *private JPanel difficulty\_panel*: Panel component as a container for difficulty check boxes.

**Note\*:** each component has its own listener implemented inline style in the code.

**Operations**

· *public void display ()*: Display method for components and containers.

**GameTypePanel Class**

****

**Attributes**

· *private JButton singlePlayerBtn***:** Button component for the playing the game in a single player mode.

· *private JButton multiPlayerBtn***:** Button component for the playing the game in a multiplayer mode.

**Note\*:** each component has its own listener implemented inline style in the code.

**Operations**

· *public void start (Boolean isMultiplayer)*: starts the game according to the game mode – single player or multiplayer

**AboutPanel Class**

****

**Attributes**

· *private JTextArea about\_text***:** Text area contains the information about the authors and project.

· *private JButton backButton***:** Button component for the coming back to the main menu of the game.

**Note\*:** each component has its own listener implemented inline style in the code.

**Operations**

· *public void display ()*: Display method for components and containers.

**HelpPanel Class**

****

**Attributes**

· *private JTextArea about\_text***:** Text area contains the information about controls and the game play.

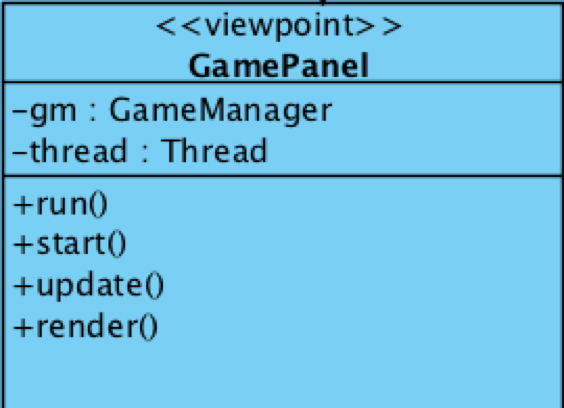
· *private JButton backButton***:** Button component for the coming back to the main menu of the game.

**Note\*:** each component has its own listener implemented inline style in the code.

**Operations**

· *public void display ()*: Display method for components and containers.

**GamePanel Class**

****

**Attributes**

· *private GameManager gm***:** Façade class of the Game Management subsystem, manages the game properties the game system.

· *private Thread thread***:** Creates and customizes the properties of the game thread.

**Operations**

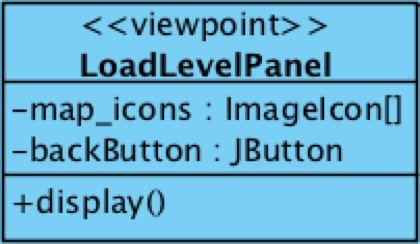
· *public void run ()*: Designed for running the game loop.

· *public void start ()***:** Designed for starting the game thread.

· *public void update ():*Updates the properties of the current level.

· *public void render (Graphics g):*Renders the properties of the current level.

**LoadPanel Class**

****

**Attributes**

· *private ImageIcon [] map\_icons***:** icons representing each level as an option for the user.

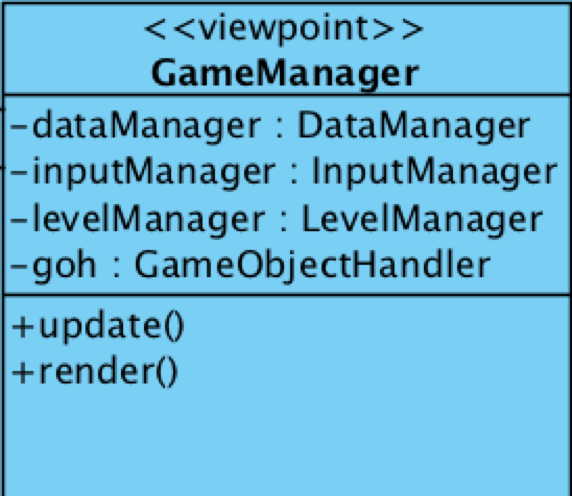
· *private JButton backButton***:** Button component for the coming back to the main menu of the game.

**Operations**

· *public void display ()*: Display method for components and containers.

Game Management subsystem

**GameManager Class**

****

**Attributes**

· *private DataManager dataManager*: Keeps the Data manager to be able to handle and manage the game data when it is necessary.

· *private Input Manager inputManager***:** Keeps the inputManger to handle the user inputs.

· *private Level Manager levelManager:*Keeps the levelManager to handle the interaction and changes between the levels.

· *private GameObjectHandler goh:*Keeps the game object handler, which is used in the level manager as a component for each level. Different levels will have different game object handlers – (different kind of game objects)

**Operations**

· *public void update ()*: Updates the properties of the game manager, primarily game object handler.

· *public void render (Graphics g)***:** Renders the graphics of the objects obtained from the game object handler.

**LevelManager Class**

****

**Attributes**

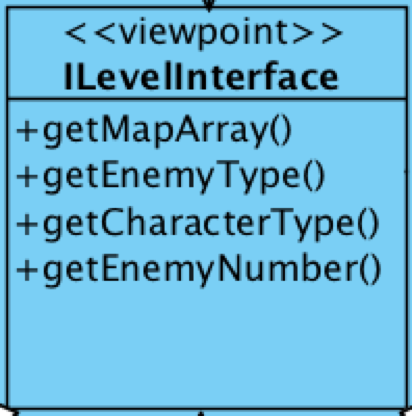
· *private ArrayList<ILevelInterface> levels*: Keeps the levels of the game as an interface**.**

**Operations**

· *public void changeLevel (int level)*: Changes the current level according to the given value. Current level is changes according to the given value index in the list.

· *public ILevelInterface getCurrentLevel ()*: Accesses the current level running on the game play.

**ILevel interface**

****

**Operations**

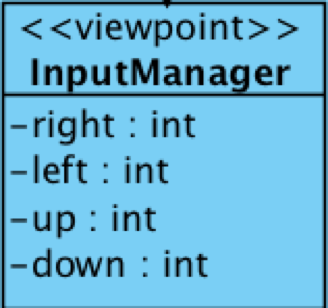
· *public String getMapArray ()*: Keeps the file name which contains two-dimensional array for the map design**.**

· *public ObjectID getEnemyType ()***:** Keeps the enemy type as an object id and accesses it.

· *public ObjectId getCharacterType ():*Keeps the character type as an object id and accesses it.

· *public int getEnemyNumber ():* Keeps the number of enemies and accesses it.

**InputManager Class**

****

**Attributes**

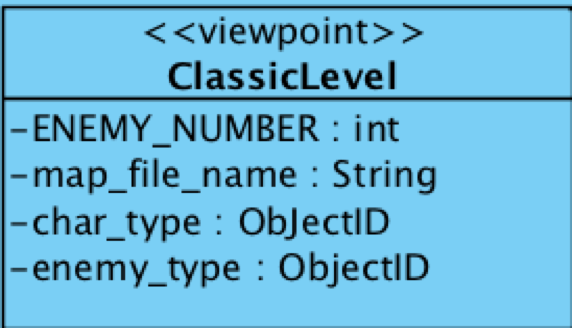
· *private static int right:* Keeps the key code for moving the character to the right.

· *private static int left:*Keeps the key code for moving the character to the left.

· *private static int up:*Keeps the key code for moving the character to the up.

· *private static int down:*Keeps the key code for moving the character to the down.

**ClassicLevel Class**

****

**Attributes**

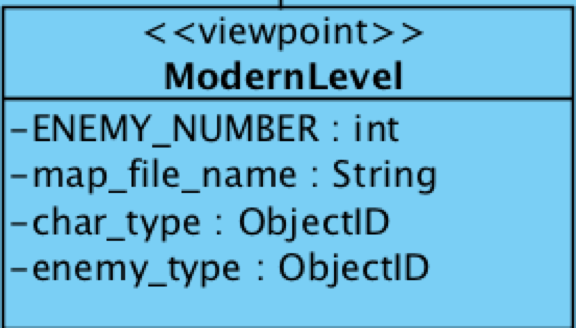
· *private final int ENEMY\_NUMBER:* Keeps the number of enemies for the classic level.

· *private String filename:*File name for the file which contains the data about the map design for classic level.

· *private ObjectID char\_type:*Keeps the character type for the classic level.

· *private ObjectID enemy\_type:*Keeps the enemy type for the classic level

**ModernLevel Class**

****

**Attributes**

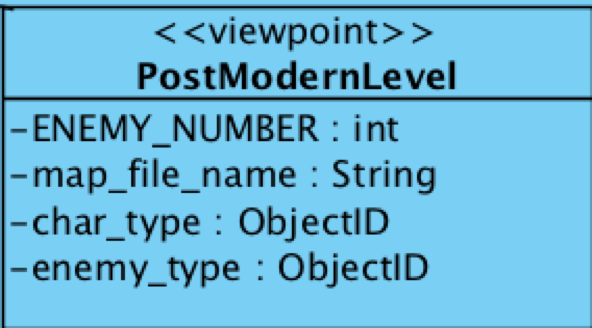
· *private final int ENEMY\_NUMBER:* Keeps the number of enemies for the modern level.

· *private String filename:*File name for the file which contains the data about the map design for modern level.

· *private ObjectID char\_type:*Keeps the character type for the modern level.

· *private ObjectID enemy\_type:*Keeps the enemy type for the modern level

**PostModernLevel Class**

****

**Attributes**

· *private final int ENEMY\_NUMBER:* Keeps the number of enemies for the post-modern level.

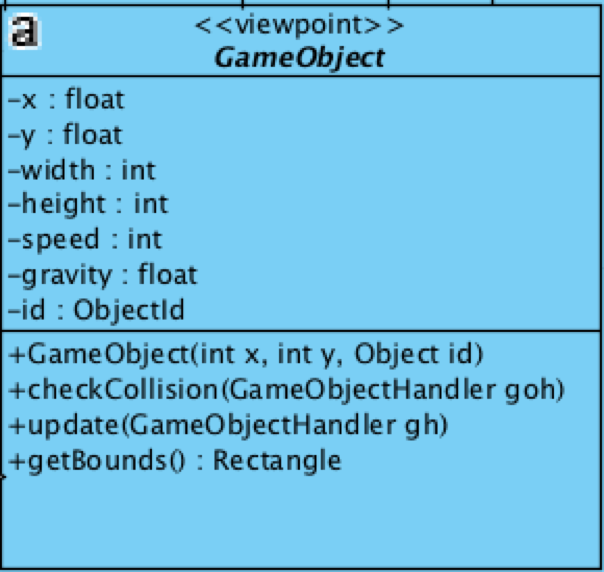
· *private String filename:*File name for the file which contains the data about the map design for post-modern level.

· *private ObjectID char\_type:*Keeps the character type for the post-modern level.

· *private ObjectID enemy\_type:*Keeps the enemy type for the post-modern level

Game Objects subsystem

**GameObject Class**



**Attributes**

· *private float x*: x coordinate of any game object on the screen.

· *private float y*: y coordinate of any game object on the screen.

· *private int width*: width dimension of the game object.

· *private int height*: height dimension of the game object.

· *private int speed*: speed of a movable game object.

· *private float gravity*: gravity value affects the game object. (could be different for different objects)

· *private TileMap tileMap*: tile map of the designed level. This attribute is included to check the collisions.

· *private ObjectID id***:** An ID which defines the type of the game object, for example, it defines whether the given game object is a character or an enemy or tile and etc.

**Operations**

· *public GameObject (int x, int y, Object id)*: Constructs the game object with the proper object properties **–** coordinates and object id.

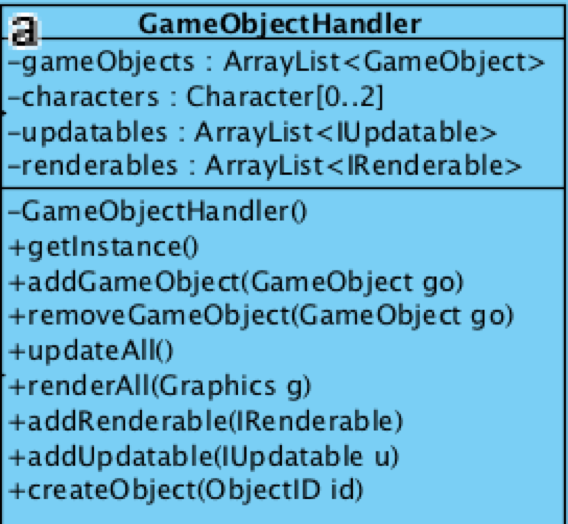
· *public void checkCollision (TileMap tm)*: Designed in order to check the collisions between the game objects and the tile map by checking the intersection between the object bounds and the tiles of tile map.

· *public void checkCollision (GameObjectHandler gh)*: Designed in order to check the collisions between the other game objects by checking the intersection between the objects bounds.

· *public void update (GameObjectHandler gh)*: This method updates the properties of the game objects – performs the action of the gravity by changing the y coordinate of the gam object.

· *public Rectangle getBounds ()*: This method accesses the rectangle which contains the game object with its coordinates on the screen.

**GameObjectHandler Class**

****

**Attributes**

· *private ArrayList<GameObject> gameObjects*: List of the all game objects in a level of the game. For example, character, enemy, tile, weapon, bonus box and so on.

· *private Character [] characters***:** To keep the characters – for multiplayer it is needed two characters for playing the game at the same time.

· *private ArrayList<IUpdatable>:* This is a list to keep all the updatable objects.

· *private ArrayList<IRenderable>:*This is a list to keep all the renderable objects.

**Operations**

· *public GameObjectHandler ()*: Constructs the game object handler, which handles all the properties and keeps all game objects.

· *public void addGameObject (GameObject go)*: This method adds a game object to the list of the handler.

· *public void removeGameObject (GameObject go)*: This method removes the given game object for the list of the handler.

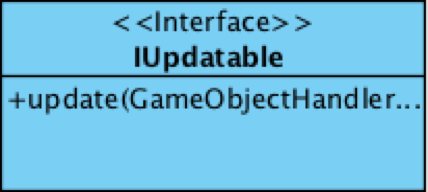
· *public void updateAll ()*: This method updates the properties of all game objects.

· *public void renderAll (Graphics g)*: This method renders the graphics of all game objects.

· *public void addUpdatable (IUpdatable u)*: This method adds an updatable object to the corresponding list of the handler.

· *public void addIRenderable (IRenderable i)*: This method adds a renderable object to the corresponding list of the handler.

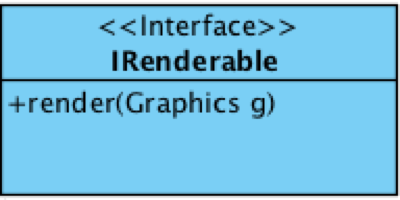
**IUpdatable interface**

****

**Operations**

· *update (GameObjectHandler gh)*: Updates the properties of the game objects, which are updatable.

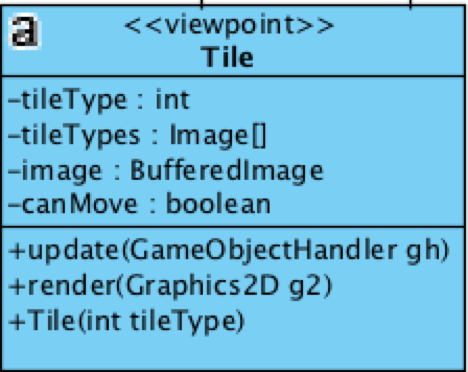
**IRenderable interface**

****

**Operations**

· *render (Graphics g)*: Render the graphics of the game objects, which are renderable.

**Tile Class**

****

**Attributes**

· *private int tileType*: This attribute defines the type of the tile. For example, tile can be a brick, grass and etc.

· *private Image [] tileTypes***:** This image array keeps the tile images according to their types.

· *private boolean isMovable:*This Boolean variable keep the property of movability of the tile. (Some tiles could be movable for mixing some challenge to the game)

**Operations**

· *public Tile (int tileType)*: Constructs a tile according to the given tile type.

· *public void update (GameObjectHandler gh)***:** Updates the properties of the tiles.

· *public void render (Graphics g)*: Renders the graphics of the tiles according the given tile type.

**TileMap Class**

****

**Attributes**

· *private int [] [] map*: This two-dimensional array keeps the values read from the file, which determines the locations of the tile on the game screen.

· *private int mapWidth*: Determines the width of the map.

· *private int mapHeight*: Determines the height of the map.

· *private float x*: x coordinate of the whole map. (Needed when filling the map with tiles)

· *private float y*: y coordinate of the whole map. (Needed when filling the map with tiles)

· *private ArrayList<Tile> tiles*: Keeps the created tiles for the map in this list.

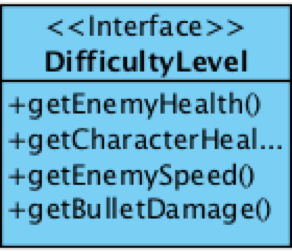
**Operations**

· *public TileMap (String filename)*: Constructs a tile map with the given file name, by reading the map data from the file with this file name.

· *public void render (Graphics g)*: Renders the graphics of all the tiles.

· *public void loadMap (String filename)*: This method loads the map by reading the given file.

**DifficultyLevel interface**

****

**Operations**

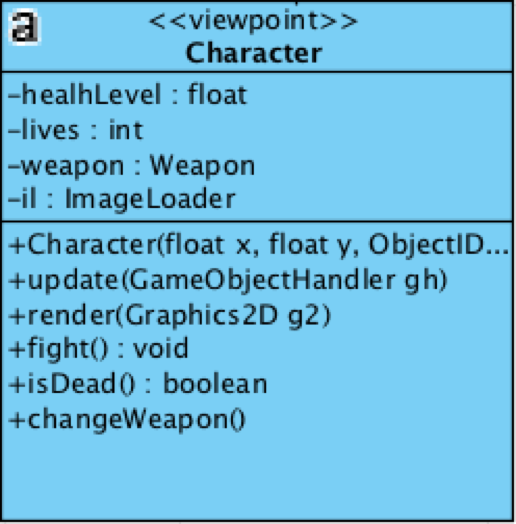
· *public float getEnemyHealth ()*: To manage the health level of the enemy according to the game level.

· *public float getCharacterHealth ()*:To manage the health level of the character according to the game level.

· *public int getEnemySpeed ()***:** To manage the speed of the enemy according to the game level.

· *public float getBulletDamage ()*: To manage the damage value of the bullets according to the game level.

**Character Class**

****

**Attributes**

· *private float healthLevel*: Defines the health level of the character.

· *private int lives*: Defines how many lives the character currently has.

· *private Weapon weapon*: Denotes the weapon of the player to fight against the enemies.

· *private ImageLoader il*: Keeps the images of the different characters.

**Operations**

· *public Character (float x, float y, ObjectID id)*: Constructs the character with the given properties like coordinates and object id.

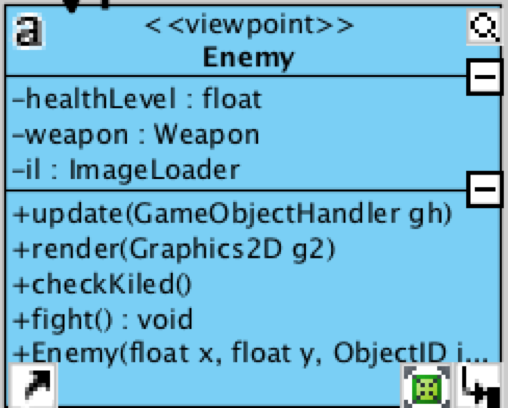
· *public void update (GameObjectHandler gh)*: Updates the properties of the character.

· *public void render (Graphics g)*: Renders the graphics of the character.

· *public void fight ()*: This method denotes the fighting feature of the character.

· *public boolean isDead ()*: This method returns true, if the character is dead.

**Enemy Class**

****

**Attributes**

· *private float healthLevel*: Defines the health level of the enemy.

· *private int lives*: Defines how many lives the enemy currently has.

· *private Weapon weapon*: Denotes the weapon of the enemy to fight against the player.

· *private ImageLoader il*: Keeps the images of the different enemies.

**Operations**

· *public Enemy (float x, float y, ObjectID id)*: Constructs the enemy with the given properties like coordinates and object id.

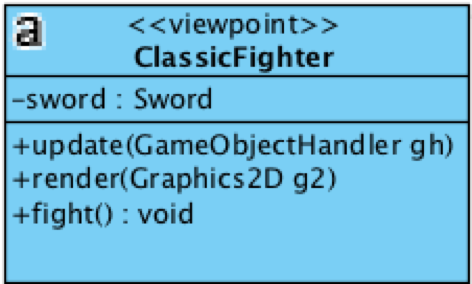
· *public void update (GameObjectHandler gh)*: Updates the properties of the enemy.

· *public void render (Graphics g)*: Renders the graphics of the enemy.

· *public void fight ()*: This method denotes the fighting feature of the enemy.

· *public boolean checkKilled ()*: This method returns true, if the enemy is dead.

**ClassicFighter Class**

****

**Attributes**

· *private Sword sword*: The weapon with which the classic fighter (player) can fight against the enemies.

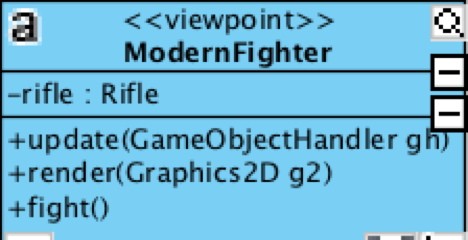
**Operations**

· *public void update (GameObjectHandler gh)*: Updates the properties of the classic fighter (character).

· *public void render (Graphics g)*: Renders the graphics of the classic fighter (character).

· *public void fight ()*: To be able to fight against the enemies.

**ModernFighter Class**

****

**Attributes**

· *private Rifle rifle*: The weapon with which the modern fighter (player) can fight against the enemies.

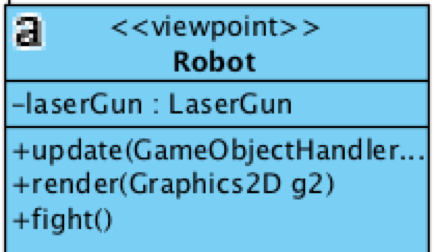
**Operations**

· *public void update (GameObjectHandler gh)*: Updates the properties of the modern fighter (character).

· *public void render (Graphics g)*: Renders the graphics of the modern fighter (character).

· *public void fight ()*: To be able to fight against the enemies.

**Robot Class**

****

**Attributes**

· *private LaserGun laserGun*: The weapon with which the robot (player) can fight against the enemies.

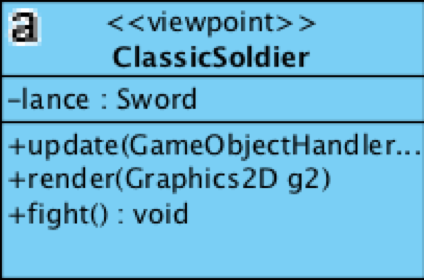
**Operations**

· *public void update (GameObjectHandler gh)*: Updates the properties of the robot (character).

· *public void render (Graphics g)*: Renders the graphics of the robot (character).

· *public void fight ()*: To be able to fight against the enemies.

**ClassicSoldier Class**

****

**Attributes**

· *private Sword sword*: The weapon with which the classic soldier (enemy) can fight against the enemies.

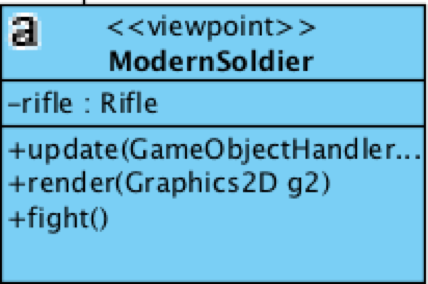
**Operations**

· *public void update (GameObjectHandler gh)*: Updates the properties of the classic soldier (enemy).

· *public void render (Graphics g)*: Renders the graphics of the classic soldier (enemy).

· *public void fight ()*: To be able to fight against the player.

**ModernFSoldier Class**

****

**Attributes**

· *private Rifle rifle*: The weapon with which the modern soldier (enemy) can fight against the player.

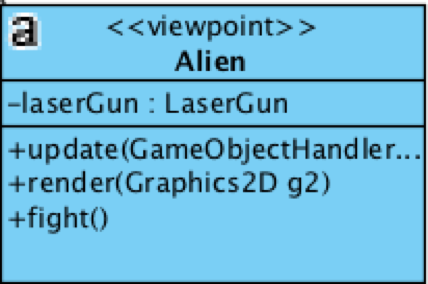
**Operations**

· *public void update (GameObjectHandler gh)*: Updates the properties of the modern soldier (enemy).

· *public void render (Graphics g)*: Renders the graphics of the modern soldier (enemy).

· *public void fight ()*: To be able to fight against the player.

**Alien Class**

****

**Attributes**

· *private LaserGun laserGun*: The weapon with which the alien (enemy) can fight against the player.

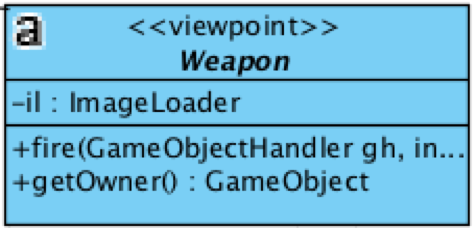
**Operations**

· *public void update (GameObjectHandler gh)*: Updates the properties of the alien (enemy).

· *public void render (Graphics g)*: Renders the graphics of the alien (enemy).

· *public void fight ()*: To be able to fight against the player.

**Weapon Class**

****

**Attributes**

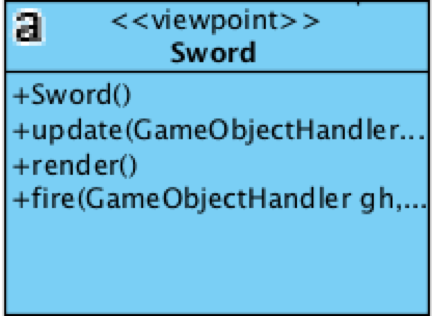
· *private ImageLoader il*: Keeps the images of the different weapons.

**Operations**

· *public void fire (GameObjectHandler gh)*: Designed to simulate the fighting mechanism of the weapon.

· *public GameObject getOwner ()*: Returns the owner of the weapon object. (Designed for proper collisions)

**Sword Class**

****

**Operations**

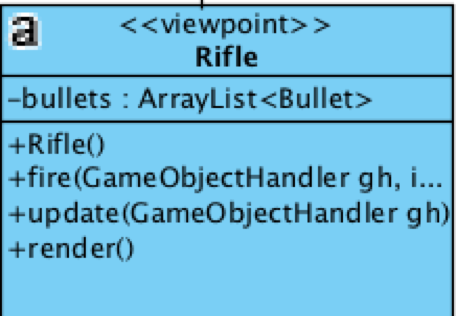
· *public Sword ()*: Constructs the sword with the game object properties.

· *public void update (GameObjectHandler gh)*: Updates the properties of the sword.

· *public void render (Graphics g)*: Renders the graphics of the sword.

· *public void fire (GameObjectHandler gh)*: Designed to simulate the fighting mechanism of the sword.

**Rifle Class**

****

**Operations**

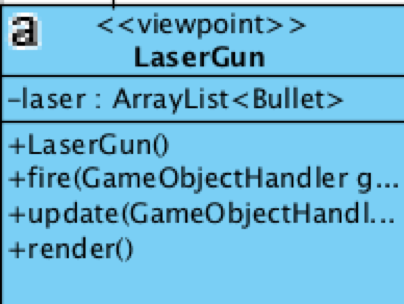
· *public Rifle ()*: Constructs the rifle with the game object properties.

· *public void update (GameObjectHandler gh)*: Updates the properties of the rifle.

· *public void render (Graphics g)*: Renders the graphics of the rifle.

· *public void fire (GameObjectHandler gh)*: Designed to simulate the fighting mechanism of the rifle.

**LaserGun Class**

****

**Operations**

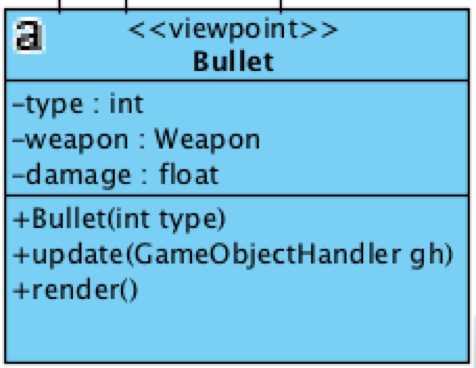
· *public LaserGun ()*: Constructs the laser gun with the game object properties.

· *public void update (GameObjectHandler gh)*: Updates the properties of the laser gun.

· *public void render (Graphics g)*: Renders the graphics of the laser gun.

· *public void fire (GameObjectHandler gh)*: Designed to simulate the fighting mechanism of the laser gun.

**Bullet Class**

****

**Attributes**

· *private int type*: Defines the type of the bullet – whether it is a laser bullet of rifle bullet – according to the weapon.

· *private Weapon weapon***:** Defines the weapon from which the bullet has been fired.

· *private float damage:*Defines the damage level of the bullet according to its type.

**Operations**

· *public Bullet (float x, float y, ObjectID id, Weapon weapon)*: Constructs a bullet with the given properties like coordinates, id and fired weapon.

· *public void update (GameObjectHandler gh)*: Update the properties of the bullet according to its type.

· *public void render (Graphics g)*: Renders the graphics of the bullet according to its type.

4.2. Final Object Design

This is the final design of the game system by connecting the subsystems.

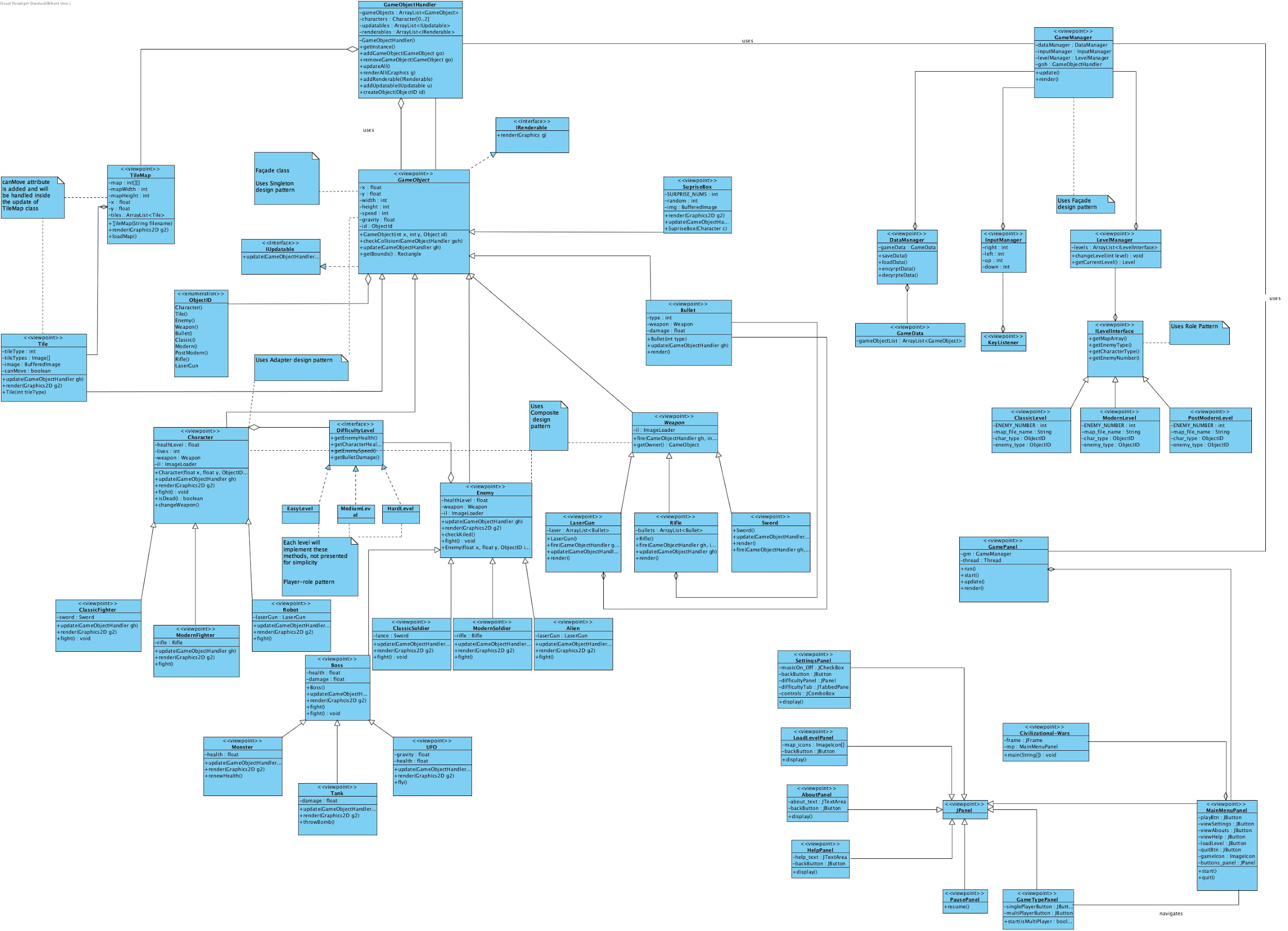
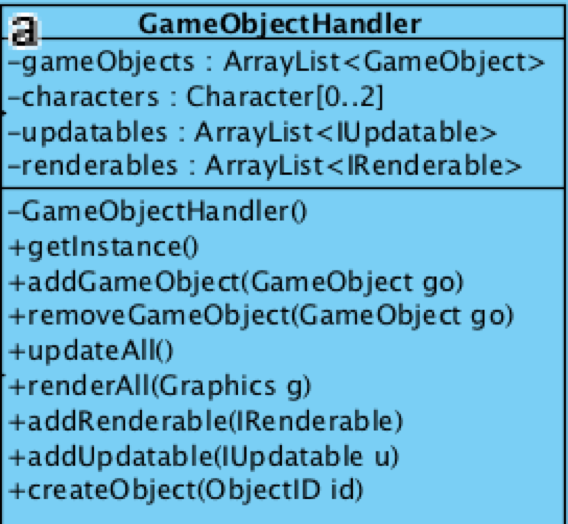
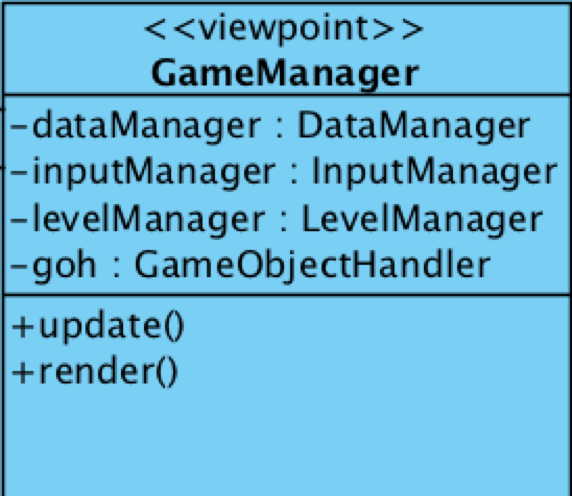
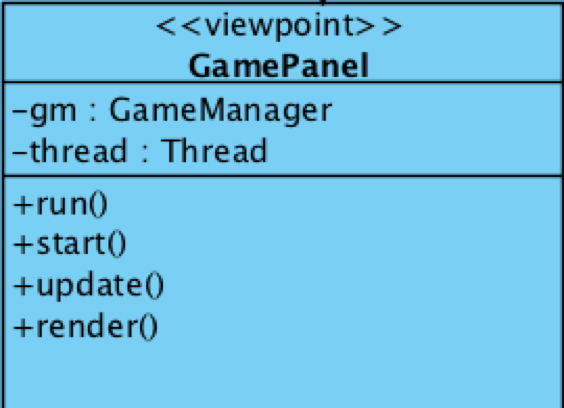


Figure 6. Final object diagram

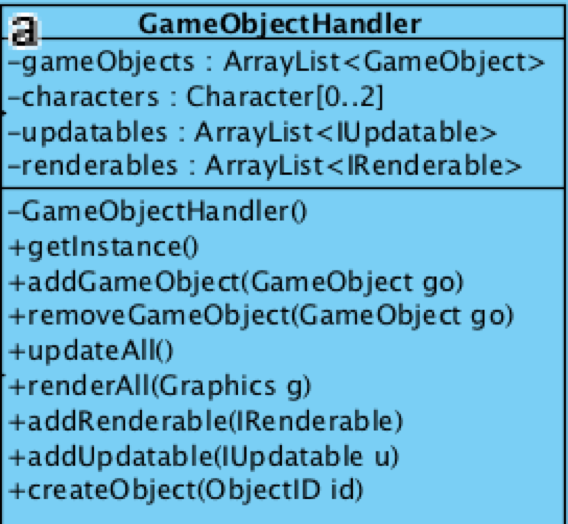
**5. Improvements summary**

For the second iteration we did track some improvements and changes in the design of our game system. Here is the list of these changes and improvements:

* We did make a change in the architectural style of the system. For the first iteration, we were using MVC (Model-View-Controller) [3] as an architectural style. However, now we are using the 3 tier architecture (Figure 1-2) for the purposed system. Although these two architectural styles are similar to each other, there is a slight topological distinction, which inclined us to utilize this architecture. This difference is about the interactions between the subsystems so that in 3-tier architecture client tier never get into interaction with the data tier, which is the same case for our game system. However, at the same time, in MVC architectural style, there is a triangular relation among the subsystems so that the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.
* We changed some design trade-offs after the first implementation of the game, which are mainly about the memory, programmability, flexibility and manageability of the system. Detailed information could be found on the section 1.3.
* Not only the missing parts of the system like difficulty of the game, surprise boxes, but also additional features like multiplayer mode, movable tiles have been added the subsystem design and final object design. (Figure 6)
* Design patterns are used to denote the relations between the classes of the subsystems. Here is the list of the design patterns used in the system:
  + Facade - In order to increase the robustness of the system, we used this pattern between the subsystems via the class of GamePanel, GameManager, GameObjectHandler, respectively for the subsystems of User interface, Game management and Game objects.

****

* + Composite - For the subsystem of Game objects, Weapon is a game object which is used both by the Character and Enemy objects, which are also game objects. (Figure 5)
  + Singleton - For the GameObjectHandler class of Game Objects subsystem, this design pattern is used, because of the over and over using and need the objects from the different classes like Character, Enemy and so on.

****

* + Player-role - This pattern is used for the classes of difficulty levels (EasyLevel, MediumLevel, HardLevel) so that there is an interface of DifficultyLevel, which is implemented by the level classes, and used by the Character and Enemy classes (Figure 7). In addition, in the Game management subsystem, this pattern is also used for the level hierarchy so that as role class there is an interface of ILevel Interface, which is implemented by the level classes (ClassicLevel, ModernLevel, PostModernLevel) and used by the LevelManager class. (Figure 8)

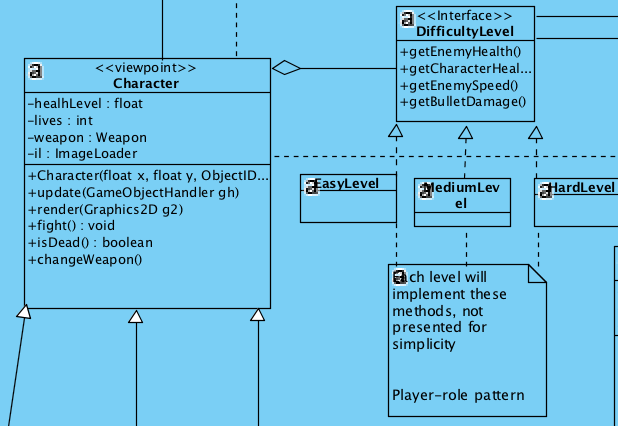


Figure 7. Player role in difficulty

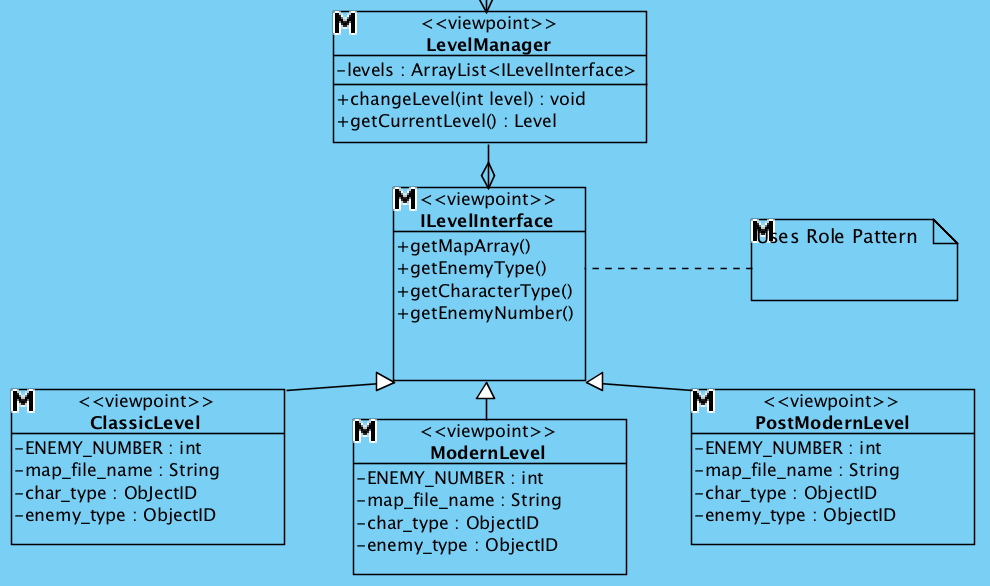


Figure 8. Player role in levels

**6. Glossary & References**

* [1] JRE – Java Runtime Environment is a set of software tools for development of Java applications. It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries. **JRE** is part of the Java Development Kit (JDK), but can be downloaded separately.
* [2] 3-tier architecture is a client-server architecture in which the functional process logic, data access, computer data storage and user interface are developed and maintained as independent modules on separate platforms.
* [3] MVC – Model View Controller is a software architectural pattern for implementing user interfaces on computers. It divides a given application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user.