Bilkent University

CS 319
Object-Oriented Software Engineering Project
Design Report

More Dangerous Dave

Section 1 / Group 03

Ahmet Namlı

Halil Selman Atmaca

Muhammed Emin Tosun

Mert Sebahattin Kutluca

Instructor: Bora Güngören

4.3)Packages/Subsystems

1. Introduction

1.1.Purpose of the System

"More Dangerous Dave" is a side viewed 2D video game. Game's purpose is similar to other arcade, adventure games. Elementarily, "Dave" should defeat his enemies by using some collectable items and get more score by collecting coins. Also, "More Dangerous Dave" improved by adding some modern and more complicated features. That is to say, "More Dangerous Dave" one move ahead of other arcade games and the "Dangerous Dave" game which inspired us.

1.2.Design Goals

Most important and principal aim of games is amusing and entertaining the player. Because player's aim is spending funny times. To make it, developers should focus on some little details. These details will put our game in front of other games. As it is understood, these details won't attract notice on them at first glance.

1.2.1 User

Well-Defined Interface:

User interface is the most important part of game to interact with user. When it is well-defined and easy on the eye, game will have got a head starts on user's impression. In menus, we'll implement a lot of well-defined buttons to make understandable. Also, help menu will help it. So, we'll guide player at every stage.

Ease of Use

In a game, easiness of play is an important issue. When player play the game, player can control the character easily. For example, if character move with keys which located on keyboard separately, player cannot reach a smooth play. We'll provide this smoothness. Also, help menu will help about keys and gameplay.

Performance

Performance of the game very effective on gameplay quality. So, we'll make "More Dangerous Dave" as a high-performance game. First of all, the passes between the maps will be smooth. "More Dangerous Dave" will run perfectly without any faults. These will make the game at better quality.

1.2.2 Maintenance

Understandability

In every part of our project, we take care about understandability for anyone who uses and analyses it. To achieve it, we design the project very simple and clearly. Especially code parts of project will be clear and well-commented.

Modifiability

When changes in project are needed, change in one part won't influence the other parts. To make it, we should keep coupling at low levels. Then, change in one part, it won't affect the whole program. To make modifiability higher, we should have some subsystems and these subsystems share few data.

Good Documentation

To make our project more understandable and clear, we will make good documentation. Connections between reports will be provided. Also, completeness of them will make project more understandable.

Extensibility

In the program lifecycle, there is an important issue as ability of adding or removing features. This provides highly maintainable software. Our project will be easy to extend. For example, map files will be read from outside. Because of that we can easily change and extend it.

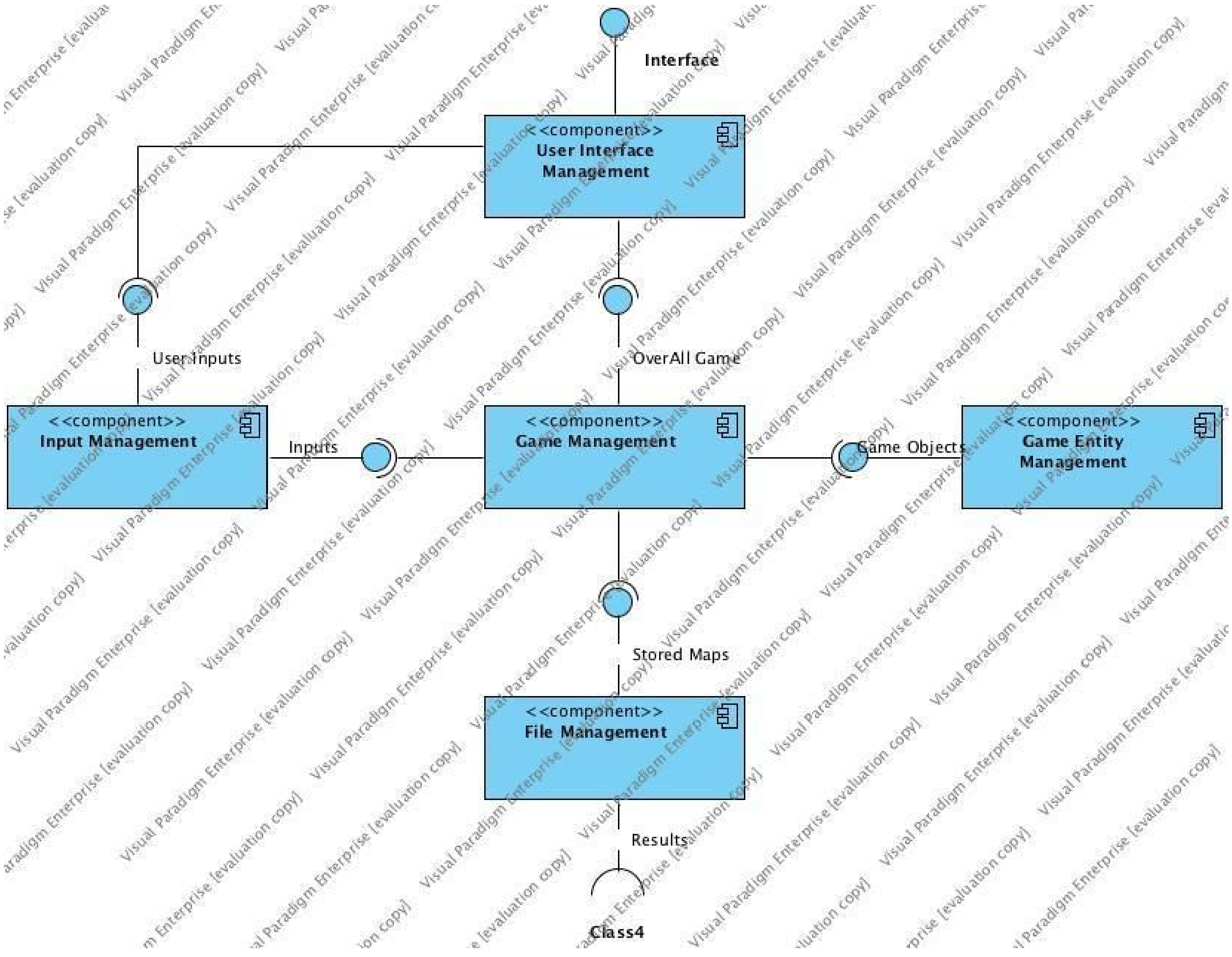
1.2.3 Trade-offs

Development time vs. Performance

To provide a high performance, we should spend more time than a project which has lower performance. We compensate this time by using Java as programming language. Because Java offers better GUI libraries. Also, Java don't have any memory leak problem.

2. Software Architecture

2.1 Subsystem Decomposition



We are using 3-tier(layer) architectural style. In this style, there are 3 layers. Components can share data to a package below itself or at the same layer. In the first layer, we have User Interface Management subsystem. This subsystem organizes the interaction between program and user. What user sees is organized by this layer. In the second layer, we have 3 subsystems. The first subsystem is Input Management subsystem. This subsystem gets inputs from User Interface Management subsystem and delivers to Game Management subsystem. The second subsystem in this layer is Game Management subsystem. This subsystem organizes game's logic. All collision checks, map loadings or saving game jobs are done by this subsystem. The third subsystem is Game Entity Management subsystem. We have all game objects in this subsystem. Object management is done by this subsystem. In third layer, we have only File Management subsystem. This subsystem is responsible for managing the data which needs to be stored. For example, keeping and getting high scores are two of this subsystem's jobs.

2.2 Hardware/Software Mapping

We are using Java in order to implement our project. The PC needs to have the latest version of Java Runtime Environment to play our game. Our game will require a keyboard to give directions to the character. User also needs to have mouse to choose menu options. To save high scores, we will use .txt file. It is also supported by almost all operating systems. The game won't need internet connection because we are developing this software as an offline game.

2.3 Persistent Data Management

In this software, storage is only needed for keeping high scores and keeping game maps. So, a simple file management system is enough for this system. We do not need to use SQL database system. This subsystem will handle read and write jobs for high scores and game maps.

2.4 Access Control and Security

Because the software is offline, we do not have any issue about access control and security. Therefore, we do not need to implement them.

2.5 Boundary Conditions

Initialization: When user starts the game, program will need to load high scores, game maps, needed images and music files to run program smoothly. When user runs the executable for our program, all files will be loaded and game will be initialized.

Termination: When user closes the game, all jobs will be terminated. Because the game only has one thread, after everything terminated game will exit.

Errors: The main aim of our project is supplying the user clear gameplay. To do this, we are going to clear errors as much as possible. If program faces errors such as failing to find correct JDK, program will show convenient message for this error.

3. Subsystem Services

3.1 Façade Design Pattern

Our design pattern is Façate design pattern. We have a façate class called GameController which manages all system. The purpose of design pattern is providing an efficient interface with logical class hierarchy. For example, beginning scene is created in menu classes, which get the information to select next scene. Thanks to objects created by classes, information can easily be carried from a scene to another scene.

3.2 Entities

There are Bullet class, Player class, Enemy class, Coin class, Diamond class, Chalice class, Health class, Jetpack, Gun, Blade classes in the character component. Some these components got by player in the game process. These components also have connection to a document which is used to store data. Using this document there is a relationship between levels.

3.3 Input Manager

Input manager gets commands from mouse and keyboards and gives the required information to GameController.

3.4 Collision Manager

Collisions are determined, handled in GameController class by using handleCollision() method.

3.5 User Interface

MainMenu class shows the main menu and it has buttons and their functions. Help class is responsible for informing user. This class is assigned to show the game graphics, for example, Dave, Boss, diamonds, coin… These 4 classes are a child of scene class. GuiManager class has fundamental methods of the game and UI.

3.6 File Manager

There is a file which stores required information to memorize. When data is needed, it is pulled from this file. High scores, map objects and game object images are stored in this file. There are methods, which are writeHighScore(), getImage(). readHighScore() led to pull high scores data from this file. Read map pulls map and images are also pulled by this file by using getImage().

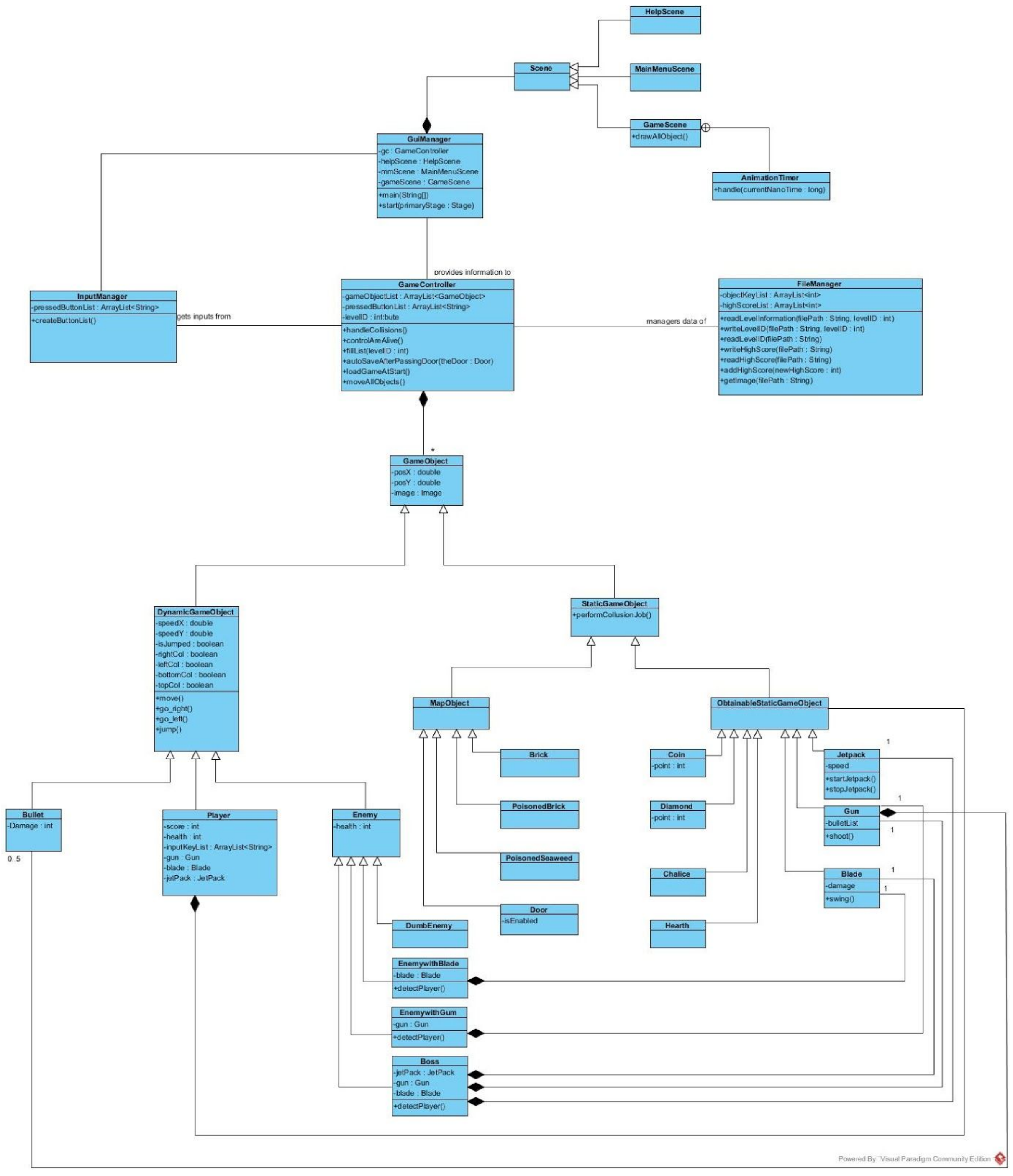
4) Low-level Design

4.1) Object Design Trade-Offs

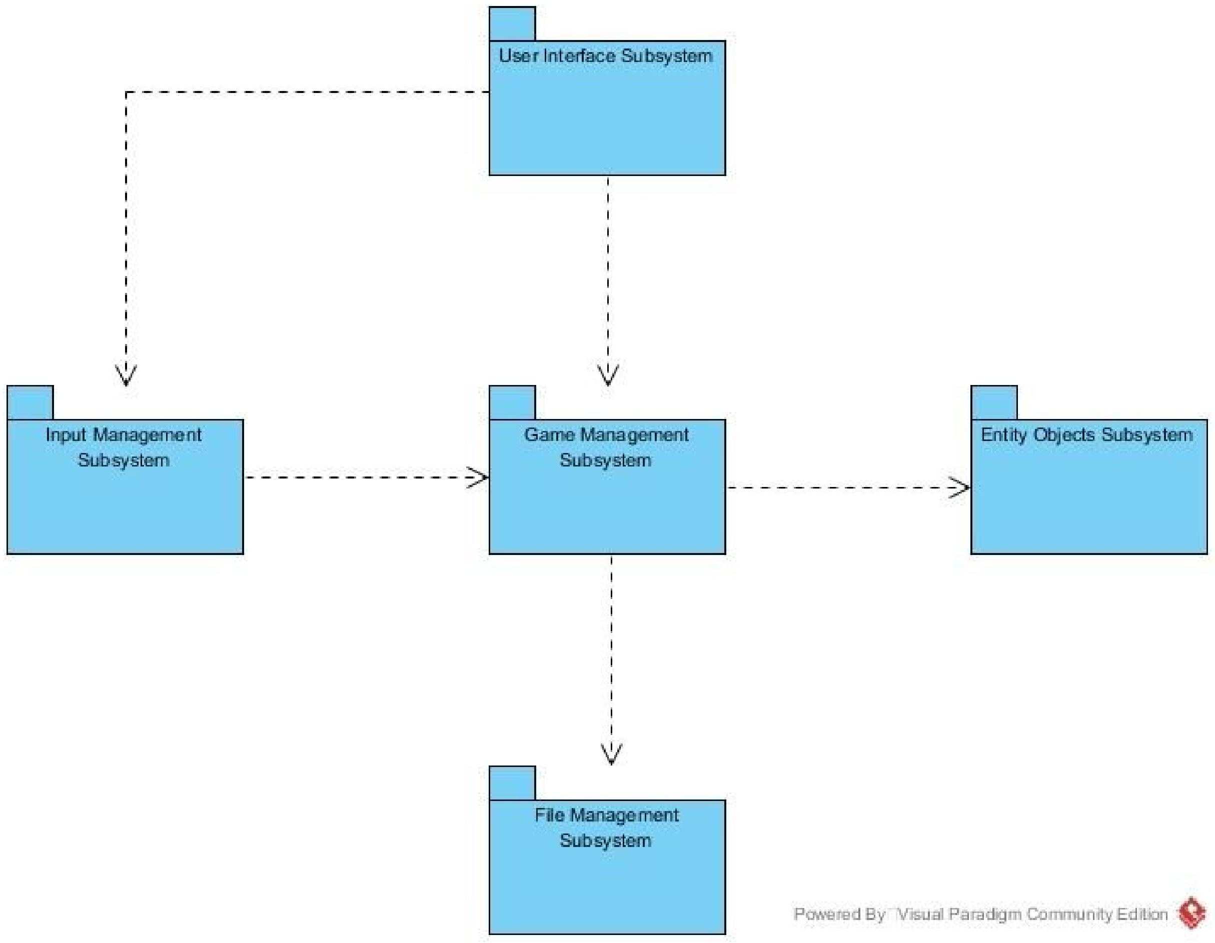
Some design goals are more important than others so we will have some trade-offs in our implementation. Our software will not check two static game objects whether they collide. This will reduce ease of writing code because we can write one method for collisions if we check all object. The software only check needed objects, this will increase run-time efficiency.

Our second tradeoff is similar to first one. It is between runtime and ease of programming. We load all game objects in init phase. This uses more memory and decrease runtime efficiency but coding it is simple.

4.2) Final Object Design

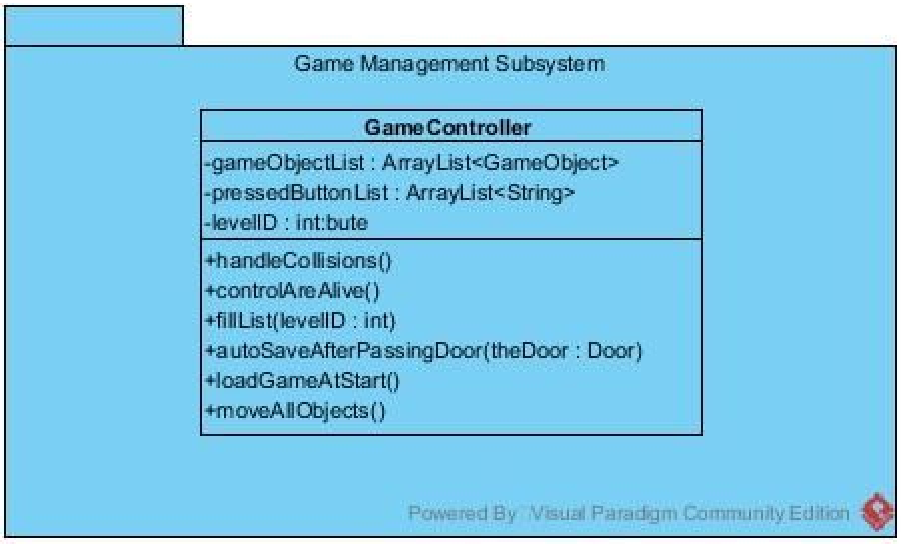


4.3) Packages/Subsystem


User Interface Subsystem
Input Management Subsystem
Game Management Subsystem
Entity Objects Subsystem
File Management Subsystem
Powered By Visual Paradigm Community Edition

Game Management Subsystem

Game Manager Subsystem coordinates the all other subsystems. It takes keyboard inputs from Input Management Subsystem and level information from File Management subsystem then it construct game objects according to these information from Game Objects subsystem. Then, User Interface subsystem draw all these objects.


Game Management Subsystem
GameController
-gameObjectList : ArrayList<GameObject>
-pressedButtonList : ArrayList<String>
-levelID : int:bute
+handleCollisions()
+controlAreAlive()
+fillList(levelID : int)
+autoSaveAfterPassingDoor(theDoor : Door)
+loadGameAtStart()
+moveAllObjects()
Powered By Visual Paradigm Community Edition

GameController Class

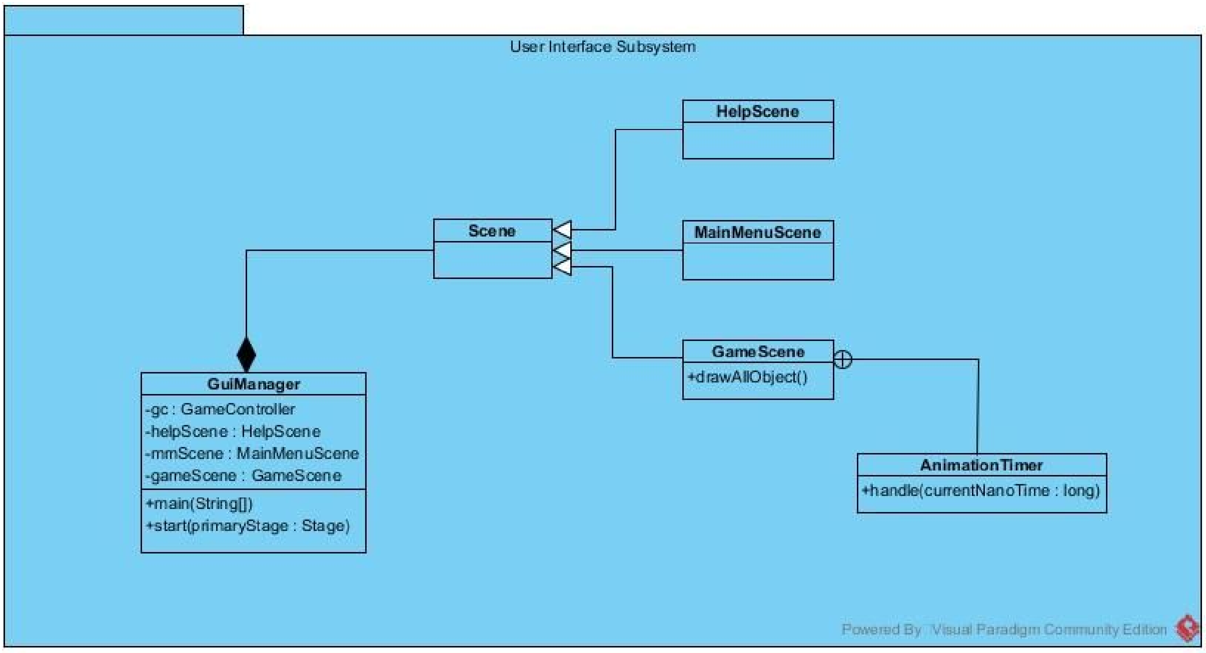
- ➔ **Attributes**
  - ❏ **gameObjectList : ArrayList<GameObject>:** This list includes all game objects. User Interface Subsystem reads this list and draw all of them. When player or an enemy is killed. This object is removed from list.
  - ❏ **pressedButtonList : ArrayList<String>:** This list is filled by Input Manager Subsystem and includes name of what key is pressed.
  - ❏ **levelID : int:** This integer is id of the current level. Its default is zero but it change in initialization phase by File Manager Subsystem.

- ➔ **Methods**
  - ❏ **controlAreAlive() :** It check health of all Dynamic Game Object if there is an object whose health is less than zero, it removes this object from GameObjectList.
  - ❏ **fillList(levelID : int) :** It reads information of related level via File Manager Subsystem, then constructs game objects and add them into the gameObjectList.
  - ❏ **autoSaveAfterPassingDoor(theDoor : Door) :** This method is triggered by collisionHandler method. It takes a door object as a parameter. Door object has level id of next level. This id is stored via File Manager Subsystem.

- ❑ **handleCollisions() :** This method includes nested loops and checks whether dynamic game objects and static game is collided, additionally it checks bullet object with all other objects.
- ❑ **moveAllObjects() :** This method calls move method of all dynamic game object. Exceptionally, player object takes pressedButtonList as a parameter.

User Interface Subsystem

This subsystem is responsible for showing game objects on screen and also show main menu and help menu.



GuiManager Class

This class has the main method of the project.

➔ **Attributes**

- ❑ **gc : GameController :** controls entire game.
- ❑ **helpScene : HelpScene :** shows help scene.
- ❑ **mmScene : MainMenuScene :** shows main menu scene.
- ❑ **gameScene : GameScene :** This scene shows the game.

➔ **Methods**

- ❑ **main(args : String[]) :** main method of the project.
- ❑ **start(primaryStage : Stage) :** this method is a special method of javaFX. It construct scene objects.

Scene Class

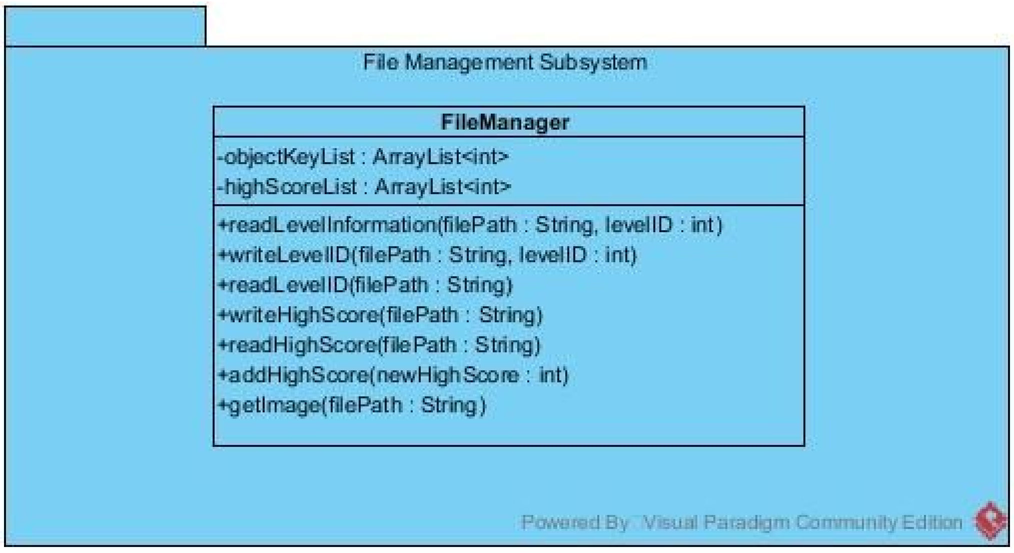
This class is parent class of HelpScene, MainMenuScene and GameScene classes.

GameSceneClass

This class includes a special class of javaFX library which is AnimationTimer. AnimationTimer also has a special method named handle(). This method constitutes our game loop.

File Management Subsystem

File Management subsystem is liable to load and store data. It loads game objects data from virtual memory and it can also store last level id to save.



File Management Subsystem
FileManager
-objectKeyList : ArrayList<int>
-highScoreList : ArrayList<int>
+readLevelInformation(filePath : String, levelID : int)
+writeLevelID(filePath : String, levelID : int)
+readLevelID(filePath : String)
+writeHighScore(filePath : String)
+readHighScore(filePath : String)
+addHighScore(newHighScore : int)
+getImage(filePath : String)
Powered By Visual Paradigm Community Edition

FileManager Class

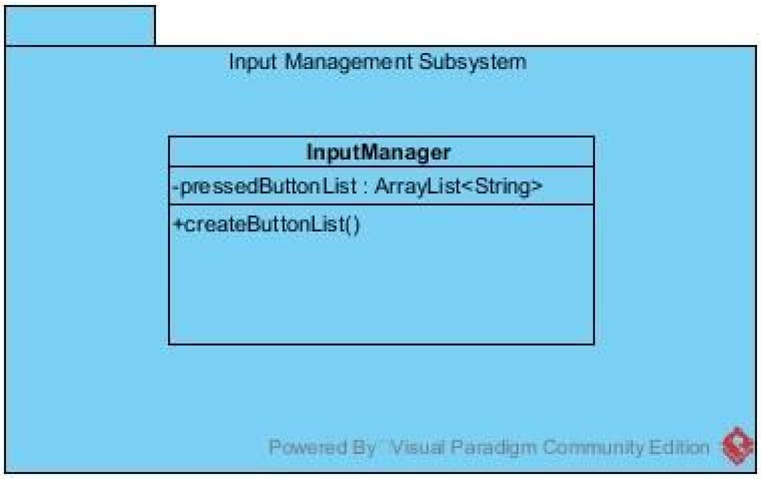
- **Attributes**
  - **objectKeyList : ArrayList<int>:** Each key in this list symbolize one object.
  - **highScoreList: ArrayList<int>:** Each key in this list symbolize one object.

- **Methods**
  - **readLevelInformation(filePath : String, levelID : int):** This method reads data in the path according to levelID and put them into a list.
  - **writeLevelID(filePath : String, levelID : int):** It writes last level's ID into a text file.
  - **readLevelID(filePath : String) : int :** It reads last level's ID from a text file.
  - **writeHighScore(filePath : String) :** It writes high score information into a text file.
  - **readHighScore(filePath : String) :** It reads high score information from a text file.

- ❑ **addHighScore(newHighScore : int) :** It adds new high score into high score list.
- ❑ **getImage(filePath : String) :** It reads image information in file path.

Input Management Subsystem

Input Management Subsystem is a bridge between user and Game Management Subsystem.


Input Management Subsystem
InputManager
-pressedButtonList : ArrayList<String>
+createButtonList()
Powered By Visual Paradigm Community Edition

Input Manager Class

➔ **Attributes**

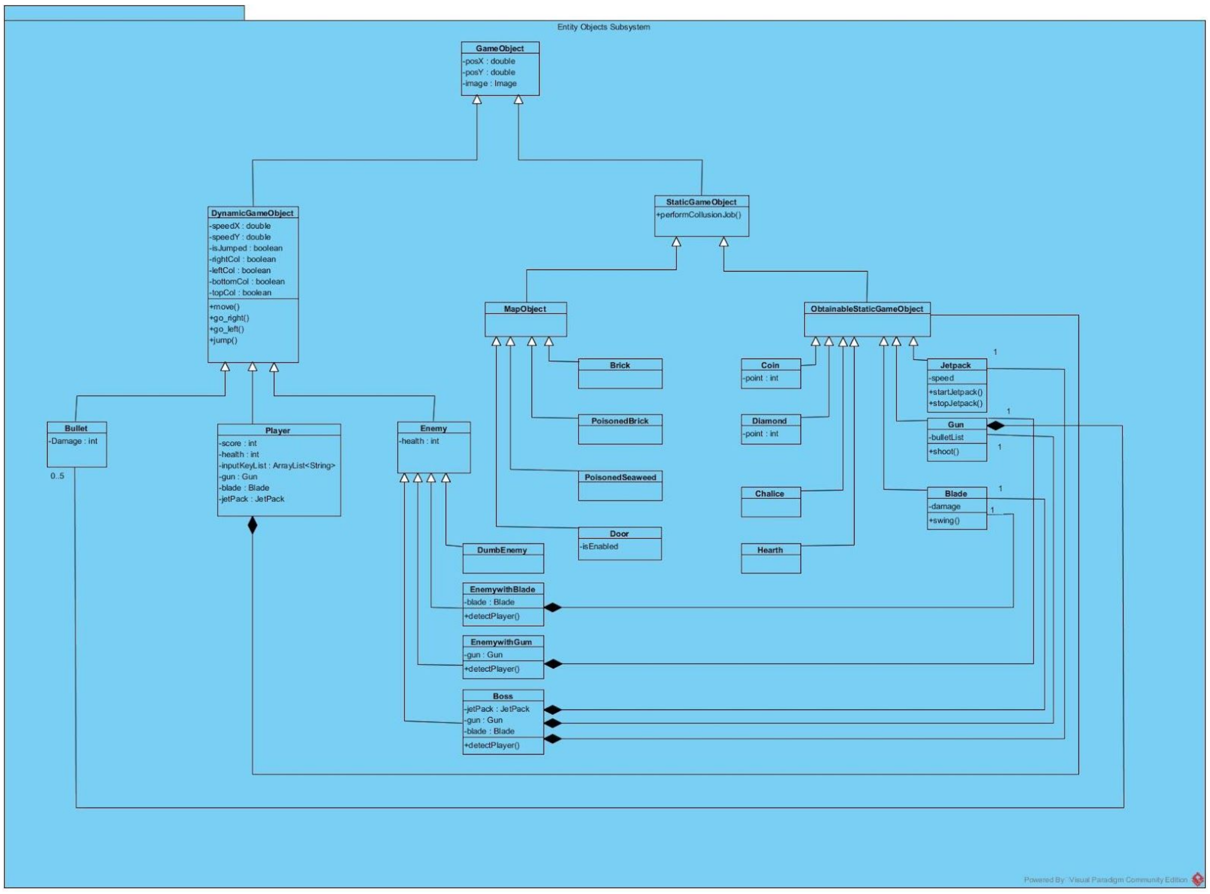
- ❑ **pressedButtonList: ArrayList<String> :** Includes the pressed buttons name.

➔ **Methods**

- ❑ **createButtonList() :** fills and clear pressedButtonList via key listener.

Entity Objects Subsystem

This subsystem includes all entity objects of the software.



GameObject Class

This class is parent of all other entity classes.

➔ **Attributes**

- ❏ **posX : double :** x coordinate of the object
- ❏ **posY : double :** y coordinate of the object
- ❏ **image : Image :** image object of javaFX library

DynamicGameObject Class

➔ **Attributes**

- ❏ **speedX : double :** position change in unit time in x coordinate
- ❏ **speedY : double :** position change in unit time in y coordinate
- ❏ **isJumped : boolean :** this boolean is false at the beginning, when object jump this becomes true, then it collides with floor, this boolean becomes false.
- ❏ **rightCol : boolean :** this boolean says there is an collision from right side.
- ❏ **leftCol : boolean :** this boolean says there is an collision from left sideç

- ❑ **bottomCol : boolean :** this boolean says there is an collision from bottom. In game map there is a gravitation. All dynamic object starts to fall if this boolean is not true.
- ❑ **topCol : boolean :** this boolean says there is an collision from top.

➔ **Methods**

- ❑ **move() :** this method is an abstract method. Children classes have their move method and they override this method.
- ❑ **go_left() :** this method increases the posX accordingly speedX.
- ❑ **go_right() :** this method decreases the posX accordingly speedX.
- ❑ **jump() :** this method set speedY and speedY change according to gravitation after jump method call.

Bullet Class

This object is child of DynamicGameObject. When it collide with a GameObject terminated by Game Management Subsystem if this object is a DynamicGameObject, its health reduces as damage.

➔ **Attributes**

- ❑ **Damage : int :** If this object collides with a DynamicGameObject, its health reduces as damage.

➔ **Player Class**

The main character of the game.

Attributes

- ❑ **score : int :** keeps the score of player
- ❑ **health : int :** health of player, when it less than zero, game ends.
- ❑ **inputKeyList : ArrayList<String> :** This keeps the pressed buttons. move method of player use this.
- ❑ **gun : Gun :** Initially this is null, when player collides with this in map, this object is constructed. The gun has five bullet.
- ❑ **blade : Blade :** Initially this is null, when player collides with this in map, this object is constructed.
- ❑ **jetPack : JetPack :** Initially this is null, when player collides with this in map, this object is constructed. When this object is activated, gravitation does not affect player.

Enemy Class

➔ **Attributes**

- ❑ **health : int :** health of enemy object.

Boss Class

Player encounters with boss at last level of the game and when player defeat the boss the game ends.

- **Attributes**
  - **jetPack : JetPack :** Boss can easily run away from bullets of player via this object.
  - **gun : Gun :** When the boss detect player, starts to shoot towards player via gun.
  - **blade : Blade :** Boss swing the blade when player near to him.

- **Methods:**
  - **detectPlayer() :** This method check whether there is a player close to him.

StaticGameObject Class

- **Methods**
  - **performCollisionJob() :** This is an abstract method that says what will be done when a collision occur with this object.

MapObject Class

MapObject class is parent class of Brick class, PoisonedBrick class, PoisonedSeaweed class and Door class. These classes are building blocks of the map. When a dynamic object collides with them, their performCollisionJob() methods is triggered. Their performCollisionJob() methods overrides StaticGameObject's. When a dynamic game object collides with Brick, it stops; when it collides with a poisonedBrick or poisonedSeaweed it lose health; when it collides with a door, it go over next level.

ObtainableStaticGameObject

ObtainableStaticGameObject class is parent class of Coin class, Diamond class, Chalice class, Health class, JetPack class, Gun class and Blade class. When player class collides with these objects, these objects is terminated and their performCollisionJob() method is triggered. If player collides with Coin and Diamond, his point increases; it he collides with Chalice, the Door become enabled; if it collides with Health, player's health increases; if player collides with JetPack, Gun or Blade, these objects is constructed inside player.