BILKENT UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

CS319
Object Oriented Software Engineering

System Design Report
Project "Rush Hour"

Group 2.C Not So Oriented

Deniz Dalkılıç
21601896
Ahmet Ayrancıoğlu
21601206
Kaan Gönç
21602670
Ali Yümsel
21601841
Sina Şahan
21602609

Table Of Contents

1. Introduction

Rush Hour is a sliding block logic game which provides a fun way to improve one's problem solving, sequential thinking and logical reasoning skills by offering many different challenges. Our aim is to redesign and improve the original Rush Hour experience by adding new features while preserving its effect on people's problem-solving abilities. In this part, we will explain the purpose of the system and our design goals that we want to achieve while developing such a system.

1.1. Purpose of the System

The purpose of our own system is to help our customers to improve their skills listed above by challenging themselves in a much more entertaining way. Our aim is to create a better Rush Hour gameplay experience in virtual life when compared to the real life, by changing the original Rush Hour game with our own additions of new features which we think will attract the customers a lot more. The players will have the opportunity to interact more with the game that they are not able to do such an action in real life except only moving real blocks. We will have different power-ups in our game that are gained by playing the game well, which will help the player tremendously when used. Players will feel more rewarded while playing our game compared to the board game because of our level-based reward system. Such a system will provide a better chance for the game to be served on different platforms with a great variety of features such as different theme or vehicle packs to different users according to their own interests which is also not possible in real life without paying an extra fee. Finally, adding new levels to the game and bringing the levels to the users will not require users to buy new games and will be done with updates contrary to the board game.

1.2. Design Goals

Performance Criteria

Response Time

- The system should be able to define each input separately in the same frame when multiple inputs are given by the player at the same time and the system should respond to inputs from the user in less than 17ms.
- Progress saving should not take more than 0.5 seconds.

Throughput

- The system should be able to support smooth animations which means the game should be able to run at 60 frames per second.
- The transition between panels must take less than 1 second.

Memory

- Only needed space should be allocated in the system for the game to run.
- No unnecessary information about the game should be kept within the local directory.
- All of the progress details should be deleted when a user deletes his account as the data will be kept in a local directory.

Dependability Criteria

Reliability

- Any crashes during the gameplay should not affect the game data so that when a crash happens the player should be able to continue to his game from where he left.
- The system should automatically save the progress including the settings adjusted the player as a backup if the player quits the game without saving his progress.
- The system should be able to provide a local backup system so that the players do not worry about having an internet connection.

Robustness

- 99.9% of users who play through the game should not encounter a system crash that is caused by bugs. The system should not have any major bugs that can disrupt the overall gameplay experience of the user.
- The user should not be able to interact with any locked feature in the game which could create a corruption in the game data. In order to achieve such a system, we will try every possible corner case that the users can encounter so that there won't be any problems revealed to the player.

Maintenance Criteria

Extendibility

- The game must be implemented with an extendible design, any additional classes created inside the Controller system should not affect the already existing behavior of other controller classes and should start working as intended when the class is added to the Game Engine.
- Graphical updates or the complete change of the User Interface should be added without making any changes inside Model or Controller systems.

Modifiability

- Object-oriented design concepts must be employed to make the game easily editable.
- Classes that depend on each other should not depend on the implementation of the methods used so that when changing the functionality of a method or

adding new functionality to a method should not affect the classes that use that method.

Readability & Traceability

- The system should allow multiple people to work on the implementation at the same time by providing a maintainable structure in order to not have any difficulties in understanding the existing code segments, implementation of features and the design choices.

End User Criteria

Usability

- Players must be able to interact with the user interface without having any difficulty and without being given any extra information about how to use the interface. 95 out of 100 people should be able to start the game from the main menu when they open the game for the first time.
- User interface should use consistent background images for buttons and the colors used across menus should also be consistent.
- 95 out of 100 people, who never played rush hour before, should be able to understand and play the game after seeing the tutorial in the help menu. The tutorial should be made using images that describe and show how tha game is played instead of long textual descriptions.

Utility

- The players should be able to feel the metaphors of real world effectively such as hearing car engines and horns while playing a level.
- The system should inform the user on how effectively they solved the puzzles, in order to let the user learn more about development of their skills. The system should inform the user on how many moves the user made and the what the average number of moves made by players on that level is.

Cost Criteria

Upgrade Cost

- Cost of translating data from the previous iterations should be low.
- Cost of adding new features to the system should be low.

Deployment Cost

- Cost of installing the system and teaching the game to the users should be low.

Maintenance Cost

- The cost for bug fixes and enhancements to the system should be low.

Trade-Offs

Space vs. Speed

- If the system we develop does not meet the performance criteria discussed above, we will expand the allocated memory of the system in order to increase the overall performance. However, we believe this will not be necessary because of our use of efficient system design and the low complexity of the game

Usability vs. Functionality

- Our design should not be complicated as it could decrease the usability of the product. Therefore, we won't include any unnecessary function to our game and keep the basic functionalities so that the user can easily interact with the product without facing any difficulty in understanding the main use cases.

2. High-level Software Architecture

In our design, our plan is to divide our system into meaningful subsystems in order to deal with the problem of complexity and efficiency. Our main purpose is to reduce the amount of coupling between different systems such that a problem in a specific subsystem is less likely to affect another subsystem whereas keeping the cohesion between classes in a subsystem high. We are going to adjust the trade-off between cohesion and coupling in a meaningful way so that different individuals could easily develop different subsystems and combine them.

2.1. Subsystem Decomposition

In order to achieve our design goals we decided to use the architectural style Model/View/Controller (MVC) in our system. By using this design style we will maintain our application domain classes in the Model subsystems, user interface components in the View subsystems and the controller classes that manage the sequence of interactions with the user in the Controller subsystems. We are going to have a number of different subsystems for all of these three subsystem types of MVC style.

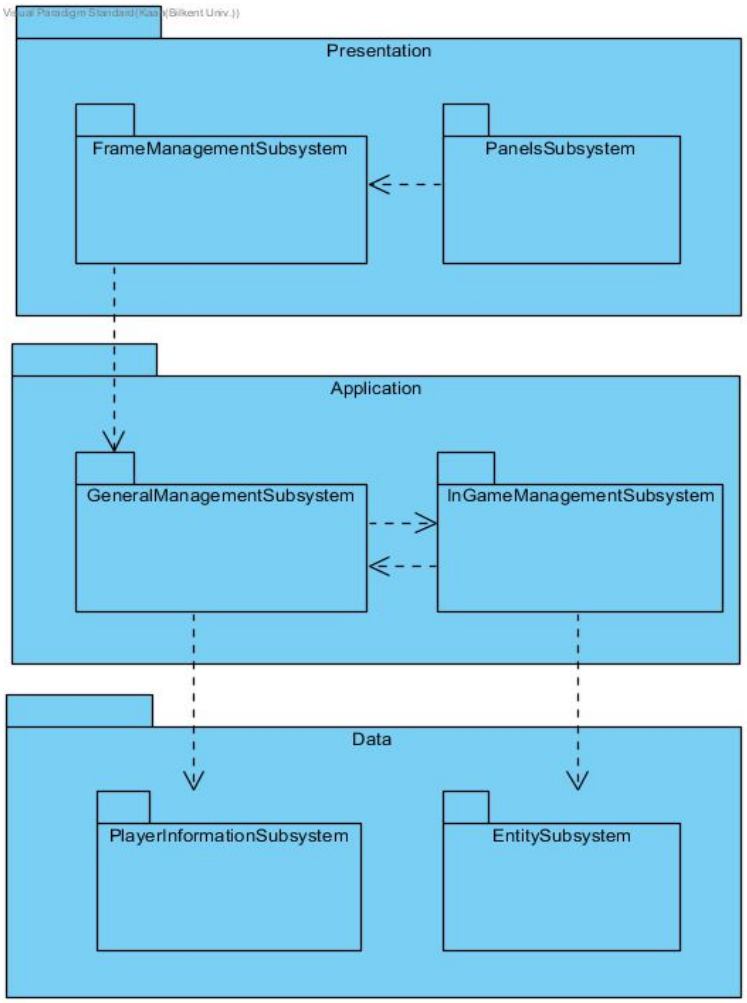


Figure 1 - Layered Subsystem Decomposition

.

As it is seen from the diagram, Presentation layer contains the user interface and visualization related subsystems, Application layer contains the subsystems that indicates the game logic, and Data layer contains subsystems that represents the model objects of the game. The purpose of layering is to show the abstract hierarchy between the subsystems regarding to MVC design. PlayerInformation and Entity subsystems are the basic ones to declare the core objects of the game. In GameManagement subsystem initiates the model objects that are used during the gameplay, and make them work in a compatible way. GeneralManagement is the main subsystem that initiates the additional feature model objects such as Player and Theme, make the game playable with GameManager and GameEngine classes, and user-interactable by providing the connection with the view. Panels subsystem designs and creates all the needed panels for the game. At last, FrameManagement subsystem gets data from the controller, shares them with the panels, and provides a reasonable harmony between the panels. Our layers can only call operations from the layer below. We used this opaque layering because it allows us to maintain the system easier and creates flexibility in our system. More details about the components of these subsystems are shown in Figure 2.

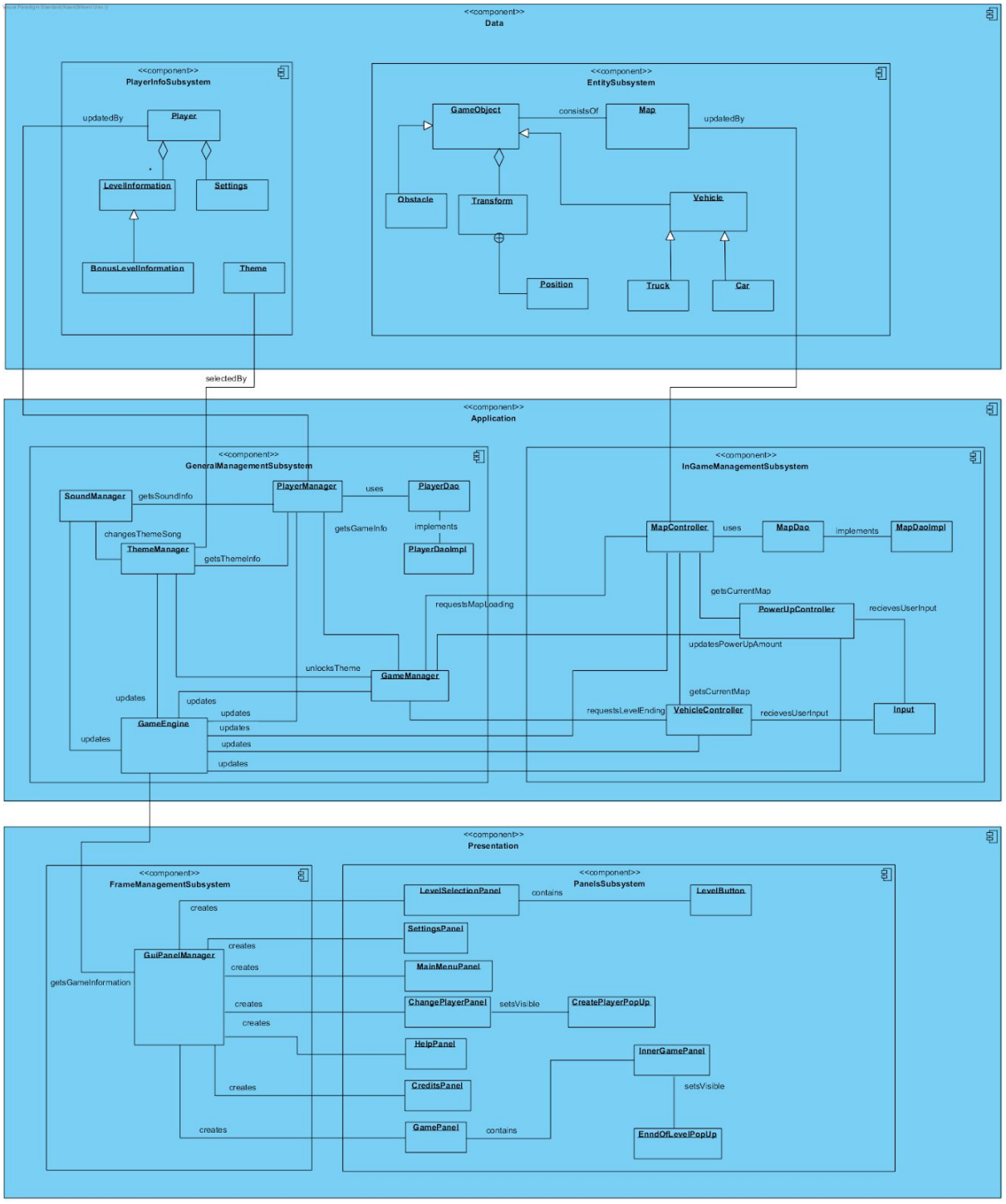


Figure 2 - Subsystems Component Diagram [1] [2]

The connections between classes and general sequences of events can be seen in this diagram. Our classes are divided into 3 major packages and 6 subsystems. We kept the amount of interactions and calls inside subsystems high and number of interactions and calls low between subsystems so that related classes can work together without knowing the implementations of unrelated classes. This allows our system to be flexible and maintainable. However, our implementation requires the GameEngine class to initialize and update every Controller class directly but still it does not know the implementations of the classes that it updates and does not depend on them. So our system stays flexible and maintainable with high cohesion and low coupling.

2.2. Hardware / Software Mapping

The subsystems of Rush Hour will be implemented in Java programming language; more specifically for the development of the View subsystems we will refer to Java Swing libraries as well as improving the design with JavaFX libraries when necessary. Therefore, in order for users to execute the game, Java Runtime Environment (JRE) should be installed in their devices. In terms of the version of Java Development Kit (JDK) that the Rush Hour will require, we need to say that it will most probably require at least JDK8, as we will be doing our implementation with additions made from JavaFX libraries.

In terms of the hardware requirements of Rush Hour, the users will need a mouse(or a mouse pad) in order to interact with the panels in the game and also a keyboard in order to move the vehicles in the game scene. Our Input system, will be identifying such different inputs from the mouse and the keyboard so that the Controller subsystems will know which method to execute during gameplay. Moreover, our game won't be exporting or importing data from the internet so that the users should not worry about having an internet connection. The data will be tuned in to special format and will be kept in text files so that the saved game progresses will not cause a memory shortness in the system. Therefore, we are sure that any user who provides himself the requirements above will be able to play our game without having any performance issues.

2.3. System Overview

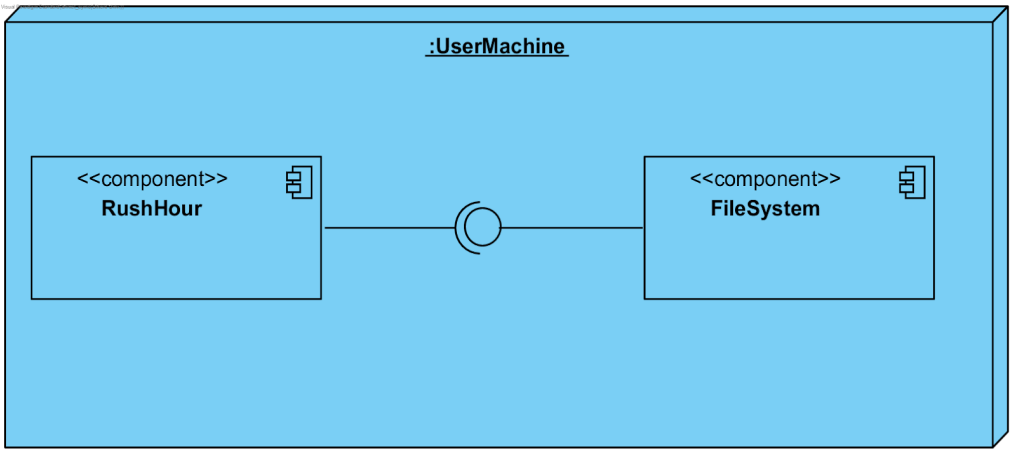


Figure 3 - Deployment Diagram Hour [1]

The physical deployment of the software artefacts in Rush Hour is displayed above. Rush Hour does not depend on an outer system which is kept on a different device, therefore both Rush Hour and the FileSystem that keeps the necessary game data are stored in the client's machine. The FileSystem provides the necessary data to Rush Hour which is required to run the game according to the latest player progresses. This includes maps, settings and level progress.

2.4. Persistent Data Management

As Rush Hour is a relatively small Project, it does not require any complex data management system like a database. It will use the client's hard drive as storage and it will not require any internet connection after the initial installation of the game. All of the settings, maps, saved games and user data will be stored on the client's hard drive as text files. Saved games will have a metadata that stores which user owns this saved game and which level are they at; and it will have a map matrix which is the last state of the game map, just before the player exits. If the player selects "Play Game" option instead of choosing a level from levels menu, this file will be read and the player will start from that last state that he left the game. All of the textures and game object images will be stored in .png format as it allows transparent background (unlike .jpg) and it supports 24-bit and 48-bit color (unlike .gif which supports only 8-bit color). The sounds will be stored in .wav format.

2.5. Access control and security

As Rush Hour won't be connected to the internet and won't be using a database, there won't be much security issues that will affect the game. We will allow multiple users to play the game and have their own settings and saved games, however there will be no extra protection that inhibit a user to access another user's profile. As this game is stored locally on the client's hard drive, we assume that only trusted users (users who have access to the computer) will have access to the game. Also, all of the user data will be machine specific (and all of the saved games will be user specific), so there won't be a way to copy user data and saved games from another computer. Maps of the game will be stored as read-only, so that no third-party will be allowed to change the maps. However, in the later versions, we might add support to add new map-packs or user-created maps.

2.6. Boundary Conditions

Although most of our system design effort is concerned with steady-state behavior, what our system does at boundary conditions such as Initialization, Termination and Failure play an important role in developing a resistant and robust system.

2.6.1. Initialization

When Rush Hour is executed the very first time, the game will be initialized with a default player which does not have any progress with the game. However, when the game is executed at any other time, the game will be initialized according to the progress details of the last player that played the game. When a player interacts with a main menu option, current player's progress details will be displayed in the associated panels. As, the progress details will be kept in local text files, the game

will not have to communicate with another system to update the data during initialization, which significantly reduces the probability of a possible error, as no internet connection is needed with text file usage. Any data that is related to functionality of the game or the user will be kept in local directory as txt files inside the installation folder. However, the game will be opened from .exe file which the user can execute.

2.6.2. Termination

In order to terminate the game there are several options. Termination of the game can happen in two ways, the user can close the application by pressing the “X” button on the panel or the user can close the application by pressing the “Exit Game” button from the main menu. As we will have a dynamic progress saving system in our game (any new progress will directly update the data in the text files), the player that terminates the game by closing the application window will not lose any progress details. The system will update the data in case of a window termination with the “***windowClosing(WindowEvent e)”*** method of Swing library which is executed right before the window is closed.

2.6.3. Failure

In case of a failure, the game data will not be lost and the player will be able to continue playing the game with his latest progress. We achieve this by maintaining a dynamic auto saving system which automatically saves each change that the user makes while Rush Hour is running. Additionally, Rush Hour is an interactive game with images and many sound effects therefore it can be hard to create a totally error-free system. Any audio or visual exception that our implementation could not catch and fix may ruin the gameplay experience. So, we will provide a reload button for the players to reload the images and the sound system if they realize an error or misconfiguration during gameplay. However, player’s progress will not be affected by such an reloading process as we will be using a dynamic progress saving system. The system will refresh the sound and images without making a change in the existing data.

2.6.4. ManageSystem Boundary Use Case

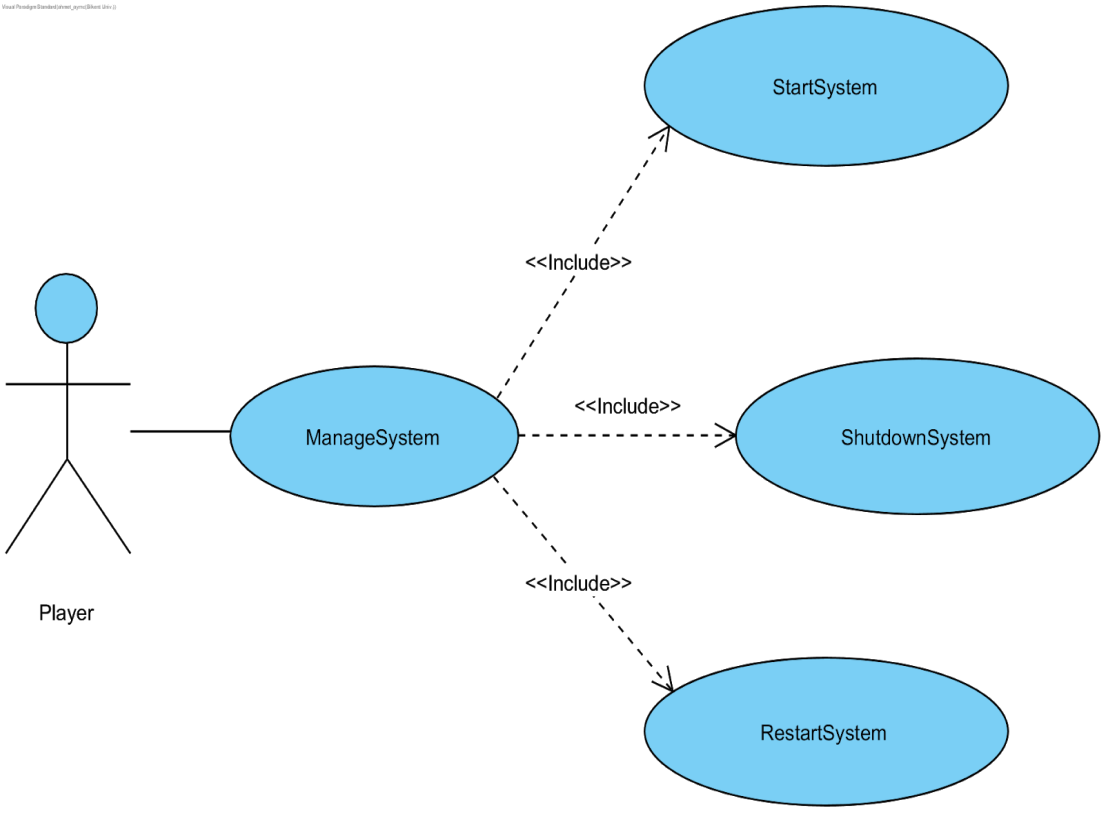


Figure 4 - ManageSystem Boundary Use Case Diagram [3]

Rush Hour will not be controlled by an outer system or any kind of administrator, therefore the user should be able to interact with boundary conditions to control the system.The player is able to manage three different boundary conditions such as starting, shut downing and restarting the system in case of a failure while managing the system.

3. Low-level design

In this part, we will explain the relationships between different classes and our way of implementing them by analyzing different design patterns and identifying class interfaces.

3.1. Object design trade-offs

During the process of low-level design, we used some designs patterns which we thought would help us in developing the game. Each design pattern has its own tradeoffs that we had to work around.

3.1.1. Singleton Design Pattern

To increase the maintainability of the game, we applied the singleton design pattern to our Manager and Controller classes which reside in Controller package. Our reasoning was, because there should only one manager or a controller that is actively controllering an aspect of the game. For example, there should only be 1 active “VehicleController” object that is controlling vehicles in the game. By using the singleton pattern we ensure that there can only be one instance of the class and other classes can easily access the instance of the class. However, using singleton design pattern has its disadvantages. As singleton object can be accessed easily by other classes and it is very convenient to use, it can be overused and lead to creating hidden dependencies between classes. Additionally, because the reference is not being actively stored in the classes that use the singleton class, tracking the dependencies becomes difficult. This leads to increased coupling between our Manager classes. Finally, using singletons can cause to problems when writing testable codes, because of the dependencies and the coupling. So for testing writing almost fully functional classes becomes necessary which is time consuming.

3.1.2. Model View Controller (MVC) Design Pattern

We wanted to use iterative engineering while developing our games to easily design, implement, test and change our implementations as necessary. By using MVC design pattern we ensured that multiple people could design and develop different aspects of our game such as Model, View and the Controller as the name of the pattern implies. This allowed us to divide the workflow between group members more effectively and lead to a faster design and development time. Another benefit of using MVC was while testing the game as we could use multiple views while using the MVC pattern and we did not have to create a fully functional user interface to make sure our game logic was working as intended. However, designing our game with MVC design pattern took longer than if we have used other patterns because deciding how to divide needed classes into three concrete parts and making sure that coupling between classes especially in different packages would be low was a time consuming process. Another, disadvantage of MVC is View depends highly on both on Controller and the Model, so to create and test a functioning user interface, we had to have a functioning Model and Controller.

3.1.3. Composite Design Pattern

We wanted all our controller and manager classes to extend from a single base class simply called “Controller”. Our GameEngine class initializes and holds every single one of our classes that are derived from “Controller” class in a Manager array. This simplifies the GameEngine class and unifies our Controller subsystem. The Manager class has an empty update() method which can be overwritten by each child class to fit their own functionalities. In our GameEngine class, update methods of all our Manager classes is called 60 times every second (60fps) regardless of their implementations. This allows us to change, add or remove any functionality of an existing classes update method without worrying about altering any other class. Another benefit of using a Manager base class is adding new classes, which derive from the Manager class, becomes very easily as we only have to add the object of the newly created class to the GameEngine without changing any code segments in our GameEngine. After that we only need to worry about the functionality of the class we added as any logic that is related to that class is contained in that class. However, using composite design patterns has its own drawbacks, It creates a too general class design that some of our Manager classes does not need to derive from but still derive from to maintain uniformity.

3.1.4. Factory Design Pattern

We wanted our UI elements to follow some standards to ensure that the graphical interface would follow our design patterns and look unified. Another reason we implemented the factory design patterns was to have clean code and avoid duplicate lines with little difference. Because, creating and adjusting Swing UI elements to our needs require multiple lines of code for every single UI element, there were multiple blocks of code that we wrote for each UI element, so we decided to put these blocks of code inside a method with various parameters for customization. We created a UIFactory class that would create various Swing UI elements for our panels. UIFactory class makes our code more robust, less coupled and easy to extend, because we only need to change the implementation of the UIFactory class to change all the panels without making any change inside the panel classes. This is also true when adding new UI elements to our panels, we only need to write a new method inside the UIFactory class to allow our panels to use the newly added UI element.

3.1.5. Data Access Object Design Pattern

In the game, Player and Map objects have to access some data that are stored in the local file system of the user. The data gets extracted or overwritten during the progress of the game. Player objects get the values for their attributes regarding the preferences of the user. These values should be saved to the local file system in order for them to get extracted later when the game is re-executed because our game ensures that any progress or preference of the user won't be lost after the game stops execution. Moreover, the maps of the levels get extracted to the game from text files. Therefore, map objects also have to extract their attribute values from the local data. In order to make these happen in a suitable way, we used the Data Access Object (DAO) design pattern. We created APIs for the Map and Player objects which are connected to their controller classes. For example, the data of a Player instance is saved to or extracted from local file system if the corresponding method of the PlayerDao interface is called by the PlayerManager. By using this design pattern, we are able to distinguish the application from the persistent data and avoid the controller classes to interfere with the user's computer.

3.2. Final Object Design

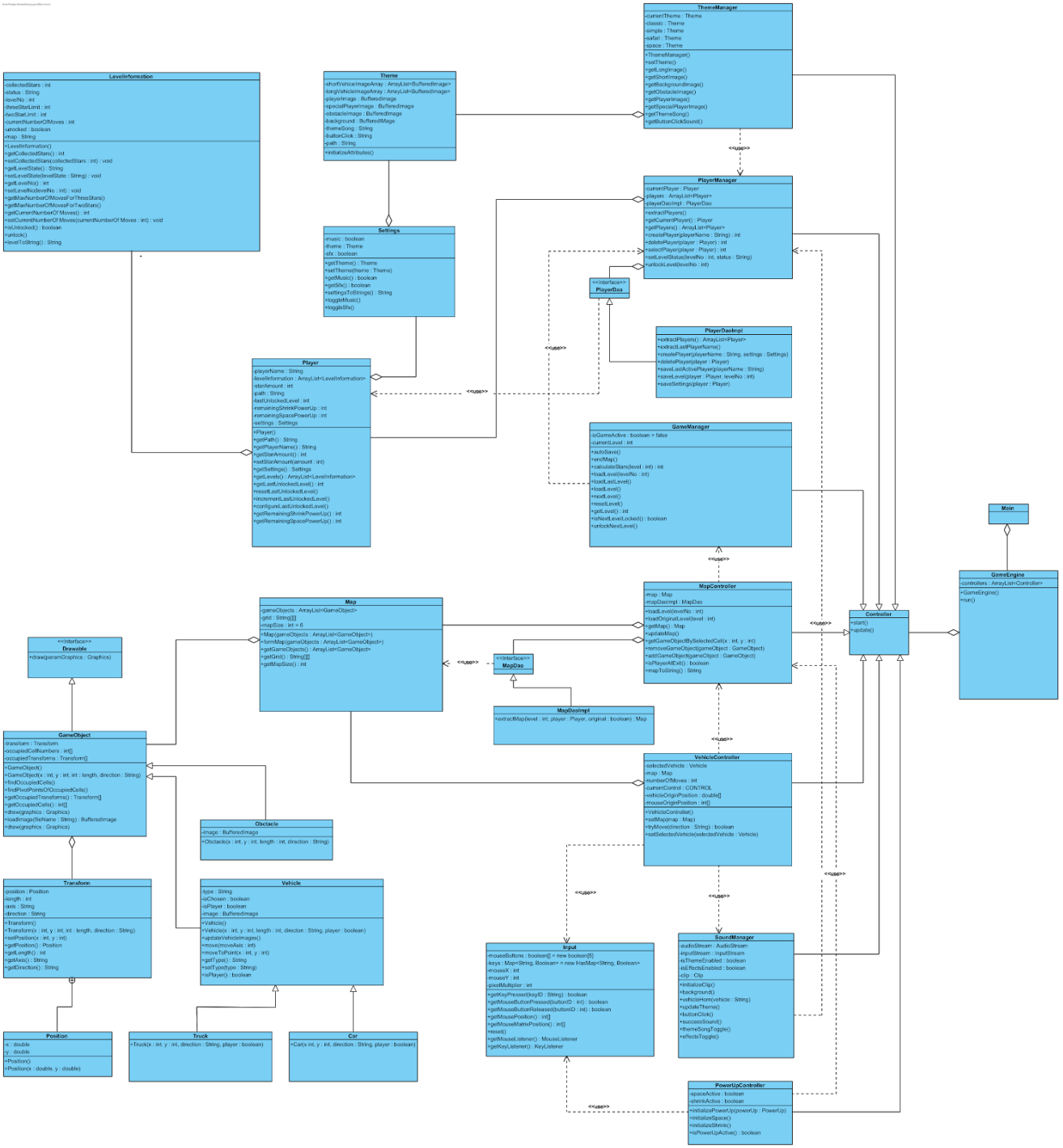


Figure 5 - Complete Class Diagram (Model - Controller)

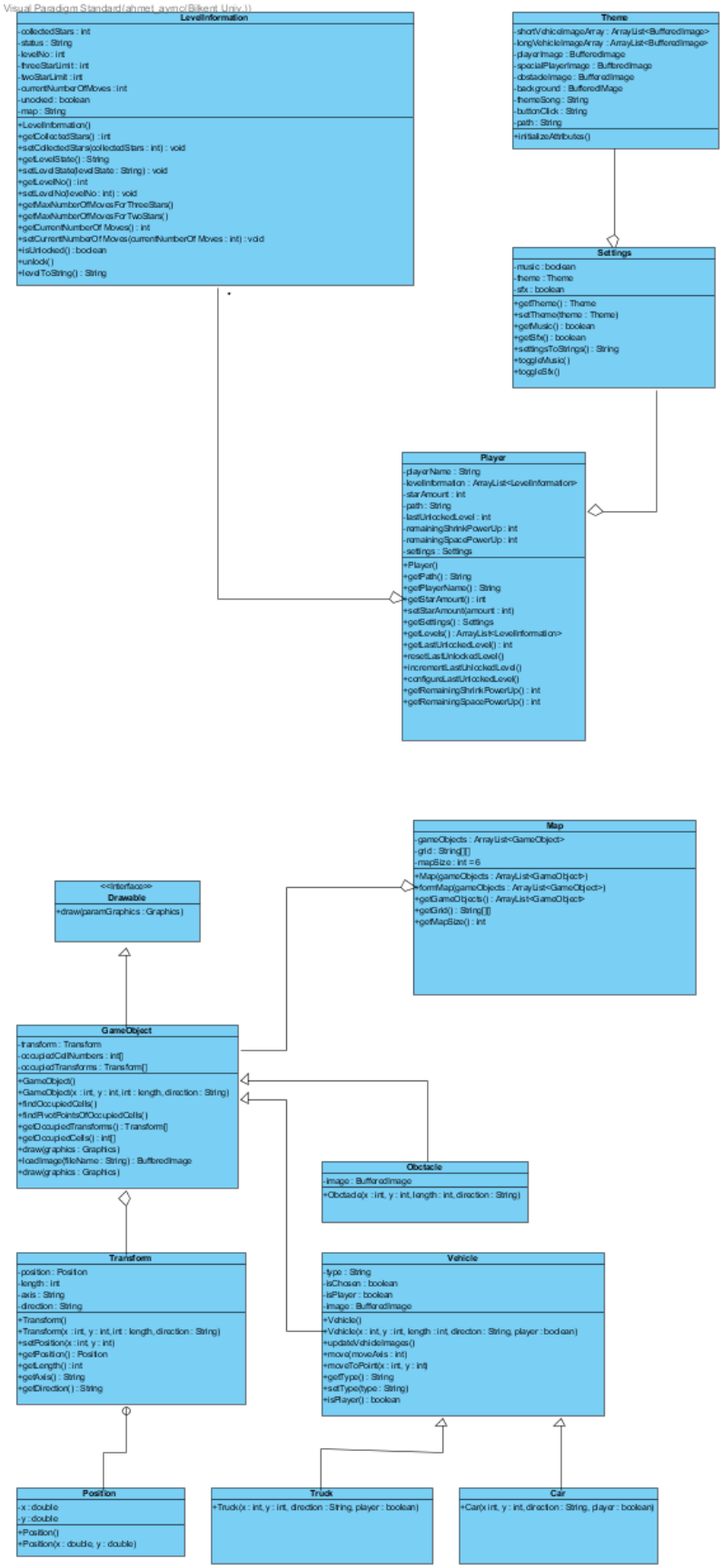


Figure 6 - Class Diagram (Model)

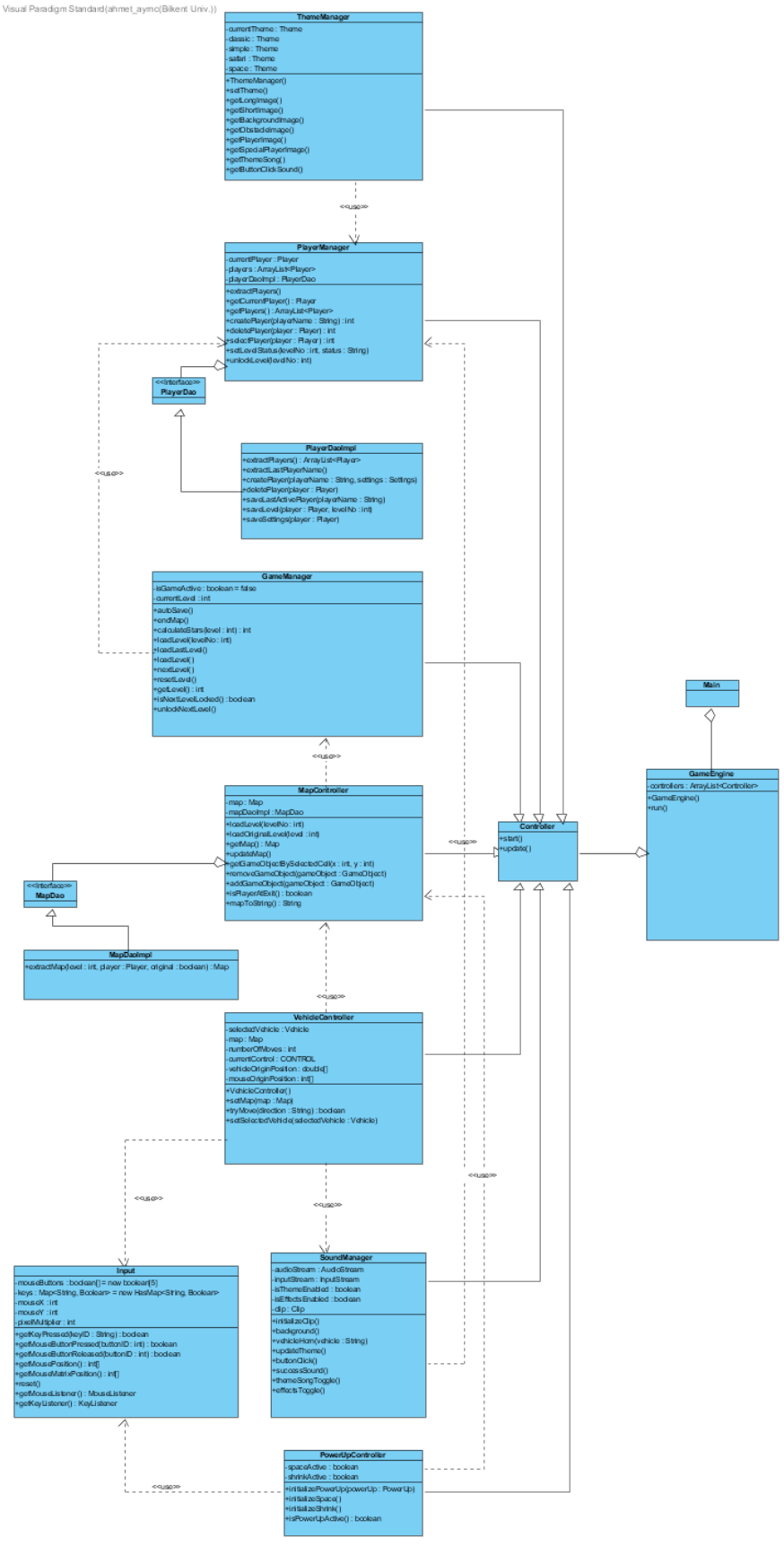


Figure 7 - Class Diagram (Controller)

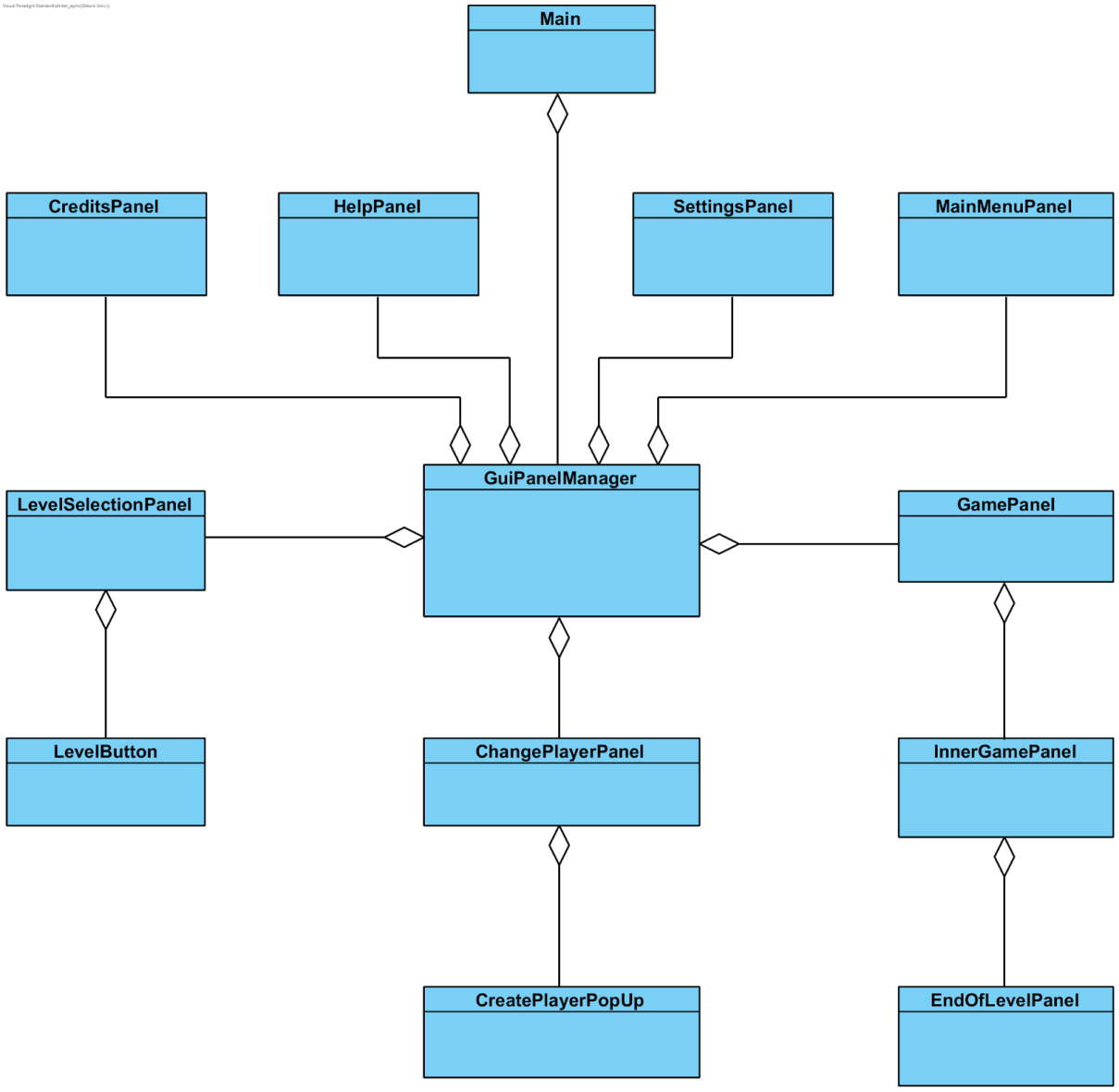


Figure 8 - Class Diagram (View)

Class Diagram of View classes are collapsed to increase readability. Detailed version the classes are in Appendix.

3.3. Packages

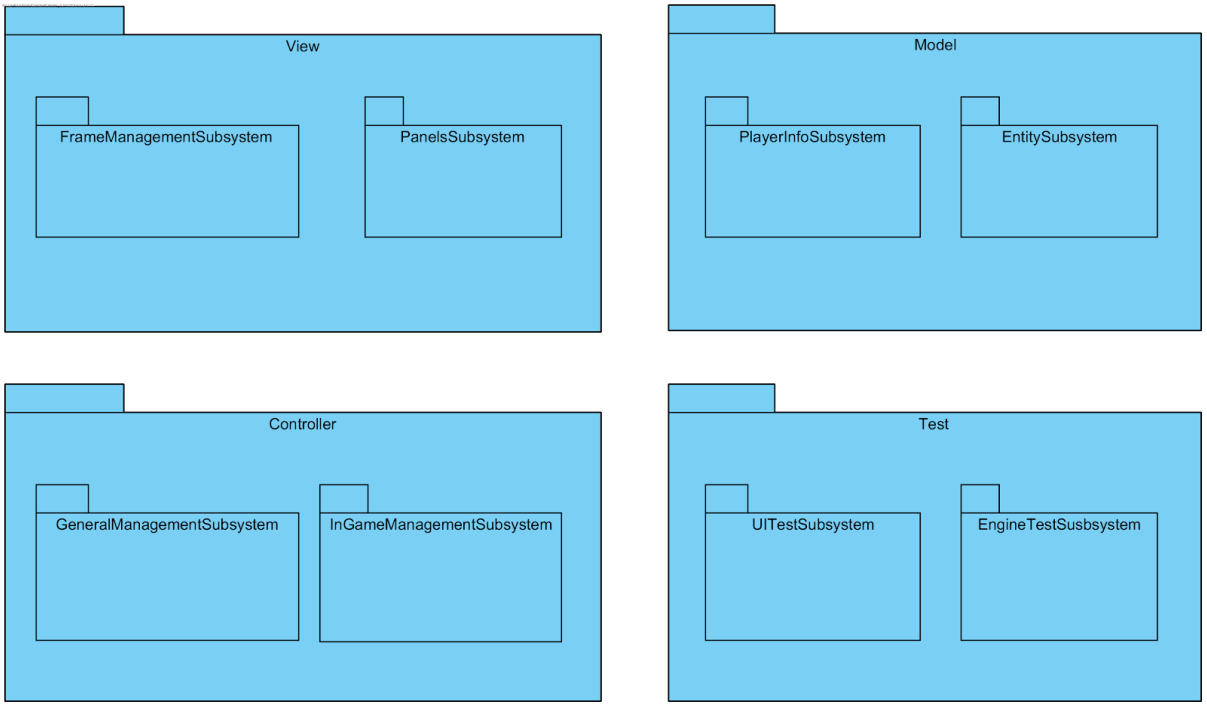


Figure 9 - Package Diagram

View, Controller, and Model packages and the packages inside them represent the Presentation, Application, and Data subsystems shown in Figure 1. Additionally, there is Test package that contains the test programs in order to test some of the crucial parts of the game individually.

3.4. Class Interfaces

3.4.1. Position Class

Position class is used to hold the coordinates of a Transform in two dimensional space.

Attributes

- **double x :** Represents the position on the x axis in two dimensional space.
- **double y:** Represents the position on the x axis in two dimensional space.

Constructors

- **public Position() :** Empty constructor that initializes both values to 0.
- **public Position(double x, double y) :** Initializes the position with the given x and y values on the two dimensional space.

3.4.2. Transform Class

Transform class is used to hold every possible information that every 2 dimensional object could have. We use position, length and direction to represent the 2d object's position, rotation and scale respectively.

Attributes

- **Position position :** Represents the position of the object in 2d space.
- **int length:** Represents the length of the object.
- **String axis:** Represents the axis of the object, horizontal or vertical.
- **String direction:** Represents the direction of the object, right, left, up or down.

Constructors

- **public Transform() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public Position getPosition():** Returns the position as a Position.
- **public void setPosition(int x, int y):** Sets the position attribute with the given x and y values.
- **public int getLength():** Returns the length as int.
- **public String getAxis():** Returns the axis as String.
- **public String getDirection():** Returns the direction as String.

3.4.3. GameObject Class

GameObject class is used to represent every possible object that is inside the game.

Attributes

- **Transform transform :** Represents all attributes that is related to 2d space.
- **int[] occupiedCellNumbers:** Holds all the cells that this game object currently occupies.
- **Transform[] occupiedTransforms:** Represents all other transforms that is related to this game object.

Constructors

- **public GameObject() :** Empty constructor that initializes values to their specified initial values.

Methods

- **private void findOccupiedCells():** Updates the cells that is occupied by this game object.
- **private void findPivotPointsOfOccupiedCells():** Updates the pivots of the occupied cells.
- **public Transform[] getOccupiedTransforms():** Returns the occupied Transforms.
- **public int[] getOccupiedCells():** Returns the occupied cells.
- **public void draw():** Draws the image connected to this game object.
- **private void loadImage:** Loads the image connected to this game object from local directory.

3.4.4. Vehicle Class

Vehicle class is used to represent the vehicles in our game. They are the main game object in the game Rush Hour.

Attributes

- **Boolean isPlayer:** To determine whether this vehicle is the playerCar.
- **BufferedImage vehicleImage:** Holds the image connected to this vehicle.
- **Boolean isChosen:** To determine whether this vehicle is currently chosen.

Constructors

- **public Vehicle() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void updateImages():** Updates the images of the vehicle according to the current theme.
- **public void moveToPoint(int x , int y):** Moves the vehicle to the given point instantly.
- **public void move(int moveAxis):** Moves the amount that is given according to its axis inside the Transform property. Moves backwards when the value is negative.
- **public boolean isPlayer():** Returns whether this vehicle is the player vehicle.

3.4.5. Obstacle Class

Obstacle class is used to represent the obstacles in our game. They are the secondary game object in the game Rush Hour.

Attributes

- **BufferedImage image:** Holds the image that represents this obstacle.

Constructors

- **public Obstacle(int x, int y, int length, String direction):** Constructor that initializes values to their specified initial values.

Methods

- **public void updateImages():** Updates the images of the obstacle according to the current theme.

3.4.6. Map Class

Map class is used to represent the map that the game is played on. It holds the vehicles that are part of the level.

Attributes

- **ArrayList<GameObject> gameObjects:** Holds the gameObjects that are currently part of the level.
- **String[] grid:** Representation of the map in textual form.
- **int mapSize:** Represents the size of the map. A size of 6 means that the map is 6 by 6.

Constructors

- **public Map() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void formMap(ArrayList<Vehicle> vehicleArray):** Forms the textual form of the map from the given array list of vehicles.Updates the textual form of the map from the given array list of vehicles.
- **public ArrayList<GameObject> getGameObjects():** Returns the vehicles on the map as ArrayList.
- **public String[][] getGrid():** Returns the textual representation of the map as 2d String array.
- **public int getMapSize():** Returns the size of the map.

3.4.7. Drawable Interface

An interface for GameObjects which are drawable.

Methods

- **public void draw(Graphics graphics):** Represents the draw method that is going to be implemented by other classes.

3.4.8. LevelInformation Class

LevelInformation class holds all information about a particular level such as the level no, number of moves and the amount of stars collected on that level.

Attributes

- **int CollectedStars:** Represents the stars collected by the player in that level.
- **String status:** Represents the state of the level. For example: Completed, NeverPlayed and such.
- **int levelNo:** Represents the level with a number.
- **int maxNumberOfMovesForThreeStars:** Holds the number of moves to get a three star in this particular level.
- **int maxNumberOfMovesForTwoStars:** Holds the number of moves to a two star in this particular level.
- **int currentNumberOfMoves:** If the level is not completed and is in a saved state, hold the current number of moves the user has made in that level.
- **boolean unlocked:** To determine whether the level is locked or unlocked.
- **String map:** Holds the current state of the map as a string so it can be saved to a text file later.

Constructors

- **public LevelInformation() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public int getCollectedStars():** Returns the number of sars as int.
- **public void setCollectedSars(int collectedStars):** Sets the collected stars on this particular level, checks if the new value is higher than the old value and does not update the value if it is lower.
- **public String getLevelState():** Returns the state of the level as String.
- **public void setLevelState(String levelState):** Changes the level state to the given value.
- **public int getLevelNo():** Returns the level number.
- **public int getCurrentNumberMoves():** Returns the current number of moves made in this particular level.
- **public void setCurrentNumberOfMoves(int amount):** Changes the current number of moves made in this particular level.
- **public int getMaxNumberOfMovesForThreeStars():** Returns the max number of moves to get three stars in this particular level.
- **public int getMaxNumberOfMovesForTwoStars():** Returns the max number of moves to get two stars in this particular level.
- **public String getMap():** Returns the string representation of the map's current state.
- **public String levelStoString():** Returns the string representation of the level's current state.

3.4.9. Settings Class

Settings class is used to hold settings information of a player

Attributes

- **boolean music:** To determine whether the music is on or off.
- **boolean sfx:** To determine whether the sound effects are on or off.
- **Theme theme:** Represents the selected theme by the player.

Constructors

- **public Settings() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public Theme getTheme():** Returns the currently selected theme by the user.
- **public void setTheme(Theme theme):** Changes the theme to the wanted theme.
- **public boolean getMusic():** Returns whether the music is on or off.
- **public void toggleMusic():** Toggles the state of the music.
- **public boolean getSfx():** Returns whether the sfx on or off.
- **public void toggleSfx():** Toggles the state of the sfx.
- **public String settingsToString():** Returns the string representation of the settings' current state.

3.4.10. Theme Class

Theme class is used to hold information of a theme.

Attributes

- **private String activeTheme:** Represents the currently selected theme.
- **private ArrayList<BufferedImage> shortVehicleImageArray:** Holds the short vehicle images for this theme.
- **private ArrayList<BufferedImage> longVehicleImageArray:** Holds the short vehicle images for this theme.
- **private BufferedImage playerImage:** Holds the player vehicle image for this theme.
- **private BufferedImage specialPlayer:** Holds the special player vehicle image for this theme.
- **private BufferedImage obstacle:** Holds the obstacle image for this theme.
- **private BufferedImage background:** Holds the background image for this theme.
- **private ArrayList<String> vehicleSoundArray:** Holds the vehicle sounds for this theme.
- **private String themeSong:** Holds the theme song for this theme.
- **private String buttonClick:** Holds the button click sound for this theme.
- **private String path:** Holds the path for this theme.

Constructors

- **public Theme(String theme):** Constructor that initializes values to their specified initial values.

Methods

- **private void initializeAttributes():** Initializes attributes for this theme.
- **private void setSounds():** Sets the sounds for this theme.
- **private void setImages():** Sets the images for this theme.

3.4.11. Player Class

Player class hold represent the Player who is playing the game. It holds every information about a player such as, level progression, rewards and settings.

Attributes

- **private String theme:** Represents the current theme that is selected by the player.
- **private ArrayList<LevelInformation> levels:** Holds the information about every level spesific to the user.
- **private String playerName:** Represent the name of the player.
- **private int starAmount:** Represents the total number of stars collected from every level.
- **private int lastUnlockedLevel:** Holds the number of the last unlocked level by this player.
- **private Settings settings**: Holds all information about the players settings.
- **private int remainingShrinkPowerup:** Holds the amount of shrink power up that the player has.
- **private int remainingSpacePowerup:** Holds the amount of space power up that the player has.
- **private String path:** Holds the player folders path inside the data folders.

Constructors

- **public Player() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public String getPath():** Returns the path of the player folder inside data folders.
- **public String getPlayerName():** Returns the name of the player.
- **public int getStarAmount():** Returns the amount of stars the player has collected in total.
- **public void setStarAmount(int starAmount):** Changes the amount of stars the player has collected in total.
- **public Settings getSettings():** Returns the current settings of the player.
- **public ArrayList<LevelInformation> getLevels():** Returns an arraylist of LevelInformation for this player.
- **public int getLastUnlockedLevelNo():** Returns the last unlocked level by this player.
- **public void configureLastUnlockedLevelNo():** Calculates and updates the last unlocked level by this player.
- **public int getRemainingShrinkPowerup():** Returns the remaining number of shrink power ups.
- **public int getRamainingSpacePowerup():** Returns the remaining number of space power ups.

3.4.12. PlayerManager Class

PlayerManager is responsible for handling the updates of player information, creating, changing and deleting users.

Attributes

- **Player currentPlayer:** Represents the currently active user,
- **ArrayList<Player> players:** Holds every player that is created for the game.
- **public int numberOfPlayers:** Holds the number of players.

Constructors

- **public PlayerManager() :** Empty constructor that initializes values to their specified initial values.

Methods

- **private void extractPlayers():** Extracts and stores the extracted players in an arraylist with the help of player extractor.
- **public Player getCurrentPlayer():** Returns the currently selected player.
- **public ArrayList<Player> getPlayers():** Returns all the players in an arraylist.
- **public void createPlayer(String playerName):** Creates a new fresh player with the given name and adds the player to the players list.
- **public void deletePlayer(Player player):** Deletes all the data of the given user and removes it from the players list.
- **public void changePlayer(Player player):** Changes the currently active player to the given Player.

3.4.13. PlayerDaoImpl Class

PlayerExtractor is responsible for extracting the player data from the local directory in to the game.

Methods

- **public Player[] extractPlayers():** Extracts the player information from local directory, creates and returns a Player array to save the extracted information.
- **public String extractLastPlayerName():** Extracts the last selected players name.
- **public Player cratePlayer(String playerName, Settings settings)**: Creates a folder and an info text for the new player.
- **public boolean deletePlayer(Player player):**Deletes the folder of the player.
- **public boolean savePlayer(Player player):** Saves all the information about a player to the representative folder.

- **private void writeFile(String path, String text):** Writes to the give file the given string.

3.4.14. GameManager Class

GameManager is a generic class that handles higher aspects of the game.

Attributes

- **boolean isGameOn:** Shows whether the game is currently on or the game is not on like paused game or menus.
- **int level:** Holds the number of the current level.

Constructors

- **public GameManager() :** Empty constructor that initializes values to their specified initial values.

Methods

- **void autoSave():** Saves the current state of the game.
- **void endMap():** This is called when the level is finished and handles all the action of level being completed.
- **private int calculateStars(int _level):** Calculates the stars that should be gained from the given level.
- **public void loadLastLevel():** Loads the last unlocked level of the currently active user.
- **public void loadLevel(int _level):** Loads the given level.
- **public void nextLevel():** Loads the next level.
- **public void resetLevel():** Resets the current level to its original state.

3.4.15. MapController Class

MapController class is responsible for loading, saving levels. Updating the current map and holding information about the level.

Attributes

- **Map map:** Represents the map that the level is played on.
- **MapExtractor mapExtractor:** Map extractor that handles level being loaded and saved from, to the local files.

Constructors

- **public MapController() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void loadLevel(int levelNo):** Load the level given the level no.
- **void loadOriginalLevel(int level):** Loads the original state of the given level.
- **public Map getMap():** Returns the map.
- **void updateMap():** Updated the grids of the map.
- **private GameObject getGameObjectBySelectedCell(int x, int y):** Returns the game object that occupies the given coordinates.
- **void removeGameObject(GameObject gameObject):** Removes the given gameObject from the map.
- **void addGameObject(GameObject gameObject):** Adds the the given gameObject to the map.
- **public boolean isPlayerAtExit():** Returns true if the player is at the exit.

3.4.16. MapDaoImpl Class

MapExtractor is responsible for extracting the map data from the local directory into the game.

Methods

- **public Map extractMap(int levelNo, Player player):** Extracts the map of the given level from the directory of the given player.

3.4.17. VehicleController Class

VehicleController class is responsible for controlling the vehicles on the map.

Attributes

- **private Map map:** Holds the map the vehicles are on.
- **private Vehicle selectedVehicle:** Represents the currently selected vehicle by the user.
- **private SoundManager soundManager:** Holds a reference to the Sound Manager.
- **private int numberOfMoves:** Holds the current number of moves that is made in the current level.
- **private CONTROL currentControl:** Holds the current selection for the control scheme.

Constructors

- **public VehicleController() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void setMap(Map _map):** Sets the map for the vehicle controller
- **public boolean tryMove(String direction):** Tries to move the vehicle 1 cell in the direction given and tests whether the move made by player is a valid move. If the vehicle can move in the given direction moves the vehicle and returns true, else it does not move the vehicle and returns false.
- **public void setSelectedVehicle(Vehicle selectedVehicle):** Sets the selected vehicle attribute to the given Vehicle.
- **public int getNumberOfMoves():** Returns the number of moves for the current level.

3.4.18. SoundManager Class

SoundManager class is responsible for all the sounds that can be heard in the game.

Attributes

- **AudioStream audioStream:** Audio stream.
- **InputStream inputStream:** Input stream.
- **private Clip clip:** Currently selected clip.
- **boolean isThemeEnabled:** Shows whether a theme is enabled or not.
- **boolean isEffectsEnabled:** Shows whether the sound effects for this player is enabled or not.

Constructors

- **public SoundManager() :** Empty constructor that initializes values to their specified initial values.

Methods

- **private void initializeClip():**
- **public void background():** Plays the background music.
- **public void vehicleHorn():** Plays the vehicle horn sound effect.
- **void updateTheme():** Updates the theme for the sounds.
- **public void buttonClick():** Plays the button click sound.
- **public void successSound():** Plays the success sound.
- **public void themeSongToggle():** Toggles the state of the theme song, on or off.
- **public void effectsToggle():** Toggles the state of the sound effects, on or off.
-

3.4.19. Input Class

Input class is responsible for handling player inputs.

Attributes

- **private static String[] mouseButtons:** Holds booleans that represents 5 mouse buttons and their states.
- **Map<String, Boolean> keys:** Holds the booleans for keys, whether they are pressed or not and connects the booleans and the strings that represent keys.
- **int mouseX:** Holds the x position of the mouse in terms of pixels.
- **int mouseY:** Holds the y position of the mouse in terms of pixels.

Constructors

- **public Input() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public boolean getKeyPressed(String keyID):** Returns true during the frame the user presses the the key given as parameter.
- **public boolean getMouseButtonPressed(int buttonID):** Returns true during the frame the user presses the the mouse button given as parameter.
- **static boolean getMouseButtonReleased(int buttonID):** Returns true during the frame the user releases the the mouse button given as parameter.
- **public int getMousePositionX():** Returns the current mouse position's x value in terms of pixels.
- **public int getMousePositionY():** Returns the current mouse position's x value in terms of pixels.
- **public int[] getMouseMatrixPosition():** Returns the current mouse position in term of cells.
- **public MouseListener getMouseListener():** Returns the mouse listener used by the input class.
- **public KeyboardListener getKeyboardListener():** Returns the keyboard listener used by the input class.

3.4.20. Controller Class

Controller class is the base class that every manager or controller is derived from.

Methods

- **public void start():** Called when the object is created before update methods. It is only called once.
- **public void update():** Called every frame from the Game Engine.

3.4.21. GameEngine Class

GameEngine class is responsible for handling the game loop and every class that is related to the game logic.

Attributes

- **private ArrayList<Controller> controllers:** Holds every controller that is being used to run the game.

Constructors

- **public GameEngine() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void run():** Calls the update methods of every Manager object that is inside managers array.

3.4.22. GuiPanelManager Class

GuiPanelManager class extends javax.swing.JFrame class, and it is responsible for managing the different panels of the game.

Attributes

- **public static GuiPanelManager instance:** The current instance of GuiPanelManager class for access from inside the panels.
- **private ArrayList<JPanel> panels:** Keeps the list of panels for quick access.
- **private GamePanel gamePanel:** Keeps the GamePanel instance.
- **private MainMenuPanel mainMenuPanel:** Keeps the MainMenuPanel instance.
- **private CreditsPanel creditsPanel:** Keeps the CreditsPanel instance.
- **private SettingsPanel settingsPanel:** Keeps the SettingsPanel instance.
- **private LevelSelectionPanel levelSelectionPanel:** Keeps the LevelSelectionPanel instance.
- **private HelpPanel helpPanel:** Keeps the HelpPanel instance.
- **private ChangePlayerPanel changePlayerPanel:** Keeps the ChangePlayerPanel instance.
- **private JPanel targetPanel:** Keeps the target panel before interactions with that panel.
- **int panelWidth:** Keeps the width of the panels.
- **int panelHeight:** Keeps the height of the panels.

Constructors

- **public GuiPanelManager() :** Initializes and configures the frame and the panels inside.

Methods

- **private void addPanels():** Function that creates the panels and adds them to the frame and the panels ArrayList.
- **public GamePanel getGamePanel():** Function to access the game panel.
- **void setPanelVisible(String panelName):** Sets the panel with the given name visible and sets the other panels invisible.
- **void updatePanels():** Updates the panels to display the latest changes to the screen.
- **public void updateImages():** Updates the images in all panels.
- **private void setListeners():** Sets the keyboard and mouse listeners.
- **public BufferedImage LoadImage(String FileName):** Returns the image with the given filename as BufferedImage.
- **int findCenter(int _panelWidth, Component _component):** Finds the center of the panel with the given length for the given component.

3.4.23. MainMenuPanel Class

The panel that holds the Main Menu Layout for our game

Attributes

- **private GuiPanelManager guiManager:** The GuiPanelManager instance for easy access to its functions.
- **private GameManager gameManager:** The gameManager instance for easy access to its functions.
- **private PlayerManager playerManager:** The playerManager instance for easy access to its functions.
- **private JLabel heading:** The label that displays the title of the game.
- **private JLabel player:** The label that displays the current player's name.
- **private JLabel starAmount:** The label that displays the amount of stars earned.
- **private JLabel lastLevel:** The label that displays the last played level.
- **private JButton changePlayer:** The button to go to ChangePlayerPanel.
- **private JButton play:** The button to go to GamePanel.
- **private JButton credits:** The button to go to CreditsPanel.
- **private JButton levels:** The button to go to LevelSelectionPanel.
- **private JButton settings:** The button to go to SettingsPanel.
- **private JButton exit:** The button to exit the game.
- **private JButton help:** The button to go to HelpPanel.
- **private BufferedImage background:** The background image.
- **private BufferedImage title:** The title image. Title is stored as image because Java does not support the font of the title.
- **private BufferedImage playButtonImage:** The image of the play button.
- **private BufferedImage playButtonImageHighlighted:** The image of the play button when the cursor is over it.
- **private BufferedImage creditsButtonImage:** The image of the credits button.
- **private BufferedImage creditsButtonHighlightedImage:** The image of the credits button when the cursor is over it.
- **private BufferedImage changePlayerButtonImage:** The image of the changePlayer button.
- **private BufferedImage changePlayerButtonHighlightedImage:** The image of the changePlayer button when the cursor is over it.
- **private BufferedImage helpButtonImage:** The image of the help button.
- **private BufferedImage helpButtonHighlightedImage:** The image of the help button when the cursor is over it.
- **private BufferedImage levelsButtonImage:** The image of the levels button.
- **private BufferedImage levelsButtonHighlightedImage:** The image of the levels button when the cursor is over it.
- **private BufferedImage exitButtonImage:** The image of the exit button.

- **private BufferedImage exitButtonHighlightedImage:** The image of the exit button when the cursor is over it.
- **private BufferedImage settingsButtonImage:** The image of the settings button.
- **private BufferedImage settingsButtonHighlightedImage:** The image of the settings button when the cursor is over it.
- **private BufferedImage starAmountImage:** The image of the StarAmount label
- **private int panelWidth:** Width of the panel.
- **private int panelHeight:** Height of the panel.
- **private ActionListener actionListener:** The ActionListener instance to capture button clicks.

Constructors

- **public MainMenuPanel(GuiPanelManager _guiManager) :** Initializes and configures the panel.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **private void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.
- **void updatePanel():** Updates the panel to display the latest changes to the components.
- **private void updateLastLevel():** Updates the lastLevel label.
- **private void updatePlayerName():** Updates the player label.
- **private void updateNumberOfStars():** Updates the starAmount label.
- **private void setBoundsOfComponents():** Sets the sizes and positions of the components in the panel.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **private void drawBackground(Graphics graphics):** The method that draws the background image to the background of the panel.

3.4.24. LevelSelectionPanel Class

The panel that holds the Levels Menu Layout for our game

Attributes

- **private GuiPanelManager guiManager:** The GuiPanelManager instance for easy access to its functions.
- **private LevelButton[] buttonArray:** The array that holds LevelButtons.
- **private JButton rightArrowButton:** The button to go to the next page of levels.
- **private JButton leftArrowButton:** The button to go to the previous page of levels.
- **private JButton menuButton:** The button to go back to the main menu.
- **private BufferedImage background:** The background image.
- **private BufferedImage rightArrow:** The image of the rightArrowButton.
- **private BufferedImage rightArrowHighlighted:** The image of the rightArrowButton when the cursor is over it.
- **private BufferedImage leftArrow:** The image of the leftArrowButton.
- **private BufferedImage leftArrowHighlighted:** The image of the leftArrowButton when the cursor is over it.
- **private BufferedImage back:** The image of the back button.
- **private BufferedImage backHighlighted:** The image of the back button when the cursor is over it.
- **private int panelWidth:** Width of the panel.
- **private int panelHeight:** Height of the panel.
- **private int page:** Holds the current page no.
- **private int pageLength:** The maximum amount of buttons that can be shown in a page.
- **private int numberOfLevels:** Total number of levels.
- **private ActionListener actionListener:** The ActionListener instance to capture button clicks.

Constructors

- **LevelSelectionPanel(GuiPanelManager _guiManager):** Initializes and configures the panel.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **private void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.
- **void updatePanel():** Updates the panel to display the latest changes to the components.

- **void updateButtons():** Updates the buttons to reflect the latest state of the game.
- **private void setBoundsOfComponents():** Sets the sizes and positions of the components in the panel.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **private void drawBackground(Graphics graphics):** The method that draws the background image to the background of the panel.

3.4.25. LevelButton Class

The the special type of button in LevelSelectionPanel which holds the level number; number of stars earned in that level; active and inactive star images; locked, unlocked and highlighted unlocked background images; and its own dimensions.

Attributes

- **GuiManagerPanel guiManager:** The GuiPanelManager instance for easy access to its functions.
- **JLabel[] stars:** Represents the collection of labels which indicates stars
- **BufferedImage levelBackground:** Represents the image of the background of the level button
- **BufferedImage levelBackgroundHighlighted:** Represents the image of the background of the level button during the mouse click.
- **BufferedImage starActive:** Represents the active stars which are earned by the player at that level.
- **BufferedImage starInactive:** Represents the inactive stars which are lost by the player at that level.
- **Dimension levelButtonDimension:** Represents the dimension of level button.
- **int LevelNo:** Represents the level number as an integer.
- **boolean isLocked:** Represents if the level is locked or not.
- **ActionListener actionListener:** The ActionListener instance to capture button clicks.

Constructors

- **private LevelButton(GuiPanelManager _guiManager):** Constructor that initializes regarding values and creates desired user interface of level button.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **private void createComponents():** Creates the components from the loaded images.

- **private void addComponents():** Adds the components to the panel.
- **private void setBoundsOfComponents(int page):** Sets the sizes and positions of the components in the panel. Parameter page controls the current page number.
- **public void updatePanel():** Updates the panel to display the latest changes to the components.
- **private void setupButton():**Method to create setup button.
- **void setLevelNo(int _levelNo):** Method to set the specific levels number.
- **void showStars(int starAmount):** Method to display the stars on desired level.
- **void toggleLock(boolean state):** Method to lock the toggle with considering the state.

3.4.26. HelpPanel Class

The panel that holds the Help Screen for our game

Attributes

- **int HELP_LABEL_WIDTH:** Represents the width of a help label.
- **Int HELP_LABEL_HEIGHT:** Represents the height of a help label.
- **GuiPanelManager guiManager:** The GuiPanelManager instance for easy access to its functions.
- **JLabel headling:** Represents the heading of the help panel.
- **JLabel[] helpLabels:** Represents the collection of labels of help panel.
- **JButton rightArrowButton:** Represents the right arrow button.
- **JButton leftArrowButton:** Represents the left arrow button.
- **JButton back:** Represents the back button.
- **BufferedImage title:** Represents the image of the title of help panel.
- **BufferedImage backButtonImage:** Represents the image of back button before the possible mouse click.
- **BufferedImage backButtonHighlightedImage:** Represents the image of back button during the possible mouse click.
- **BufferedImage rightArrowImage:** Represents the image of right arrow button before the possible mouse click.
- **BufferedImage rightArrowHighLightedImage:** Represents the image of right arrow button during the possible mouse click.
- **BufferedImage leftArrowImage:** Represents the image of left arrow button before the possible mouse click.
- **BufferedImage leftArrowHighlightedImage:** Represents the image of left arrow button during the possible mouse click.
- **BufferedImage[] helpImages:** Represents the image collection of the help panel.
- **Int panelWidth:** Represents the width of a panel.
- **Int panelHeight:** Represents the height of a panel.

- **Int page:** Represents the number of pages.
- **Int pageLength:** Represents the length of each page.
- **actionListener ActionListener:** The ActionListener instance to capture button clicks.

Constructors

- **private HelpPanel(GuiPanelManager _guiManager):** Constructor that initializes regarding values and creates desired user interface of help screen.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **private void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.
- **void updatePanel():** Updates the panel to display the latest changes to the components.
- **private void setBoundsOfComponents(int page):** Sets the sizes and positions of the components in the panel. Parameter page controls the current page number.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **private void drawBackground(Graphics graphics):** The method that draws the background image to the background of the panel.
- **private void updateVisiblePage():** Method to update the number of visible pages.

3.4.27. GamePanel Class

The panel that holds the Layout for the Play Game Screen. The actual gameplay is in the innerGamePanel panel which is held inside this panel as the game attribute.

Attributes

- **GuiPanelManager guiManager:** The GuiPanelManager instance for easy access to its functions.
- **InnerGamePanel innerGamePanel:** Represents the instance of the panel that the actual game is running.
- **JButton menu:** Represents menu button to return on the main menu.
- **JButton reset:** Represents reset button to start over the same level.
- **JButton settings:** Represents the setting button in order to go to settings.
- **JButton shrink:** Represents the special case button to shrink the desired obstacle to make the level easier to solve.

- **JButton space:** Represents the space feature button which changes the background of inner game to the space theme.
- **JLabel moveLabel:** Represents the move label. (?)
- **JLabel numberLabel:** Represents the number of the label. (?)
- **JProgressBar timer:** Represents the timer of the game.
- **BufferedImage timerIcon:** Represents the image of timer icon.
- **BufferedImage background:** Represents the image of the background of the game.
- **BufferedImage menuButtonImage:** Represents the image of menu button before the possible mouse click.
- **BufferedImage menuButtonHighlightedImage:** Represents the image of menu button during the possible mouse click.
- **BufferedImage resetButtonImage:** Represents the image of reset button before the possible mouse click.
- **BufferedImage resetButtonHighlightedImage:** Represents the image of reset button during the possible mouse click.
- **BufferedImage settingsButtonImage:** Represents the image of settings button before the possible mouse click.
- **BufferedImage settingsButtonHighlightedImage:** Represents the image of settings button during the possible mouse click.
- **BufferedImage shrinkButtonImage:** Represents the image of shrink button before the possible mouse click.
- **BufferedImage shrinkButtonHighlightedImage:** Represents the image of shrink button during the possible mouse click.
- **BufferedImage spaceButtonImage:** Represents the image of space button before the possible mouse click.
- **BufferedImage spaceButtonHighlightedImage:** Represents the image of space button during the possible mouse click.
- **BufferedImage movesImage:** Represents the image of number of moves.
- **Int panelWidth:** Represents the panel's width.
- **Int panelHeight:** Represents the panel's height
- **ActionListener actionListener:** The ActionListener instance to capture button clicks.

Constructors

- **private GamePanel(GuiPanelManager _guiManager):** Constructor that initializes regarding values and creates desired user interface of total game.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **private void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.

- **private void setBoundsOfComponents(int page):** Sets the sizes and positions of the components in the panel. Parameter page controls the current page number.
- **public void updatePanel():** Updates the panel to display the latest changes to the components.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **private void drawBackground(Graphics graphics):** The method that draws the background image to the background of the panel.
- **public void setEndOfLevelPanelVisible(int starAmount):** Method to make visible the end of level panel and demonstrate the earned stars with considering the number of moves.
- **public void setInnerGamePanelVisible():** Method to set visible the inner game panel which is a panel that game is played.
- **public void setInnerGameVisible():** Method to set visible the inner game which is the screen that game is played.
- **private void createInnerGamePanel():** Method to set visible the inner game panel which is a panel that game is played.
- **private InnerGamePanel getInnerGamePanel():** Method to get the inner game panel which returns desired inner game panel.
- **private void updateNumberOfMoves():**Method to update the number of moves with considering the moves of the player.
- **private void setTimerBar(boolean isActive, int msTime):** Method to set the timer bar to start.

3.4.28. InnerGamePanel Class

The panel that the actual game is running.

Attributes

- **private GuiPanelManager guiManager:** The GuiPanelManager instance for easy access to its functions.
- **public EndOfLevelPanel endOfLevelPanel:** The endOfLevelPanel instance to display when the level has finished
- **private Map map:** The current state of the map while the game is progressing

Constructors

- **InnerGamePanel(GuiPanelManager guiManager):** Initializes and configures the panel.

Methods

- **void updatePanel():** Updates the panel to display the latest changes to the components.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **void setEndOfLevelPanelVisible(boolean visible, int starAmount):** Sets the endOfLevelPanel visible, shows stars in the endOfLevelPanel, and plays the success sound.
- **private void createEndOfLevelPanel():** Creates and configures the endOfLevelPanel and adds it to itself as a component.

3.4.29. EndOfLevelPanel Class

EndOfLevel panel is to demonstrate a User Interface at the end of levels, it has particular buttons and labels and with their pictures to form the end of level's interface.

Attributes

- **GuiManagerPanel guiManager:** Represents the instance of GuiManagerPanel in order to form the user interface.
- **JButton retry:** Represents retry button to retry the same level at the end of a level.
- **JButton menu:** Represents menu button to return on the main menu at the end of a level.
- **JButton nextLevel:** Represents next level button in order to move on next level at the end of a level.
- **JLabel headling:** Represents the heading of the end of the level
- **JLabel[] stars:** Represents the amount of star that player earns after that level.
- **BufferedImage menuButtonImage:** Represents the image of menu button before the possible mouse click.
- **BufferedImage menuButtonHighlightedImage:** Represents the image of menu button during the possible mouse click.
- **BufferedImage retryButtonImage:** Represents the image of retry button before the possible mouse click.
- **BufferedImage retryButtonHighlightedImage:** Represents the image of retry button during the possible mouse click.
- **BufferedImage nextLevelButtonImage:** Represents the image of nextLevel button before the possible mouse click.
- **BufferedImage nextLevelHighlightedImage:** Represents the image of nextLevel button during the possible mouse click.
- **BufferedImage starImage:** Represents the image of star to demonstrate the earned stars from that level.
- **BufferedImage starLockedImage:** Represents the image of star to demonstrate the lost stars from that level.
- **Int panelWidth:** Represents the width of a panel.

- **Int panelHeight:** Represents the height of a panel.
- **actionListener ActionListener:** The ActionListener instance to capture button clicks.

Constructors

- **private EndOfLevelPanel(GuiPanelManager _guiManager):** Constructor that initializes regarding values and creates desired user interface of end of a level.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **private void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.
- **private void setBoundsOfComponents(int page):** Sets the sizes and positions of the components in the panel. Parameter page controls the current page number.
- **public void updatePanel():** Updates the panel to display the latest changes to the components.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **private void drawBackground(Graphics graphics):** The method that draws the background image to the background of the panel.
- **private void showStars(int StarAmount):):** Method to show the stars that earned by the end of a level.

3.4.30. ChangePlayerPanel Class

The panel that holds the Change Player Screen Layout for our game.

Attributes

- **private GuiPanelManager guiManager:** The GuiPanelManager instance for easy access to its functions.
- **private GameManager gameManager:** The gameManager instance for easy access to its functions.
- **public CreatePlayerPopUp popUp:** The pop-up panel that is displayed when the user wants to create a new player.
- **private ArrayList<JButton> buttonArray:** The list that holds player selection buttons.
- **private JButton rightArrowButton:** The button to go to the next page of players.

- **private JButton leftArrowButton:** The button to go to the previous page of players.
- **private JButton menuButton:** The button to go back to the main menu.
- **private JButton addButton:** The button to add a new player.
- **private JButton deleteButton1:** The button to delete the first player of the current page.
- **private JButton deleteButton2:** The button to delete the second player of the current page.
- **private JButton deleteButton3:** The button to delete the third player of the current page.
- **private JButton editButton1:** The button to edit the first player of the current page.
- **private JButton editButton2:** The button to edit the second player of the current page.
- **private JButton editButton3:** The button to edit the third player of the current page.
- **private ArrayList<String> playerNameArray:** The list that holds the player names.
- **private BufferedImage background:** The background image of the panel.
- **private BufferedImage levelBackground:** The background image of the player selection buttons.
- **private BufferedImage levelBackgroundH:** The background image of the player selection buttons when the cursor is over it.
- **private BufferedImage rightArrow:** The image of rightArrowButton.
- **private BufferedImage leftArrow:** The image of leftArrowButton.
- **private BufferedImage leftArrowH:** The image of leftArrowButton when the cursor is over it.
- **private BufferedImage rightArrowH:** The image of rightArrowButton when the cursor is over it
- **private BufferedImage add:** The image of addButton.
- **private BufferedImage addH:** The image of addButton when the cursor is over it
- **private BufferedImage edit:**The image of editButton.
- **private BufferedImage editH:**The image of editButton when the cursor is over it
- **private BufferedImage delete:**The image of deleteButton.
- **private BufferedImage deleteH:**The image of deleteButton when the cursor is over it
- **private BufferedImage back:**The image of menuButton.
- **private BufferedImage backHighlighted:**The image of menuButton when the cursor is over it
- **private int panelWidth:** Width of the panel.
- **private int panelHeight:** Height of the panel.
- **private int currentPage:** Holds the current page no.
- **private int numberOfPlayers:** Total number of players.

- **private int limit:** The lower bound of the buttons that are currently being displayed on the screen. (currentPage * pageLength)
- **private ActionListener actionListener:** The ActionListener instance to capture button clicks.

Constructors

- **ChangePlayerPanel(GuiPanelManager _guiManager):** Initializes and configures the panel.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **private void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.
- **void updatePanel():** Updates the panel to display the latest changes to the components.
- **private void setBoundsOfComponents(int page):** Sets the sizes and positions of the components in the panel. Parameter page controls the current page number.
- **private void selectPlayer(String name):** Selects the player with the given name as the active player.
- **void addPlayer(String name):** Creates a new player with the given name and adds it to the game.
- **private void deletePlayer(String name):** Deletes the player with the given name from the game.
- **public void updatePanel():** Updates the panel to display the latest changes to the components.
- **private void updatePages():** Updates the pages of the panel.
- **private void updateButtons():** Updates the buttons of the panel.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **private void drawBackground(Graphics graphics):** The method that draws the background image to the background of the panel.

3.4.31. CreatePlayerPopUp Class

The panel that holds the pop-up layout for the create new player functionality our game

Attributes

- **private GuiPanelManager guiManager:** The GuiPanelManager instance for easy access to its functions.
- **private ChangePlayerPanel changePlayerPanel:** The ChangePlayerPanel instance that created this panel.
- **private JTextField playerName:** The text field to enter the player name.
- **private JButton close:** The close button to discard creating a new player.
- **private JButton confirm:** The confirm button to confirm creating a new player.
- **private BufferedImage background:** The background image of this pop-up.
- **private BufferedImage closeImage:** The image of close button.
- **private BufferedImage closeHighlightedImage:** The image of close button when the cursor is over it.
- **private BufferedImage confirmImage:** The image of confirm button.
- **private BufferedImage confirmHighlightedImage:** The image of confirm button when the cursor is over it.
- **private int panelWidth:** Width of the pop-up panel.
- **private int panelHeight:** Height of the pop-up panel.
- **private ActionListener actionListener:** The ActionListener instance to capture button clicks.

Constructors

- **CreatePlayerPopUp(GuiPanelManager _guiManager, ChangePlayerPanel _changePlayerPanel):** Initializes and configures the panel.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.
- **void updatePanel():** Updates the panel to display the latest changes to the components.
- **private void setBoundsOfComponents():** Sets the sizes and positions of the components in the panel.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **private void drawBackground(Graphics graphics):** The method that draws the background image to the background of the panel.

- **void requestFocusForTextField():** Requests focus for the playerName text field.

3.4.32. SettingsPanel Class

The panel that holds the Settings Screen Layout for our game

Attributes

- **GuiPanelManager guiManager:** The GuiPanelManager instance for easy access to its functions.
- **JButton music:** Represents the music button to control the music, open it or lower it.
- **JButton sfx:** Represents the sfx button to activate special effects.
- **JButton back:** Represents the back button in order to go back to the page that user was before.
- **JButton minimalistic:** Represents the minimalistic button to activate the minimalistic feature.
- **JButton classic:** Represents the classic button to activate the classic feature.
- **JButton safari:** Represents the safari button to activate the safari feature.
- **JButton space:** Represents the space button to activate the space feature.
- **JButton volume:** Represents the volume button to decrease or increase the volume.
- **JButton heading:** Represents the heading of the settings.
- **JButton theme:** Represents the button of theme.
- **BufferedImage title:** Represents the image of title of settings.
- **BufferedImage backButtonImage:** Represents the image of back button before the possible mouse click.
- **BufferedImage backButtonHighlightedImage:** Represents the image of back button during the possible mouse click.
- **BufferedImage musicImage:** Represents the image of music button before the possible mouse click.
- **BufferedImage musicHighlightedImage:** Represents the image of music button during the possible mouse click.
- **BufferedImage musicOffImage:** Represents the image of music is off before the possible mouse click.
- **BufferedImage musicOffHighlightedImage:** Represents the image of music is off during the possible mouse click.
- **BufferedImage sfxImage:** Represents the image of sfx button before the possible mouse click.
- **BufferedImage sfxHighlightedImage:** Represents the image of sfx button during the possible mouse click.
- **BufferedImage sfxOffImage:** Represents the image of sfx is off before the possible mouse click.

- **BufferedImage sfxOffHighlightedImage:** Represents the image of sfx is off during the possible mouse click.
- **BufferedImage simpleImage:** Represents the image of minimalistic button before the possible mouse click.
- **BufferedImage simpleHighlightedImage:** Represents the image of minimalistic button during the possible mouse click.
- **BufferedImage classicImage:** Represents the image of classic mode button before the possible mouse click.
- **BufferedImage classicHighlightedImage:** Represents the image of classic mode button during the possible mouse click.
- **BufferedImage safariImage:** Represents the image of safari mode button before the possible mouse click.
- **BufferedImage safariHighlightedImage:** Represents the image of safari mode button during the possible mouse click.
- **int panelWidth:** Represents the width of a panel.
- **int panelHeight:** Represents the height of a panel.
- **String previousPanel:** Represents the previous panel as a string.
- **ActionListener actionListener:** The ActionListener instance to capture button clicks.

Constructors

- **SettingsPanel(GuiPanelManager _guiManager):** Initializes and configures the panel.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.
- **void updatePanel():** Updates the panel to display the latest changes to the components.
- **private void updateSoundButton(String type):** Updates the sound button to reflect the current state of the sounds in game.
- **private void updateThemeButton(String type):** Updates the theme button to reflect the current state of the themes in game.
- **private void setBoundsOfComponents():** Sets the sizes and positions of the components in the panel.
- **public void paintComponent(Graphics g):** The method that paints the panel to the screen.
- **private void drawBackground(Graphics graphics):** The method that draws the background image to the background of the panel.

3.4.33. CreditsPanel Class

The panel that holds the Credits Screen for our game.

Attributes

- **private JLabel heading:** The label that displays the title of the panel.
- **private JLabel name1:** Name of the first developer of the game.
- **private JLabel name2:** Name of the second developer of the game.
- **private JLabel name3:** Name of the third developer of the game.
- **private JLabel name4:** Name of the fourth developer of the game.
- **private JLabel name5:** Name of the fifth developer of the game.
- **private JButton back:** The back button to go back to the main menu.
- **private BufferedImage background:** The background image.
- **private BufferedImage title:** The title image. Title is stored as image because Java does not support the font of the title.
- **private BufferedImage backButtonImage:** The image of back button.
- **private BufferedImage backButtonHighlightedImage:** The image of back button when the cursor is over it.
- **private int panelWidth:** Width of the panel.
- **private int panelHeight:** Height of the panel.
- **private ActionListener actionListener:** The ActionListener instance to capture button clicks.

Constructors

- **public CreditsPanel(GuiPanelManager _guiManager) :** Initializes and configures the panel.

Methods

- **public void loadImages():** Loads the images from the images directory into the memory.
- **void createComponents():** Creates the components from the loaded images.
- **private void addComponents():** Adds the components to the panel.
- **void updatePanel():** Updates the panel to display the latest changes to the components.
- **private void setBoundsOfComponents():** Sets the sizes and positions of the components in the panel.

3.4.34. UIFactoryClass

The class holds static functions and Dimension variables to create the needed UI elements in a convenient way.

Attributes

- **private static Dimension longButtonDimension:** Dimensions for long buttons.
- **private static Dimension squareButtonDimension:** Dimensions for square buttons.
- **private static Dimension playButtonDimension:** Dimensions for play button. (in MainMenuPanel)
- **private static Dimension arrowButtonDimension:** Dimensions for arrow buttons. (in paged panels)
- **private static Dimension levelButtonDimension:** Dimensions for level buttons. (in LevelSelectionPanel)
- **private static Dimension playerButtonDimension:** Dimensions for player buttons. (in ChangePlayerPanel)
- **private static Dimension miniStarDimension:** Dimensions for mini stars. (in LevelButton)
- **private static Dimension movesCarDimension:** Dimensions for moveLabel. (in GamePanel)
- **private static Dimension starAmountDimension:** Dimensions for starAmount label. (in MainMenuPanel)

Methods

- **static LevelButton createLevelButton(ActionListener actionListener):** Creates level buttons.
- **static JButton createPlayerButton(BufferedImage normalImage, BufferedImage highlightedImage, String playerName, ActionListener actionListener):** Creates player buttons.
- **static JButton createButton(BufferedImage normalImage, BufferedImage highlightedImage, String buttonType, ActionListener actionListener):** Creates other types of buttons.
- **static JLabel createLabelIcon(BufferedImage image, String labelType):** Creates labels with icons.
- **private static void setupButton(JButton button, BufferedImage normalImage, BufferedImage highlightedImage, String buttonType, ActionListener actionListener):** Configures the created buttons.
- **private static void setupLabelIcon(JLabel label, BufferedImage image, String labelType):** Configures the created labels.

3.4.35. Main Class

Main class is responsible for connecting the Model and Controller with the View. Additionally it is responsible for setting up the Game Engine and starting a thread for the GameEngine. In this Game Engine thread, the run method of Game Engine is called 60 times every frame.

4. Improvements Summary

- New design patterns, Data Access Object (DAO) and Factory have been added.
- Deployment diagram has been added.
- Boundary Use Case diagram has been added.
- New classes such as Theme, ThemeManager, BonusLevelInformation, Obstacle, PowerUpController have been added. Some existing classes such as Reward, RewardSystem have been removed. PlayerExtractor, MapExtractor, MapSaver classes have been replaced with PlayerDao, MapDao interfaces and PlayerDaoImpl, MapDaoImpl classes in order to fit the newly added Data Access Object design pattern.
- Subsystem Decomposition diagram and Subsystem Component diagram have been updated according to the renovated structure.
- Test Package has been updated in the Package diagram.

5. References

[1]‘Chapter 6 System Design: Decomposing the System’. Presentation, Bilkent University, 2018. https://mcavus.github.io/cs319/material/ch06lect1_ET.pdf

[2]B. Bruegge and A. H. Dutoit, Object-oriented software engineering using UML, patterns, and java. Harlow: 3rd Edition, 2010, 227.

[3]‘Chapter 7 - System Design: Addressing Design Goals’. Presentation, Bilkent University, 2018. https://mcavus.github.io/cs319/material/ch07lect1_ET.pdf

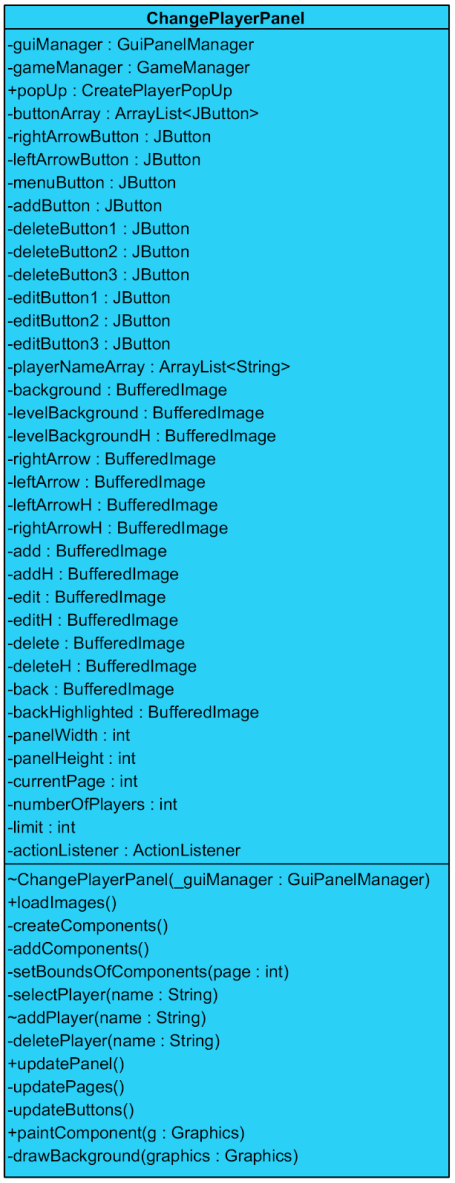
6. Appendix

6.1. View Class Diagram

The attributes and methods were omitted from the class diagram for readability. Here are the full attributes and methods for each class of the view package. There is also the UIFactory class which only has static attributes and methods.


EndOfLevelPanel
+guiManager : GuiPanelManager
-retry : JButton
-menu : JButton
-nextLevel : JButton
-heading : JLabel
-stars : JLabel[]
-background : BufferedImage
-menuButtonImage : BufferedImage
-menuButtonHighlightedImage : BufferedImage
-retryButtonImage : BufferedImage
-retryButtonHighlightedImage : BufferedImage
-nextLevelButtonImage : BufferedImage
-nextLevelHighlightedImage : BufferedImage
-starImage : BufferedImage
-starLockedImage : BufferedImage
-panelWidth : int
-panelHeight : int
-actionListener : ActionListener
~EndOfLevelPanel(_guiManager : GuiPanelManager)
+loadImages()
~updatePanel()
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)
-createComponents()
-addComponents()
-setBoundsOfComponents()
~showStars(starAmount : int)
LevelSelectionPanel
-guiManager : GuiPanelManager
-buttonArray : JButton[]
-rightArrowButton : JButton
-leftArrowButton : JButton
-backButton : JButton
-background : BufferedImage
-backButtonImage : BufferedImage
-backButtonHighlightedImage : BufferedImage
-rightArrowImage : BufferedImage
-rightArrowHighlightedImage : BufferedImage
-leftArrowImage : BufferedImage
-leftArrowHighlightedImage : BufferedImage
-panelWidth : int
-panelHeight : int
-page : int
-pageLength : int
-numberOfLevels : int
-popUp : LevelSelectionPopUp
-actionListener : ActionListener
~LevelSelectionPanel(_guiManager : GuiPanelManager)
+loadImages()
-createComponents()
-addComponents()
-setBoundsOfComponents()
-updateButtons()
+findNoOfLevels() : int
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)
HelpPanel
-HELP_LABEL_WIDTH : int
-HELP_LABEL_HEIGHT : int
-guiManager : GuiPanelManager
-heading : JLabel
-helpLabels : JLabel[]
-rightArrowButton : JButton
-leftArrowButton : JButton
-back : JButton
-background : BufferedImage
-title : BufferedImage
-backButtonImage : BufferedImage
-backButtonHighlightedImage : BufferedImage
-rightArrowImage : BufferedImage
-rightArrowHighlightedImage : BufferedImage
-leftArrowImage : BufferedImage
-leftArrowHighlightedImage : BufferedImage
-helpImages : BufferedImage[]
-panelWidth : int
-panelHeight : int
-page : int
-pageLength : int
-actionListener : ActionListener
~HelpPanel(_guiManager : GuiPanelManager)
+loadImages()
-createComponents()
-addComponents()
-setBoundsOfComponents()
-updateVisiblePage()
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)
CreatePlayerPopUp
-guiManager : GuiPanelManager
-playerName : JTextField
-close : JButton
-confirm : JButton
-background : BufferedImage
-closeImage : BufferedImage
-closeHighlightedImage : BufferedImage
-confirmImage : BufferedImage
-confirmHighlightedImage : BufferedImage
-panelWidth : int
-panelHeight : int
-actionListener : ActionListener
-changePlayerPanel : ChangePlayerPanel
~CreatePlayerPopUp(_guiManager : GuiPanelManager, _changePlayerPanel : ChangePlayerPanel)
+loadImages()
~createComponents()
-addComponents()
-setBoundsOfComponents()
~updatePanel()
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)
~requestFocusForTextField()
GuiPanelManager
-instance : GuiPanelManager
-panels : ArrayList<JPanel>
-gamePanel : GamePanel
-mainMenuPanel : MainMenuPanel
-creditsPanel : CreditsPanel
-settingsPanel : SettingsPanel
-levelSelectionPanel : LevelSelectionPanel
-helpPanel : HelpPanel
-changePlayerPanel : ChangePlayerPanel
-targetPanel : JPanel
~panelWidth : int
~panelHeight : int
+GuiPanelManager()
-addPanels()
+getGamePanel() : GamePanel
~setPanelVisible(panelName : String)
~updatePanels()
+updateImages()
-setListeners()
+LoadImage(FileName : String) : BufferedImage
~findCenter(_panelWidth : int, _component : Component) : int
CreditsPanel
-guiManager : GuiPanelManager
-heading : JLabel
-name1 : JLabel
-name2 : JLabel
-name3 : JLabel
-name4 : JLabel
-name5 : JLabel
-back : JButton
-background : BufferedImage
-title : BufferedImage
-backButtonImage : BufferedImage
-backButtonHighlightedImage : BufferedImage
-panelWidth : int
-panelHeight : int
-actionListener : ActionListener
~CreditsPanel(_guiManager : GuiPanelManager)
+loadImages()
-createComponents()
-addComponents()
-setBoundsOfComponents()
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)
LevelButton
-guiManager : GuiPanelManager
-stars : JLabel[]
-levelBackground : BufferedImage
-levelBackgroundHighlighted : BufferedImage
-starActive : BufferedImage
-starInactive : BufferedImage
-lockedBackground : BufferedImage
-levelButtonDimension : Dimension
-levelNo : int
-isLocked : boolean
-actionListener : ActionListener
~LevelButton(_guiManager : GuiPanelManager)
-loadImages()
-createComponents()
-addComponents()
-setBoundsOfComponents()
-setupButton()
~setLevelNo(_levelNo : int)
~showStars(starAmount : int)
~toggleLock(state : boolean)
InnerGamePanel
-guiManager : GuiPanelManager
-endOfLevelPanel : EndOfLevelPanel
-map : Map
~InnerGamePanel(_guiManager : GuiPanelManager)
~updatePanel()
+paintComponent()
~setEndOfLevelPanelVisible(visible : boolean, starAmount : int)
-createEndOfLevelPanel()


GamePanel
~guiManager : GuiPanelManager
-innerGamePanel : InnerGamePanel
-menu : JButton
-reset : JButton
-settings : JButton
-shrink : JButton
-space : JButton
-moveLabel : JLabel
-numberLabel : JLabel
-timer : JProgressBar
-timerIcon : BufferedImage
-background : BufferedImage
-menuButtonImage : BufferedImage
-menuButtonHighlightedImage : BufferedImage
-resetButtonImage : BufferedImage
-resetButtonHighlightedImage : BufferedImage
-settingsButtonImage : BufferedImage
-settingsButtonHighlightedImage : BufferedImage
-shrinkButtonImage : BufferedImage
-shrinkButtonHighlightedImage : BufferedImage
-spaceButtonImage : BufferedImage
-spaceButtonHighlightedImage : BufferedImage
-movesImage : BufferedImage
-panelWidth : int
-panelHeight : int
-actionListener : ActionListener
~GamePanel(_guiManager : GuiPanelManager)
+updatePanel()
-loadImages()
-createComponents()
-addComponents()
-setBoundsOfComponents()
+setEndOfLevelPanelVisible(starAmount : int)
+setInnerGamePanelVisible()
-createInnerGamePanel()
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)
~getInnerGamePanel() : InnerGamePanel
-updateNumberOfMoves()
-setTimerBar(isActive : boolean, msTime : int)
MainMenuPanel
-guiManager : GuiPanelManager
-gameManager : GameManager
-playerManager : PlayerManager
-heading : JLabel
-player : JLabel
-starAmount : JLabel
-lastLevel : JLabel
-changePlayer : JButton
-play : JButton
-credits : JButton
-levels : JButton
-settings : JButton
-exit : JButton
-help : JButton
-background : BufferedImage
-title : BufferedImage
-playButtonImage : BufferedImage
-playButtonImageHighlighted : BufferedImage
-creditsButtonImage : BufferedImage
-creditsButtonHighlightedImage : BufferedImage
-changePlayerButtonImage : BufferedImage
-changePlayerButtonHighlightedImage : BufferedImage
-helpButtonImage : BufferedImage
-helpButtonHighlightedImage : BufferedImage
-levelsButtonImage : BufferedImage
-levelsButtonHighlightedImage : BufferedImage
-exitButtonImage : BufferedImage
-exitButtonHighlightedImage : BufferedImage
-settingsButtonImage : BufferedImage
-settingsButtonHighlightedImage : BufferedImage
-starAmountImage : BufferedImage
-panelWidth : int
-panelHeight : int
-actionListener : ActionListener
~MainMenuPanel(_guiManager : GuiPanelManager)
+loadImages()
-createComponents()
-addComponents()
~updatePanel()
-updateLastLevel()
-updatePlayerName()
-updateNumberOfStars()
-setBoundsOfComponents()
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)
SettingsPanel
~guiManager : GuiPanelManager
-music : JButton
-sfx : JButton
-back : JButton
-minimalistic : JButton
-classic : JButton
-safari : JButton
-space : JButton
-heading : JLabel
-volume : JLabel
-theme : JLabel
-title : BufferedImage
-background : BufferedImage
-backButtonImage : BufferedImage
-backButtonHighlightedImage : BufferedImage
-musicImage : BufferedImage
-musicHighlightedImage : BufferedImage
-musicOffImage : BufferedImage
-musicOffHighlightedImage : BufferedImage
-sfxImage : BufferedImage
-sfxHighlightedImage : BufferedImage
-sfxOffButtonImage : BufferedImage
-sfxOffHighlightedImage : BufferedImage
-simpleImage : BufferedImage
-simpleHighlightedImage : BufferedImage
-classicImage : BufferedImage
-classicHighlightedImage : BufferedImage
-safariImage : BufferedImage
-safariHighlightedImage : BufferedImage
-spaceImage : BufferedImage
-spaceHighlightedImage : BufferedImage
-panelWidth : int
-panelHeight : int
-actionListener : ActionListener
-previousPanel : String
~SettingsPanel(_guiManager : GuiPanelManager)
~updatePanel()
+loadImages()
-createComponents()
-addComponents()
-setBoundsOfComponents()
-updateSoundButtons(type : String)
-getThemeNameByButton(button : JButton) : String
-updateThemeButtons()
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)
UIFactory
-longButtonDimension : Dimension
-squareButtonDimension : Dimension
-playButtonDimension : Dimension
-arrowButtonDimension : Dimension
-levelButtonDimension : Dimension
-miniStarDimension : Dimension
-movesCarDimension : Dimension
-starAmountDimension : Dimension
~createLevelButton(actionListener : ActionListener) : LevelButton
~createPlayerButton(normalImage : BufferedImage, highlightedImage : BufferedImage, playerName : String, actionListener : ActionListener) : JButton
~createButton(normalImage : BufferedImage, highlightedImage : BufferedImage, buttonType : String, actionListener : ActionListener) : JButton
~createLabelIcon(image : BufferedImage, labelType) : JLabel
-setupButton(button JButton, normalImage : BufferedImage, highlightedImage : BufferedImage, buttonType : String, actionListener : ActionListener)
-setupLabelIcon(label : JLabel, image : BufferedImage, labelType : String)


ChangePlayerPanel
-guiManager : GuiPanelManager
-gameManager : GameManager
+popUp : CreatePlayerPopUp
-buttonArray : ArrayList<JButton>
-rightArrowButton : JButton
-leftArrowButton : JButton
-menuButton : JButton
-addButton : JButton
-deleteButton1 : JButton
-deleteButton2 : JButton
-deleteButton3 : JButton
-editButton1 : JButton
-editButton2 : JButton
-editButton3 : JButton
-playerNameArray : ArrayList<String>
-background : BufferedImage
-levelBackground : BufferedImage
-levelBackgroundH : BufferedImage
-rightArrow : BufferedImage
-leftArrow : BufferedImage
-leftArrowH : BufferedImage
-rightArrowH : BufferedImage
-add : BufferedImage
-addH : BufferedImage
-edit : BufferedImage
-editH : BufferedImage
-delete : BufferedImage
-deleteH : BufferedImage
-back : BufferedImage
-backHighlighted : BufferedImage
-panelWidth : int
-panelHeight : int
-currentPage : int
-numberOfPlayers : int
-limit : int
-actionListener : ActionListener
~ChangePlayerPanel(_guiManager : GuiPanelManager)
+loadImages()
-createComponents()
-addComponents()
-setBoundsOfComponents(page : int)
-selectPlayer(name : String)
~addPlayer(name : String)
-deletePlayer(name : String)
+updatePanel()
-updatePages()
-updateButtons()
+paintComponent(g : Graphics)
-drawBackground(graphics : Graphics)