# CS 319 - Object-Oriented Software Engineering
# Design Report


Group 2E:

Ferhat Serdar Atalay
Aylin Çakal
Ali Bulut
İsmail Serdar Taşkafa

**Submission: 17.04.18**

# 1. Introduction

## 1.1. Purpose of the System

Planet Trip is a 2-D arcade game similar to DX-Ball. User tries to destroy all meteors in current level of the map. To do this user only controls a paddle for giving new direction to the ball. When the ball destroys a meteor, a game bonus can appear. If user wants to get this bonus it has to collect that bonus by moving paddle. The user's aim is to finish all levels successfully. We designed the game in such a way that at each level the speed of ball increases and olsa levels became more difficult. Also to increase challenge, a enemy paddle could appear. Planet Trip aims to improve quick response ability to a sudden changes and increase hand-eye coordination and also good time to spend.

## 1.2. Design Goals

### 1.2.1. End User Criteria

- Usability

From the welcoming screen, every part of the game experience should be self explaining. For this purpose we find it useful to use the terms that players are familiar with in the gaming culture. The main purpose is to help the user find their way through without further explanations needed.

- Good Documentation

While developing PlanetTrip game, we aim to well document all the work that we will do.

### 1.2.2. Maintenance Criteria

- Extensibility

Extensibility is the ease of adding new features to the software. It is an important criteria in development of any software system as new requirements can be added to the system as result of feedback from users or the product owner.

- Adaptability

Usage of object oriented design principles implies each component of the software is independent of each other. Because of that, changes in one part does not affect the functioning of other parts. This makes making modifications on our software and testing it more straightforward.

### 1.2.3. Performance Criteria

- Reusability

We designed our classes in a fashion that they will be able to reused again for further upgrades in project. Our aim is to have the least amount of need for modification on our code when another part is changing.

### 1.2.4. Trade-Offs

- Functionality vs. Usability

Since we want to user friendly game, it should be easy to use. In other words, users should not encounter any usage difficulties. Planet Trip game will provide user-friendly interface to user. Because of that, we give priority to usability rather than functionality.

- Portability vs. Efficiency

Our application will have portable environment so it will work on various devices that has different specs. Since we can not cover all devices, the game will not be running efficiently on all devices.

# 2. Software Architecture

## 2.1. Subsystem Decomposition

In order to clearly show how our system works, we divided it into pieces. To provide a good organisation, we categorised the subsystems by their missions. This

method enables us to choose between different architectural styles, and we decided to go on with Model View Controller since it seem the most suitable for our project. . To capitalize on this interface-logic separation, the game is designed on MVC (Model-View-Controller) pattern. This will help minimize coupling between main subsystems and maximize cohesion within those subsystems.

In Figure 2.1, three main subsystems of our project are shown. The “View” subsystem that is User Interface Management deals with handling user interfaces and getting user input. “Model” subsystem that is Game Entities handles the data, rules and the logic of the game. Finally, “Controller” subsystem that is Game Management gets input from the other subsystems and updates them.

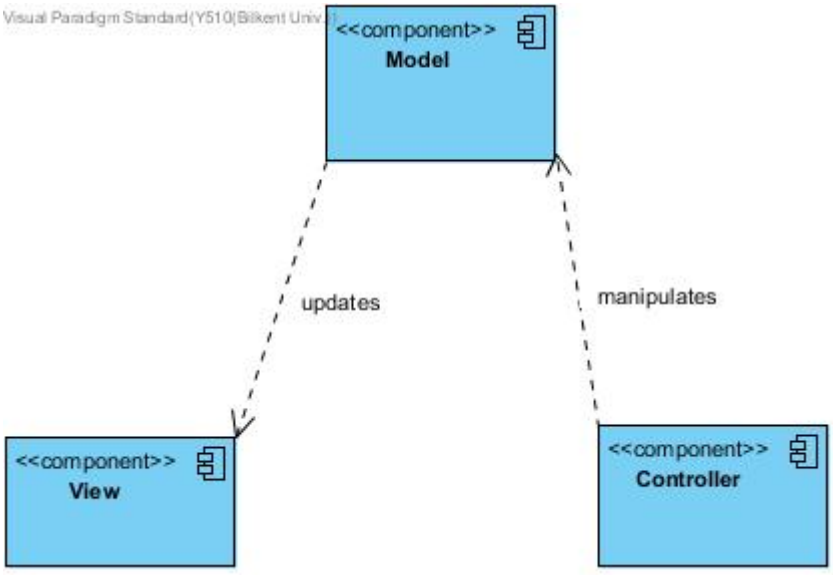



Figure 2.1: MVC Architecture

MVC architecture as shown in Figure 2.1 is expanded to show the subsystems of these three components and their interactions.

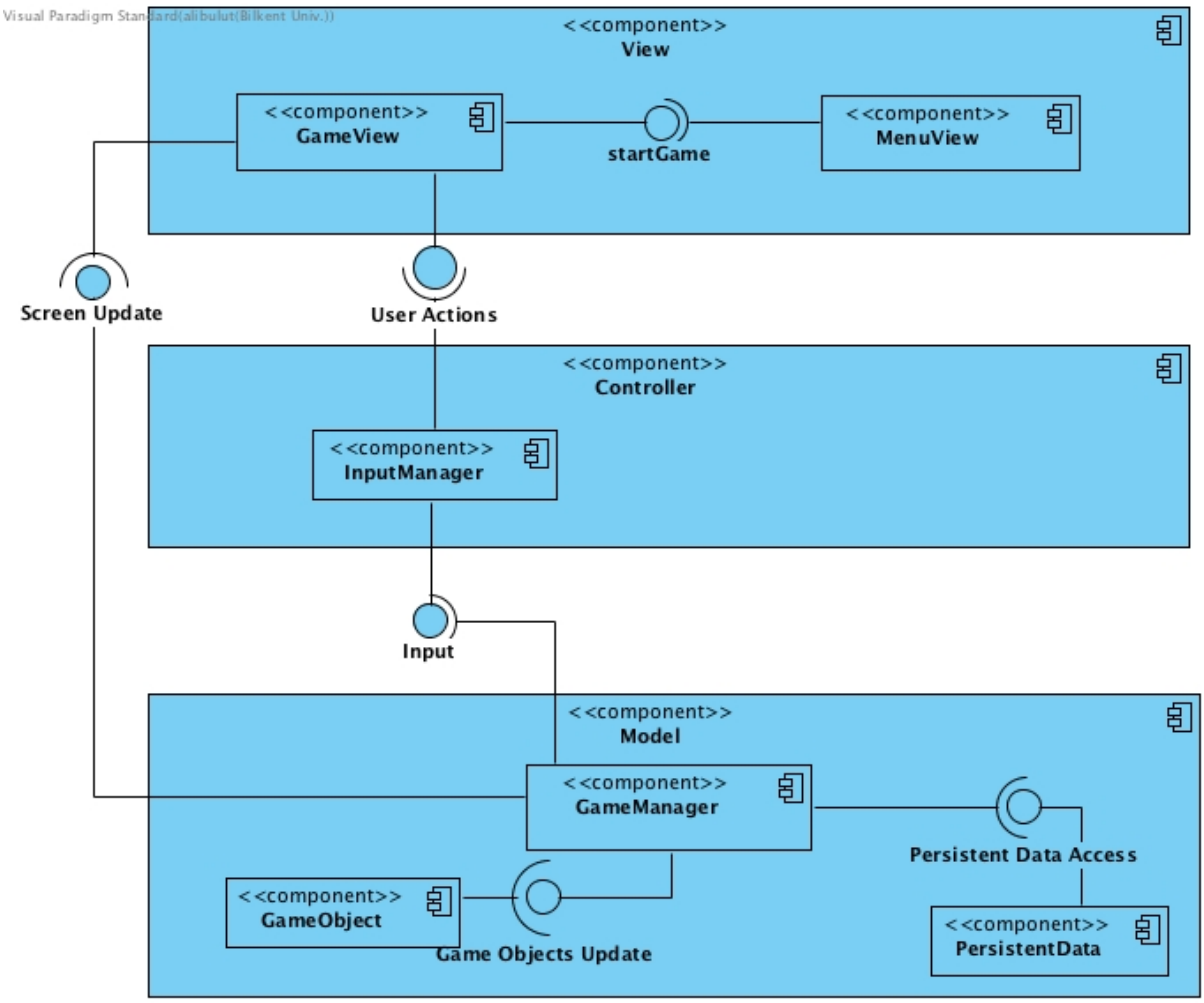



Figure 2.2: Subsystem Decomposition

## 2.2. Hardware/Software Mapping

Our game needs a keyboard for the inputs, and a mouse for the settings adjustment. Since we will implement the project with java, a basic computer with java compiler to compile & run the .java file will fulfill the requirements of the game.

## 2.3. Persistent Data Management

Planet Trip does not need for any databases or data management system like SQL. The game files such as images and text files will be stored in hard drive. So every time the game has started, the software loads needed files from the related file path on the hard drive. Also to find the where data is, we will keep path of a file in the codes. On the other hand, the software has to save options and progress of an user. To do this, the software also override related text files.

## 2.4. Access Control and Security

Since Planet Trip game has nothing to do with any network connection or database, there is no need to any security protocols. The game also does not need for any personal data and because all game data is stored separately for each device, there is no need to any access control. Any change in a file will only affects that local programme.

## 2.5. Boundary Conditions

### 2.5.1. Initialization

Our game will be able to played without installation, since it will come with an executable .jar file.

### 2.5.2. Termination

User will be able to leave the game through clicking “Exit” button in Main Menu. For those who want to quit during the game play, the “Exit” button also exists in Pause Menu. And since the game will run in a window, it is possible to close the application with “x” button on the upper right.

### 2.5.3. Failure

If the game cannot load a file, related area for that object will left blank. In that case there is a problem with the file paths or file source.

# 3. Subsystem Services

## 3.1. View: User Interface Subsystem

View subsystem that is User Interface Subsystem conducts the graphical interfaces of the PlanetTrip game. With the help of the other subsystems, view subsystem manages the user interfaces.

### 3.1.1 Menu View

Menu view is the first screen and the first activated component of the game. It shows the user menu buttons such as “Play Game”, “Options” to choose what actions to do. After getting inputs from the user and sending the inputs to the Input Manager, according to the feedbacks of the Input Manager, Game Engine decides which action to do that are placed in Controller. According to the selection of the user and feedback of the Game Engine, this component leads proper part of the Controller subsystem and the game is executed for the request of the user.

### 3.1.2 Game View

Game View is to show different game maps for each level. According to the feedbacks of Input Manager to Game Engine component, Game Engine component selects the interface of the current game map and sends it to the Game View. Therefore, Game View component is updated through Game Engine and provides the user with the interface of the current game map.

## 3.2. Controller: Game Management Subsystem

Controller subsystem that is Game Management Subsystem manages and conducts logic of the game and also handles the logical background of the helper subsystems by taking inputs from user, processing it and sending the feedback to the proper subsystems.

### 3.2.1 Input Manager

Input Manager is a component that takes inputs from the user and sends it to the Game Engine to process the input. After Game Engine processes the information which comes from the Input Manager, it executes the proper actions and creates links between subsystems.

## 3.3. Model: Game Entities Subsystem Interface

Model Subsystem that is Game Entities Subsystem includes the components which handles the data, rules and the logic of the game. This subsystem notifies the game objects that will be updated to controller.

### 3.3.1 Game Manager

Game Manager is the brain of the game. It is the most important component that initializes other important parts. This component handles all the logic of the game. It calls proper components, calls proper functions and decides all the proper actions which game logic requires. It executes the game by processing inputs which comes from the user via Input Manager and executing the actions by linking the other proper subsystems. Therefore, Game Manager can access and activate all the other subsystems and components.

### 3.3.2 Game Object

Game Object component holds the general data which is same for every different game object such as Bonus, Paddle or Ball. This component holds the information which includes the rules about game. For example, information such as the position of the objects is held in the Game Object component and affects the logic of the game. Then, Game Engine reaches the features of the game objects via this component and manages the game according to these features.

### 3.3.3 Persistent Data

Persistent Data component holds the .txt or .jpg files that will be used in the game. Game Engine can reach this part through File Manager. File Manager can update and reach the data which is held by this component that is Persistent Data.

# 4. Low-Level Design

## 4.1. Final Object Design

The diagram below demonstrates our project's final object design.

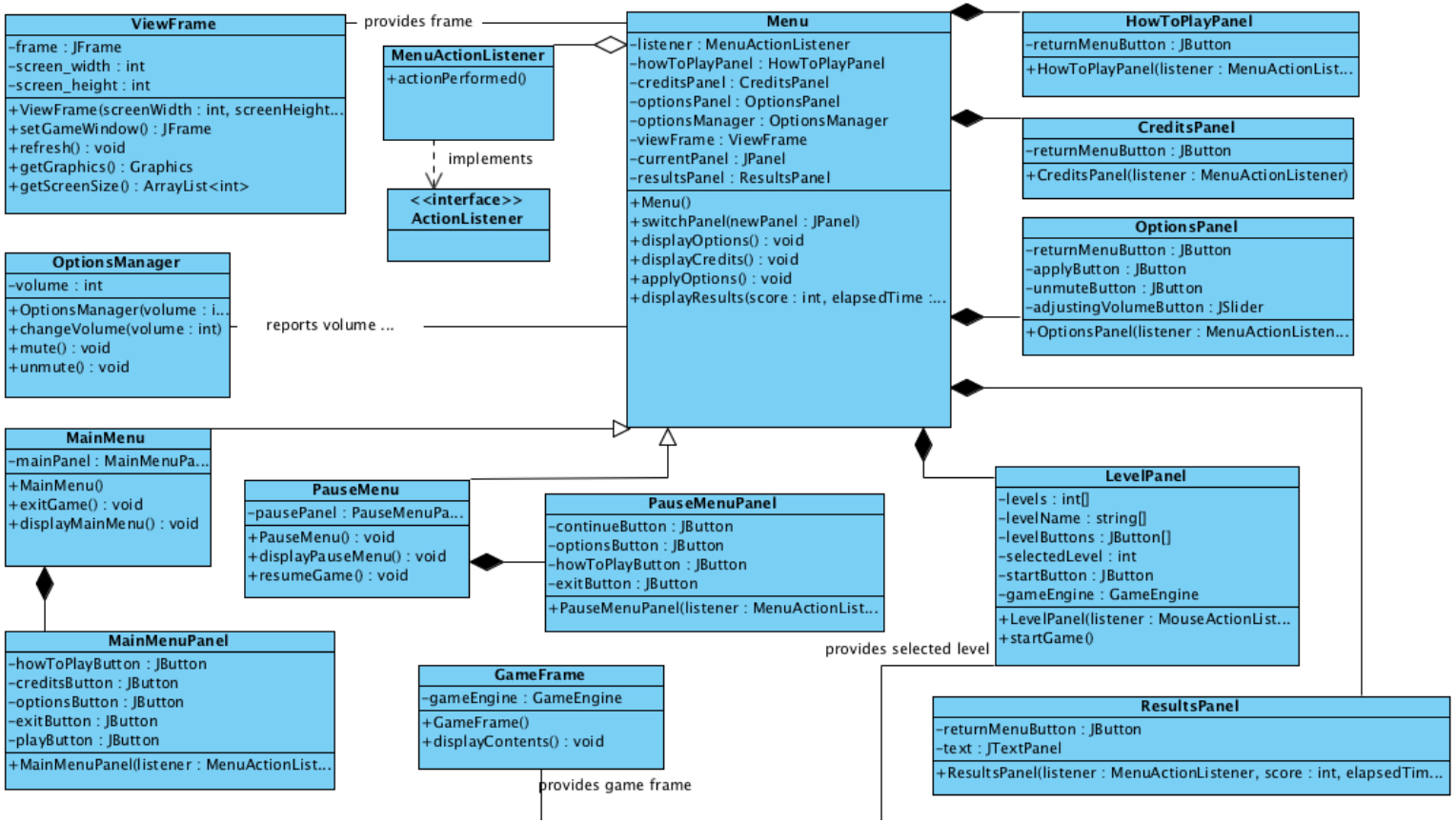



Figure 4.1 : Object Design Part1

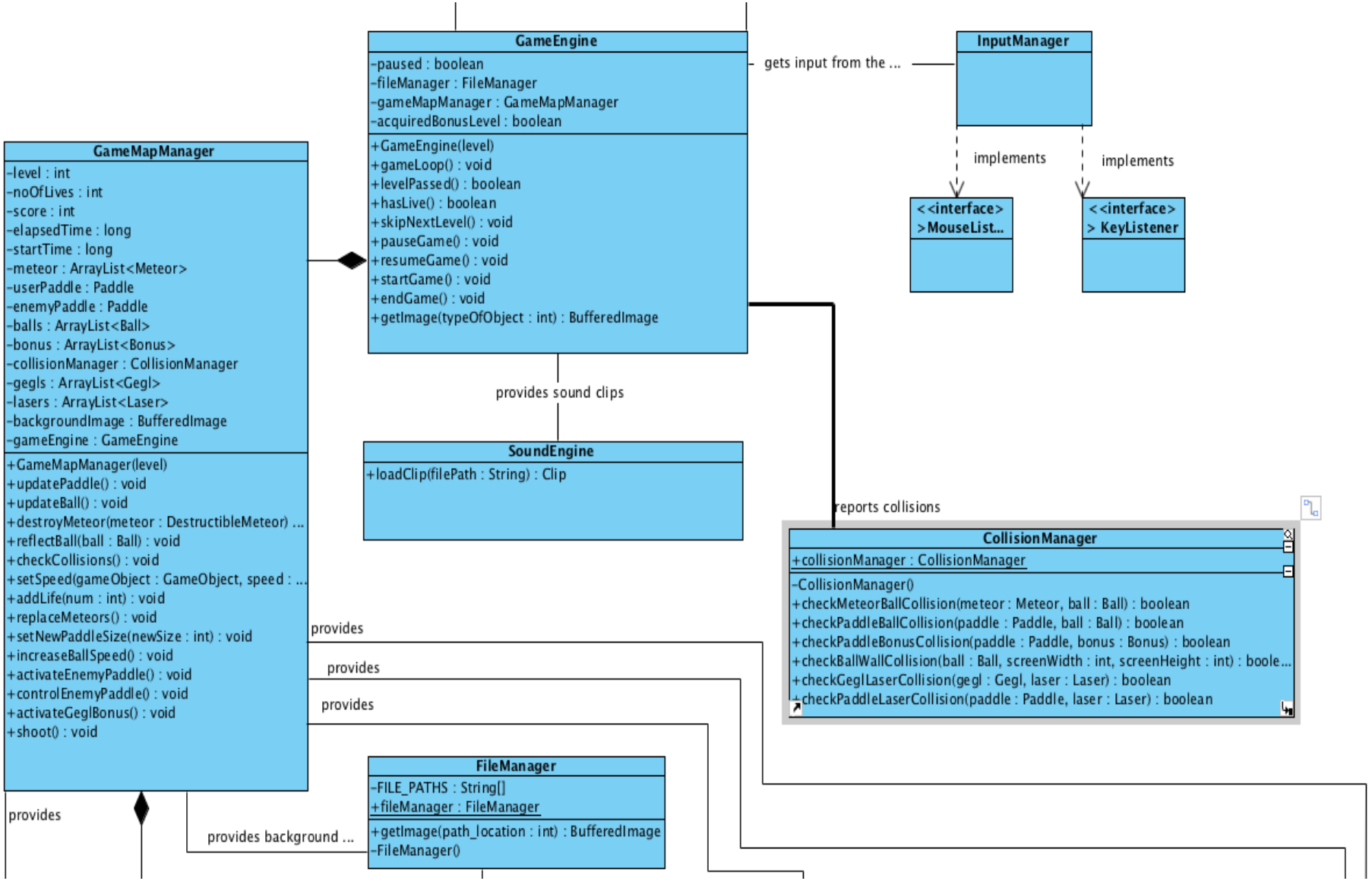



Figure 4.2 : Object Design Part2

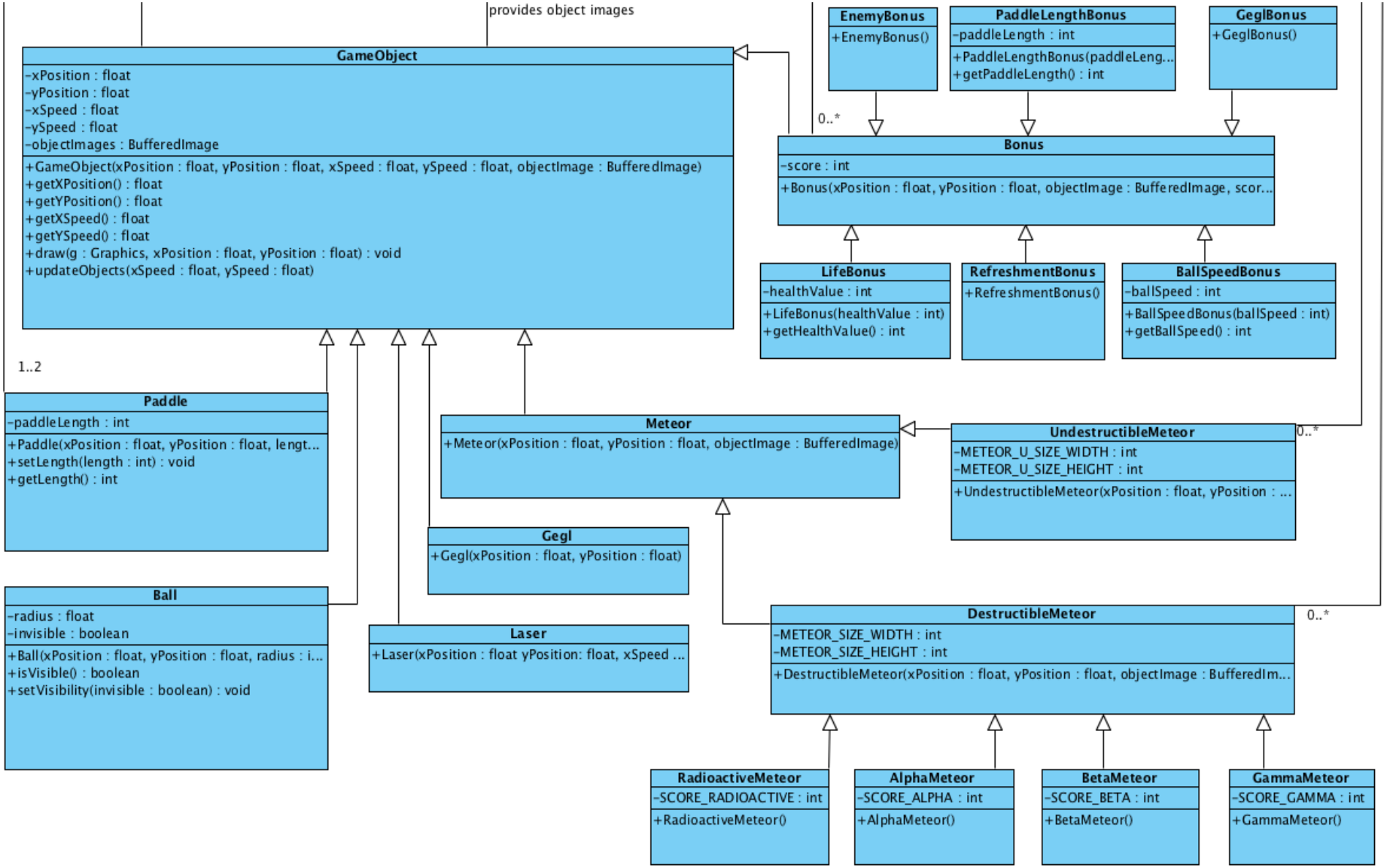



Figure 4.3 Object Design Part3

## 4.2. Object Design Trade-offs

### 4.2.1. Façade Design Pattern

To reduce complexity of the interaction between classes we used façade design pattern for MenuView component. By using façade design pattern we provided set of interfaces for the subsystem. However, using this diagram will increase number of interaction between classes. Changes in interacted classes will also affect this class.

### 4.2.2. Singleton Design Pattern

To increase the maintainability of the game we decided to use singleton design pattern. We used this pattern in CollisionManager and FileManager classes. For CollisionManager, there is only at most game sessions running, so we don't need more than one CollisionManager instance present at a given time. In case of FileManager, if there are more than one instance of that class it means more than one object may be modifying the same data. That may cause corruption in game data and cause program to crash. Since there should be only one instance for these classes, multiple creation of these classes will be unnecessary. On the other hand, singleton design pattern inherently cause code to be tightly coupled. This makes mocking them under test rather difficult in many cases.

## 4.3. Packages

### 4.3.1 Java.util:

We used the Java util library in order to benefit from ArrayLists in project.

### 4.3.2 javafx.scene.layout:

Scene Layout package is used since it provides classes to support user interface layouts.

### 4.3.3 java.awt.Graphics:

Awt Graphics application to draw onto the component or on image.

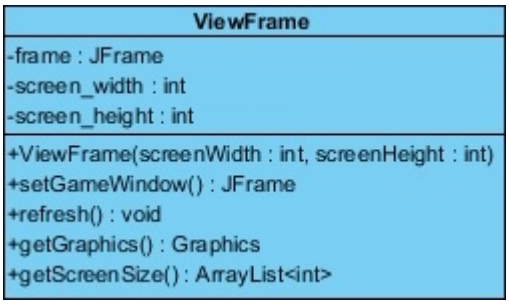
### 4.3.4 java.lang.System:

We used java.lang.System library for measuring time. The calculation will be based on the system time.

## 4.4. Class Interfaces

### ViewFrame Class

This class provides frame for Menu class.




### Attributes:

**private JFrame frame:** The main frame for our program. All visual content, such as menu or game screen will be displayed through that.

**private int screen_width:** Width of the frame.

**private int screen_height:** Height of the frame.

### Constructor:

**public ViewFrame(int screenWidth, int screenHeight):** It takes width and height and creates a frame with given sizes.

### Methods:

**public JFrame setGameWindow():** Returns a JFrame object to display the game screen.

**public void refresh():** Refreshes the frame and applies changes after the latest refresh or load to frame.

**public Graphics getGraphics():** Returns a graphics context. With that graphics, drawing of off-screen images will be enabled.

**public ArrayList<int> getScreenSize():** Size of the frame will be returned as an array list with length two.

## MenuActionListener Class

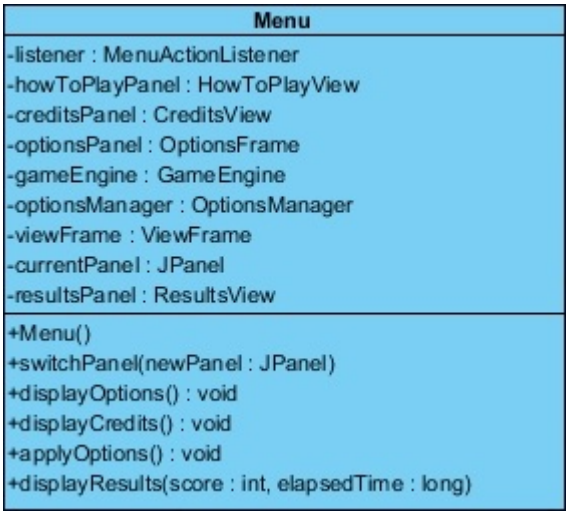
Listens for any user activity on menu. If user presses one of the buttons on the screen or scrolls the sound level bar on settings menu, it will inform the controller accordingly.


### Methods:

**public void actionPerformed():** This method is called after user performs any action.

## Menu Class

This is the main class from which all menu classes inherit. Child classes override display methods according to its type. Also, it will contain the frames for all screens.



## Attributes:

**private MenuActionListener listener:** This listener listens for any activity on menu. In case of an action, it will notify the system accordingly.

**private HowToPlayView howToPlayPanel:** This is the panel for “How To Play” screen. All objects related to that screen will be added to that panel.

**private CreditsView creditsPanel:** This is the panel for “Credits” screen. All objects related to that screen will be added to that panel.

**private GameEngine gameEngine:** That object is the main controller of our game. It controls the game logic and flow of gameplay.

**private OptionsFrame optionsManager:** This is the panel for “Options” screen. All objects related to that screen will be added to that panel.

**private ViewFrame viewFrame:** All panels are put on that frame.

**private JPanel currentPanel:** This variable keeps track of the panel which is currently being displayed.

**private ResultsView resultsPanel:** This is the panel on which results are shown.

## Constructor:

**public Menu():** This constructor creates a new menu without any parameter.

## Methods:

**public void switchPanel(JPanel newPanel):** This method changes the current panel by taking new panel as parameter. This method is invoked as result of user action.

**public void displayOptions():** This method shows options menu on screen.

**public void displayCredits():** This method shows credits section on screen.

**public void applyOptions():** This method applies changes to settings, enabling user to see the difference.

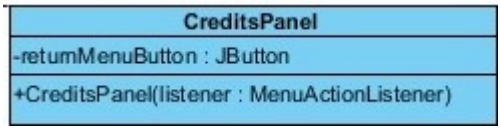
**public void displayResults(int score, long elapsedTime):** By taking score and elapsed time as parameters, it shows results on screen.

## ActionListener Class

This is the general interface for all action listeners. Menu actionlistener used implements that interface. It does not have any attributes, constructors or methods.


## CreditsPanel Class

This is the panel for showing credits section. It also has a button for returning to main menu. This class does not have any methods, apart from a constructor.




### Attributes:

**private JButton returnMenuButton:** This is the button by which user can return back to main menu.

### Constructor:

**public CreditsPanel(MenuActionListener listener):** This constructor creates a new credits panel by taking a listener as parameter. That listener will listen if user presses return to main menu button.

## HowToPlayPanel Class

This is the panel for showing “How to Play” section. It also has a button for returning to main menu. This class does not have any methods, apart from a constructor.


### Attributes:

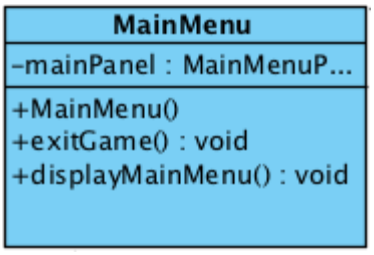
**private JButton returnMenuButton:** This is the button by which user can return back to main menu.

### Constructor:

**public HowToPlayPanel(MenuActionListener listener):** This constructor creates a new panel for “How to Play” section by taking a listener as parameter. That listener will listen if user presses return to main menu button.

## MainMenu Class

This is the class for main menu screen. It has controller for main menu and it also contains the panel to display contents of main menu.




### Attributes:

**private MainMenuPanel mainPanel:** This is the panel with which contents of main menu will be displayed. It also has five buttons which user can press.

### Constructor:

**public MainMenu():** It is the constructor for main menu and it has no parameters.

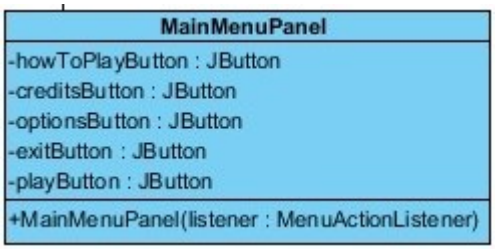
### Methods:

**public void exitGame():** This function terminates the application. It quits the game.

**public void displayMainMenu():** This function sets current frame contents to main menu contents.

## MainMenuPanel Class

This is the panel for main menu. All related components, such as buttons will be put on that panel. Apart from a constructor, it does not have any methods.




### Attributes:

**private JButton howToPlayButton:** If user presses that button, “How To Play” section will be opened.

**private JButton creditsButton:** If user presses that button, “Credits” section will be opened by setting credits panel as the current panel.

**private JButton optionsButton:** If user presses that button, “Options” section will be opened by setting options panel as the current panel.

**private JButton exitButton:** If user presses that button, the program will terminate.

**private JButton playButton:** If user presses that button, a game is started.

### Constructor:

**public MainMenuPanel(MouseActionListener listener):** This listener will keep track of mouse movements of user.

## OptionsPanel Class

This is the panel for options menu.


### Attributes:

**private JButton returnMenuButton:** With this button, user can return to main menu.

**private JButton applyButton:** User can apply changes to settings by pressing that button.

**private JButton unmuteButton:** User can unmute the sound by pressing that button.

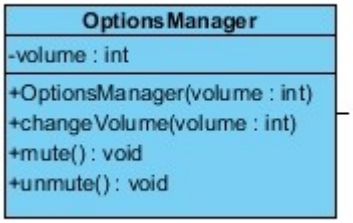
**private JSlider adjustingVolumeButton:** User can adjust volume by sliding it.

### Constructor:

**public OptionsPanel(MenuActionListener listener):** This constructor creates an options panel by taking the listener as parameter.

## OptionsManager Class

This class is the controller for options but at this point, only volume can be changed. According to user inputs, this class adjusts settings.




### Attributes:

**private int volume:** This variable represents the sound level in application.

### Constructor:

**public OptionsManager(int volume):** This constructor creates an options manager by taking initial sound value as parameter.

### Methods:

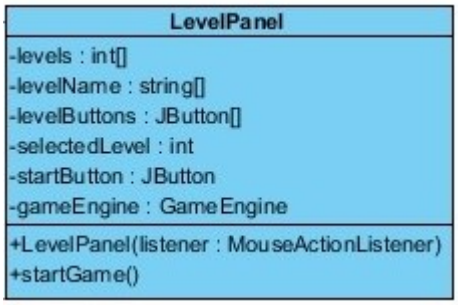
**public void changeVolume(int volume):** This method sets volume level to a specified integer level.

**public void mute():** This method mutes the sound.

**public void unmute():** This method unmutes the sound.

## LevelPanel Class

This class contains both controller and view for level part. It has a connection between GameEngine to start the game for the selected level.



### Attributes:

**private int[] levels:** This variable holds each level selection of the game such as 1,2,3.

**private string[] levelName:** This variable holds the name of the planets means levels. Each number of the levels corresponds to name of the planet such as Neptun, Pluton.

**private JButton[] levelButtons:** This variable holds the button of the planets.
**private int selectedLevel:** This variable keeps the selected level as a number.

**private JButton startButton:** This variable keeps the start button and sends a signal to game engine to start the game.

**private GameEngine gameEngine:** This variable initializes the game engine into this class to start the game for the selected level.

### Constructor:

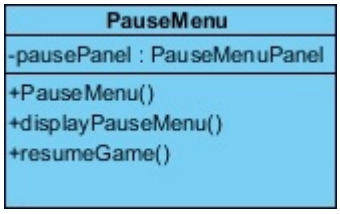
**LevelPanel(listener: MouseActionListener):** This constructor initializes level panel by listening the user selection.

### Methods:

**startGame():** This method is used to start the game for the selected level. It provides selected level to GameEngin.

## PauseMenu Class

This class contains both controller and view for pause menu. For the view part, it has a panel.



### Attributes:

**private PauseMenuPanel pausePanel:** This panel comprises the view part for pause menu.

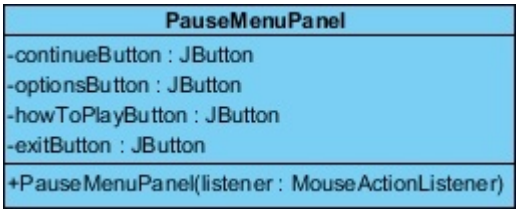
### Constructor:

**public PauseMenu():** This constructor creates a pause menu by taking no parameter.

### Methods:

**public void displayPauseMenu():** This method shows the pause menu by setting pause menu panel as current panel.

## PauseMenuPanel Class




### Attributes:

**private JButton continueButton:** By pressing that button, user can continue playing the game by removing pause panel from the frame.

**private JButton optionButton:** By pressing that button, user can go to settings section by putting settings panel on the frame.

**private JButton howToPlayButton:** “How to Play” section is opened if user presses that button.

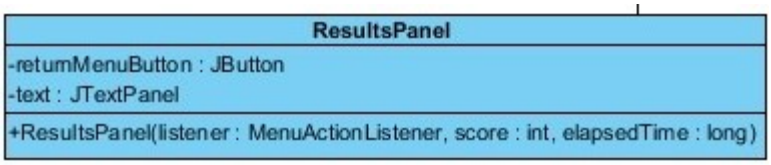
**private JButton exitButton:** Pressing this button will open main menu by removing pause panel and setting main menu panel on frame.

### Constructor:

**public PauseMenuListener(MouseActionListener listener):** This constructor creates a panel for pause menu by taking a mouse listener. That will keep track of user's mouse actions.

## ResultsPanel

This class has a text field and it shows score for current game on that. Score is shown only after game is over. It also has a button for going back to main menu.




### Attributes:

**private JButton returnToMenuButton:** By pressing that button, user can return to main menu as main menu panel will be set to panel.

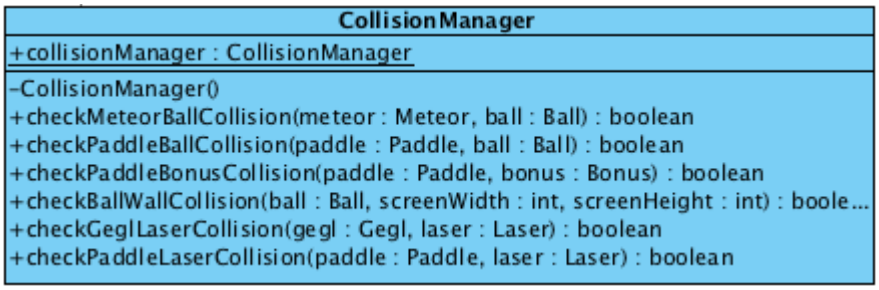
**private JTextPanel text:** This text field shows total score and elapsed time from initializing the game to endgame.

### Constructor:

**public ResultsPanel(MenuActionListener listener, long elapsedTime):** This constructor creates a results view by taking a listener for the button and the time passed from the start of game as parameters.

## CollisionManager Class

This class will provide functions that does maths of the several type of collision between the objects.




## Attributes:

**public static CollisionManager collisionManager:** This attribute created initially for using this class' methods. This is for singleton design pattern.

## Methods:

**public boolean checkBallMeteorCollision(meteor : Meteor, ball : Ball):** This method checks whether there is a collision between the meteor and the ball. If there is a collision this method returns true, otherwise returns false as a response.

**public boolean checkPaddleBallCollision(paddle : Paddle, ball : Ball):** This method checks whether there is a collision between the paddle and the ball. If there is a collision this method returns true, otherwise returns false as a response.

**public boolean checkPaddleBonusCollision(paddle : Paddle, ball : Ball):** This method checks whether there is a collision between the paddle and the ball. If there is a collision this method returns true, otherwise returns false as a response.

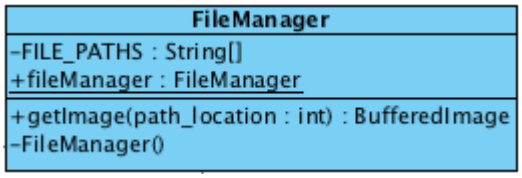
**public boolean checkWallBallCollision(ball: Ball, screenWidth: int, screenHeight: int):** This method checks whether the ball reached to map boundaries. If the ball reaches the map boundaries this method returns true, otherwise returns false as a response.

**public boolean checkGeglLaserCollision(gegl: Gegl, laser : Laser):** This method checks whether there is a collision between a Gegl and a laser. If there is a collision this method returns true.

**public boolean checkPaddleLaserCollision(paddle: Paddle, laser Laser):** This method checks whether there is a collision between the paddle and a laser. If there is a collision this method returns true.

## FileManager Class

This class loads related files from directories. It also keeps background image of the game levels.




### Attributes:

**public static FileManager fileManager:** This attribute created initially for using this class' methods. This is for singleton design pattern.

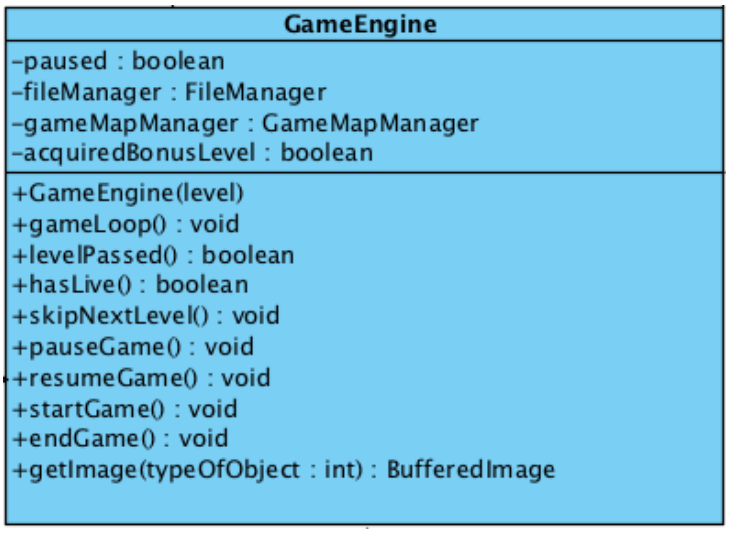
**private static final string[] FILE_PATHS:** This attribute keeps file paths of the object. At each place of string array there is a file path. This array is preloaded.

### Methods:

**public BufferedImage getImage(pathLocation: int):** This method returns related type of objects of image. It finds them from path location.

## GameEngine Class

This class's task is the managing status of the game. It performs actions according to the user requests.



### Attributes:

**private boolean paused:** This attribute keeps status of the game whether the game has paused or not.

**private FileManager fileManager:** This attribute is defined for using FileManager class's functions.

**private GameMapManager gameMapManager:** This attribute is for creating GameMapManager object.

### Constructor:

**public GameEngine():** This constructor initialize the attributes of the GameEngine class for the first run.

### Methods:

**public void gameLoop():** This method runs a loop for the continuity of the game and calls needed functions for next actions.

**public boolean levelPassed():** This method checks the current level condition and decides whether the level has finished or not.

**public boolean hasLive():** This method checks if the user has still life.

**public void skipNextLevel():** This method changes current level to next level and calls needed methods for next level.

**public void pauseGame():** This method pauses the game and stops the loop till the next user instruction.

**public void resumeGame():** This method resumes the game.

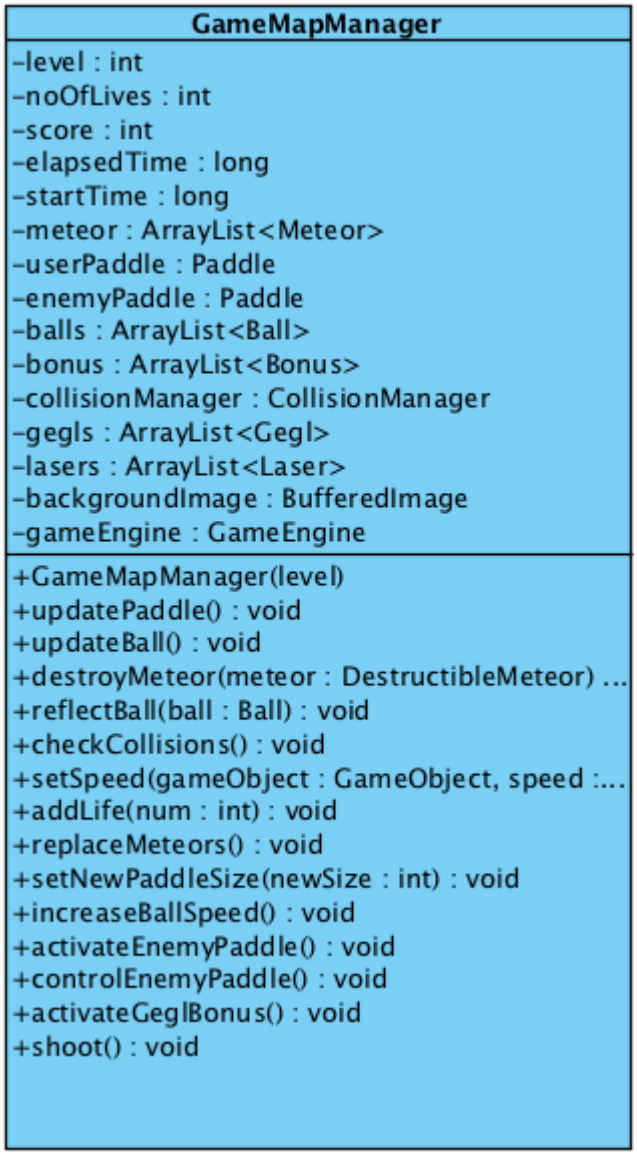
**public void startGame():** This method runs only at the start of the game.

**public void endGame():** This method finishes the game and quit from the game.

**public BufferedImage getImage(typeOfObject: int):** This method returns images that is requested by the object.

## GameMapManager Class

GameMapManager class keeps objects that will be used in the game. It provides functions for game object modification and performs needed actions. This class also keeps progress informations of the game.




### Attributes:

**private int level:** This attribute keeps current level of the game.

**private int noOfLives:** This attribute keeps remaining number of lives.

**private int score:** This attribute is for keeping the user score.

**private long elapsedTime:** This attribute keeps total elapsed time while user playing the game. When game paused elapsed time does not continue to count.

**private long startTime:** This attribute holds starting time of the level. It is updated when a level starts or user returns to the game from the pause condition.

**private ArrayList<Meteor>:** This method keeps the meteors that are currently on the map. It does not keep destroyed meteors.

**private UserPaddle Paddle:** This is an object for user. User controls this paddle.

**private EnemyPaddle Paddle:** This is an object for enemy paddle. This paddle is controlled by the programme algorithm.

**private ArrayList<Ball> balls:** This arraylist keeps ball objects on the map.

**private ArrayList<Bonus> bonus:** This arraylist attribute keeps bonuses on the map.

**private CollisionManager collisionManager:** This collisionManager object will provide related collision functions.

**private ArrayList<Gegl> gegls:** This object arraylist keeps alive gegls in the level. Only used in bonus level.

**private ArrarayList<Laser> lasers:** This object arraylist keeps active lasers in the level. Only used in bonus level.

**private BufferedImage backgroundImage:** This attribute keeps background image of current level.

## Constructor:

**public GameMapManager():** This constructor initialize needed objects for the begging of the map.

## Methods:

**public void updatePaddle():** This method updates speed, length and position of a paddle.

**public void updateBall():** This method updates speed and position of a ball.

**public void destroyMeteor(meteor: DestructibleMeteor):** This method destroys a meteor and removes it from the map.

**public void reflectBall(ball: Ball):** This method does new speed direction calculation of a ball when ball hits an obstacle. It reflects according to collision angle between center of ball and center of meteor.

**public void checkCollisions():**This method test all possible kind of collisions. It calls all methods in the CollisionManager class one by one with all possibilities.

**public void setSpeed(gameObject: GameObjects, speed: int):** This method set speed of an given object to a new speed.

**public void addLife(num: int):** This method increase or decrease remaining life respect to taken bonus.

**public void replaceMeteors():** This method replaces all destroyed meteors. It sets meteors to initial state of the level.

**public void setNewPaddleSize(newSize: int):** This method changes size of a paddle.

**public void increaseBallSpeed():** This method increase speed of a ball.

**public void activateEnemyPaddle():** When a related bonus collected, this method activates enemy paddle.

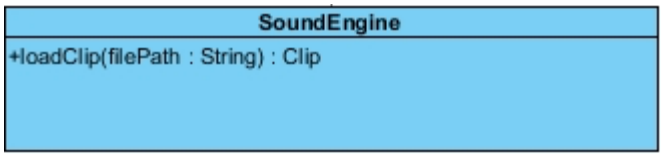
**public void controlEnemyPaddle():** This method gives command to enemy paddle to give direction to a ball.

**public void activateGeglBonus():** This method activates bonus level when related bonus is collected.

**public shoot():** This method creates laser objects in the bonus level.

## SoundEngine

This class plays game sound clips. The game music will be controlled on this class.

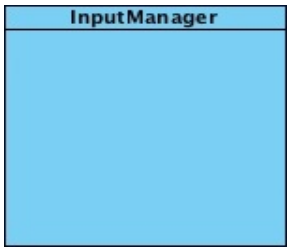



### Methods:

**public Clip loadClip(filePath: String) :** This method gets related sound clip from realated file path and starts to play that clip.

## InputManager

This class receives user inputs from keyboard and mouse, and passes them to game engine. It will then act according to these inputs. This class implements KeyListener and MouseListener interfaces.




## KeyListener

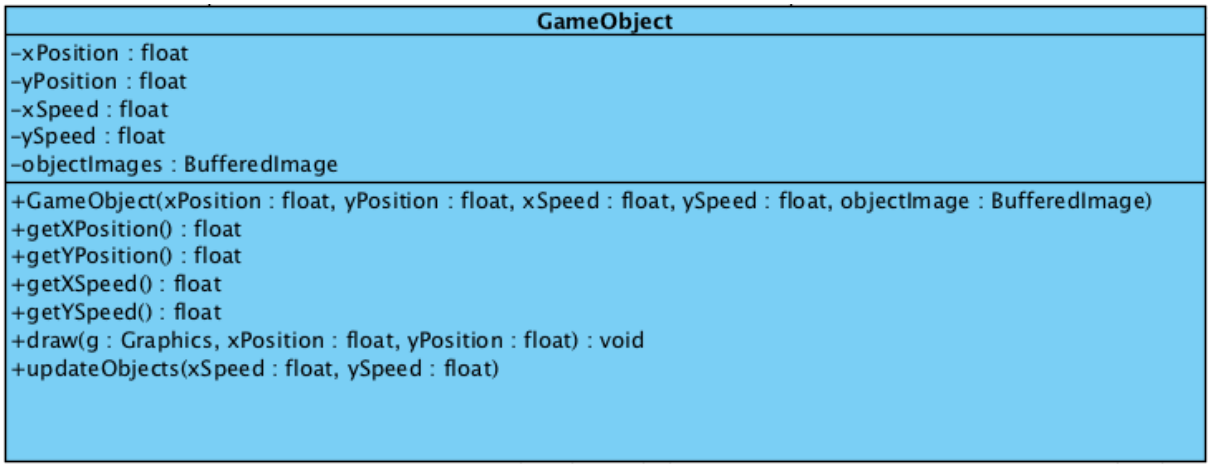
It is a general interface to receive inputs from keyboard and it can be implemented by different classes with separate implementations. In this case, input manager class will provide these implementations.


## MouseListener

It is a general interface to receive inputs from mouse and it can be implemented by different classes with separate implementations. In this case, input manager class will provide these implementations.


## GameObject Class

This class provides a template for all types of objects in a game. This type keeps parameters common to all objects, namely location, object type and image. It also has two-dimensional speed but it will not be used by all classes.




### Attributes:

**private float xPosition:** Float variable for the horizontal position of the game objects.

**private float yPosition:** Float variable for the vertical position of the game objects.

**private float xSpeed:** Float variable of the horizontal speed of a game object in screen.

**private float ySpeed:** Float variable of the vertical speed of a game object in screen.

**private BufferedImage objectImage:** The image of the object to represent in the screen.

### Constructor:

**public GameObject(xPosition: float, yPosition: float, xSpeed: float, ySpeed: float, objectImage: BufferedImage):** Constructor initialises each game object by taking their positions and speeds in float format, and image as parameters to extract from FileManager.

### Methods:

**public float getXPosition():** Public method to get the horizontal position of a game object in float.

**public float getYPosition():** Public method to get the horizontal position of a game object in float.

**public float getXSpeed():** This method returns horizontal speed of a game object in float format.

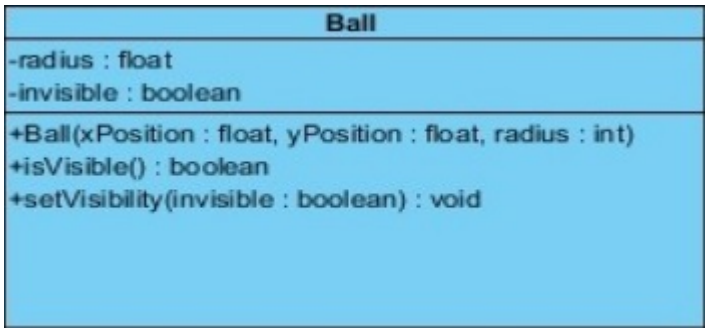
**public float getYSpeed():** This method returns vertical speed of a game object in float format.

**public void draw(g: Graphics, xPosition: float, yPosition: float):** This method draws the instance on a given graphics context at a given location.

**public void updateObjects(xSpeed: float, ySpeed: float):** This method updates the speed of a game object.

## Ball Class

It is an object with which user can hit and destroy meteors. In order not to fail the game, user should not let it fall through the void.




### Attributes:

**private float radius:** It is a size parameter, representing ball's radius.

**private boolean invisible:** It represents whether ball is visible or not.

### Constructor:

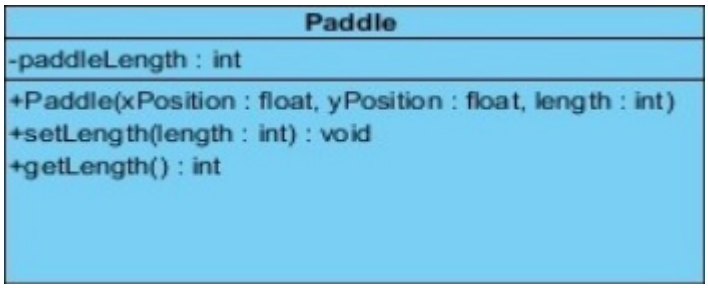
**public Ball(xPosition: float, yPosition: float, radius: int):** This constructor creates a new ball and takes ball's initial location and size as parameters. Initially, a ball is set visible.

### Methods:

**public boolean isVisible():** It returns whether the ball is visible or not.

**public void setVisibility(invisible: boolean):** This method is used to change visibility status of ball.

## Paddle Class




### Attributes:

**private int paddleLength:** This is the length of a paddle instance.

## Constructor:

**public Paddle(xPosition: float, yPosition: float, length: int):** This constructor returns a paddle instance by taking its initial location and length as parameters.

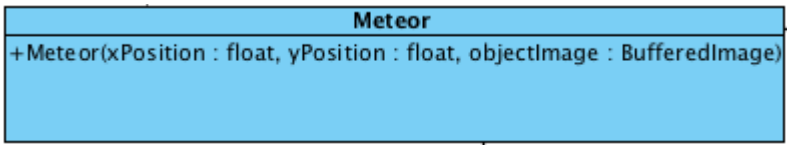
## Methods:

**public void setLength(length: int):** This method sets length of the paddle to a specified integer value.

**public int getLength():** This method returns length of the paddle instance.

## Meteor Class

Meteor Class contains the Destructible and Undestructible meteors.

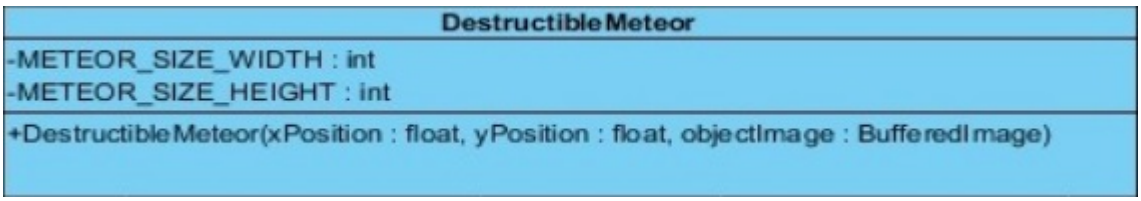



## Constructor:

**public Meteor(xPosition: float, yPosition: float, objectImage: BufferedImage):** Constructs the meteor objects with their position and images according to their type.

## DestructibleMeteor Class

This is the parent class for all meteor types which are destructible. Specific types of meteors inherits from this type.




## Attributes:

**private static final int METEOR_SIZE_WIDTH:** Constant width of the destructible meteor objects in integer format.

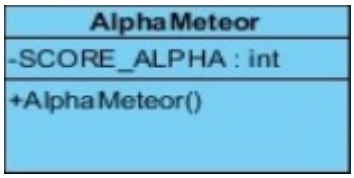
**private static final int METEOR_SIZE_HEIGHT:** Constant height of the destructible meteor objects in integer format.

### Constructor:

**public DestructableMeteor(xPosition: float, yPosition: float, objectImage: BufferedImage):** Constructor initialises destructible meteors by taking their positions in float variables and image as parameters.

## AlphaMeteor Class

This class inherits from destructible meteors. This type of meteors can be destroyed after single hit. After it is destroyed, user will gain a specified amount of score.




### Attributes:

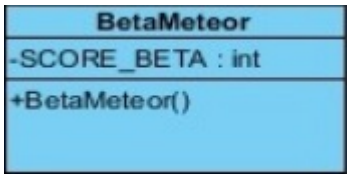
**private static final int SCORE_ALPHA:** Constant integer holds value for the amount of score player gets when an alpha type brick is destroyed.

### Constructor:

**public AlphaMeteor():** Empty constructor class initialises Alpha meteors without taking any parameters.

## BetaMeteor Class

This class inherits from destructible meteors. This type of meteors can be destroyed after two hits. After it is destroyed, user will gain a specified amount of score.

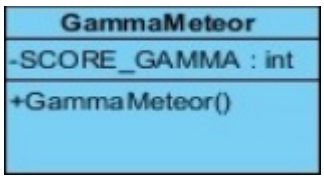


### Attributes:

**private static final int SCORE_BETA:** Constant integer holds value for the amount of score player gets when a beta type brick is destroyed.

## GammaMeteor Class

This class inherits from destructible meteors. This type of meteors can be destroyed after three hits. After it is destroyed, user will gain a specified amount of score.

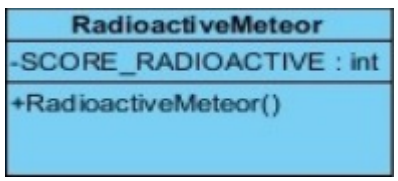



### Attributes:

**private static final int SCORE_GAMMA:** Constant integer holds value for the amount of score player gets when a gamma type brick is destroyed.

## RadioactiveMeteor Class

These type of meteors inherit from destructible meteors. They will destroy surrounding meteors if one of its instances gets hit.

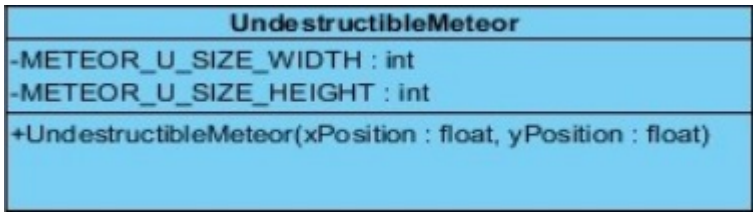



### Attributes:

**private static final int SCORE_RADIOACTIVE:** Constant integer holds value for the amount of score player gets when a radioactive brick is destroyed.

## UndestructibleMeteor Class

This class inherits from game objects. This type of meteors cannot be destroyed even if they get hit.




## Attributes:

**private static final int METEOR_U_SIZE_WIDTH:** Constant integer value of the undestructible meteors width.

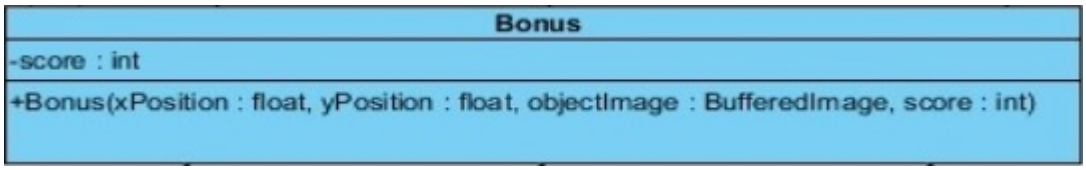
**private static final int METEOR_U_SIZE_HEIGHT:** Constant integer value of the undestructible meteor height.

## Constructor:

**public UndestructableMeteor(xPosition: float, yPosition: float):** This constructor creates an indestructible meteor with given initial location.

## Bonus Class

This class inherits from game objects. This type of objects keep some amount of score and if it gets hit, user will receive that score. After being hit, it will be destroyed.




## Attributes:

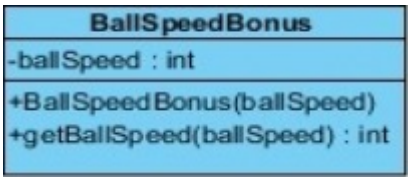
**private int score:** This variable stores the amount of “reward” will user receive after hitting the bonus object.

## Constructor:

**public Bonus(xPosition: float, yPosition: float, objectImage: BufferedImage, score: int):** This constructor creates a bonus instance with given initial location, image and score it will contain. These values are received as parameters to constructor.

## BallSpeedBonus Class

BallSpeedBonus increases or decreases the speed of balls in the screen when it is collected by the user paddle. It changes the speed of the ball in a random manner when obtained.




### Attributes:

**private int ballSpeed:** Value of the speed changes when bonus is taken.

### Constructor:

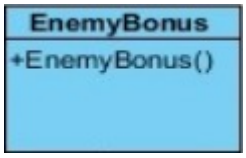
**public BallSpeedBonus(ballSpeed: int):** Constructor takes the speed value to adjust the speed of ball.

### Methods:

**public int getBallSpeed():** This method gets the new speed of the ball.

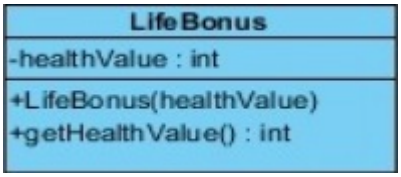
## EnemyBonus Class

EnemyBonus creates a new paddle at the top of the screen when collected. Just like the user paddle, it can reflect the ball. However the user has no control on it.




## LifeBonus Class

LifeBonus can give the player an extra life, or randomly take their life.



### Attributes:

**private int healthValue:** The value of the bonus life to be added.

### Constructor:

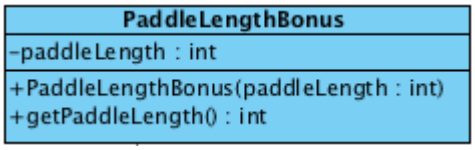
**public LifeBonus(healthValue: int):** Initializes the life bonus with desired health value.

### Methods:

**public int getHealthValue():** Method gets the health value to be added when bonus is obtained.

## PaddleLengthBonus Class

PaddleLengthBonus can increase or decrease the length of the paddle randomly.




### Attributes:

**private int paddleLength:** The integer value of paddle length to be set.

### Constructor:

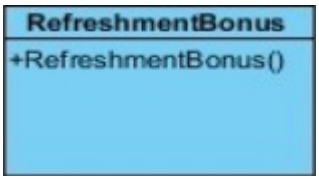
**public PaddleLengthBonus(paddleLength: int):** Initializes the paddle length bonus with adjustable length value.

### Methods:

**public int getPaddleLength():** Method gets the new paddle length to be adjusted when bonus is collected.

## RefreshmentBonus Class

RefreshmentBonus refreshes the meteors in the map as they were at the beginning of the level. However the points and bonuses that the user has collected remains the same.

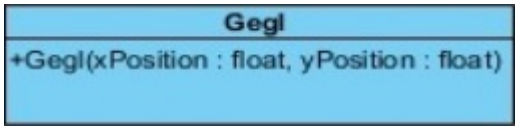



## GeglBonus Class

GeglBonus unlocks a bonus level after finishing the current one. The new level provides a different game mechanic which includes lasers and gegls. It can be thought as a Space Invaders level.


## Gegl Class

Gegls are created at the top of the screen and move downwards slowly. They shoot lasers, which can make the user lose their life. They die when shot with users laser.

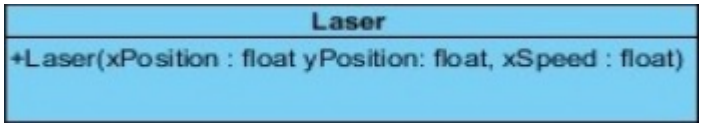



### Constructor:

**public Gegl(xPosition: float, yPosition: float):** The constructor initializes Gegls at their desired locations.

## Laser Class

Laser is used in the bonus level to shoot gegls, and also they can shoot the player with lasers they shoot. They move vertically and disappear at collision.




### Constructor:

**public Laser(xPosition: float, yPosition: float, xSpeed: float):** Initializes the laser object with its initial position and vertical speed.

# 5. Improvement Summary

Improvements in the second iteration can be grouped into three. Firstly, we changed our design to fit MVC architecture. Our initial object design and subsystem decomposition were wrong in that sense. Secondly, we added new classes to low-level design for the new functionality for game. Thirdly, we fixed errors regarding organization of the report. These are changing nested numbering and moving class descriptions to low-level design.

- We have changed our MVC design by carrying the GameEngine from Control group to the Model.
- We modified our subsystem decomposition to properly reflect the MVC architecture. We grouped the components into Model, View and Controller and we were notified that some of the interactions should not exist or wrong. Our current model fits the MVC pattern correctly. Also, we simplified our diagram by focusing on components, not classes. We removed components which are composed of only one class.
- We modified the organization of report by fixing nested numberings of sections. Also, we moved class descriptions to under low-level design section.
- We added two design pattern, Singleton and Facade to our low-level design and modified our diagrams accordingly.
- We no longer hold the object types as integers.
- We have added Gegl, GeglBonus and Laser classes for the new bonus level.
- Bonus types (RefreshmentBonus, PaddleLengthBonus, BallSpeedBonus, EnemyBonus and LifeBonus) are identified and explained in the game entities.
- Instead of holding image of objects as attribute in each class, now GameObject handles the BufferedImage according to the object (except for Bonus and DestructibleMeteor, which have more than one image).

# 6. Glossary & References

https://docs.oracle.com/javase/8/docs/api/