

# CS319 PROJECT DESIGN REPORT

# GROUP G

### **ERKUT ALAKUŞ**

### **DOĞANCAN DEMİRTAŞ**

### **GÜNÇE KALYONCU**

### **CANSU TÜZMEN**

**TABLE OF CONTENT**

1. Introduction3
   1. Purpose of the System3
   2. Design Goals3
2. Software Architecture4
   1. Subsystem Decompostion4
   2. Hardware/Software Mapping……………………………………………5
   3. Persistent Data Management…………………………………………….6
   4. Access Control and Security…………………………………………….6
   5. Boundry Conditions………………………………………………………………..6
3. Subsystem Services7
4. Low-Level Design…………………………………………………………….11
   1. Object Design Trade-offs……………………………………………………………....11
   2. Final Object Design…………………………………………………………………….12
   3. Packages………………………………………………………………………………..14
   4. Class Interfaces………………………………………………………………...............17
5. **Introduction**
   1. **Purpose of the System**

Digger Unlimited is a basic arcade game and it main aim is entertaining players with a good design. Thus, the levels are sorted from easy to hard and each of them consists of some bonuses changing the course of the game to increase the attraction. It will also have well-designed interface so it is easy to use and learn its tools.

* 1. **Design Goals**

**Response time**

Since Digger Unlimited is an arcade game, the response time is very important so

the average respond time will be 0.5 second in order not to distract the player’s attention or interest.

**Extensibility**

In order to make game more exciting and intersting, new features like new maps, bonuses, characters and concepts could be added over time because its object oriented structure enables them easiliy**.**

**Usability**

The system will have user-friendly interface to make the usage more easy . Thus, it provides players well designed menus and a “help” option to learn the goal of the game and its tools.

**Trade-Off's:**

**Space vs. speed**

Speed is more important than space, because the players should collect coins without being caughted by the monsters so the system should respond in seconds to make the game more entertaining and interesting.

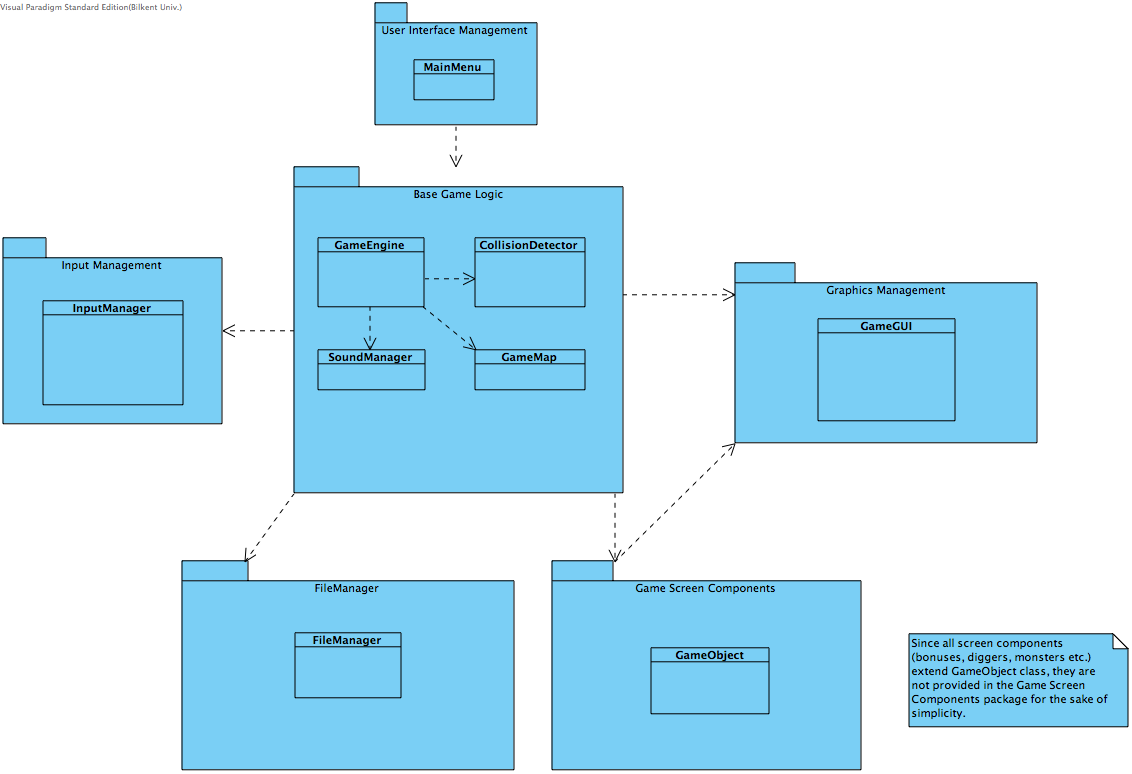
**Usability vs. Functionality**

Usablity is more important than functionality so in order to reach wide range of customers, the system interface should not be prepared too complex.

**2. Software Architecture**

**2.1 Subsystem Decomposition**

We choose 3-tier architectural style although it is widely used in DBMS applications. Since our current design will be constructed on 5 main stages –**Engine**, **Graphical User Interface**, **Menu**, **Input Manager** and **Game Objects**, we decided to represent these stages in 3-tier architecture. In the below diagram, interaction of 3 layers can be seen:

**** *Figure 1- Subsystem Decomposition*

## 

## **2.2. Hardware / Software Mapping**

"Digger Unlimited" will require a JRE or JDK to be executed since it is developed in Java language. As hardware requirements, game requires a keyboard to give input and interact with the system. Minimum system requirements will be minimal; a basic computer with an operating system should be sufficient to run the game. Input are only 4 controls, which is nothing for today's operating computers. We will also have stored information on .txt files, so system should support .txt file format. It does not require any Internet connection.

## **2.3.** **Persistent Data Management**

“The Digger Unlimited” does not require a complex database, because we hold small amount of information and it can be done within client’s hard disk drive, which will include text and sound files and images. Game Map will be included in the folders as a text file, and before every new stage, game map will be created from these text files. There will be five stages that means five text files. If system would unable to open text file, game will not start and ask user for a restart the game or repair the game files. Sound effects and object images will also be in the client’s hard disk drive already with simple formats, this helps user to add his own sounds to the game and relieve us from having a database.

## **2.4.** **Access Control and Security**

"Digger Unlimited" does not have any user authentication and there is also no database to store this information. Only "GameEngine" has access to the file system and to prevent changes on game logic, necessary security measurements were satisfied with the critical decision variables to protect the flow of information. We gave input, image and sound controls to another managers; specific classes for jobs to increase the robustness of the game and at the same time reduce the complexity of "GameEngine". "Digger Unlimited" will not need any kind of network connection, which decreases our job to prevent in case of security vulnerabilities we might face.

## **2.5.** **Boundary Conditions**

**Initialization**

Due to the “Digger Unlimited” is a desktop application; it does not require an install. Users only need to have JRE or JDK than can run .jar file of the game.

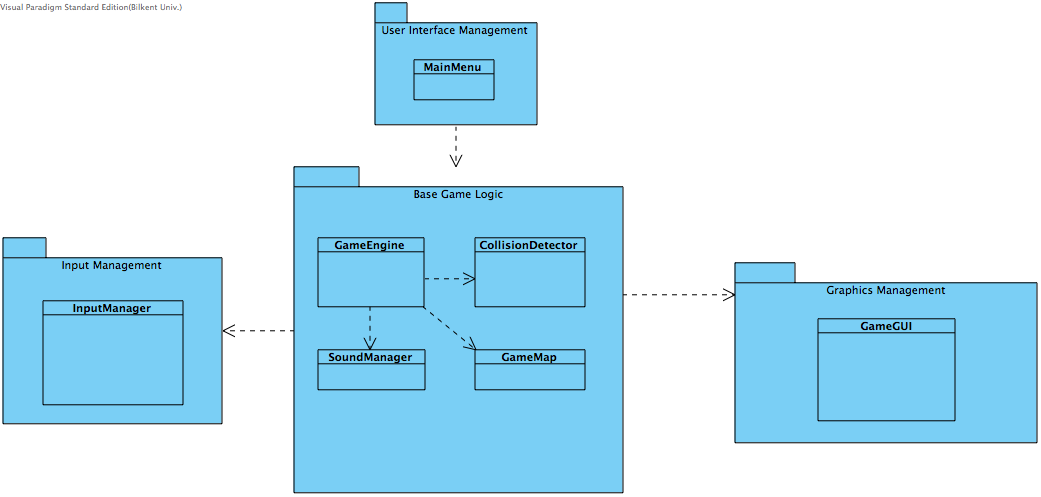
**Termination**

It is possible to terminate the game by clicking “X” button on the upper right side of the game screen. For in-game termination, user could click “Exit” in the main menu, or pressing “Esc” to open pause menu and return “Main Menu” by selecting it from the pause menu than perform exit operation.

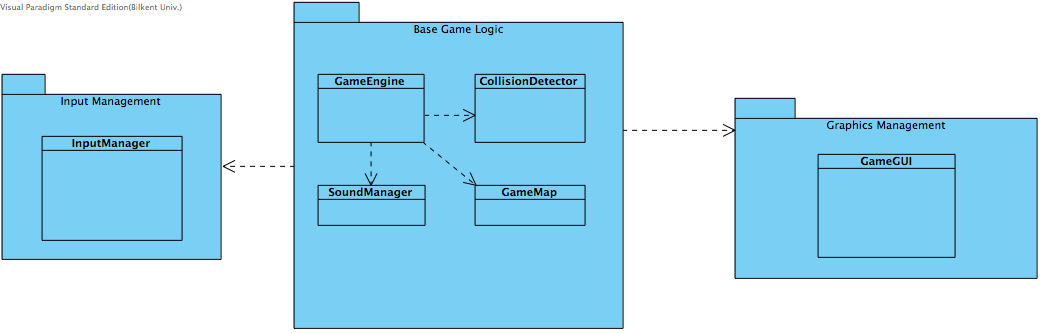
**Error**

If system encounters an error that prevent game resources to load, such as maps, sounds and images, “Digger Unlimited” will close itself and return an error message to user and ask for a reopen or repair. An error based on a performance issue, which is unlikely, will cause loss of current data.

# 3. Subsystem Services

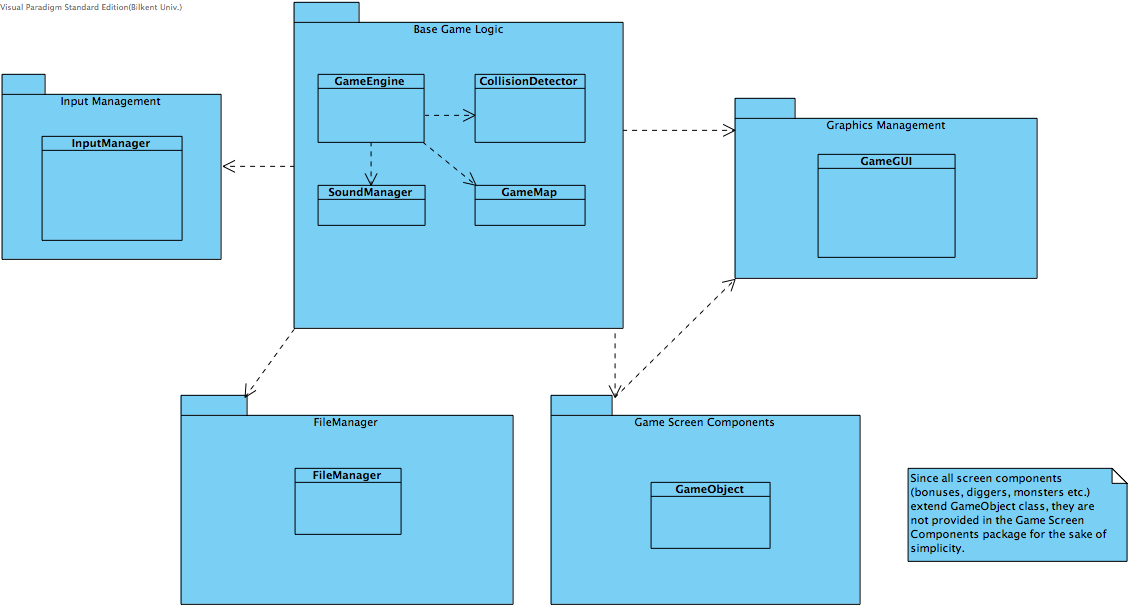
Since user will first interact with MainMenu, it is assigned into the presentation layer. Depending on user’s choice game concept –characters- will be changed. Game theme and sound options will be passed to the GameEngine class, which is the main logic of the game, from MainMenu.

*Figure 2– Interaction of Presentation and Application Layers*

****

*Figure 3 – Application Layer*

As indicated above, GameEngine is the logic of the game. Initialization of game map, input manager and collision detector, GameMap, InputManager and CollisionDetector classes, is done here. Base Game Logic, Input Management and Graphic Management packages are related with each other such that InputManager class provides every user input to the GameEngine class. Depending on these inputs GameEngine updates the instance of GameMap class and triggers GameGUI class in every iteration to draw the map again.

****

*Figure 4 – Interaction of Application and Data Layers*

File Management and Game Screen Components packages are considered as Data Layer. All I/O is handled with these packages. FileManager class in the File Management package is responsible to read game map contents from a text file. GameMap and GameObject instances are created from this text file and provided to GameEngine and Game GUI classes.

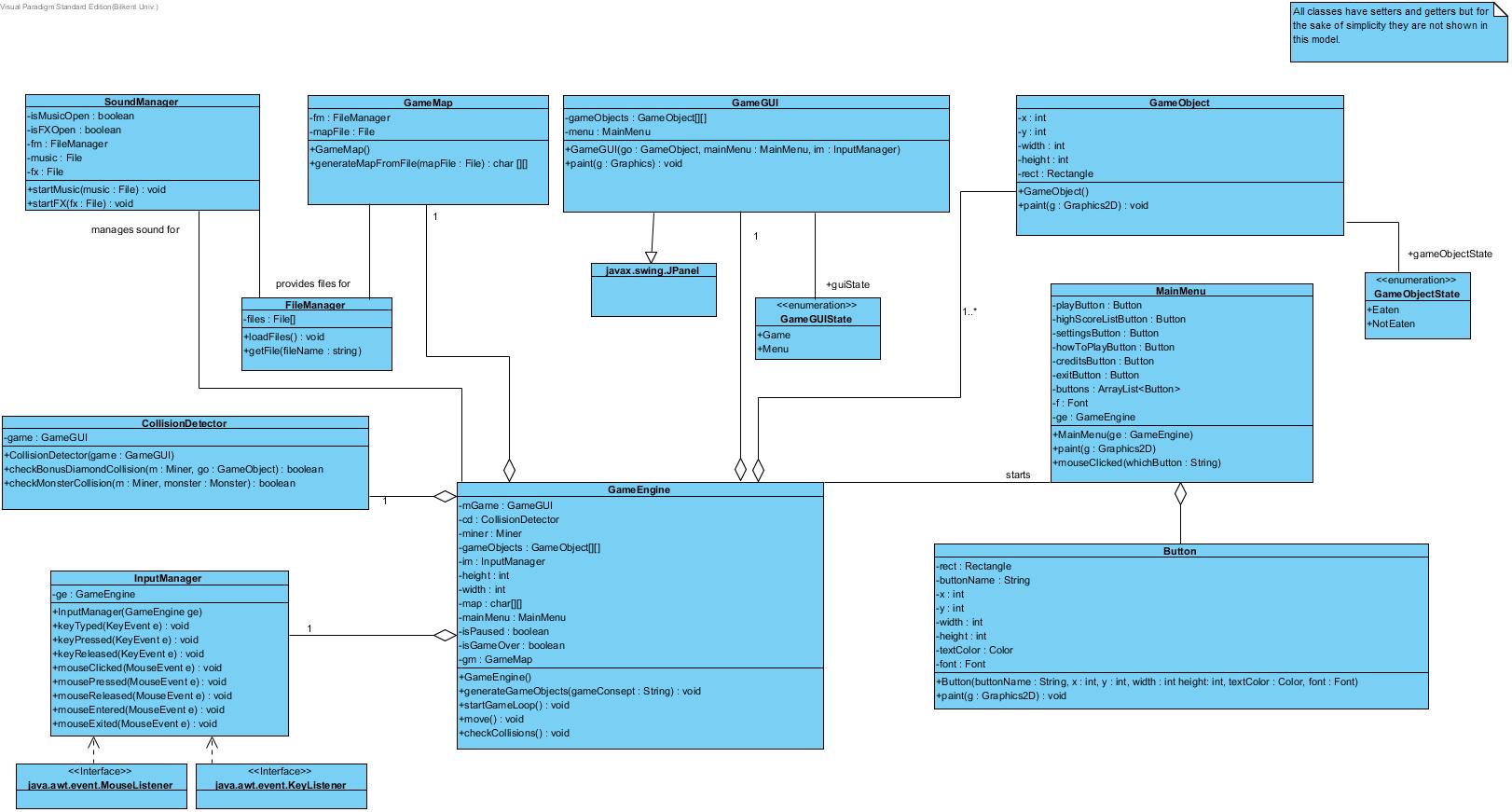
# 4. Low Level Design

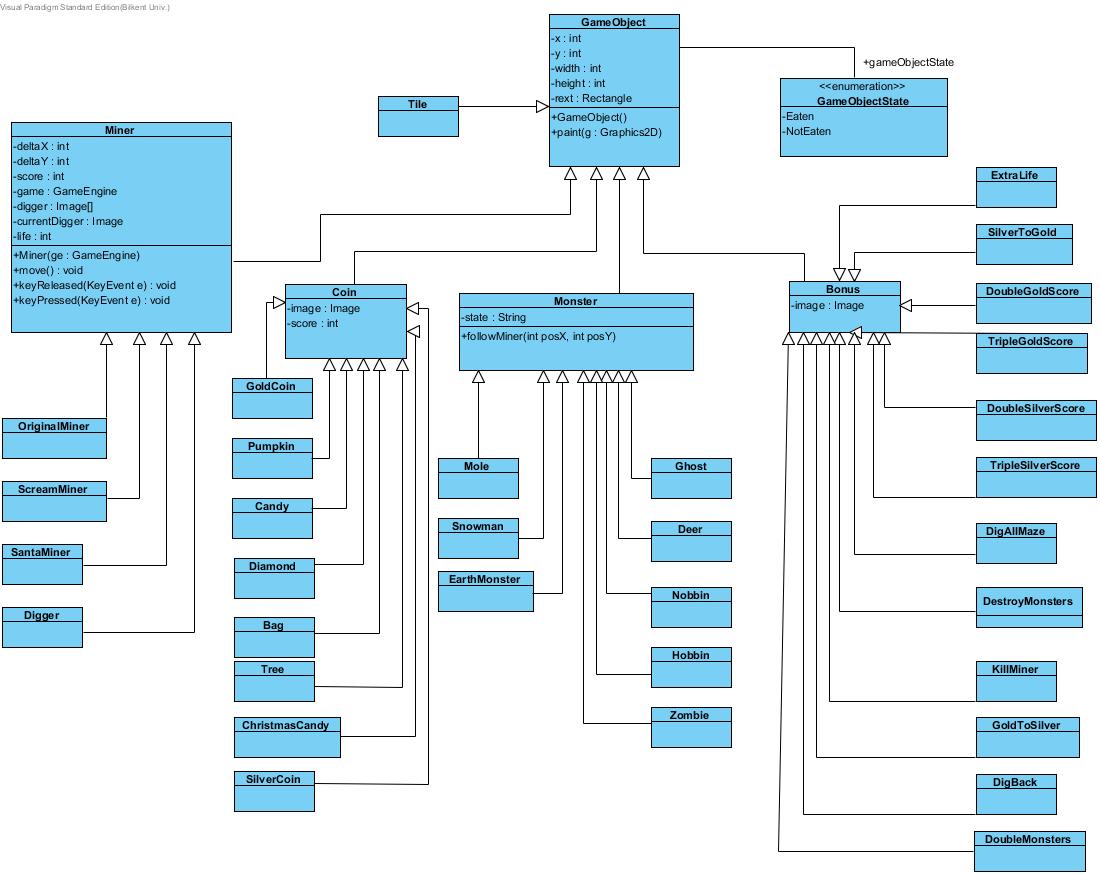
4.1. Object Design Trade-offs

We followed Singleton Design Principle in order to be make sure particular classes are only instantiated once. Since having more than one Miner or GameGUI or CollisionDetector creates a mess during implementation phase, singleton is a good choice to reduce errors. As can be seen from the class diagram below, most of the constructors are private and a method to get instance of this class is provided.

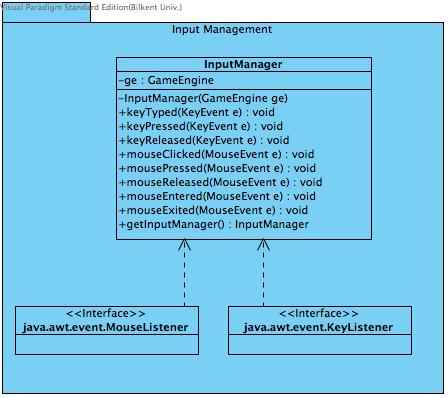
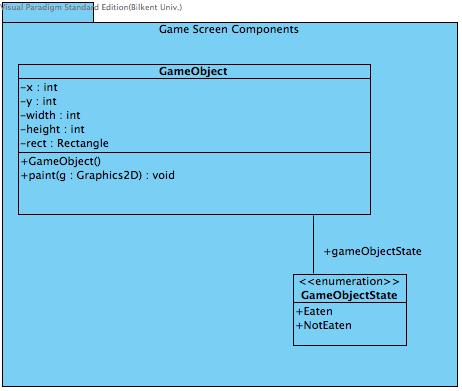
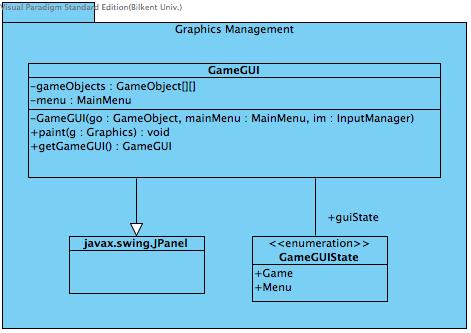
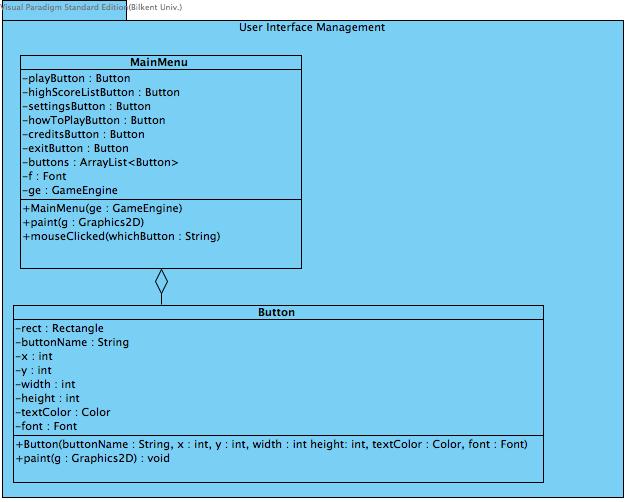
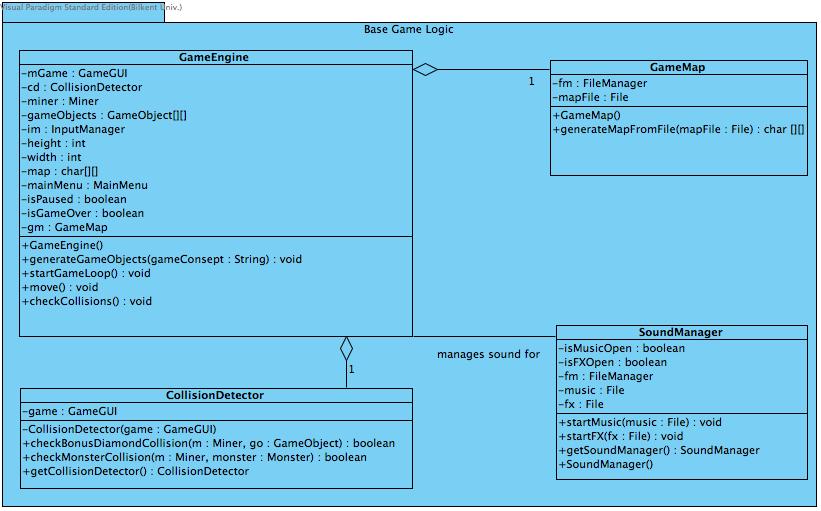
We also decided to create game map from a text file. Representing a 2D world is easier when we use a 2D array. After iterating the text file line by line, map will be loaded into memory with a 2D array.

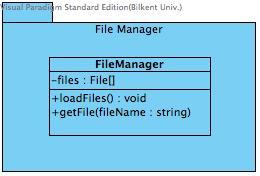
As can be seen from the class diagram, only one class that extends JPanel is present. That means there is only one canvas and depending on the state of the game, which can be menu or paused or high score screen, GameGUI draws components on the screen.

**4.2. Final Object Design**

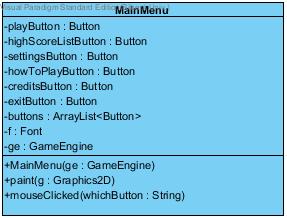
****

**4.3. Packages**

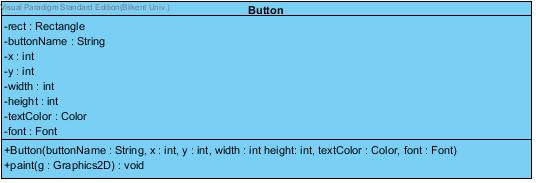




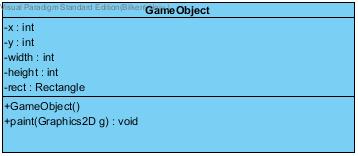
**4.4 Class Interfaces**

**Main Menu Class**

This class is created for main menu of the game. It stores an arraylist for Button objects. Each Button object represents an option the player could choose. To do that, it has “mouseClicked(string whichButton)” method to recognize which button is clicked on according input come from “InputManager”. It also has “paint(Graphics2D g)” method to draw all buttons main menu has.

**Button Class**

The “Button” class stores some features of button like its name, shape and its width and height, color and position. The paint(Graphics2D g) of this class only draws one button according to these features.

**Game Object Class**

“GameObject” class is created because all game objects has width, height, positions and Rectangle so all fundemental objects will use it as their parent class. It has paint(Graphics2D g) operation to draw game objects and this method is overrided by all classes which extends “GameObject”.

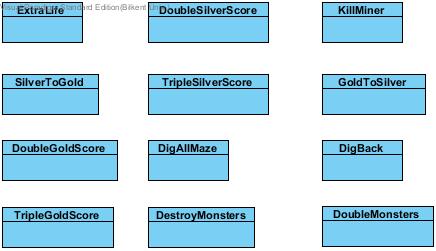
**Miner Class**

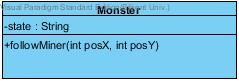
Addittion to “GameObject” class, the “Miner” class has some extra attributes like an Image array for walking animation of miner. Moreover, it has some operations: move(), keyReleased() and keyPressed() to control miner’s movement and positions. move() sets image of the miner and keyReleased() and keyPressed() set the postion of miner according to input which comes from inputManager.

 **Coin Class**

The “Coin” class additionally has attributes such as the image of coin as an Image object and its score according to being gold or silver.

**C:\Users\Cansu\Desktop\cs319\sketch\bns.jpgBonus Class**

This class represents bonus objects so it store its iamge as an Image object and each bonus is represented as class and they extends Bonus class.

**Monster Class**

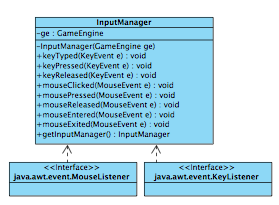
This class additionally has followMiner(int posX, int posY) operation to lead monster to follow miner.

**C:\Users\Cansu\Desktop\cs319\sketch\GameObjects.jpg** **Tile Class**

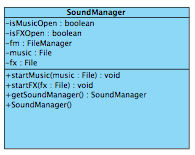
The “Tile” class also extends “GameObject” and it is created to draw background of the game.

**GameObjectState**

All game objects have states and when an object’s position intersects with the miner’s the state of it is changed from not eaten to eaten.

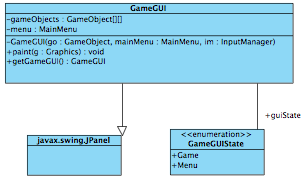
**InputManager Class**

"InputManager" is reponsible for listening inputs from the user. It is the listener of the program with "KeyEvent"s, which is needed for gameplay, and "MouseEvent"s, which is needed for menu selections. "GameEngine" uses "InputManager" to change the character accordingly.

**SoundManager Class**

"SoundManager" is the class for audio settings of the game. It manages sounds for "GameEngine". "SoundManager" uses files that "FileManager" provided. Sound files are loaded through "FileManager" and processed by "SoundManager".

**GameGUI Class**



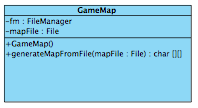
"GameGUI" handles the rendering operations for the game. It uses "GameObject", "MainMenu" and "InputManager" classes to render the screen according to gameplay and user input.

"MainMenu" is used in "GameGUI" class for showing Main Menu and with the help of "InputManager" class, display the appropriate screen (e.g. Settings, Game Themes etc.).

"GameObject" holds the current state of an object in the screen and GameGUI class display the object according to its state. "Input Manager" class helps "GameGUI" to determine which object will change its state, than "GameGUI" will update the screen view.

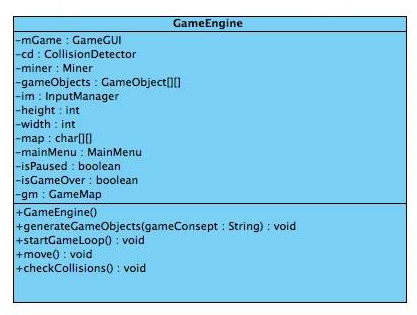
"GameGUIState" holds states of GUI as enumeration. "GameGUI" will prepare and display GUI according to its state.

**GameMap Class**



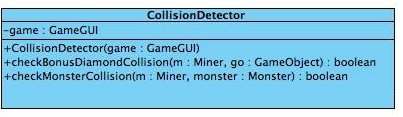
"GameMap" is the class for initializing first state of objects in a new level. It is called by "GameEngine" at the start of each level. "GameMap" uses files that "FileManager" provided. Map files, which is .txt's, are loaded through "FileManager" and processed by "GameMap".

**GameEngine Class**



“GameEngine” class controls the entire game logic and it has the game loop. It reads and generates game objects from the file and sends them to “GameGUI” class. This class performs the proper operations according to the requests that came from “InputManager” class such as starting game, moving miner, collecting coins. Also it checks collisions between “GameObject” objects with the help of “CollisionDetector” class and ends these collisions with killing miner, destroying tiles, adding bonuses.

**CollisionDetector**



“CollisionDetector” is a class detects collisions and notify “GameEngine” class.