
Bilkent University

Department of Computer Engineering

# CS-319 Project

*Quadrillion*

## Design Report

### Group No: 1H
### Group Name: COGENE

Osman Orhan Uysal

Samet Özcan

Mehmet Alper Karadağ

Ziya Erkoç

Talha Murathan Göktaş

# 1. Introduction

## a. Purpose of the system

Board games have been one of the ways of entertaining, socializing and sharing since they have been designed and created. There are many versions of them available to fulfill these necessities of people from all ages. However, like many other things, they also have been virtualized which is the starting point of this project. Many additional features increasing factors from competition to entertainment are possible to be added into the game thanks to digital environment. There are other advantages of virtualized board games compared to the real ones such as mobility and easiness of set-up and playing etc. Thus, the purpose of the system is to provide entertainment, imagination and preferably competitive atmosphere together in a virtual environment.

Quadrillion is a board game in which the board is composed of 4 board pieces and there are 12 pieces in different shapes to be placed on the board. The aim in the game is to fill all the blanks on the board by placing all 12 pieces successfully onto it. The 4 board pieces can be flipped, rotated and be bonded from all sides in many combinations which make many possible board styles possible. In this virtualized version of the Quadrillion Game many levels with distinct board arrangements will be provided for the players along with features such as leaderboard and hint usage.

## b. Design goals

1. Usability

   We want our game to be easily used by most of the players. Therefore, we aim to make the components of the game more flexible and more easy to use.

2. Performance

   Our game want much require performance requirements but we aim to decrease the time to reach database and do database related actions and also to connect to the Pos Service in a shorter time not to much devastate the experience.

3. Reliability

   We aim to make our game less frequent to get crashed.

4. Security

We want our game to protect private information of the users by avoiding the storage of confidential information such as credit card information and also to apply encryption methods.

5. Functionality
   We want to have lots of functionality in the game and give the user a wider range of activities to do within the game.

6. Supportability
   Our game can run on any machine which has Java installed regardless of the operating system (Windows, Mac OS, Linux) thanks to JVM (Java Virtual Machine).

## c. Trade-Offs

- **Security vs Usability**
  Although we ask user his/her credit-card information over and over again in order not to create security vulnerabilities, this indeed is going to decrease usability because user will be bored of entering his/her credit card information repeatedly.

- **Reliability vs Usability**
  We promised the ability to move the piece by holding from its any parts instead of assigning a pivot point on the piece and only allowing the movement of the piece through that point. However, creating multiple places to hold the piece may increase the chance of bug while rotating and flipping the piece and decrease the reliability.

- **Supportability vs Performance**
  We wanted our application to work on Windows, Mac and Linux and to achieve this we used Java which helps creating cross-platform applications. However, Java's cross-platform feature is not without its limitations which is performance. Hence, by using Java we sacrificed performance for the sake of supportability.

- **Functionality vs Usability**
  As mentioned, usability is very significant for a virtualized versions of the board games since this is the first thing the players looks for when they try to play the game because no matter single-player or multiplayer, the concept of board games is based on the entertainment of the player(s) without confronting them too many or rough difficulties. In other words, we thought that simplicity must be the most important factor of these

type of games. However, we decided to add some features such as time, move counter, competitive style and hint to vary the functions in the game. Nonetheless, the common issue for these new functions is that they have basic concepts and limited functional requirements. In other words, we planned to increase functionality as long as it does not disturb the overall simplicity and therefore usability of the game. In short, we considered to pay attention to both functionality and usability, but usability has the priority.

# 2. High-level software architecture

## a. Three-tier Architecture

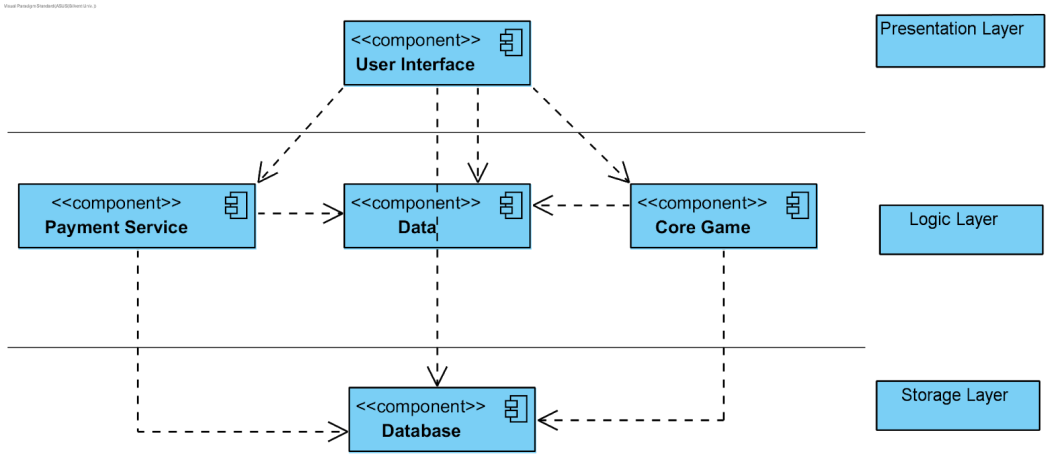


Figure 1: Component Diagram for the Architecture

## b. Subsystem Decomposition

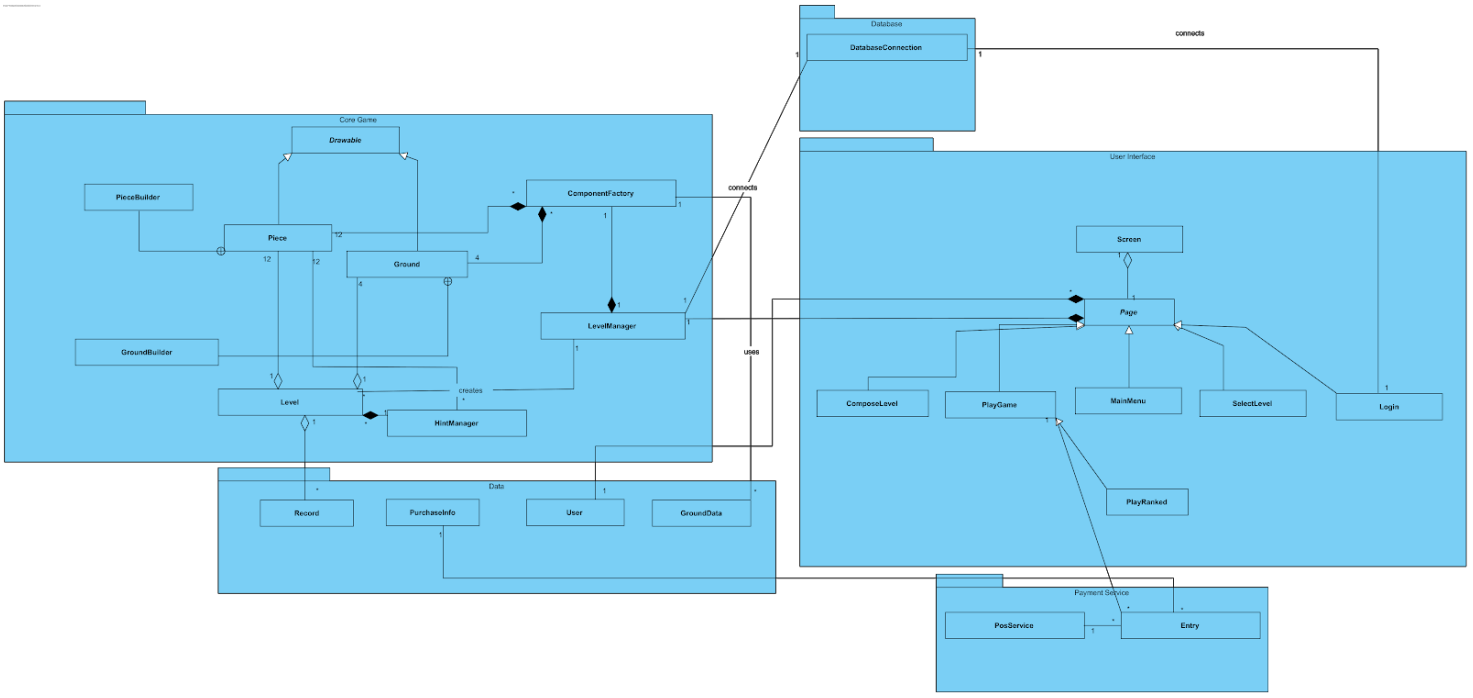


Figure 2: Subsystem Diagram

Our subsystem consists of 5 subsystems each of which has a common role. This decomposition is beneficial for us because each subsystem matches with the skills of one or two members of our team. Although all subsystems and classes it encapsulates will be explained in detail below, we shall briefly describe what each subsystem does.

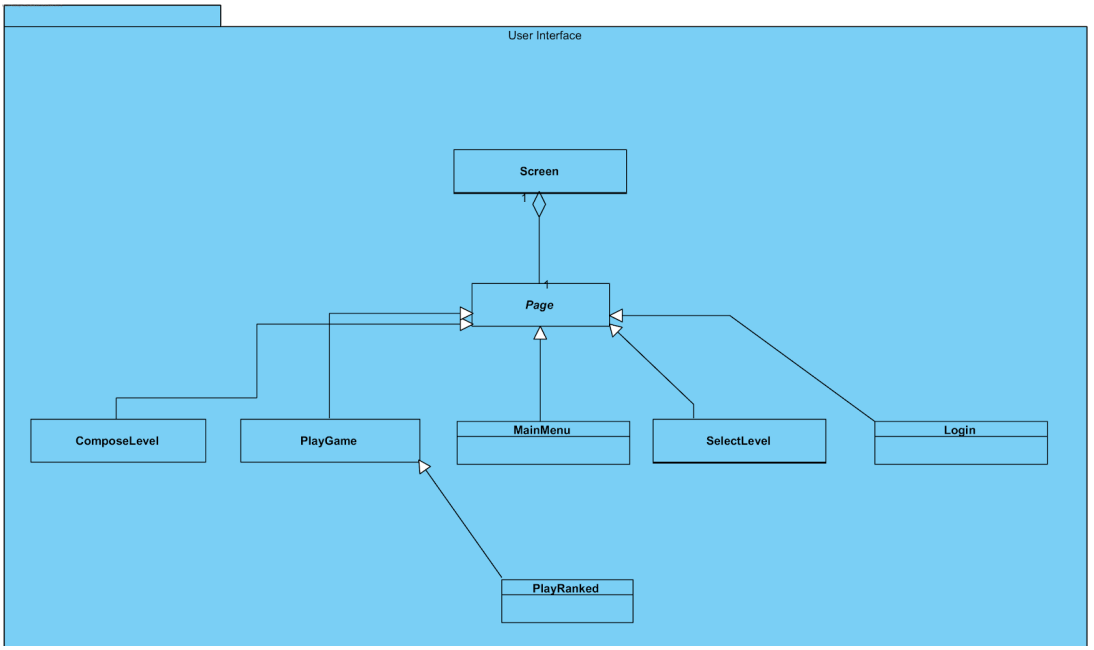


Figure 3: User Interface Subsystem

i. **User Interface:** This subsystem consists of classes that are responsible for how user interface looks like. That is, how labels, buttons and text-fields are laid-out and also what happens when user interacts with them.

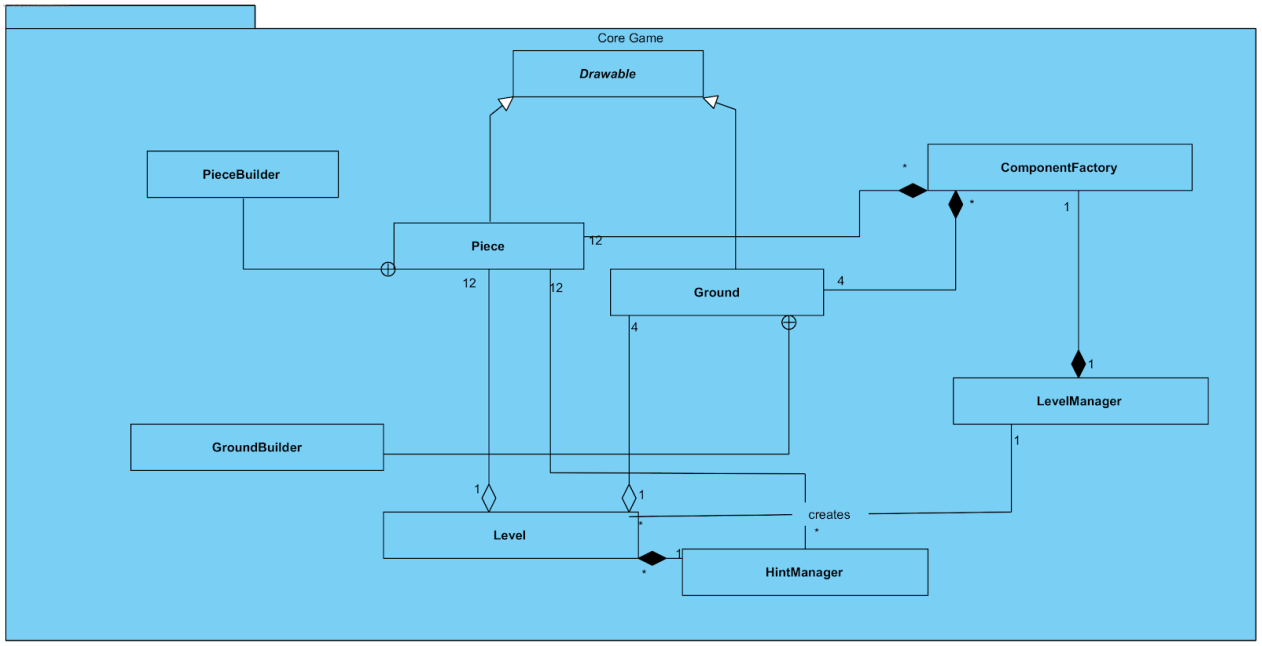


Figure 4: Core Game Subsystem

ii. **Core Game:** This subsystem is responsible, firstly, for checking main conditions such as if piece is placed correctly or if game is completed or this level is valid by doing calculations. Apart from that it is responsible for creating level, ground and piece as well as getting level from the database. Lastly, drawing pieces and grounds and handling their movement (move, rotate, flip, drag) are responsibilities of this subsystem.

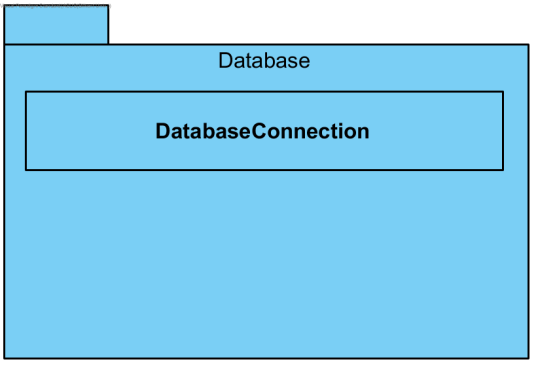


Figure 5: Database Subsystem

iii. **Database:** It is responsible for connecting to the database and executing SQL commands on it.

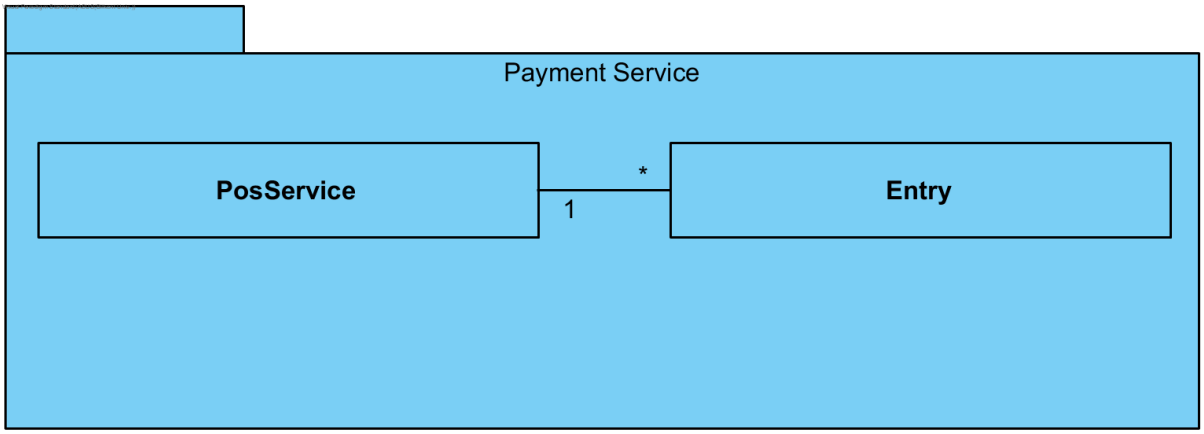


Figure 6: Payment Service Subsystem

iv. **Payment Service:** It is responsible for establishing connection with iyzipay transaction service and conducting transactions.

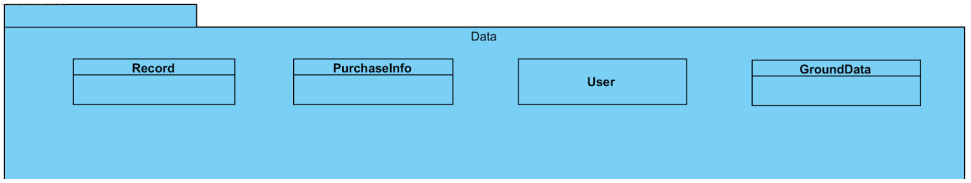


Figure 7: Data Subsystem

v. **Data:** This subsystem includes common data types that will be used in the application.

## c. Hardware/software mapping

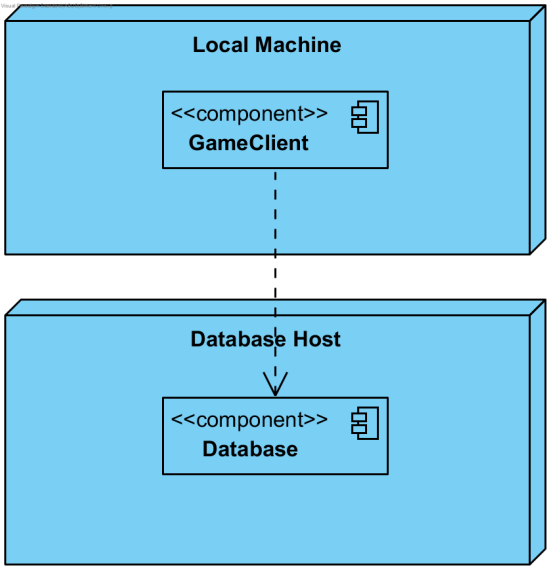


Figure 8: Deployment Diagram

Quadrillion is implemented using Java and JavaFX.

Player will need to download appropriate Java JRE to their computer. For gameplay inputs only Mouse will be necessary. Speaker is not necessary but will make the player hear sound effects.

The player needs to open .jar file of the game deployed in the file system of the operating system.

## d. Persistent data management

In our project, game data will be persistently stored in MySQL database. At first we will keep our database at local level. Having completed the test, we plan to rent a MySQL database that is located at a remote server. The name of our database is *quadrillion* and we have three different tables.

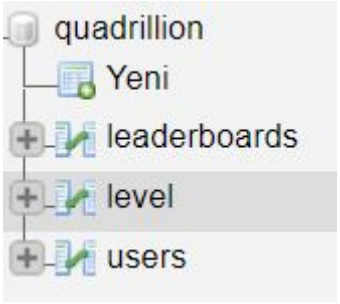


Figure 9: Database Structure

i. level

| # | Name | Type | Collation | Attributes | Null | Default | Comments | Extra |
|---|---|---|---|---|---|---|---|---|
| 1 | ID | int(11) | | | No | *None* | | AUTO_INCREMENT |
| 2 | ROTATIONS | text | utf8_general_ci | | No | *None* | | |
| 3 | LOCATIONS | text | utf8_general_ci | | No | *None* | | |
| 4 | ISFRONT | text | utf8_general_ci | | No | *None* | | |

Figure 10: Level Table Structure

This table holds the information about how each of the 4 sub-boards (grounds) are combined; namely, what their position is, how many times they are rotate and whether front face or back face is active. Below figure shows an example entry. Locations and Rotations are kept as Strings but in the code they will be tokenized.

| ID | ROTATIONS | LOCATIONS | ISFRONT |
|---|---|---|---|
| 1 | 1;2;3;4 | (100,100);(100,340);(340,160);(340,400) | 1;0;1;0 |

Figure 11: Level Table Example

ii. leaderboards

| # | Name | Type | Collation | Attributes | Null | Default | Comments | Extra |
|---|---|---|---|---|---|---|---|---|
| 1 | ID | int(11) | | | No | *None* | | AUTO_INCREMENT |
| 2 | USER_NICK | text | utf8_general_ci | | No | *None* | | |
| 3 | LEVEL_ID | int(11) | | | No | *None* | | |
| 4 | TIME_ELAPSED | int(11) | | | No | *None* | | |
| 5 | MOVES | int(11) | | | No | *None* | | |
| 6 | TOTAL_SCORE | int(11) | | | No | *None* | | |

Figure 12: Leaderboards Table Structure

This table holds information about rankings of each user for particular level. USER_NICK holds the nickname of the user, LEVEL_ID holds which level user played. TIME_ELAPSED and MOVES represent how much time elapsed for complete the level and number of moves played respectively. Lastly, TOTAL_SCORE will be calculated in the code based on the TIME_ELAPSED and MOVES. To calculate the leaderboard for a particular level WHERE and ORDER_BY keywords of SQL will be used. Below image is an example entry.

| ID | USER_NICK | LEVEL_ID | TIME_ELAPSED | MOVES | TOTAL_SCORE |
|---|---|---|---|---|---|
| 1 | Rgtemze | 1 | 100 | 5 | 10 |

Figure 13: Leaderboards Table Example

iii. users

| # | Name | Type | Collation | Attributes | Null | Default | Comments | Extra |
|---|---|---|---|---|---|---|---|---|
| 1 | ID | int(11) | | | No | *None* | | AUTO_INCREMENT |
| 2 | NICKNAME | text | utf8_general_ci | | No | *None* | | |
| 3 | PASS | text | utf8_general_ci | | No | *None* | | |
| 4 | HINT | int(11) | | | No | *None* | | |

Figure 14: Users Table Structure

This table holds information of the users. NICKNAME is the unique username of the user, PASS is the hashed version of the password of the user and hint is the number of hints he/she has. Below is the example entry.

| ID | NICKNAME | PASS | HINT |
|---|---|---|---|
| 1 | Rgtemze | sdfjsjgsgjsdj5h45545454jk | 15 |

Figure 15: Users Table Example

Our database holds user information and how grounds are oriented in a level. However, we did not want to persistently store how the Pieces are structured because they are static and won't change at all; hence, we hard-coded it. ComponentFactory class knows how each of the 12 pieces are structured and produces them.

## e. Access control and security

| | PlayGame | MainMenu | Login | ComposeLevel | Piece | Ground | PlayRanked |
|---|---|---|---|---|---|---|---|
| Player | PlayGame()<br>buyHint()<br>showHint()<br>retry()<br>exit() | startGame()<br>startRankedGame()<br>exit()<br>composeLevel() | login()<br>register() | submitLevel() | rotate()<br>flip()<br>draw()<br>move() | rotate()<br>flip<br>draw()<br>move() | uploadResults()<br>openRandomLevel() |

Figure x: Access Matrix

Quadrillion is a desktop game with single-player and multi-player options. Players are asked to choose their option after signing in. If they do not have an account they are asked to create one and accounts are kept in database system. In database system precautions for keeping the data secure will be taken. Users are also able to compose new levels for the game. In the game play the data of board and pieces is kept in database and if the mode was ranked the time duration is uploaded to the database system. However there is no option for users to delete other levels created by other users. In our database implementation discussed in above part, the security checks provide the security. Methods not related with the database cannot change the game thus pose no threat.

## Control Flow

Control flow of the game Quadrillion is event-driven which the events are in flow dependant to users decisions. The decisions are made in menus and

during the game-play. These events created according to decisions are controlled by instances of PlayGame, MainMenu and Level in the game Quadrillion.

### f. Boundary conditions

#### i. Start-Up

Players start playing the game Quadrillion by executing a .jar file.

#### ii. Shutdown

Players exit the game Quadrillion by clicking the Exit button on the main menu or clicking the cross button on the right corner of the window.

#### iii. Errors

In the game Quadrillion we will focus on the parts that are possible to cause errors. These parts are expected to be related with database interaction due to network problems and overrides, game end conditions and impossible combinations.

In database uploads we will keep the data locally before making sure that it is uploaded. This will happen when

1. user composes new level and uploads it to the system
2. ranked game is finished and result is uploaded to system

Also when hint is used, precautions will be taken to check whether hint is true and hint count reduces only when it is used.

When user closes the game without finishing the game moves will be deleted.

## 3. Subsystem services

### a. User Interface

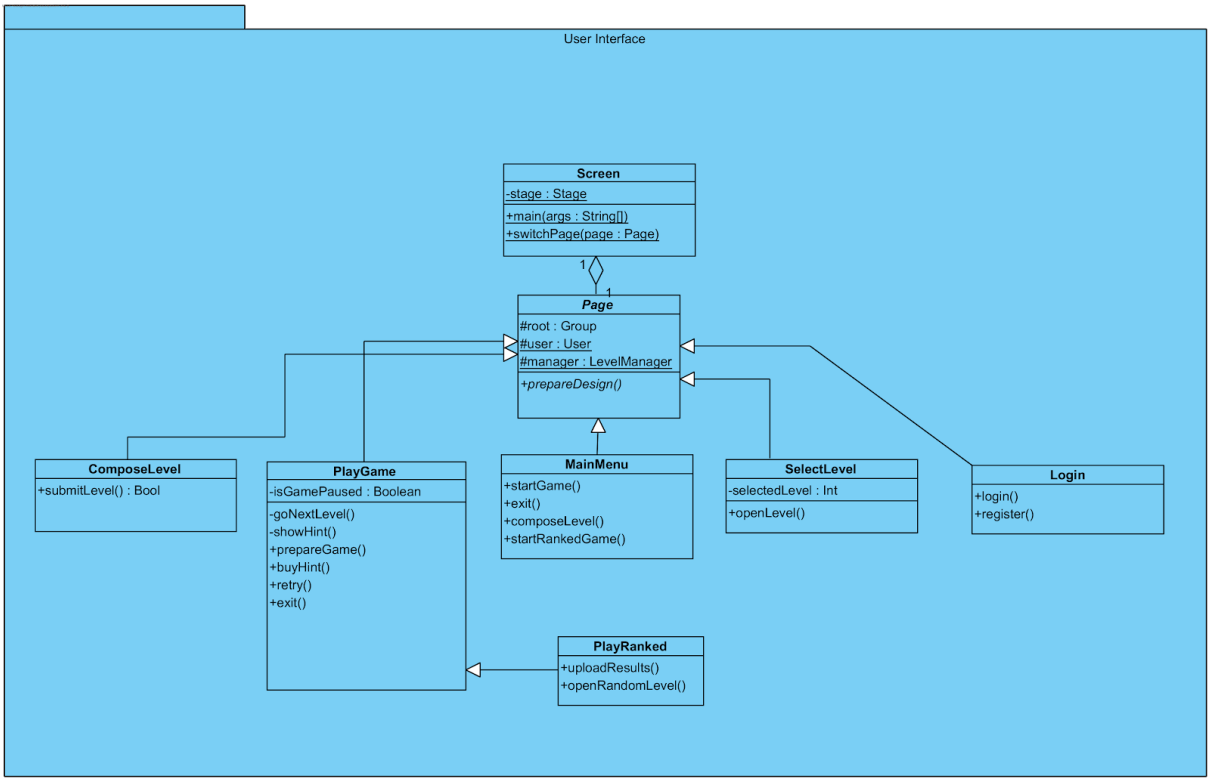


Figure 16: User Interface Subsystem (with non-collapsed fields)

i. Screen

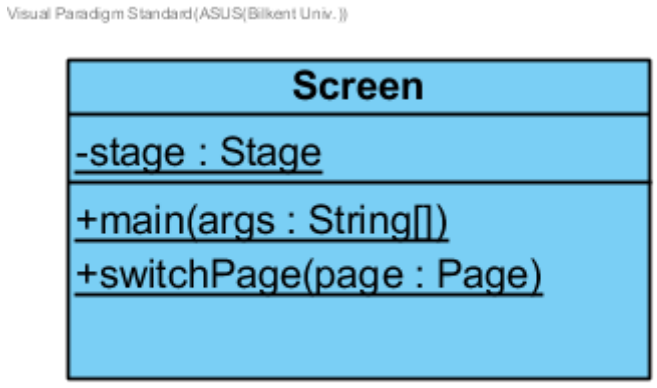


Figure 17: Screen Class

This class is the starting point of the application. It holds and controls the current game scene

Attributes:

- **private static Scene gameScene**

  This attribute holds the current scene that is shown to the user.

Methods:

- **public void start(Stage primaryStage)**

Starting point for every javafx program. It initializes the gameScene attribute as Login screen, sets the dimensions and title of the game window and shows everything on screen.

- **public static void switchPage(Page page)**
  This method sets the gameScene as given page.
- **public static void main( String[] args )**
  Main method that every Java program must have.

ii. Page

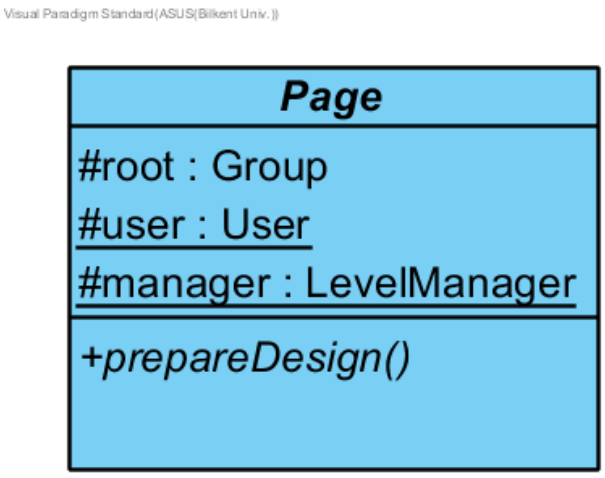


Figure 18: Page Class

This is an abstract class which every different scene in the game extends to prepare its unique design.

Attributes:

- **protected Group root**
  This holds Group object to draw and remove on the screen.
- **protected static User user**
  This holds the sole user instance that is used throughout the game.
- **protected static LevelManager manager**
  This is the only LevelManager instance that is initialized at the beginning of the game and used throughout.

Methods:

- **public abstract void prepareDesign()**
  This is an abstract method that is implemented in child classes.

iii. ComposeLevel

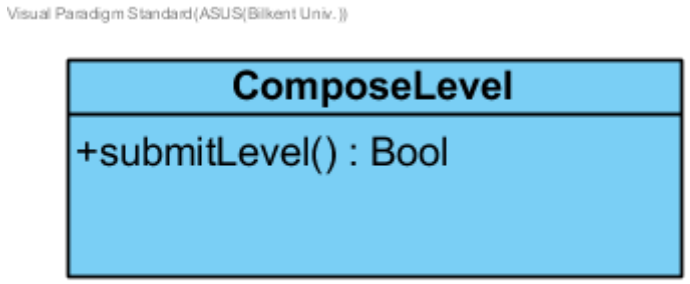


Figure 19: ComposeLevel Class

This class handles the compose level screen of the game.
LevelManager class takes all of the burden of this class. This class will communicate it through LevelManager object.
Methods:

- **private boolean submitLevel()**
  This method delegates LevelManager to submit level. If level is valid true is returned; otherwise; false.

iv. PlayGame

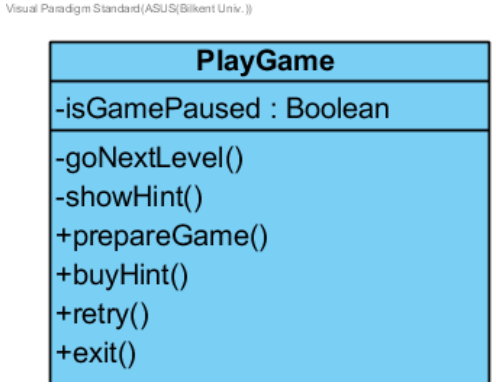


Figure 20: PlayGame Class

This class handles the game and its state.
Attributes:

- **private boolean isGamePaused**
  Holds if the game is paused or being played at the moment.

Methods:

- **private void goNextLevel()**
  Changes the current level with the next level.
- **private void showHint()**
  It delegates LevelManager class to show hint.
- **public void prepareGame()**

Creates current level with pieces and grounds in it.

- **public void buyHint()**
  Gets an instance of PosService and calls buy method.
- **public void retry()**
  Restarts same level again.
- **public void exit()**
  Exits to the main menu.

v. PlayRanked

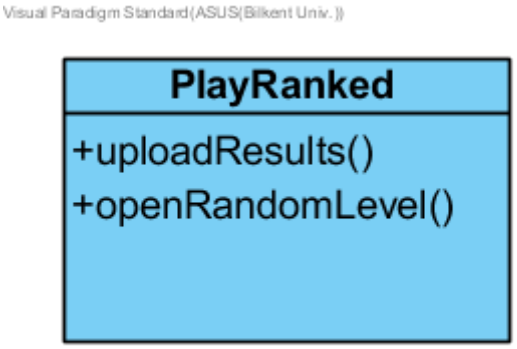


Figure 21: PlayRanked Class

This is a child class of PlayGame class. It handles additional properties that are necessary for ranked games.
Methods:

- **public void uploadResults()**
  Uploads the records of the finished game to the leaderboard of the corresponding level in database.
- **public void openRandomLevel()**
  Selects a random level between all levels and sets as current level.

vi. MainMenu

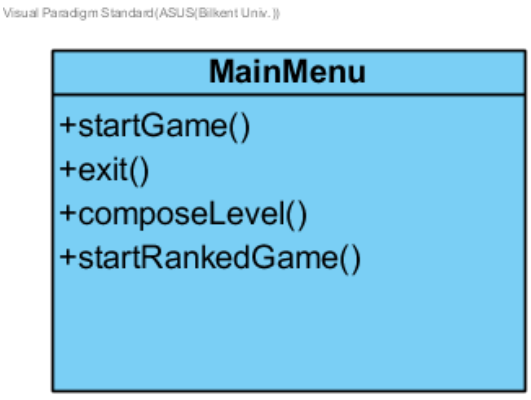


Figure 22: MainMenu Class

This is the screen that user sees after log in. User has 4 different options: Play casual game, Play ranked game, Compose level, Exit.
Methods:

- **public void startGame()**
  Changes the scene as SelectLevel scene.
- **public void exit()**
  Closes the game.
- **public void ComposeLevel()**
  Initiates the level composition screen using ComposeLevel class.
- **public void startRankedGame()**
  Initiates a ranked game using RankedGame and PlayGame classes.

vii. SelectLevel

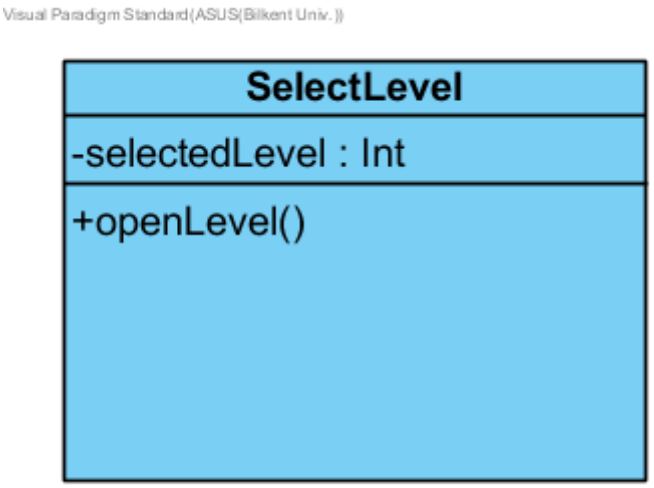


Figure 23: SelectLevel Class

This class handles the select level screen of the game.
Attributes:

- **private int selectedLevel**
  Holds the number of selected level

Methods:

- **public void openLevel( int levelNo )**
  Prepares the given level and initiates game using PlayGame class.

viii. Login

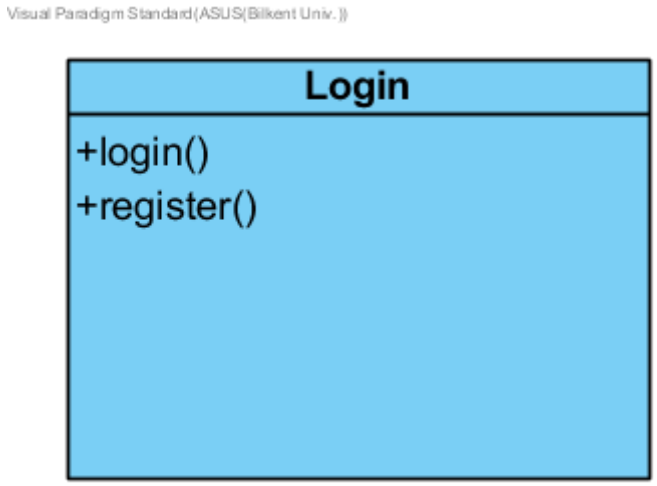


Figure 24: Login Class

This is the first screen that user will encounter when s/he opens up the game.

Methods:

- **public void login()**
  Tries to connect with database. When connection is established checks if the given ID and password matches. If the result is successful changes the gameScene to MainMenu.
- **public void register()**
  Tries to connect with database. When connection is established checks if there is a user with same ID. If the user ID is unique, registers the user to the database. Then, calls the login method.

b. Data

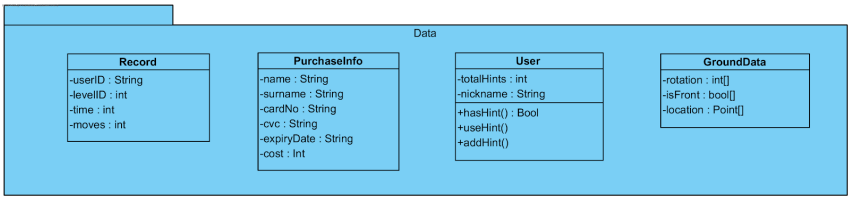


Figure 25: Data Subsystem (with non-collapsed fields)

i. Record

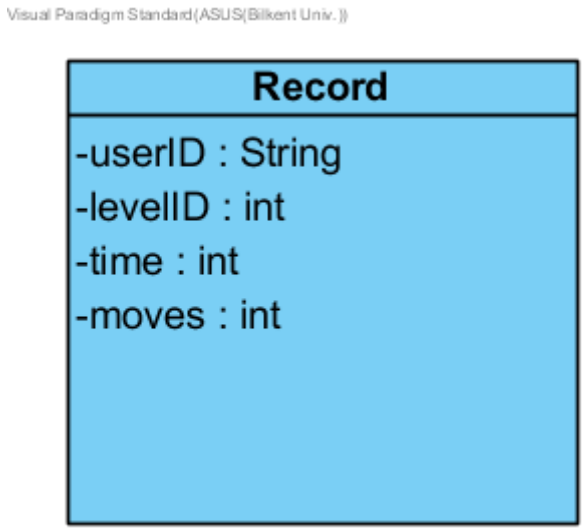


Figure 26: Record Class

This class is used for keeping the ranked game records.

Attributes:

- **private String userID**
  Holds the ID of the User
- **private int levelID**
  Holds the number of the Level
- **private int times**
  Keeps the time that user took to finish that Level.
- **private int moves**
  Keeps the number of moves that user made to finish that Level.

ii. PurchaseInfo

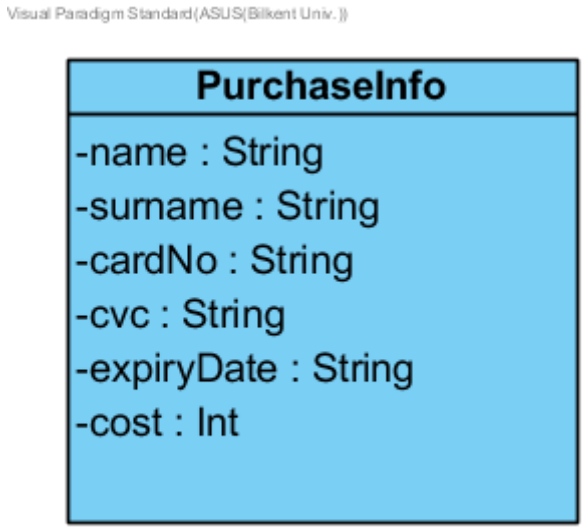


Figure 27: PurchaseInfo Class

This class keeps the purchase info.

Attributes:

- **private String name**
  The name of the card owner
- **private String surname**
  The surname of the card owner
- **private String cardNo**
  The credit card number
- **private String cvc**
  cvc number of the credit card
- **private String expiryDate**
  Expiration date of the credit card
- **private int cost**
  Total cost of the purchase

iii. GroundData

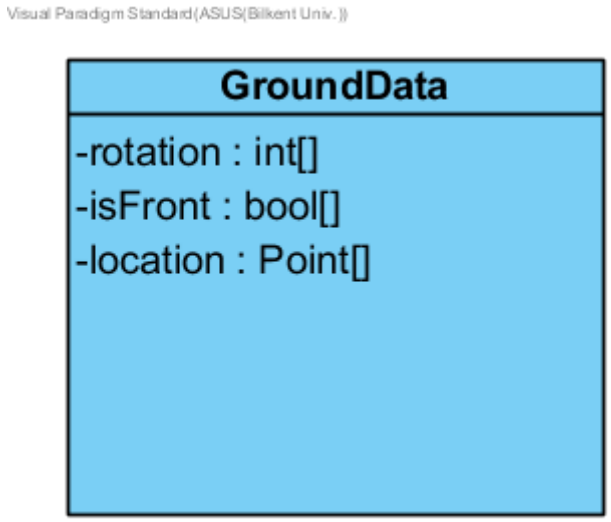


Figure 28: GroundData Class

This class represents information about the organizations of grounds fetched from the database. Hence, this class represents each row in our database. Since each Level has 4 grounds each array in the attributes will have 4 elements.

Attributes:

- **private int rotation[]**
  It holds how much each ground is rotated.
- **private int isFront[]**
  It holds whether each ground has frontboard as the active board or backboard as the active board.
- **private Point location[]**
  It holds each location of the grounds.

iv. User

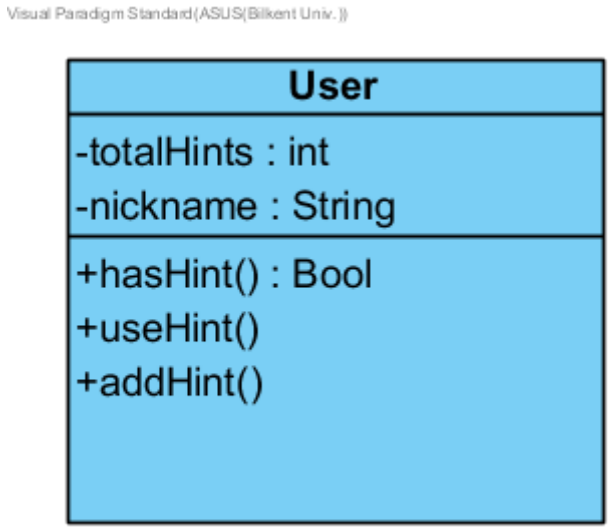


Figure 29: User Class

This class keeps the hint information of the user.

Attributes:

- **private int totalHints**
  Keeps the number of hints that user has.

Methods:

- **public boolean hasHint()**
  Returns true if the user has at least 1 hint.
- **public void useHint()**
  Decreases the number of hints user has by 1.
- **public void addHint()**
  Increases the number of hints user has by 1.

c. PaymentService

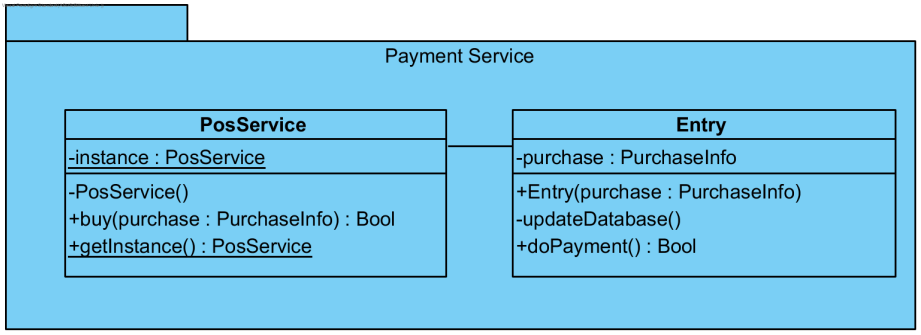


Figure 30: Payment Service Subsystem (with non-collapsed fields)

i. PosService

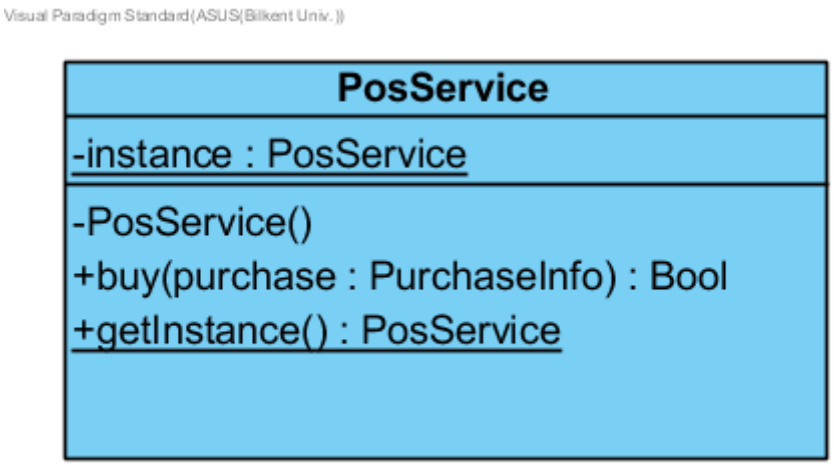


Figure 31: PosService Class

This is a singleton class to connect with Pos Service.

Attributes:

- **private static PosService instance**
  It is the unique object of this class that is created when object is constructed and continues to exist as long as program runs.

Constructor(s):

- **private PosService ()**
  It is a private constructor which means the *instance* object can be only initialized inside the class because of Singleton pattern.

Methods:

- **public static PosService getInstance()**
  It returns the unique instance of the class.
- **public bool buy(PurchaseInfo purchase)**
  It executes purchase operation by using given purchase information.

ii. Entry

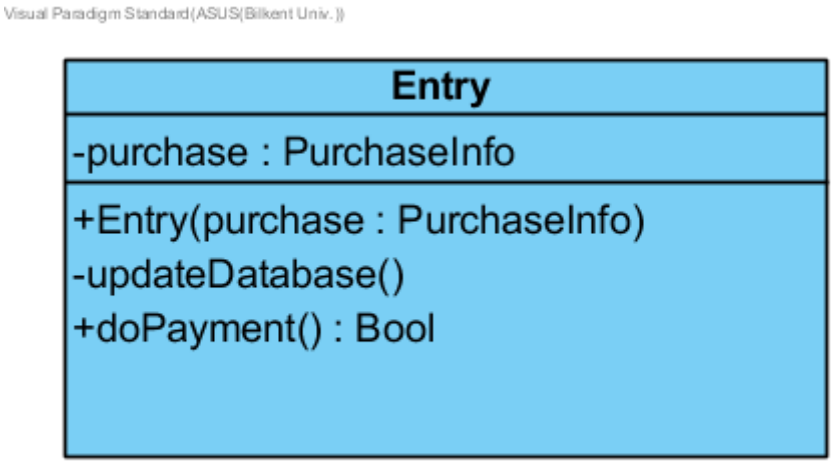


Figure 32: Entry Class

This class is responsible for purchase operation and updating the database when user successfully purchases hints.

Attributes:

- **private PurchaseInfo purchase**
  It holds information about purchase such as credit card information and the amount of money the user will be charged.

Constructor(s):

- **public Entry(PurchaseInfo purchase)**
  It constructs the object by using existing purchase object.
  Methods:
- **private void updateDatabase()**
  If payment operation is successfull this method updates the number of hints the user have in the database.
- **public bool doPayment()**
  It delegates ‘buy’ method of PosService method to complete the payment. This method returns the success status of the ‘buy’ method.

## d. Database

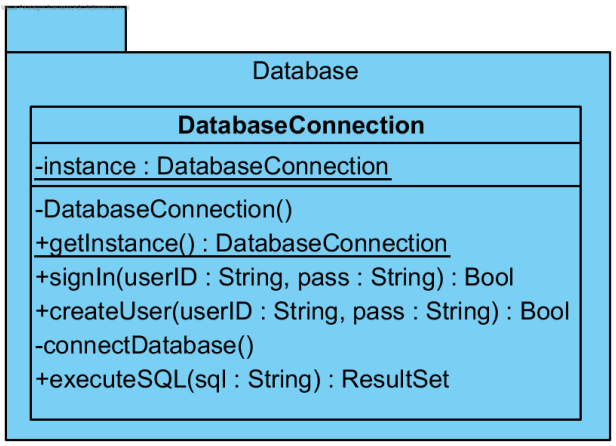


Figure 33: Database Subsystem (with non-collapsed fields)

### i. DatabaseConnection

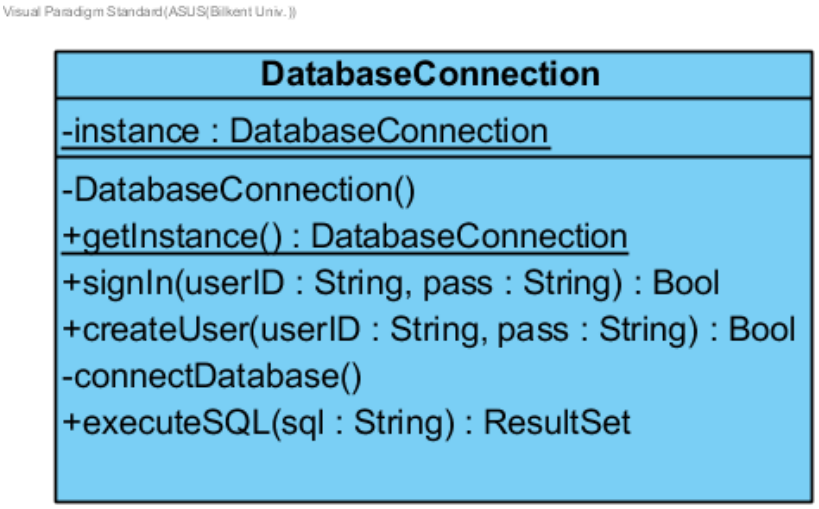


Figure 34: DatabaseConnection Class

This is a singleton class that helps establishing connection with the MySQL database.

Attributes:

- **private static DatabaseConnection instance**
  It is the unique object of this class that is created when object is constructed and continues to exist as long as program runs.

Constructor(s):

- **private DatabaseConnection()**
  It is a private constructor which means the *instance* object can be only initialized inside the class because of Singleton pattern.

Methods:

- **public static DatabaseConnecion getInstance()**
  It returns the unique instance of the class.
- **public bool signIn(String id, String pass)**
  It asks database to check if given ID exists and matches with the password. It also returns if the operation is successful.
- **public bool createUser(String id, String pass)**
  It asks database to check if given ID already taken exists. If so user is signed-up. It also returns if the operation is successful.
- **public bool connectDatabase(String id, String pass)**
  It established the initial connection with the database. Success status is returned from this method.
- **public bool executeSQL(String sql)**
  It executes SQL commands. Additional measurements will be taken to avoid SQL-injection attack.

## e. Core Game

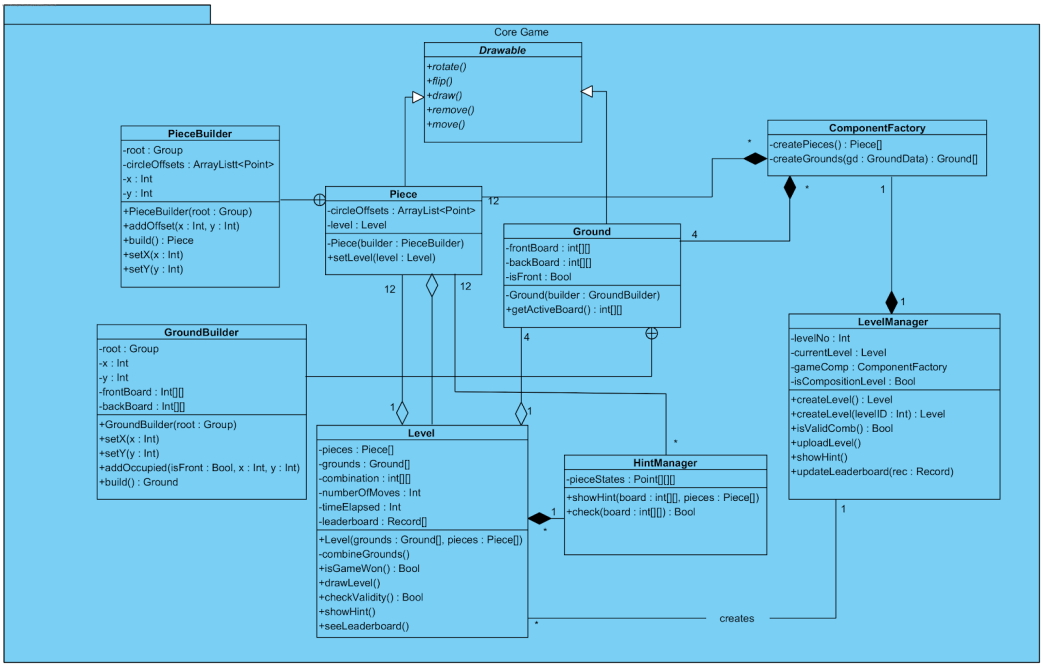


Figure 35: Core Game Subsystem

i. Level

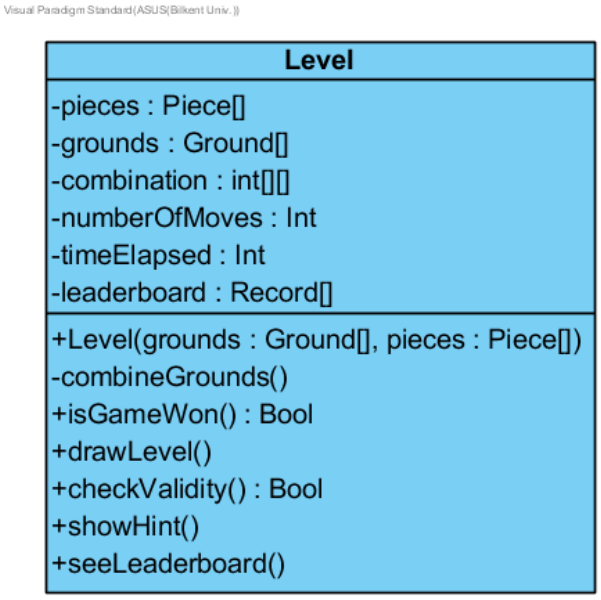


Figure 36: Level Class

This class holds all information about a particular setting and state of the particular level; namely, how sub-boards (Grounds) are oriented, how pieces are located. Besides, it regulates hint revelation process, induces other components to be drawn and controls if game is completed.

Attributes:

- **private Piece pieces[]**
  This attribute holds reference to Piece objects that are created beforehand.
- **private Ground grounds[]**
  This attribute holds reference to Ground objects that are created beforehand.
- **private int combination[][]**
  It holds a matrix that is the data representation of the board that is created by 4 4x4 subboards (Grounds). In this matrix, 0 means that a piece can be put there while 1 means the opposite.
- **private int numberOfMoves**
  A counter for player's move count.
- **private int timeElapsed**
  A counter for elapsed time.
- **private Record leaderboard[]**
  This attribute holds the rankings of the players for this level.

Constructor(s):

- **public Level(Grounds[] ground, Pieces[] piece)**

Pieces and Grounds that are created elsewhere
are sent here to be used.

Methods:

- **private void combineGrounds()**
  Grounds that are feeded into this class are combined to create the combination matrix that will be used to check if piece is placed correctly or board is valid.
- **public boolean isGameWon()**
  This method checks if the game is completed; namely, all if all pieces are placed correctly.
- **public boolean checkValidity()**
  This method checks if the combination of the grounds is valid and level has at least one solution. If both conditions are satisfied it returns true.
- **public void showHint()**
  This method reveals the location of one of the pieces.
- **public void seeLeaderboard()**
  This method opens up a leaderboard pop-up which shows the best performances played in this level.

ii. Piece



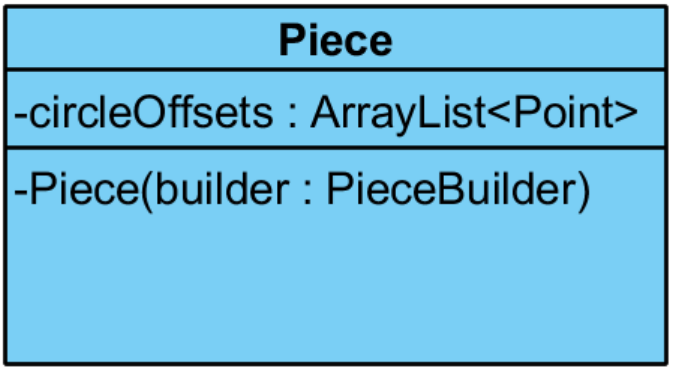


Figure 37: Piece Class

This class is responsible for holding the information of how marbles are arranged to constitute a Piece. Here marble means each spherical object which constitute the Piece when sticked together. It includes EventHandler<MouseEvent> interface in order to manipulate (rotate, move, flip, drag) the piece.

Attributes:

- **private ArrayList<Point> circleOffset**

This attribute holds how marbles are oriented as an dynamic array of 2D Points (e.g [(0,0), (1,0), (2,0)] is equal to the piece of **OOO**.

Constructors:

- **private Piece(PieceBuilder builder)**
  This is a private constructor because Piece object can only be constructed using PieceBuilder.

iii. PieceBuilder

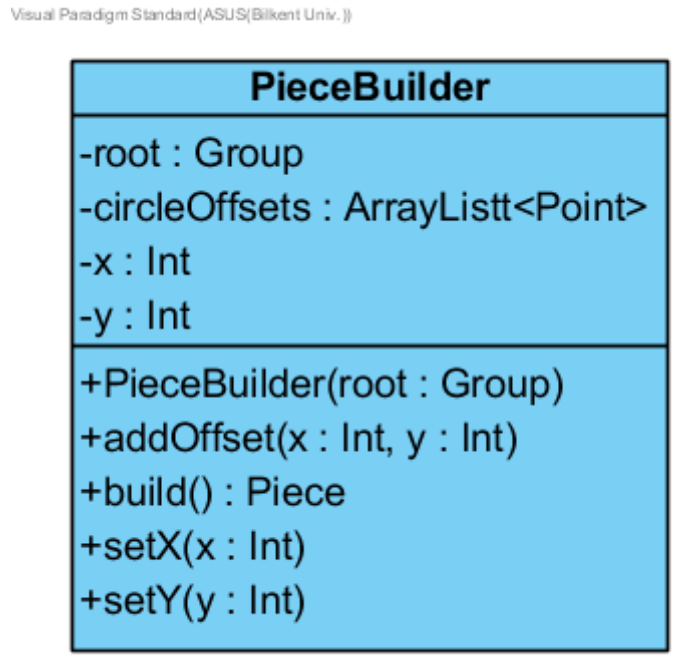


Figure 38: PieceBuilder Class

This class is a builder class of Piece class and initializes some of the attributes of it.

Attributes:

- **private Group root**
  This holds Group object to draw and remove on the screen.
- **private ArrayList<Point> circleOffsets**
  This is the same attribute as in Piece class.
- **private int x**
  This holds x coordinate of the ground.
- **private int y**
  This holds y coordinate of the ground.

Constructors:

- **public PieceBuilder(Group root)**
  This constructs builder object by taking a root object as a parameter

Methods:

- **public void addOffset(int x, int y)**
  This adds new offsets to the circleOffsets list.
- **public void setX(int x)**
  It sets X position if the ground.
- **public void setY(int y)**
  It sets Y position if the ground.
- **public Piece build()**
  It calls the private constructor of Piece class to create a new object by using the root, circleOffsets, x and y attributes.

iv. Ground

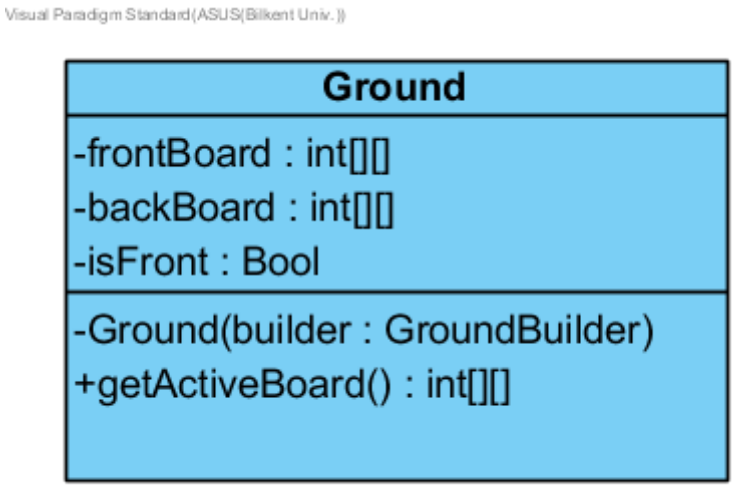


Figure 39: Ground Class

This class is a subclass of Drawable class holds information of 4x4 sub-board. It includes EventHandler<MouseEvent> interface in order to manipulate (rotate, move, flip, drag) the ground.

Attributes:

- **private int frontBoard[][]**
  This matrix holds front face of the 4x4 board.
- **private int backBoard[][]**
  This matrix holds back face of the 4x4 board.
- **private boolean isFront**
  This holds if front front or back face is active.

Constructors:

- **private Ground(GroundBuilder builder)**
  Ground object is constructed using GroundBuilder.

Methods:

- **public int[][] getActiveBoard()**
  It returns the active face of the board either front or back.

v. GroundBuilder

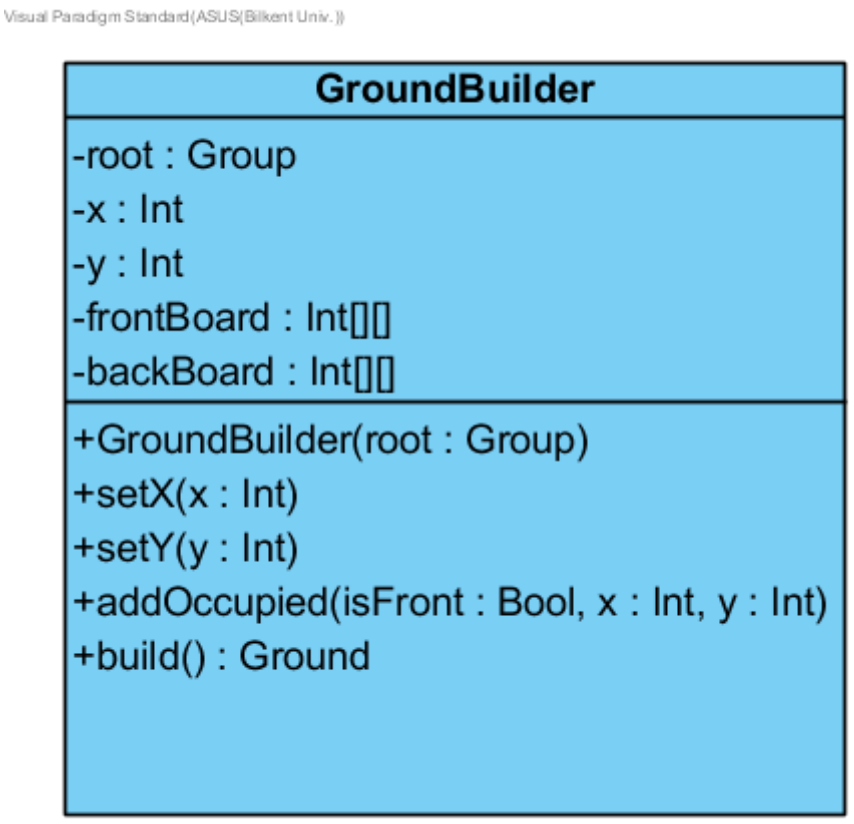


Figure 40: GroundBuilder Class

This class is a builder class of Ground class and initializes some of the attributes of it.

Attributes:

- **private Group root**
  This holds Group object to draw and remove on the screen.
- **private int x**
  This holds x coordinate of the ground.
- **private int y**
  This holds y coordinate of the ground.
- **private int frontBoard[][]**
  This holds front face of the ground as a matrix.
- **private int backBoard[][]**
  This holds back face of the ground as a matrix.
  Constructors:
- **public GroundBuilder(Group root)**
  This constructs builder object by taking a root object as a parameter.

Methods:

- **public void setX(int x)**
  It sets X position if the ground.
- **public void setY(int y)**
  It sets Y position if the ground.
- **public addOccupied(bool isFront, int x, int y)**
  It sets given x, y coordinates of either front or back face of the board as occupied.
- **public Ground build()**
  It calls the private constructor of Ground class to create a new object by using the attributes of this class.

vi. Drawable

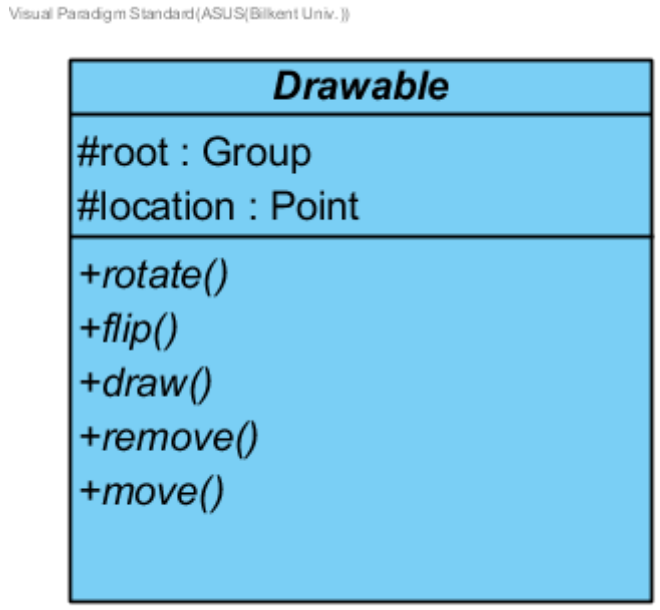


Figure 41: Drawable Class

This methods is an abstract method for the main components of the game that can be drawn, rotated, flipped, clicked, dragged, removed. This class will enforce its subclasses to EventHandler<MouseEvent> interface in order to rotate(), flip(), draw() and move() objects.
Attributes:

- **protected Group root**
  This attribute can be thought of as a brush and an eraser that is used for components to be drawn on and removed from the screen.
- **protected Point location**
  This attribute holds the position of the component on the main window.

Methods:

- **public abstract rotate()**
  This method rotates the components.
- **public abstract flip()**
  This method flips the components.
- **public abstract draw()**
  This method draw the components using the root object.
- **public abstract remove()**
  This method removes the components from the window using the root object.

vii. ComponentFactory




Figure 42: ComponentFactory Class

This method knows initial versions of the Pisces and Grounds and manufactures them in order for other classes to use them.
Methods:

- **public Piece[] createPieces()**
  This method creates Pieces and returns them.
- **public Ground[] createGrounds(GroundData gd)**
  This method creates Grounds based on the GroundData and returns them.

viii. LevelManager

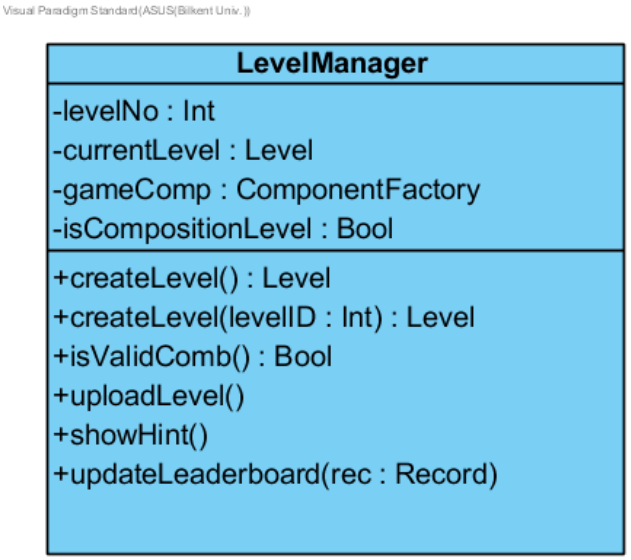


Figure 43: LevelManager Class

This method fetches the information about the level; namely, how grounds are combined from the database and creates a new level out of it.

Attributes:

- **private int levelNo**
  ID number of the level that will be fetched from the database.
- **private ComponentFactory gameComp**
  Instances of the Factory that creates grounds and pieces that will be used to create the level object.
- **private Level currentLevel**
  Holds current level.
- **private boolean isCompositionLevel**
  Since level it contains can either be played in the game or can be currently being composed in ComposeLevel.

Methods:

- **public void createLevel(levelID)**
  This method creates a normal playable level. This method contacts database fetches information about the formation of the grounds. It rotates, flips and sets the locations of the grounds and sends it to the constructor of level class along with pieces.
- **public void createLevel()**
  This method creates level for composition (ComposeLevel). It creates level with initial configurations of Grounds.
- **private void uploadLevel()**

This method checks the validity of the composed level. If it is valid, it uploads the composed level to the database.

- **public bool isValidComb()**
  This method delegates Level class to check validity.
- **public void showHint()**
  This method delegates Level class to show hint.
- **public void updateLeaderbaord(Record rec)**
  This method updates the leaderboard table by adding new entry/record.

ix. HintManager

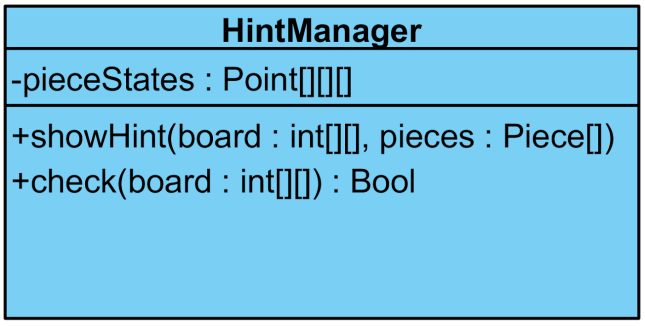


Figure 44: HintManager Class

This class is responsible for providing hints based on the current situation of the board and also for checking if the level that is composed on ComposeLevel has a solution.

Attributes:

- **private Point[][][][] pieceStates**
  This attribute holds all of the states of all piecec.

Methods:

- **public void showHint(int [][] board, Piece[] pieces)**
  This method finds a hint; namely, a suitable place for a piece which is not yet embedded into the board.
- **public boolean check(int [][] board)**
  This method checks if the level being composed in the ComposeLevel has a solution.

# 4. Low-level design

## a. Design Patterns

### i. Strategy Pattern

We have used this behavioral pattern to regulate user interface; namely, which page to show to the User. We have an abstract Page class with prepareDesign method which is extended by ComposeLevel, PlayGame, MainMenu, SelectLevel and Logic classes. These each sub-classes have its own implementation of prepareDesign method along with their own specific methods. Here Page class represents the Strategy class and by using Screen class which Strategy is to be used (i.e. which subclass to show) will be chosen. [1]

### ii. Factory Pattern

This pattern gets quite well along with builder pattern. ComponentFactory method creates Piece and Ground objects using their builders. It is important to have Factory Pattern in our project because the ones who will use the Piece and Ground do need not use what their initial properties are and how they should be constructed. ComponentFactory knows how to construct them and their initial configurations. The ones who will use the Piece and Ground objects just demand pieces and grounds from ComponentFactory and the factory produces and delivers them back. [2]

### iii. Singleton Pattern

Since Database and Pos Service are contacted through internet it is essential to create a single object for them in order not to create an overhead of establishing multiple connection to the Database and Pos Service. That is if multiple objects are created to connect with Database and Pos Service this would (1) put extra load on the server, (2) simultaneous connections may lead to race condition. Hence, we want to make sure that only one object is active for each class and enforced Singleton Pattern. [3]

### iv. Builder Pattern

We used BuilderPattern to enforce immutability for certain attributes of Piece and Ground classes. For Piece class we wanted to ensure that after the object is constructed adding new offset is impossible, it would severely damage the state of the game. Some would argue that we could add addOffset function to the Piece class and get rid of

PieceBuilder class. Before all, if we were to add such function to the Piece class it would have been public since we want our ComponentFactory class to use it. However, in that way we could not make sure if by mistake addOffset method is called after object is constructed which disrupts state of the game. [4] [5]

v. Observer Pattern

We used this pattern so that Piece class can express to PlayGame class the number of moves. PlayGame class implements the MoveObserver interface (for detaily see Class Interfaces section at the bottom). In that way, we did not give exactly the PlayGame object and unnecesarry priveleges to the Piece class but just the interface. Level manager playes an intermediary role here which is connecting MoveObserver interface to the Piece class. [6]

vi. Facade Pattern

We used Facade Pattern for Core Game subsystem. LevelManager class is the facade of Core Game subsystem. Because Core Game subsystem contains key classes such as Ground, Piece and Level, we wanted to prevent other subsystem to easily manipulate them. Only through LevelManager can classes from other subsystems manipulate these objects. [7]

## b. Final object design

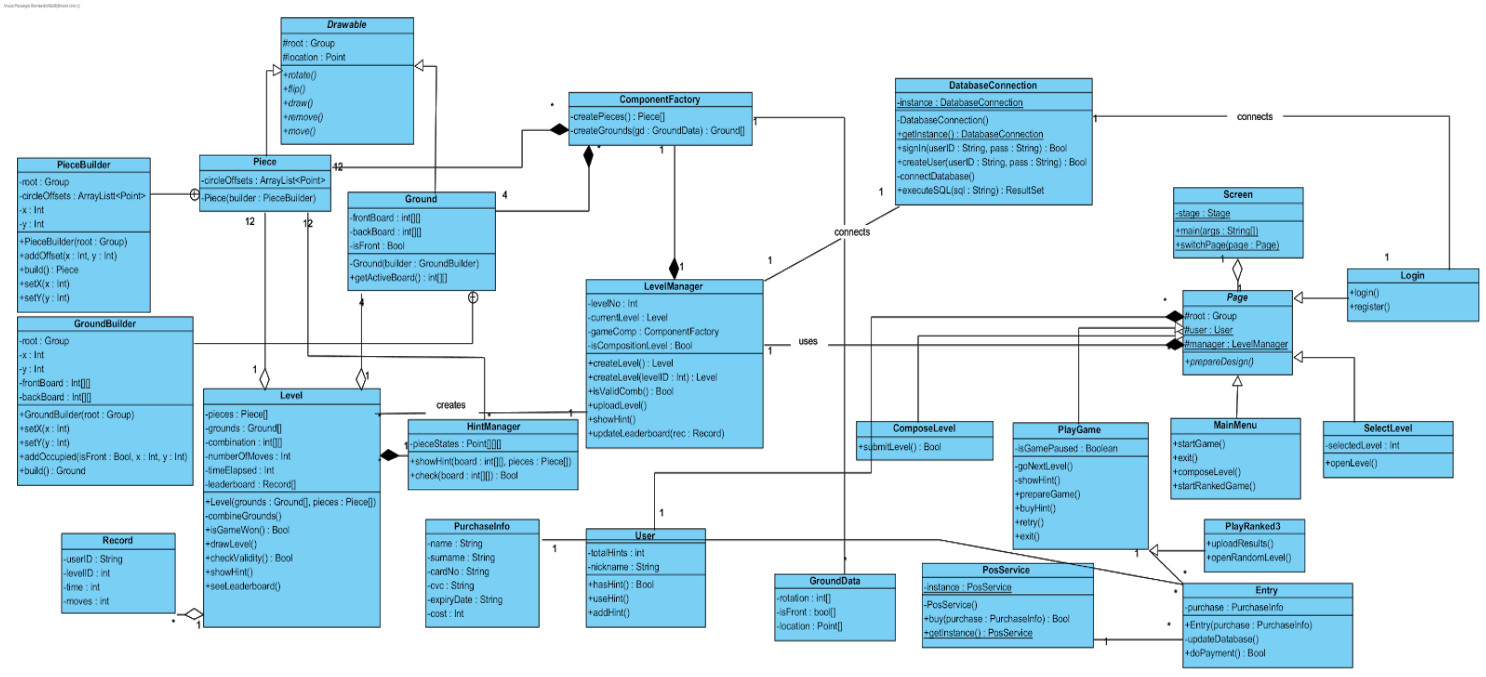


Figure 45: Final Object Design

## c. Packages

1. com.iyizipay.*

   We are using iyizico as our pos service. Their iyizipay package will be used to do all transactions. [8]

2. com.javafx.*

   We will use JavaFx framework to build our user interface. [9]

3. java.sql.*

   This package is used for Database related functionality such as connecting to database and executing query on database. [10]

## d. Class Interfaces

i. EventHandler<KeyEvent>

This interface will be used to listen keyboard inputs. Using keyboard user can rotate and flip the pieces. This interface has a single method cold *handle(KeyEvent event)* which is the callback when key is pressed.

ii. MoveObserver

This interface is used to apply Observer pattern. Number of moves are calculated in Piece class but we must show it in PlayGame and its subclass PlayRanked. PlayGame implements this interface. This interface has a method called *notifyMovesChanged(int move).* The classes that are involved are Piece, Level Manager and PlayGame. Here PlayGame class is observer and the Piece class is Subject or the class the PlayGame observes. They are connected together through LevelManager class. In other words, observer of the Piece class is set by LevelManager class as the PlayGame class.

iii. EventHandler<MouseEvent>

This interface will be used to get user inputs. Especially, when user moves, rotates, flips grounds and also when he/she clicks buttons the actions will be taken through this interface. This interface has a single method cold *handle(MoustEvent event)* which is the callback when mouse is pressed.

# 5. Glossary & References


[1] https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

[2] https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

[3] https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

[4] https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

[5] https://medium.com/@ajinkyabadve/builder-design-patterns-in-java-1ffb12648850

[6] https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

[7] https://www.tutorialspoint.com/design_pattern/facade_pattern.htm

[8] https://github.com/iyzico/iyzipay-java

[9] https://openjfx.io/javadoc/11/

[10]https://docs.oracle.com/javase/8/docs/api/index.html?java/sql/package-summary.html