Bilkent University

Object Oriented Software Engineering Project

Project short-name: Mr.&Mrs. Pac-Man Ext.

Design Report

Ecem İLGÜN, Aziz Osman KOZHAN, Başak Şevval EKİCİ, Talha ŞEKER

Supervisor: Bora Güngören

Progress Report

Oct 21, 2017

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object Oriented Software Engineering Project course CS319.

Contents

1. Introduction

1.1 Purpose of the System

Mr.&Mrs. Pac-Man Extended is 2D arcade game which is based on classical Pac-Man game. Therefore, the main purpose of the game is to eat all food on the map without colliding with ghosts. In addition to that, our game has new features such as new food, shield option and creating your own map. By these new features, interest of players will be sustained. This system will be constructed by considering object-oriented design techniques

1.2 Design Goals

The main purpose of a game is to entertain player(s). To do that, throughout the project, we have to focus on details which do not directly have an effect on the system. Moreover, with the help of object-oriented design, we aim making a project which is easy to extend. This part details the design goals of the system such as end user criteria, maintenance criteria, performance criteria and trade-offs.

1.2.1 End User Criteria

Usability: From user's perspective, a game should be easy to use and learn. To provide this, we do not change commonly used keyboard buttons to move Pac-Mans. First user plays with arrows and other one plays with W, A, S, D combination. Additionally, the purpose is to avoid confusing GUI design; therefore, GUI will have simple design. On the main menu, there is help section to adapt user to new features. User can realize what to do with suitable icons and prompts. Shield, pause and help panel will appear as pop-up for simplicity. Therefore, user does not have to pass windows each time.

Performance: For a good game experience, we try to overcome input lags and low fps. For example, user can hit a ghost due to keyboard input lag and this is frustrating. To achieve this, we focus on writing code as efficient as possible. The game is chosen to be applied in a 2 dimensional world not to experience rendering problems.

1.2.2 Maintenance Criteria

Extensibility: The game is designed in a way that allows developers to add new features and components to meet the user expectations. If deemed necessary, new shields, food or ghosts can be added. For now, two players can play the game locally; however, in the future, number of players can be incremented. All of these are beneficial for player experience.

Modifiability: The system designed will be centered around a multi layered architecture. We have chosen three-tier architecture: Presentation tier includes user interface parts, logic tier makes logical decisions and calculations, and data tier stores and fetches data. Adapting three-tier architecture model allows us to modify each subsystem easily, without affecting any of the remaining ones.

Reusability: Subsystems can be used in other games or similar systems because the system is designed adhering to multitier architecture. Therefore, subsystems can carry out their functionality with other subsystems.

Portability: We have chosen to implement the game in Java. Java provides Java Virtual Machine (JVM) which is runnable in many operating systems. This feature of Java increases portability of the game.

1.2.3 Trade-Offs

Portability – Performance & Memory: In this project, we will use Java programming language which is runnable in many operating systems. This decision results in increased portability of the game.

However, it also means that Java will first compile the program into byte code and then has to call its interpreter, Java Virtual Machine, to interpret the code into machine code. Including this middle process in machine code generation, namely using JVM, may incur in performance penalty.

A second trade-off occurs in memory usage. Java's object model requires a lot more space. For instance, it makes use of wrapper classes for some of the basic types in C++ (e.g. int vs Integer wrapper class). These wrapper classes carry the overhead of an Object class. Therefore, Java uses more resources and makes less function with more code compared to C++.

Functionality – Usability: At first glance, new features decrease usability. With each new feature, user encounters something different from the basic Pac-Man game and getting used to new features takes time. Therefore, since we have aimed to decrease orientation time of a new user, there will not be too much functionality which might cause confusion. However, the game might be quite boring without new features. This made us to add the new features mentioned in 'Purpose of the System'.

As a result, our purpose was to reach a middle ground between functionality and usability. We have added new features to increase functionality, but have refrained from making the game logic or the UI too complicated for the user to play/get used to.
Reusability – Performance: Multitier architecture will be followed to increase reusability of the system and its components. However, this design increases function calls; therefore, our choice of architecture will have a negative effect on overall performance.

2. Software Architecture

In this part, we will examine composition of our system. As it has been mentioned before, the system will be constructed by smaller subsystems. In this way, we can prevent huge chunk of codes and codes can be ordered systematically. In our project, we will use Model View Controller (MVC) design which is suitable for this kind of game project.

2.1 Subsystem Decomposition

In this section, we will demonstrate subsystems roughly. Detailed information about subsystems is in third chapter. While deciding subsystems, we aim loosely coupling each system and followed three-tier architectural style. Throughout the project, some code blocks will need to be changed; therefore, we do not couple subsystems strictly. Otherwise, all subsystems are affected due to a change in one subsystem. Additionally, such a design is beneficial to develop the game in the future. Briefly main concerns are modifiability and extensibility. Our system consists of three subsystems: GUI Layer, Game Logic Layer and Data Layer

- GUI Layer will have classes which is responsible for providing user interface and having interaction with user.
- Game Logic Layer will carry out all decisions and calculations.
- Data Layer will be responsible for protecting data about high scores and maps.

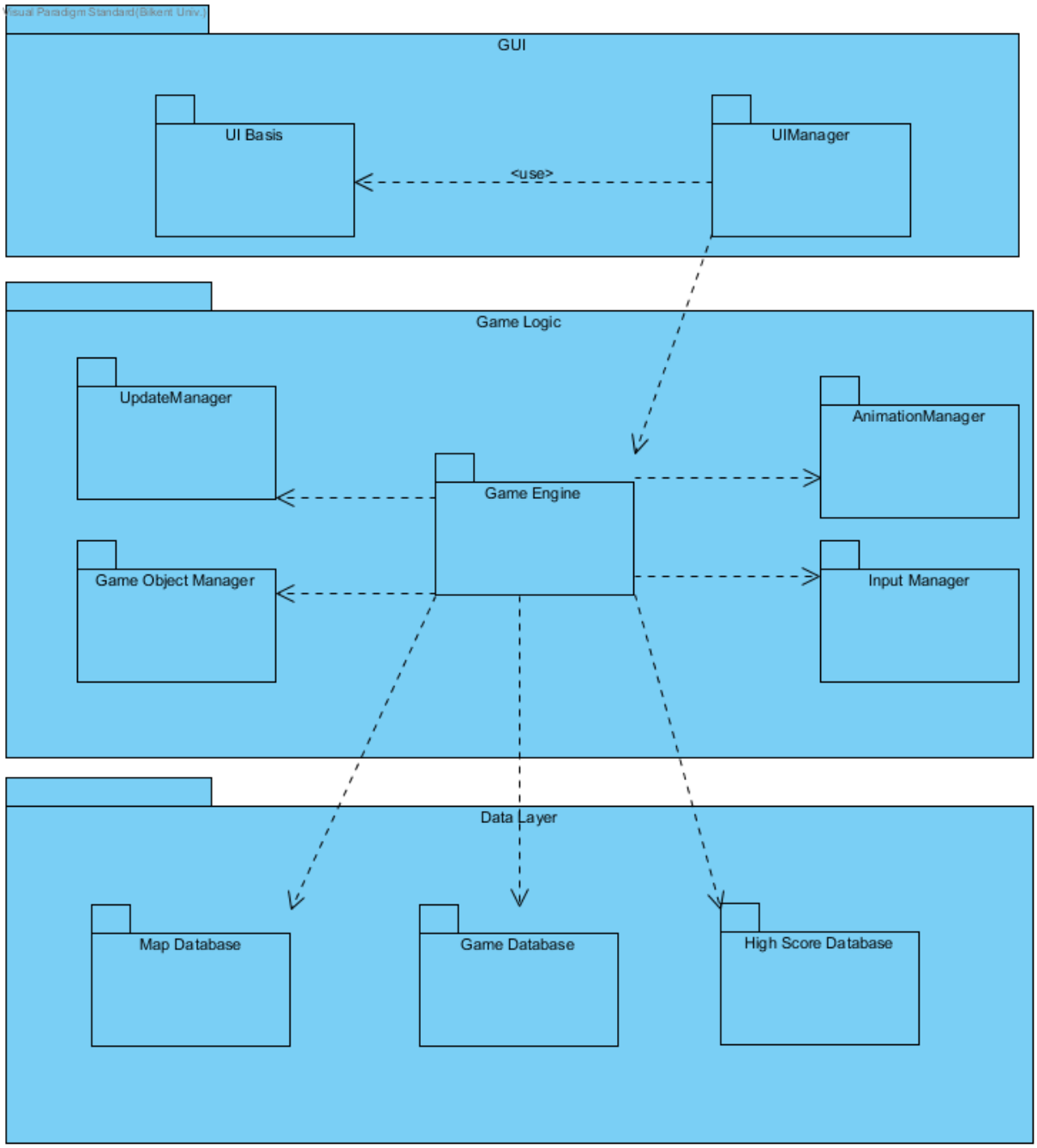

Figure 1

2.2 Hardware/Software Mapping

Our game will be implemented by using Java programming language. All user interface and graphics components will be constructed by using Java libraries. Most computers can run this game because it does not contain any 3D rendering and intense calculations. Our game requires a bit of memory because in data tier layer the system will make some operations such as saving high scores, map and game. To give inputs, basic mouse and keyboard will be enough. Due to these simple hardware and software

requirements, this game is runnable in many computers and in many operating systems such as Windows, Linux and mac.

2.3 Persistent Data Management

In our game high scores, created map and current position of a game level are persistent identity that needs to be stored after each game. We will use local database for all of them because our game will not have any network connection. It means that user can only see high scores which obtained his/her own computer. By using file system, data will be written into txt. files.

2.4 Access Control and Security

Our game does not require any authentication or account because all data stored locally and do not require any private information from user. Therefore, the project does not require any access control or security.

2.5 Boundary Conditions

Execution

Mr.&Mrs. Pac-Man will be launched by double clicking .jar file of the game. The game does not need any special software other than Java Runtime Environment.

Termination

Scenario 1: Player clicks exit button on the pause menu.

Scenario 2: The game can be terminated by using Alt+f4 combination or clicking X icon on the window (for Windows).

Failure

Scenario 1: If there is an error with database system, a prompt will be appeared and user will not be able to save what s/he wants to save.

Scenario 2: If electricity or computer problem happen, the game will not give any opportunity to save the last situation. The user will lose all progress.

3. Subsystem Services

In this section, there are detailed expressions of subsystems.

3.1 GUI Layer

GUI Layer contains all user interface components. These components will be contained in user interface package. In this package, there will be main menu screen, help screen, pause menu screen, high scores screen, create map screen, game over screen, game screen, shield panel and name request screen.

If user clicks play game, game screen will be activated and GUI Layer and Logic Layer will communicate with each other to carry out logic. If first or second level is achieved, shield panel will appear as pop-up and again these two layers interact. If user saves a map, Logic Layer interacts with Data Layer to store the map.

3.2 Game Logic Layer

Game Logic Layer is responsible for carrying out main dynamics of the game. The main class of this layer is GameEngine class. GameEngine class is responsible for implementation of game preparation, start game, pausing and saving game. It also controls and holds level, number of players, score and number of lives Pac-Mans. GameEngine executes main processes; however, while doing that, it uses some helper subsystems. One of them is InputManager system which provides interaction between user and GameEngine. InputManager takes arrow keys; W, A, S, D combination and Esc key as input. Another significant subsystem is UpdateManager which provides a basis for frame life cycles. UpdateManager is mainly responsible for cycle and object management and action handling. For example, UpdateManager handles the situation where Pac-Man hits a ghost. Another subsystem is AnimationManager which handles all the animation related cases of game such as movement animation of Pac-Man.

3.3 Data Layer

Data Layer contains high scores database, map database and current game database. Map database saves created maps and names. High scores database saves high scores and names. Current game database saves last situation of a saved game.

4. Low Level Design

4.1 Object Design Trade-Offs

Designing the objects need some principles applied in order to get rid of the possible bugs in the program and also make the program in the most efficient way. In game, we design the objects in a reusable way to deal the complexity. We also keep the data background in an uncompressed form and this leads to a space-time tradeoff.

4.1.1 Reusability of the Objects

The program is divided into three layers, which are complex subsystems itselves. The subsystems are connected in suitable forms and there is a hierarchicy in between the classes.

4.1.1.1 Inheritance

Using inheritance between classes had reduced the complexity of the program and the amount of code would have written. Extending helps to have the main properties and abilities of the super class and use them in another object, which is based on the super class but formed in a more complex and specific object. For example in the game, the game panel designed in a grid form so that every object needs to have coordinates, an image of the character etc. So we created a Pacman Object that will be a base to every character object in the game. For example Pacman is extended to the Pacman Object, but has characteristic methods and properties (Figure 1). Similarly, in Figure 2, it can be seen that Food extends to Pacman Object and some other types of Food, which are Green, Yellow or Big Food, also extend to the Food class.

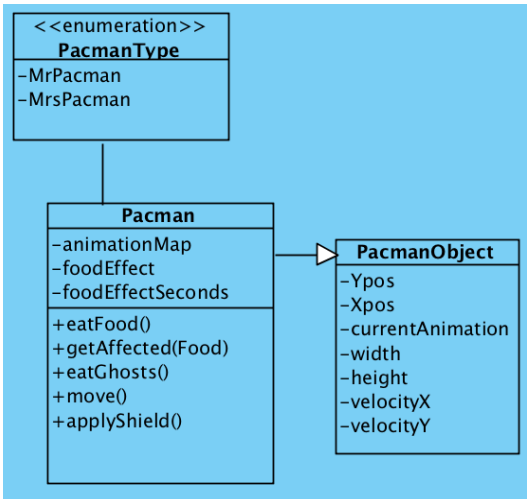

Figure 2

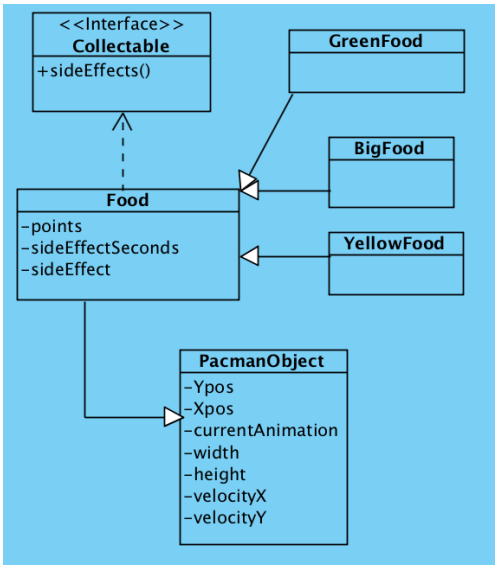

Figure 3

4.1.1.2 Interface

To have a reusable object design, interfaces are an essential component of the system. It helps decoupling separate components of the program and provides a polymorphic behavior. The usage of interfaces in our program is good for breaking up the complex designs. Although Java interfaces are slower and more limited than the other ones, using interface clarifies the dependencies between the classes of the game. In figure 2, it can be seen that the class Food uses an interface Collectable, which is also used by the Food-extended classes such as Yellow, Green or Big Food.

4.1.1.3 Polymorphism

Polymorphism is another important object oriented programming concept that we had used in our game. It allows the game objects to take on many forms. We used polymorphism in our project and by overriding the methods of parent class we were able to call the same method for all children of that class.

4.1.2 Space-Time Trade-Offs

Data storage brings some problems with itself. In our program, the storage method caused the tradeoff. We store the data of the game in an uncompressed way, which takes more space. But by not compressing data, the program does not run the decompressing algorithm and the program runs in less time, which is more practical in our case. To reduce the

transmission time and so the cost at the expense of CPU time to perform the compression and decompression, we used the uncompressed form of the data.

4.1.3 Encapsulation

Programmers use encapsulation to prevent clients accessing program data when there is no need. Hence, we can prevent possible bugs. However, performance of the system is negatively affected due to extra procedures as a trade-off. To implement encapsulation, we made all attributes and operations which should not been accessible and not necessary to be accessed by client private.

4.2 Detailed Object Design

Detailed class diagram showing GUI, Data Layer and Game Logic subsystems is provided in the next page in order to provide an overlook of the project.

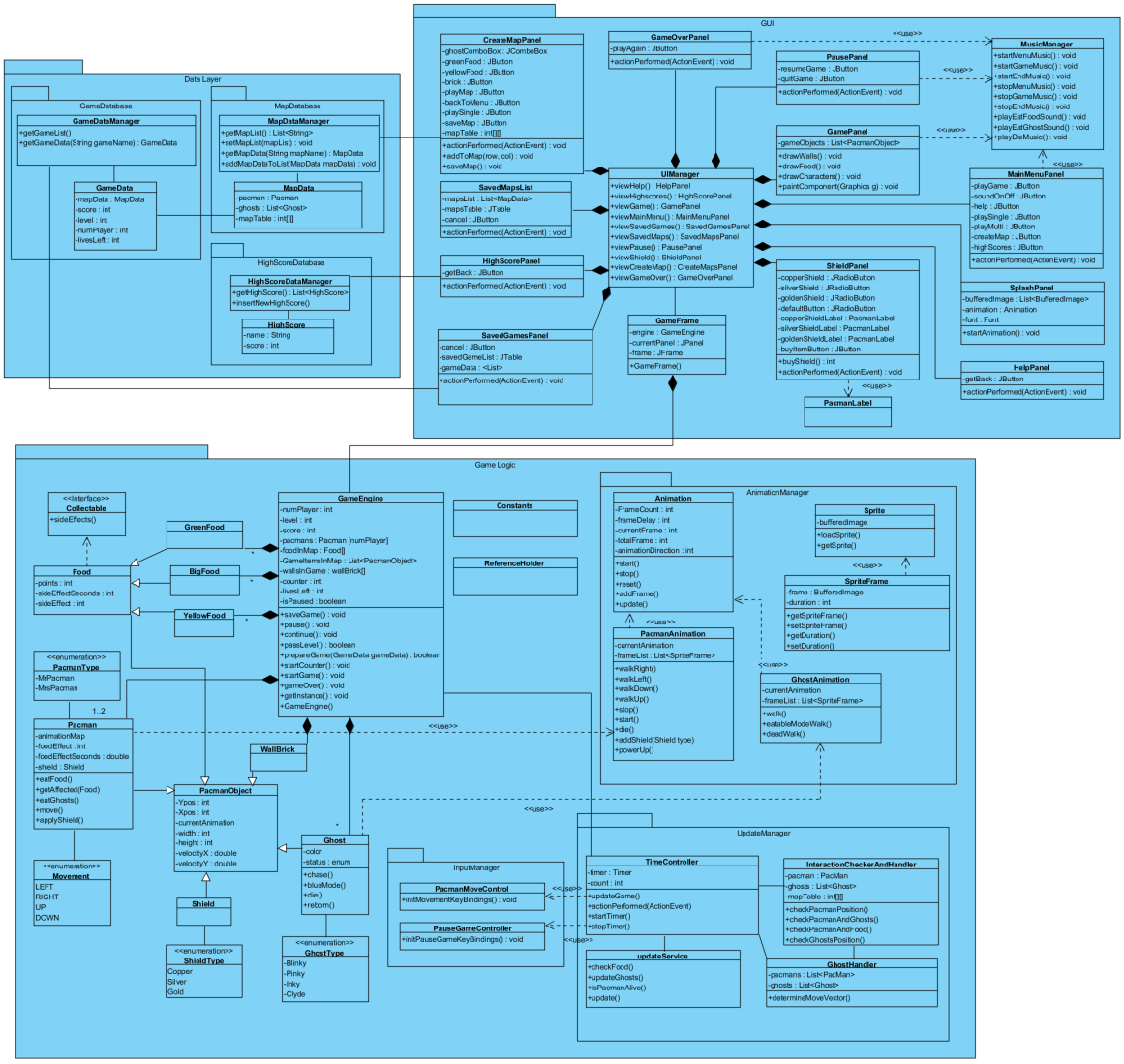

Figure 4 – Object Design Diagram (interfaces are currently excluded for readability)

4.3 Packages

4.3.1 GUI Package

This package is used to provide our subsystem with graphical components, including the game frame and its panels, which are used to provide the user a different screen for each scenario mentioned in our analysis report.

The package includes two crucial classes: UIManager and GameFrame.
UIManager is GUI Package's *service class* which provides GameFrame with any kind of panel the user input may require, e.g. GameOverPanel will be invoked if user fails the game.
GameFrame, the other crucial class, includes an instance GameEngine and a currentPanel which will be initialized to the panels provided by UIManager. It is the *core class* of GUI Package. It is used to interact with the Game Logic layer (via GameEngine), which is analized in the next section (4.3.2).

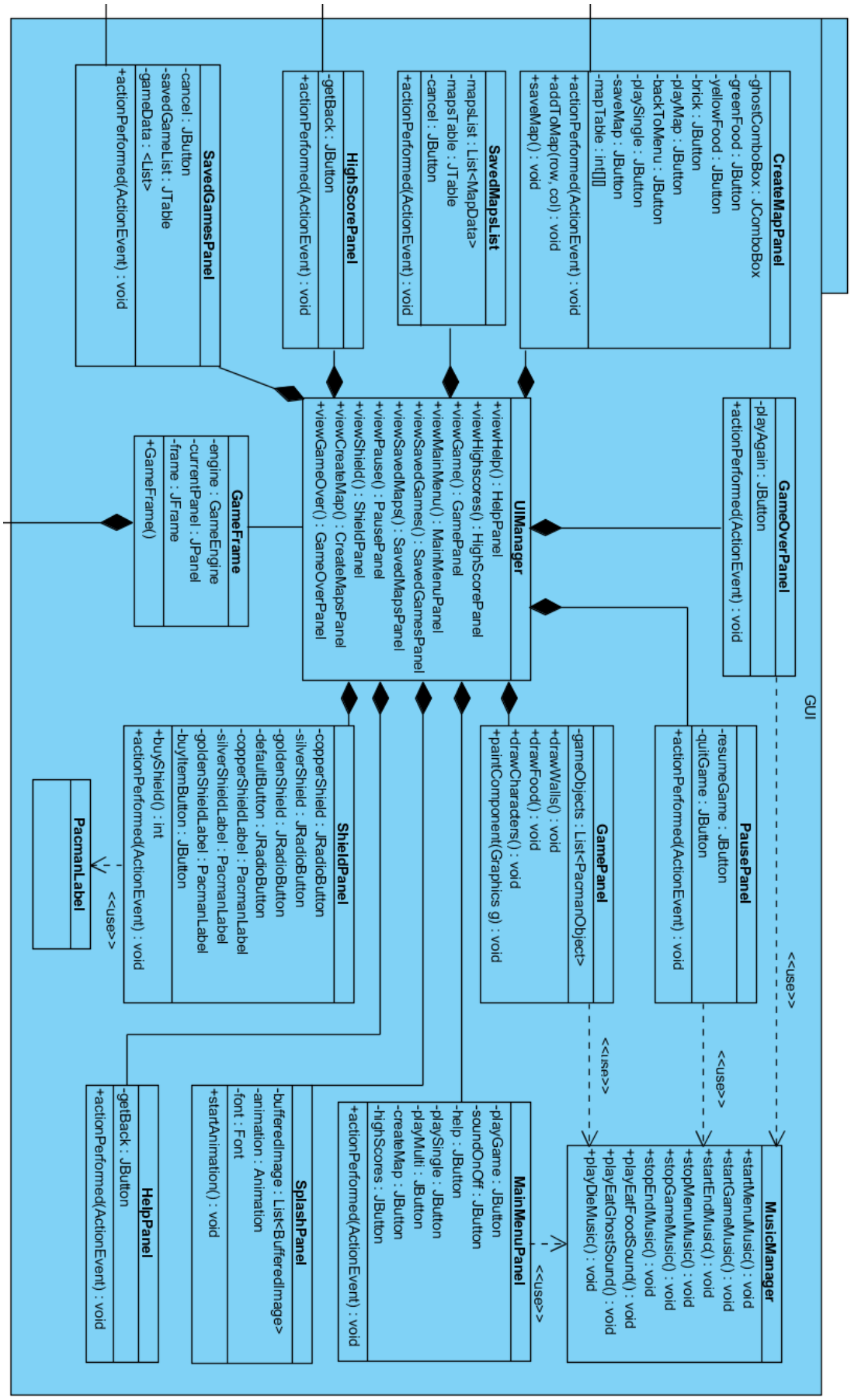

Figure 5 – Graphical User Interface Package

4.3.1.1 GameFrame Class

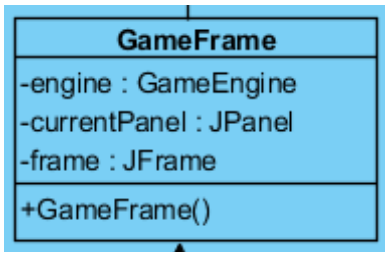

Figure 5.1 – GameFrame Class

Attributes:

- private Jframe frame: This frame is used to display all visual context of our game.
- private GameEngine engine: GameFrame class uses this GameEngine instance to interact with the Game Logic subsystem when user selects to play the game. Game Logic subsystem controls all decisions regarding the game, takes keyboard inputs and updates each game logic object.
- private currentPanel JPanel: currentPanel will hold one of the panels provided by UIManager methods.

Constructor:

- public GameFrame(): This is called each time a GameFrame object is initialized.
  - Initializes frame, currentPanel and engine.
  - Calls UIManager.viewSplash(). This method sets currentPanel to SplashPanel.
  - Calls UIManager.viewMainMenu() after intro animation ends. This method sets currentPanel to MainMenuPanel.

4.3.1.2 UIManager Class

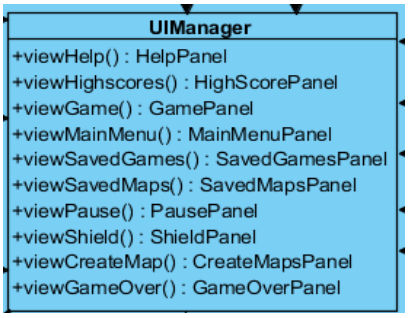

Figure 5.2 – UIManager Class

Methods:

- protected HelpPanel viewHelp(): Creates and returns a HelpPanel.

- protected HighScorePanel viewHighScore(): Asks for HighScoreDataManager class to provide it with high scores. Creates a HighScorePanel instance and returns it.
- protected GamePanel viewGame(): Calls GameEngine.prepareGame() to change GameFrame's engine attribute. Creates a GamePanel instance and returns it.
- protected MainMenuPanel viewMainMenu(): Creates a MainMenuPanel instance and returns it.
- protected PausePanel viewPause(): Creates a PausePanel instance and returns it.
- protected CreateMapPanel viewCreateMap(): Creates a CreateMapPanel instance and returns it.
- protected ShieldPanel viewShield(): Creates a ShieldPanel instance and returns it.
- protected SavedGamesPanel viewSavedGames(): Asks for GameDataManager class to provide it with a list of saved games. Creates a SavedGamesPanel instance and returns it.
- protected SavedMapsPanel viewSavedMaps(): Asks for MapDataManager class to provide it with a list of saved maps. Creates a SavedMapsPanel instance and returns it.
- protected GameOverPanel viewGameOver(): Creates a GameOverPanel instance and returns it.
- protected SplashPanel viewSplash(): Creates a SplashPanel instance and returns it.

4.3.2 Game Logic Package

Game Logic Package is responsible for handling all of the game mechanics.
We have grouped various kinds of PacmanObject's children (Food, Pacman, WallBrick, Ghost and Shield) each with its own attributes inside this package. Therefore, all data of the game while the game is still running is handled inside Game Logic package.
We have also grouped service classes which will change these PacmanObjects inside the Game Logic Package. Therefore, Game Logic Package is also responsible for updating each objects status and position according to manager subpackages: UpdateManager and AnimationManager.

Core class of Game Logic Package is GameEngine class, which controls all of the updates each PacmanObject will receive. GameEngine will share the information it has with GUI subsystem as mentioned in the previous section (4.3.1).

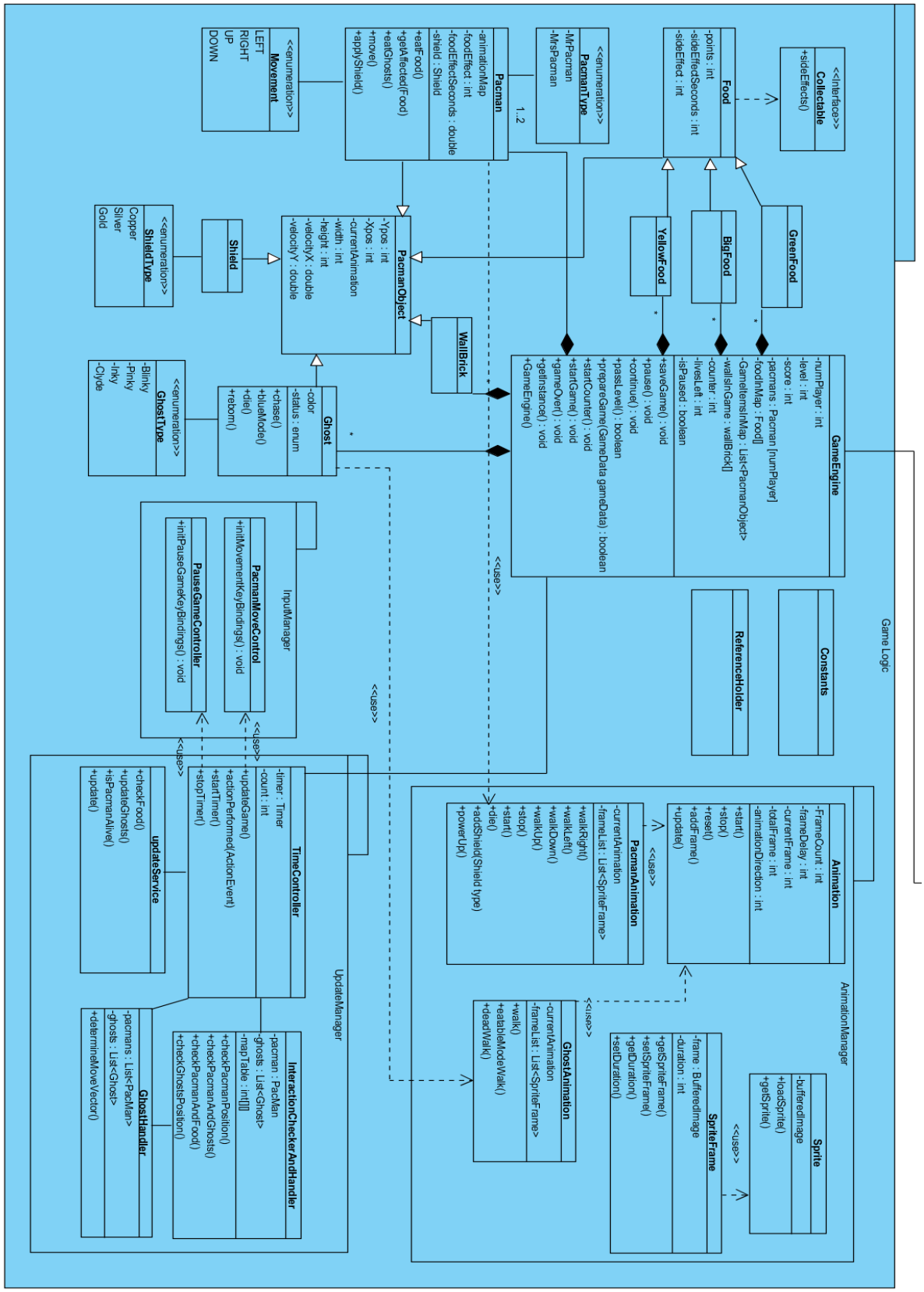

Figure 6 – Game Logic Package

4.3.2.1 GameEngine Class

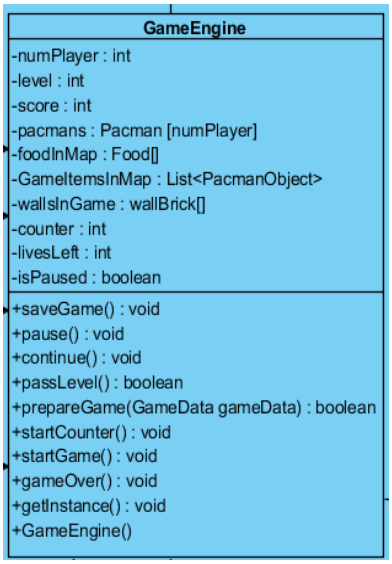

Figure 6.1 – GameEngine Class

Attributes:

- private int numPlayer: Keeps number of players in the game.
- private int level: Keeps the current leves user is in, or was in in if user loads a saved game.
- private int score: Keeps the sum of the scores reached by user(s) in that game.
- private Pacman[numPlayer] pacmans: Keeps 1 or 2 pacman instances, depending on whether the user has chosen single or multiplayer modes.
- private int[][] foodInMap: Keeps a 2D map of integers where each element will correspond to a grid and the integer value inside each element will determine whether the grid has a food and if so, the food's type.
- private List<PacmanObject> gameItemsInMap: Keeps different kinds of PacmanObjects.
- private int[][] wallsInGame: Keeps a 2D map of integers where each element will correspond to a grid and the integer value inside each element will determine whether the grid has a wallBrick.
- private int counter: It keeps the number of seconds left until the game starts each time it's paused. It is initialized to 3 seconds inside GameEngine's constructor.
- private int livesLeft: Keeps number of lives the player(s) has left in total. It is decremented each time user collides with a Ghost which is in its default chase mode.

- private boolean isPaused: It is False while game is Running, True while the PausePanel is active. Also True as long as the countdown until game starts has not finished yet.

Constructor:

- public GameEngine(): Initializes each of its attributes to their defaults values.
  - Initializes numPlayer to 1.
  - Initializes level to 1.
  - Initializes score to 0.
  - Initializes pacmans to Pacman array of length 1 with a Mr. Pacman inside.
  - Initializes foodInMap according to default game map.
  - Initializes gameItemsInMap according to default game map.
  - Initializes wallsInGame according to default game map.
  - Initializes counter to 3.
  - Initializes livesLeft to 3.
  - Initializes isPaused to True (since each game starts with countdown).

Methods:

- public void saveGame(): Constructs a GameData object with all of the attributes in GameEngine. Calls DataLayer.GameDataBase.GameDataManager.setGameData() function in order to save this GameData object into our database.
- public void pause(): Sets isPause to True. Sets counter to 3, so that game will count down to 3 and then start after pause.
- public void continue(): Calls startCounter(). Countdown has now started. Continues the game once startCounter returns.
- public void passLevel(): This function is called when all the food in the map is eaten and the user(s) still has at least one life.
  - If level is not 3, increments it.
  - Else, calls setHighScore().
- public boolean prepareGame(): This function is called when users pass a level and continue after choosing their shields. It updates the shield choices and sets game map back to its default, returns true if it succesfully does so.

- public boolean prepareGame(GameData gameData): This function is called when users choose to play in any different mode other than the default one (default mode is single player, default map, starting a new game from level one: a.k.a the initialization values). It loads the gameData in order to start a game in a different mode. Returns true if it succesfully loads the data.
- public void startCounter(): This function is called in two occasions: if the user starts/loads a game, or if the user continues the game after pause. It makes the game stay paused for 3 seconds and then continue.
- public void gameOver(): This function is called if the users have lost their last life. It is first used to inform the GameFrame to display a GameOverPanel, then it calls setHighScore() function.
- public void setHighScore(String Player): It calls DataLayer.HighScoreDataBase.HighScoreDataManager.insertnewHighScore() to set a new high score (insertNewHighScore method checks if the new highScore is actually in its top 10 list).

4.3.2.1 PacmanObject Class

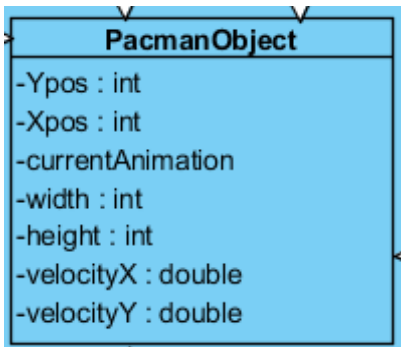

Figure 6.2 – PacmanObject Class

Attributes:

- private int YPos: Keeps vertical location of the center of object.
- private int XPos: Keeps horizontal location of the center of object.
- private int width: Keep the total width of the object.
- private int height: Keep the total height of the object.
- private double velocityX: Keeps the current velocity of object in x axis.
- private double velocityY: Keeps the current velocity of object in y axis.

4.3.2.2 Pacman Class

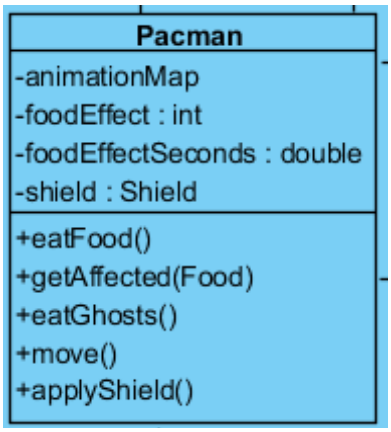

Figure 6.3 – Pacman Class

Attributes: Pacman class has all of PacmanObject's attributes since it inherits from PacmanObject class.

- private int foodEffect: Keeps an enumeration of foodEffects, it's initialized to 0 if Pacman is not under any effect.
- private double foodEffectSeconds: Keeps the amount of seconds left until food's effect decays.
- private Shield shield: Keeps the shield if user has purchased one.
- private enum currentAnimation: Keeps an enumeration corresponding to different PacmanAnimation functions.

Methods:

- public int eatFood(Food food): This function is called when Pacman object collides with a food object. Pacman object will get affected depending on the food type. It returns the score gained from eating that food type.
- public int eatGhosts(): This function is called when Pacman object collides with a ghost in its blueMode. It returns the score gained from eating a ghost.
- public void move(int x, int y): This function is used to translate Pacman in the map.
- public void applyShield(ShieldType type): This function is used to update Pacman's shield attribute. It will be called if user buys a shield.

4.3.2.3 Food Class

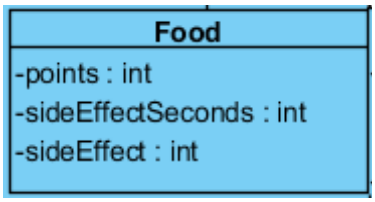

Figure 6.4 – Food Class

Attributes: Food class has all of PacmanObject's attributes since it inherits from PacmanObject class.

- private int points: Determines how many points will be gained if Pacman eats a food.
- private double sideEffectSeconds: Determines the seconds it will take until the food's side effect decays.
- private int sideEffect: Determines the type of sideEffect Pacman will have if it eats the food.

4.3.2.2 Ghost Class

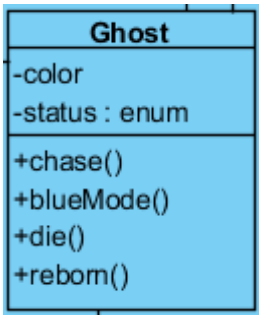

Figure 6.5 – Ghost Class

Attributes: Ghost class has all of PacmanObject's attributes since it inherits from PacmanObject class.

- private GhostType type: Determines if the ghost is Blinky, Pinky, Inky or Clyde.
- private int currentAnimation: Keeps the status of the ghost. Its used to determine its animation. Its also used to determine what happens when Pacman and Ghost collides, e.g. if currentAnimation enum equals to blueMode, then pacman can eat the ghost.

Methods:

- public void chase(): Changes currentAnimation to its default, where Ghost is colorized properly(red, pink etc.) and chases Pacman.
- public int blueMode(): Changes currentAnimation to ghost's blueMode, where ghost looks neon blue and is eatable by a Pacman.
- public void die(int x, int y): Changes currentAnimation to dead, where ghost tries to return to its cage to be reborn.

4.3.3 Data Layer Package

This package is where all the database methods and objects are handled. This packaged is used to save, get and update each essential data component (GameData, MapData, HighScore(s)). These action are done using three different manager classes : GameDataManager, MapDataManager and HighScoreDataManager.

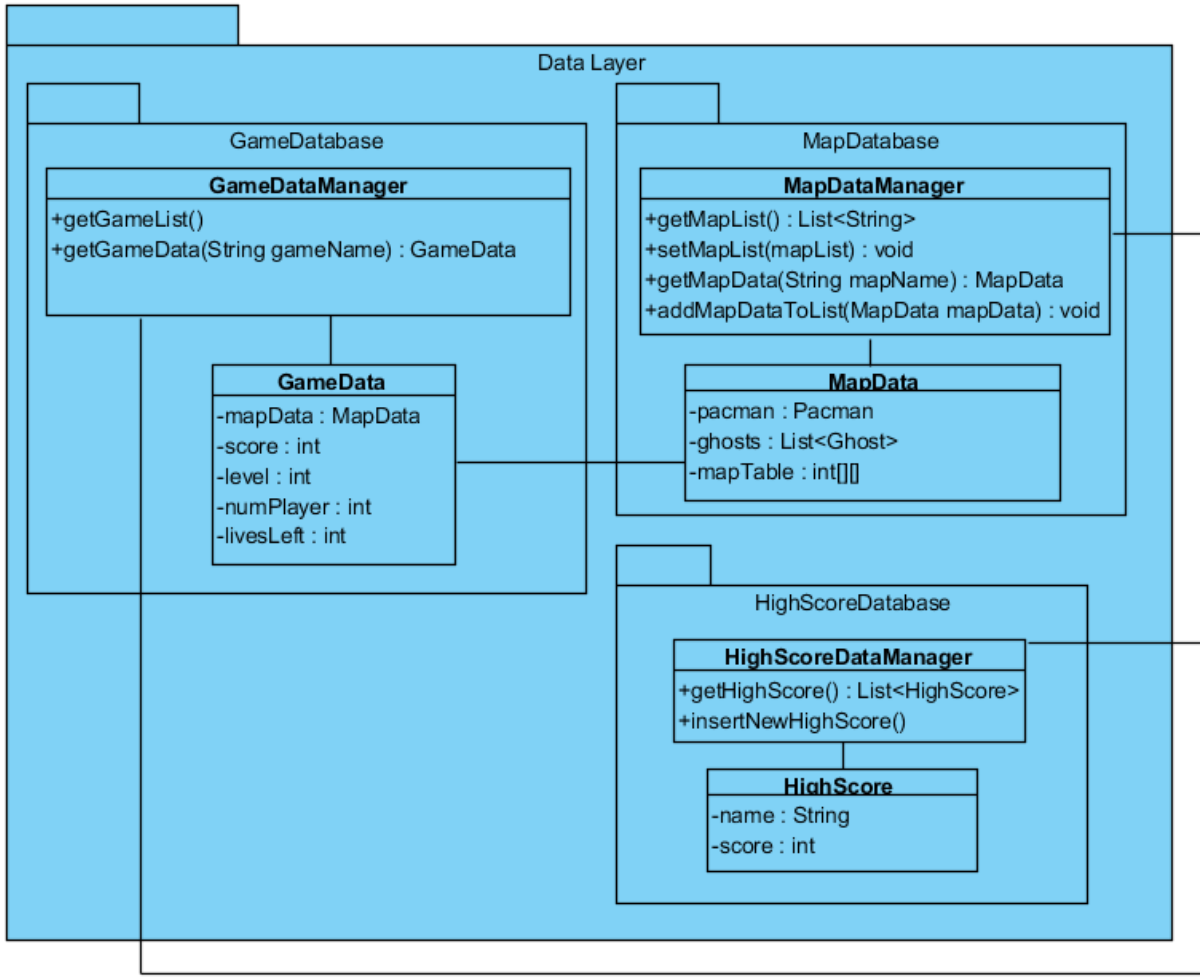

Figure 7 – Data Layer Package

4.3.3.1 GameDataManager Class

Methods:

- public List<String> getGameList(): Returns the names of saved games, in order to display in GUI.SavedGamesPanel.
- public GameData getGameData(String name): This function is called when user tries to load a previously played game. Returns the GameData object of that game if it exists, else returns null.
- public void setGameData(String name, GameData gameData): This function is called when users try to save their game. It attaches this games name into name list and creates a separate file for this game's data.

4.3.3.1 MapDataManager Class

Methods:

- public List<String> getMapList(): Returns the names of saved games, in order to display in GUI.SavedGamesPanel.
- Public MapData getMapData(String name): This function is called when user tries to load a user made map. Returns the MapData if the query matches with one of the previously made maps; else, returns null.
- public void setMapData(String name, MapeData mapData): This function is called when users try to save a map. It attaches this map's name into a list of map names and creates a separate file for this map's data.

4.3.3.1 HighScoreDataManager Class

Methods:

- public List<Highscore> getHighScore(): Returns the list of highScore objects, each with a username and a score.
- public void insertNewHighScore(String name, int score): Creates a HighScore instance and pushes it into the list of HighScores.