Bilkent University

Department of Computer Engineering

**CS 319 Project**

*Project name: Piece12*

Design Report v1.0

**Fantastic 4**

Berk Mehmet Gürlek

İsmail Yavuzselim Taşçı

Nursena Kurubaş

Ali Kemal Özkan

November 8, 2018

**Contents**

**1.** Introduction ..……………………………………………………………………………………………………………….. 2

**1.1.** Purpose of the System …….……………………………………………………………………………… 2

**1.2.** Design Goals ……………..…….……………………………………………………………………………… 2

**i.** End User Criteria ..…………..………………………………………………………………………… 2

**ii.** Maintenance Criteria ……..………………………………………………………………………… 3

**iii.** Performance Criteria ..…………..………………………………………………………………… 4

**2.** Software Architecture …………………………………………………………………………………………………… 4

**2.1.** Subsystem Decomposition ……………………………………………………………………………… 4

**2.2.** Hardware / Software Mapping ..……………………………………………………………………… 7

**2.3.** Persistent Data Management …………………………………………………………………………. 7

**2.4.** Access Control Security …………………………………………………………………………………… 7

**2.5.** Boundary Conditions ………………………………………………………………………………………. 8

**3.** Subsystem Design …………………………………………………………………………………………………………. 8

**3.1.** Game Manager Subsystem ..…………………………………………………………………………… 8

**3.2.** Game States Subsystem ..……..………………………………………………………….……………. 12

**3.3.** Game Entities Subsystem ..……..………………………………………….…………….……………. 34

**3.4.** Detailed System Design ..……..….……………………………………………………….……………. 40

**1. Introduction**

At this section, we introduce our game’s design in detail. We also provide a detailed class diagram and we explain all subsystems.

**1.1. Purpose of the Game**

Piece12 is 2D puzzle and mind game. Main purpose of the game is entertaining people by making them use their minds. In order to get maximum satisfaction from the game, we design different modes and levels to push player’s limits. In addition, we added multiplayer feature to let players share the same enthusiasm with their friends and to let them taste sense of competition. The game is planned to be a compatible, user friendly mobile application game because of the fact that everyone can play easily anytime and anywhere. Game encourage to player use their minds effectively, solve puzzles, problems and make the right decisions. With these opportunities game has a lot of benefits in itself.

**1.2. Design Goals**

Design is a very important step when it is time to make a game. We decide where we should focus on and what functionalities our game should have. There are some criteria that we should focus on in the design stage. These criteria will help the make the game more functional. In our analysis report we mentioned many non-functional requirements that our game will/should have. In following sections these functionalities will be mentioned. Design of the game based on three criteria such as; end user, maintenance and performance.

**i. End User Criteria**

**User-Friendliness**

Piece12 is designed to people that who can use mobile phone. So our age range is very wide. To make it possible the game user interface will be easy that everyone can play easily. All options will help user to get better game experiences. Player will not need a training to play the game. Just reading the how to play part will be enough to play the game. Also, player can learn the game as they play many times. Passing the levels may be become hard close to the end, but playing the game will always be easy.

**ii. Maintenance Criteria**

**Extensibility**

In the game we let user to make some changes on their own. Most of them are design changes but, user will adopt the with these features. In shop, user can buy different themes to make the game special to him/herself. Also, in later, player can unlock the augmented reality option to get more enjoyment from the game. With the shop, the game will be extensible. New features can be added in later.

**Portability**

The game itself is designed to be portable because we wanted to all people can play the game everywhere and every time they want. “Everywhere” means that all android devices. Since, android is most used mobile operating system in the world, most of people can reach the game. In addition, game will support most of the android versions so, older phone owners could play the game. With these, we aim to appeal to most people.

**Modifiability**

Piece12 game is designed wisely that all subsystem can be modified without affecting other subsystems and functions. With that, if we want make some changes or add some features, process will be easy and smoothly.

**iii. Performance Criteria**

**Game Performance**

Performance is really important part of the game because, user will want to play the game without freezing. Game response time should be less than 100 ms to increase the efficiency. Also frame per second of the game will 60 in minimum. With less fps, user can see the difference and it will create bad ideas on player’s mind. Our game will not require high system requirements so it will work on low quality devices too.

**2. Software Architecture**

**2.1 Subsystem Decomposition**

Since we are implementing a game with multiple functionalities, such as shop, leaderboard… etc. we will build a system, that is composed of subsystems. These subsystems are the milestones of our applications. Moreover, since we are producing a game, which will be playable on Android devices, it will have connections with both users and Google Play Store itself.

As it is shown in the class diagram, our subsystems are composed of states of the application. Then we have 10 different states, which are shop, menu, leaderboard, credits, player number, mode, level, settings, pause and game. Shop, leaderboard, credits, player number, mode, level states are the sub-states of the menu state.

Game State Manager will supply transaction of the states. Menu will be the first subsystem that will be available when the game launches. User is provided 7 subsystems that are directly approachable from the first state. Then, there will be a transaction between subsystems if the user decides to play the game. After level and mode are selected, pieces and the board will be loaded, which means that the application is in the game state.

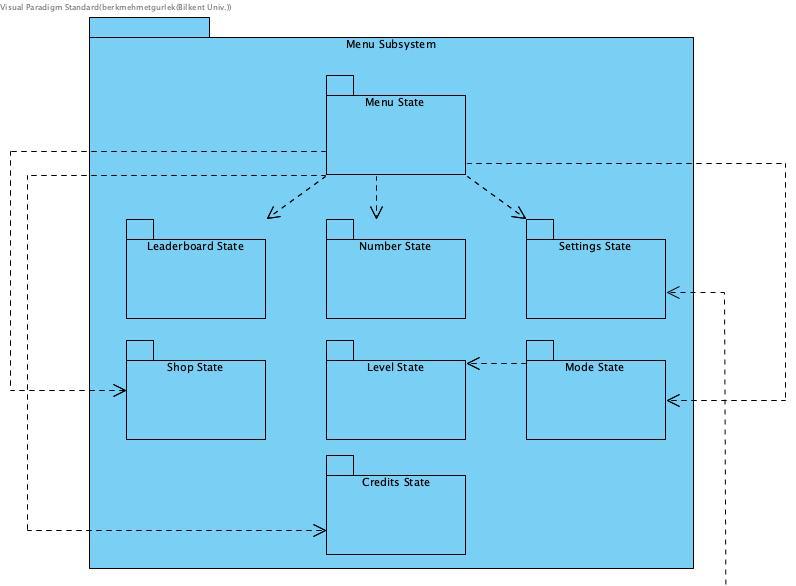


Figure-1 Menu Subsystem

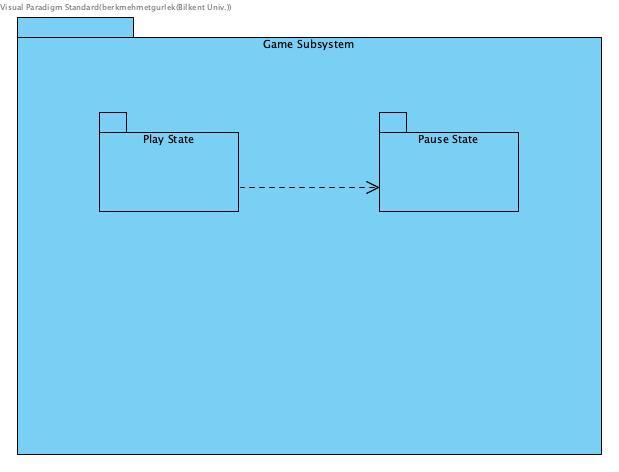


Figure-2 Game Subsystem

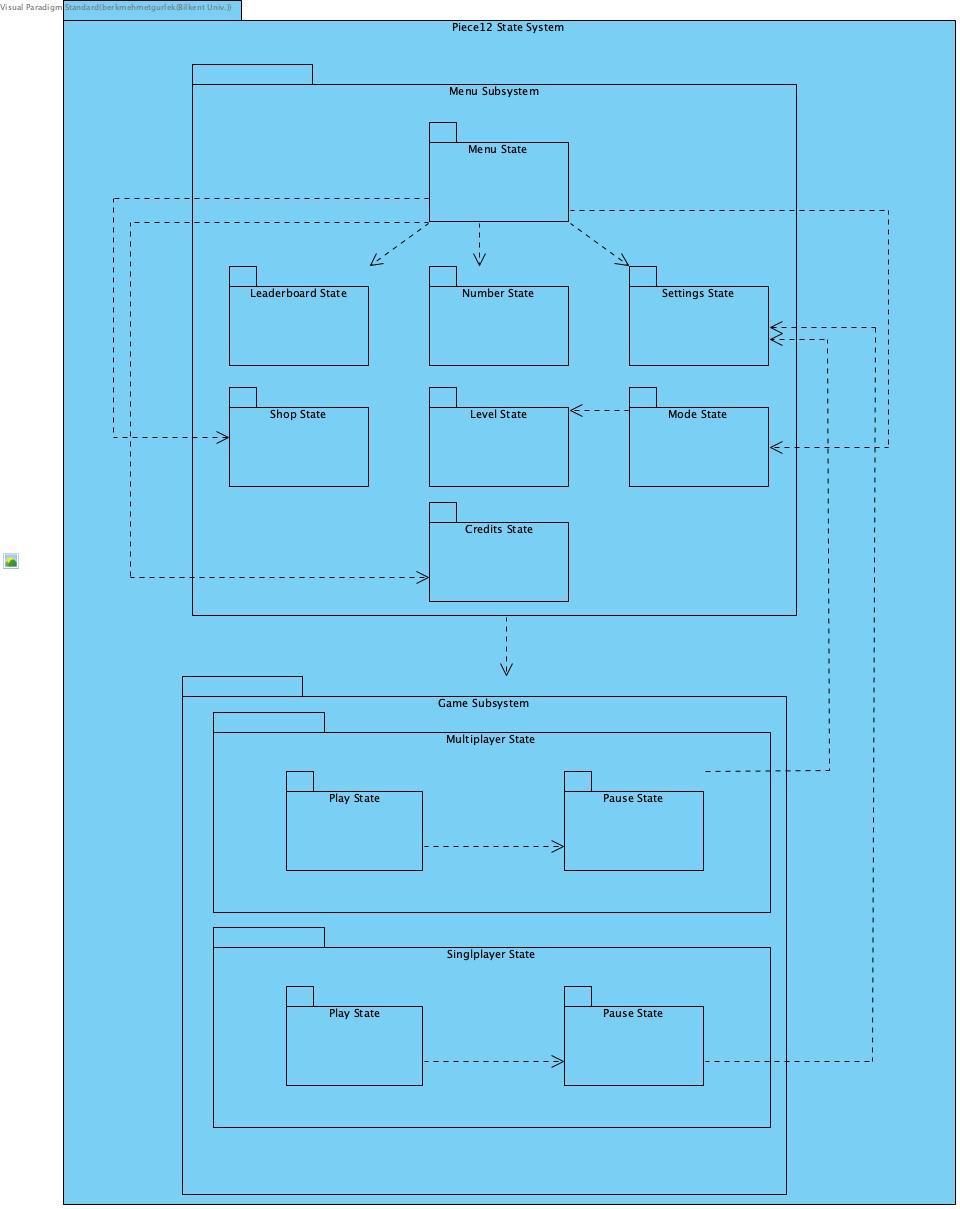


Figure – 3 Subsystem Decomposition

**2.2 Hardware/ Software Mapping**

As it is mentioned before, Piece12 is an android application, which will be installed via Google Play Store or manually from an APK file. Since it is an android application, an android device is required in order to use our application. We will implement our application as optimized and consistent as it is possible in order to produce an application that will not cause any problem among android devices that differ from each other. Moreover, since Piece12 has a connection with Google Play Store user needs to have an internet connection if he/she wants to play the game online or look up for the leaderboard. However offline playing is available in our application so that users able to play the game while they do not have an internet connection. Thus, our application has a very basic hardware/ software mapping that only require an android device and only constraint is to have an internet connection in order to reach some features of the game.

**2.3 Persistent Data Management**

Application will be managing the data in two ways. First, offline database that will track and save the progress of the player that played the game in single player mode. This feature enables users to play the same level more than one time to achieve a better score. On the other hand, online database that Google supports will hold the same data, if there exists an internet connection, for updating the leaderboard and providing the user daily quests. Moreover, online database and offline database will work synchronous when the user establishes an internet connection.

**2.4 Access Control and Security**

Piece12 will hold data that are only valuable and useful for the game. Thus, there will be no user that is authenticated more than another user. Each user will have the same rights, which is being a player. The shop system will use the in-game currency that will be only earned by playing the game. Therefore, there will be a total secure environment that no personal information is used by the game or asked from the game and, each user is authenticated as a player thus, there will be no approach provided a user to change the data of the leaderboard.

**2.5 Boundary Conditions**

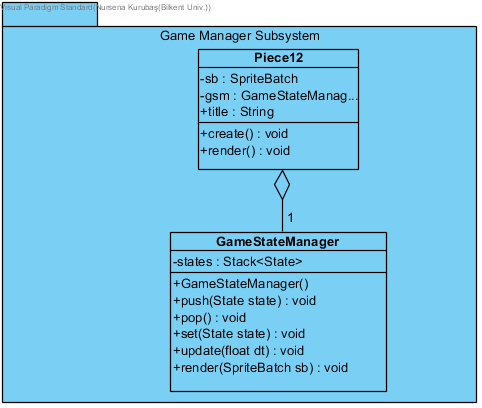
In main menu, user will transact between various states as it is shown in the diagram so that, shop, settings, how to play options are provided without any restriction. However, when the user selects the leaderboard option, if the user has no internet connection, he/she will be warned of this situation and led back to the main menu. The same procedure will be done when user selects multiplayer option. Moreover, if the user’s internet connection is lost while playing multiplayer user will be warned again and sent back to the main menu. Furthermore, the levels in single player that are not playable if the previous level is not passed. If the user selects to return to the menu while playing in single player mode or, quits the game the last level that the user played will be reset and user will not be able to play the next level. Last but not least game, there is a small chance that game will crash, in that case same procedure will be applied as the user quits from the game.

**3. Subsystem Design**

Our design is composed of three subsystems: Game manager subsystem, game entities subsystem and game states subsystem. In this section we will mention the detailed design of our project and we will explain each subsystem and classes in detail.

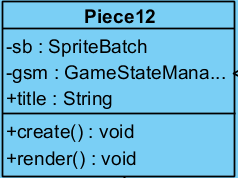
**3.1. Game Manager Subsystem**

Function of this subsystem is creating and managing the game. The tasks like page transitions, getting user information at the beginning of the game happens at this subsystem.



**Piece12 Class**

This class creates the game’s initial state and GameStateManager. Also this class runs the game in a loop that renders and updates the game. This class also implements runnable interface, since the game loop runs as a thread.



**Attributes:**

**private SpriteBatch sb:** SpriteBatch is a type that comes from libGDX library that we will use for gui elements. Basically this is a container for all gui elements in a state. We will use this attribute when we want to draw a new screen or update it’s contents.

**private GameStateManager gsm:** This is an instance of our GameStateManager. And this instance will be created here since our initial state will be pushed our gsm. And our initial state will also have this gsm as parameter in it’s constructor.

**public String title:** This is the title of our game. Since we will implement this game in android environment, this title will be shown as game name.

**Constructors:**

This class has no constructor since it is the main class of our game.

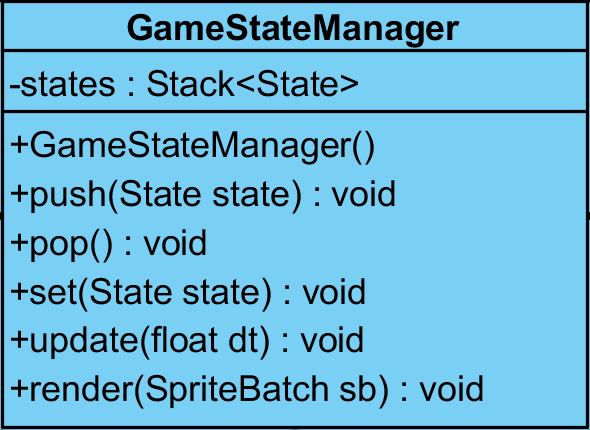
**Methods:**

**public void create():** This method will only be called when a game changes state and a new state should be rendered. This method will clear the whole screen and draws the initial state.

**public void render():** This method is the loop method for the game. So that this method will be always in a loop and will work all the time that game is open. This method will handle the calling update methods of the states. This will be possible by using GameStateManager so that state that is on the top of the state stack will be the active state and update method of that state will be called.

**Game Manager System Class**

This class creates a stack of states to manage the state transitions which are the screen states that users see and use. For example, when users is at main menu at the beginning, if they click credits, this class will push Credits State to the stack and when user goes back, it will pop that state. Also this class ensures that only the top State is working and keeps being updated.



**Attributes:**

**Stack<State> states:** This property is stack of States. It’s job is handling State transition. Our main purpose of using stack is when a new State comes in that State will be pushed in to the stack and it will be the stack that is on top of the stack so our game will know which State is active and where to go back if user press back button.

**Constructors:**

**public GameStateManager():** This is the constructor for initializing this class. In this constructor we will only create an empty stack of States for using in other methods.

**Methods:**

**public void push( State state ):** This method’s purpose is pushing new states into the stack. This method will be called when a new state comes in. Parameter *State state* is the State that we want to push.

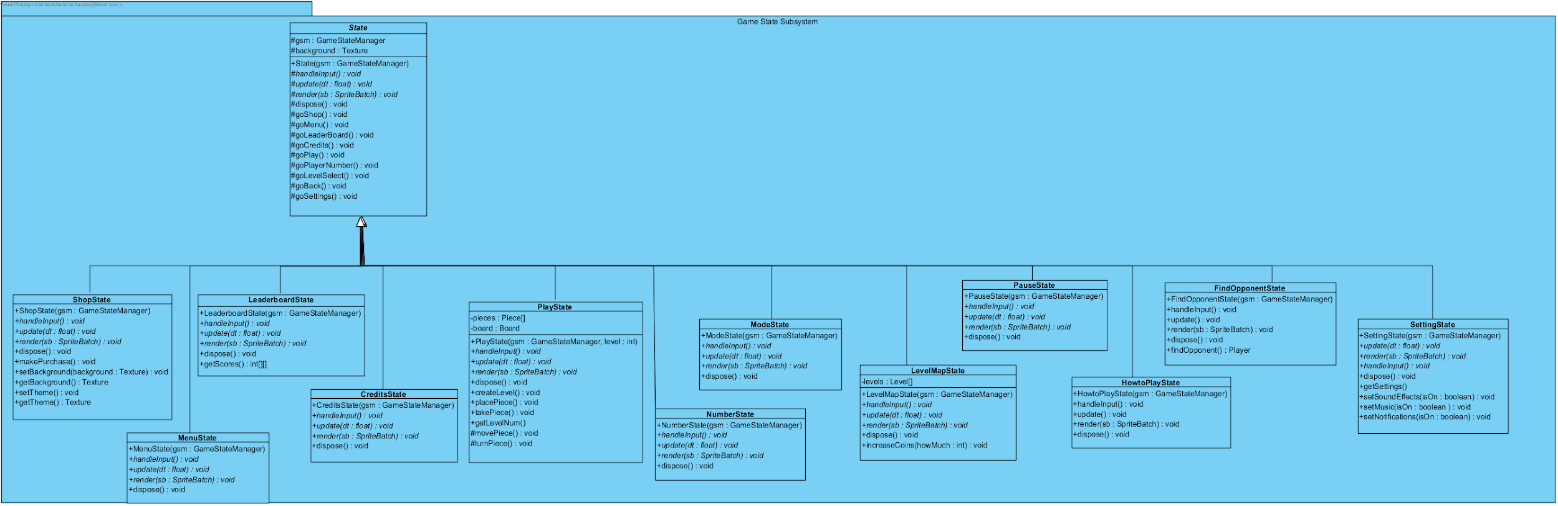
**public void pop():** This method is responsible for popping top State from Stack so that when we do not need that State anymore we will pop it and that State is no more our current state.

**public void set( State state ):** This method will first pop then push a new State so that second State will be the same but top State will be changed so our current State will change. This is used for when we need to change the current State but we want a new State rather than second State of the stack. Parameter State state is the State that we want to be our current State.

**public void update( float dt ):** This method for calling update methods of the current State that is on top of the State stack. This method will be called by render method of Piece12 class. So that this method will be called in loop just like render method of Piece12. float dt is delta time which is time that is really small and this will come from libGDX library that is the speed of screen refresh rate.

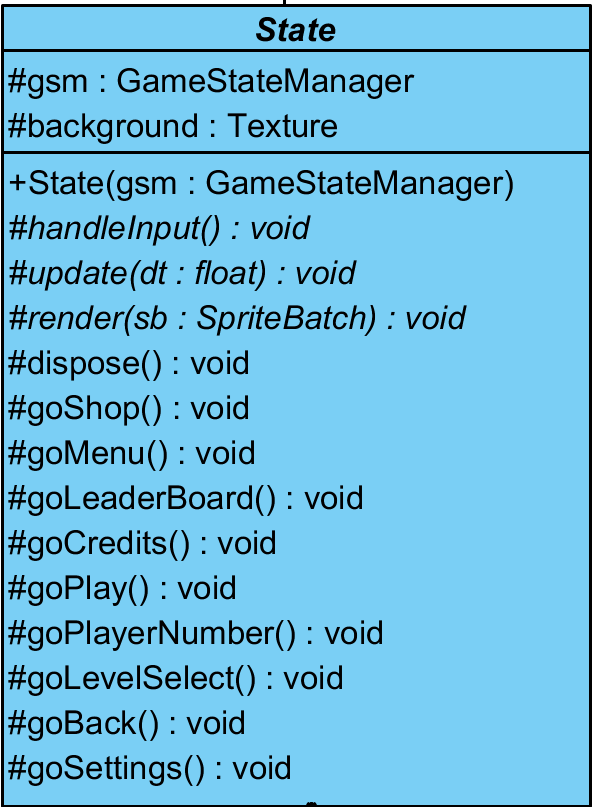
**public void render( SpriteBatch sb ):** This method will be used for rendering states on the screen and this method will also be called within render of Piece12 class. In this method we will call render methods of the States so States can be printed on the screen and when something changes screen will also change. SpriteBatch sb is the spritebatch that we initialize on the Piece12. Because they are heavy files we do not want to use more than one spritebatch but we can use same one everywhere.

**3.2. Game States Subsystems**

Game States are simply the different screens that user will see and use. Each State has link to one or more other States so there are many paths that user can surf through the application. Also, the game playing state remains in this subsystems as *PlayState*. This subsystem controls the user interface of the game as well as managing states.

**State Abstract Class**

This class is a representative class for each special States of the system. All states are subclasses of this super class. A Stack of these states are kept in Game State Manager to provide the transitions between States. All states’ user interfaces will be implemented on Android Studio using XML layout.



**Attributes:**

**protected GameStateManager gsm:** This attribute is the instance of the our gsm that we create on Piece12. We will have only one gsm so we can manage States properly. This attribute will initialized in other States in constructor by taking GameStateManager as parameter.

**protected Texture background:** This atttribute is the background of our game. Since all States will have the same background this abstract class has it as attribute.

**Constructors:**

**public State( GameStateManager gsm ):** This constructor will be used in other States as calling with super. This constructor will initialize gsm of this class as gsm that we take as parameter so we will have only one gsm and we will pass it through States.

**Methods:**

**protected abstract void handleInput():** This method is an abstract method for using in other states. However, this method’s purpose is same in every state but will change in detail. This method will handle user input as it states in the name of the method. This method will understand where the user just touched or is touching and will call methods accordingly to this. For example when user click it will get x, y coordinates of it and by looking that coordinations method will understand which button user clicked and will call a proper go method for changing states.

**protected abstract void update( float dt ):** This method is an abstract method for using in other States. But the purpose of this method will remain same but will differ in implementation. This method is the method that will call handleInput method in a loop and render method so that when user give some input or screen should be changed we will catch those and process it. Parameter float dt is the delta time.

**protected abstract void render( SpriteBatch sb ):** This method is an abstract method for using in other States. But the purpose of this method will remain same but will differ in implementation. This method will be called in update method and job of this method printing gui elements on the screen and updating them if anything changes. Parameter SpriteBatch is the SpriteBatch that we initialize on Piece12.

**protected abstract void dispose():** This method is an abstract method for using in other States. But idea behind this method is same for every State but implementation will differ. Purpose of this method will be blocking memory leaks by deleting gui elements from old states. Because of every state has different gui elements implementation of this class will change.

**protected void goShop():** This method’s purpose is pushing ShopState in to the stack. So this method is one of the go methods that will be called in handleInput method. This method is an abstract method since this method is not differ in implementation for every State and it is same for everyone of them. So if a state needs this method it can call it.

**protected void goMenu():** This method’s purpose is pushing MenuState in to the stack. So this method is one of the go methods that will be called in handleInput method. This method is an abstract method since this method is not differ in implementation for every State and it is same for everyone of them. So if a state needs this method it can call it.

**protected void goLeaderboard():** This method’s purpose is pushing LeaderboardState in to the stack. So this method is one of the go methods that will be called in handleInput method. This method is an abstract method since this method is not differ in implementation for every State and it is same for everyone of them. So if a state needs this method it can call it.

**protected void goCredits():** This method’s purpose is pushing CreditsState in to the stack. So this method is one of the go methods that will be called in handleInput method. This method is an abstract method since this method is not differ in implementation for every State and it is same for everyone of them. So if a state needs this method it can call it.

**protected void goPlay():** This method’s purpose is pushing PlayState in to the stack. So this method is one of the go methods that will be called in handleInput method. This method is an abstract method since this method is not differ in implementation for every State and it is same for everyone of them. So if a state needs this method it can call it.

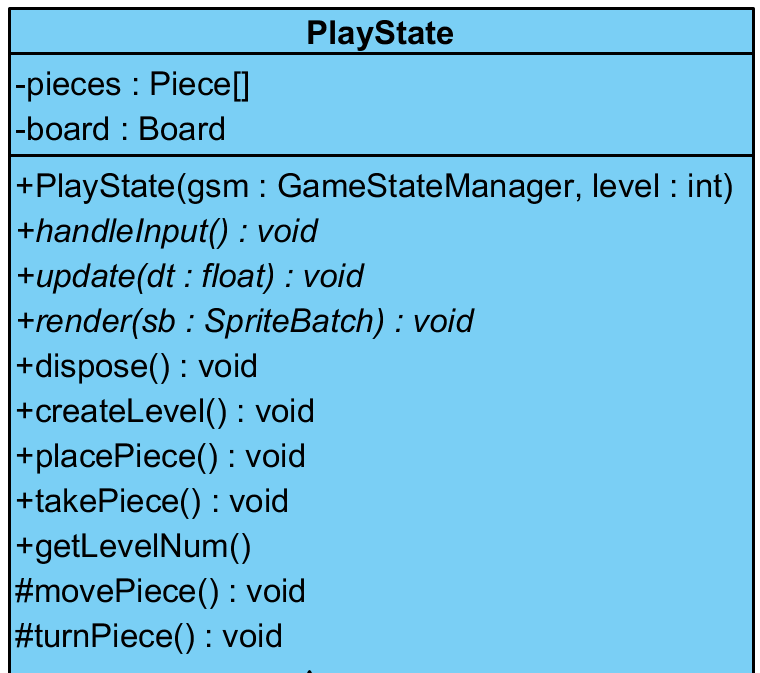
**protected void goPlayerNumber():** This method’s purpose is pushing NumberState in to the stack. So this method is one of the go methods that will be called in handleInput method. This method is an abstract method since this method is not differ in implementation for every State and it is same for everyone of them. So if a state needs this method it can call it.

**protected void goLevelSelect():** This method’s purpose is pushing LevelMapState in to the stack. So this method is one of the go methods that will be called in handleInput method. This method is an abstract method since this method is not differ in implementation for every State and it is same for everyone of them. So if a state needs this method it can call it.

**protected void goBack():** This method’s purpose is popping a State form the stack and by doing that oor new current State will be second State of the stack which is the previous State that user saw. So by doing that user can go back. This method is one of the go methods that will be called in handleInput method. This method is an abstract method since this method is not differ in implementation for every State and it is same for everyone of them. So if a state needs this method it can call it.

**PlayState Class**

This class creates the playable game and user will play the game on this state and all main logical calculations of the system happens at this class. So, this class is major for game playing.



**Attributes:**

**Piece[] piece:** This attribute is the piece array. It will hold all pieces of a level. So that we can make calculations on them.

**Board board:** This attribute is the instance of the game board of our game. We initialized it so that we can make calculations by using it’s methods.

**Constructors:**

**public PlayState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12. Then it will call getLevelNum and createLevel methods.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**public void createLevel():** This method’s purpose is creating level by filling board with initial pieces and in memory mode showing user the solution manual card so that player can look it and try to memorize. This method first needs to call getLevelNum method to find out which level it will create then it needs to connect database to fetch level data.

**public void placePiece():** This method will handle the placing pieces. So that it will place pieces when user drag them and drop. It may change piece coordinates little bit to find holes and place pieces. When it tries to place it will look up to board’s info if all places that user is trying to place piece is empty or not. If one or more holes is not empty it will not place it and will vibrate phone little and take the piece to the initial position. And this method will be called in handleInput method.

**public void takePiece():** This method will find out that a piece is selected and taken out it’s initial place. Then it will invoke the movePiece method so that piece can be moved while player drags it. This method will be called in handleInput method when handleInput method finds out that player touching a piece and dragging.

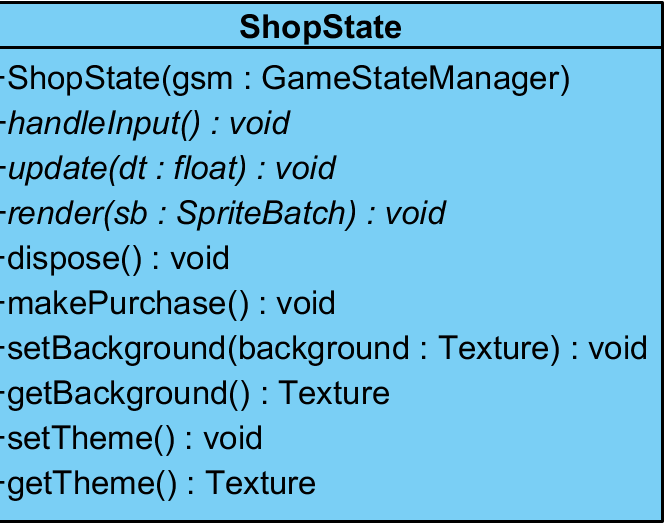
**public void getLevelNum():** Purpose of this method is informing createLevel method to which level will created. Since we are implementing this method in Android environment, we will be using it’s methods to get information from old state. So this method will catch information sended from LevelMapState.

**protected void movePiece():** This method will be responsible from moving pieces. So this method will be invoked when takePiece method finds out user takes a piece and tries to drag. This method will change the coordinates of asset of the selected piece.

**protected void turnPiece(): T**his method will change the mode attribute of a piece and pieceInfo. Since we represent direction of piece as integer from 1 to 4 when user tries to rotate the piece this method will be invoked and then according to input this method change attributes of selected piece. Also this method will rotate asset of the selected piece.

**ShopState Class**

This class represents the shop feature of our game. Player will make purchase and use the purchased products from here and system will set themes & background and manage the coin amount of Player from here.



**Attributes:**

This class has no attributes since we will use database. We will hold datas on database.

**Constructors:**

**public ShopState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12. Then it will call getTheme and getBackground methods to find out selected background and theme and show it to the user.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**public void makePurchase():** This method will be responsible from making purchases from our in-game shop. Player will be able to buy background and themes with coins. This method will change attribute of item that user bought in database so that it will available for user. Also this method will update the coins of the user. This method will be called in handleInput method when it understands that player click the buy button.

**public void getTheme():** This method will fetch the current theme from database.

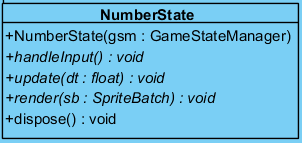
**public void setTheme():** This method will update the database when user change the theme. handleInput method will invoke this method when needed.

**public void getBackground():** This method will fetch the current background from database.

**public void setBackground():** This method will update the database when user change the background. handleInput method will invoke this method when needed.

**NumberState Class**

This class represents the state that shows user two choices to play: Singleplayer or Multiplayer. User’s choice is taken as an input and kept, then this state connects to ModeState.



**Attributes:**

This class has no attributes since this class is only a transition class that user will select singleplayer or multiplayer and that selection will be sended.

**Constructors:**

**public NumberState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

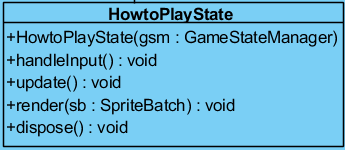
**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**HowtoPlayState Class**

This State will show user a screen with images and texts, explaining how to play the game. Most of the States have access to this State through a button. This State only will have a *back* button that calls *goBack()* from *State abstract class* to *pop* the State from the *Stack* and show user the previous State. This state is meant to help player through the gameplay.



**Attributes:**

This class has no attributes since this class is only for showing some simple instructions to the user about how can he or she play the game.

**Constructors:**

**public HowtoPlayState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

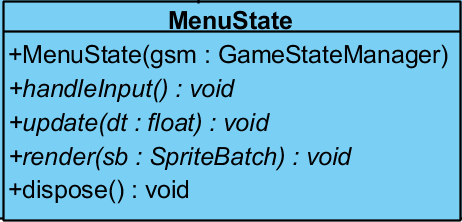
**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**MenuState Class**

This state represents the main menu of the game. Main menu consist of several buttons that connects it to the other states. Since those buttons will be arranged by Android Studio, we didn’t feel any need to have them in our class diagram.

****

**Attributes:**

This class has no attributes since this class is only for showing menu screen and handling inputs from the user.

**Constructors:**

**public MenuState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

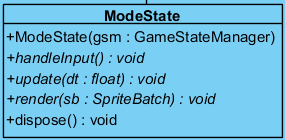
**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**ModeState Class**

This class represents the mode choosing state of the game. User will have several options and will choose one of them to move on.

****

**Attributes:**

This class has no attributes since this class is only for showing game mode selection screen and handling inputs from the user. This selection will send to the other States to process.

**Constructors:**

**public ModeState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

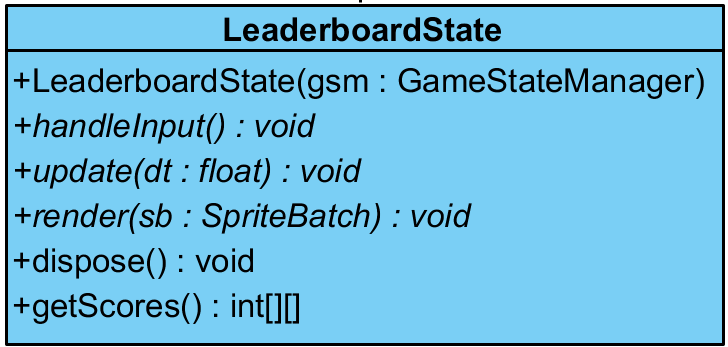
**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**LeaderboardState Class**

This class represents user the leaderboard by fetching the datas from our database. User can reach this page from *MainMenuState*. There will also be a main menu button at top left for user to go back to main menu.

****

**Attributes:**

This class has no attributes since this class will show leaderboard information of time race mode and data will be fetched from database.

**Constructors:**

**public LeaderboardState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

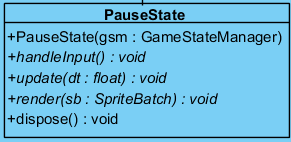
**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State. In this state getScores method will be invoked and by looking it’s return value leaderboard table will be drawn.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**public int[][] getScores():** This method is responsible from fetching data from leaderboard database and it will return 2-D integer array with playerID and highScore.

**PauseState Class**

This class represents the state that occurs when player pauses the game while playing it. So only the PlayState has connection to this state. This state pauses the game and reveals user several buttons to make a choice.



**Attributes:**

This class has no attributes since this class is responsible from showing pause screen to the user when user press pause button while playing. Only thing that we need to do in this class handling user inputs and taking actions according to it.

**Constructors:**

**public PauseState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

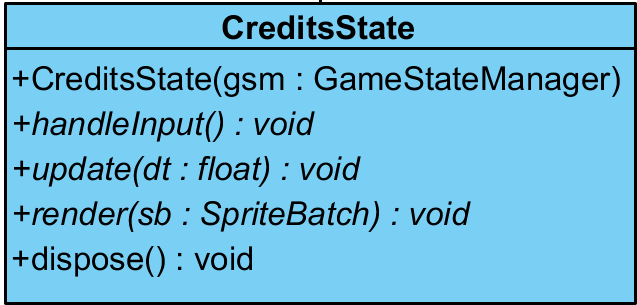
**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State. In this state getScores method will be invoked and by looking it’s return value leaderboard table will be drawn.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**CreditsState Class**

InCredits State, a screen full of text and main menu button will be revealed. Texts will be about us as creators of the game and our references. This State will be reachable from MainMenuState and it will also has a link to MainMenuState.

****

**Attributes:**

This class has no attributes since this class will only show the credits.

**Constructors:**

**public LeaderboardState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

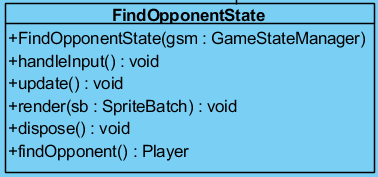
**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State. In this state getScores method will be invoked and by looking it’s return value leaderboard table will be drawn.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**FindOpponentState Class**

This state is for users who choose to play game as multiplayer. This state matches two random players who wants to play in same mode at same time interval.



**Attributes:**

This class has no attributes since this class is responsible finding opponent according to previous choices. Since it will use database there is no need to any attributes.

**Constructors:**

**public FindOpponentState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

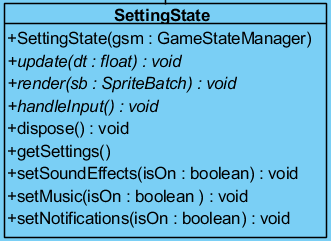
**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State. In this state getScores method will be invoked and by looking it’s return value leaderboard table will be drawn.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**public void findOpponent():** This method will send the user choices to the database and when it matches any other user it will start PlayState for multiplayer game.

**SettingsState Class**

This state arranges the sound and notification settings according to the decisions of player. Player can reach here from PauseState or MainMenuState and then go back to where s/he came by using a *back* button.



**Attributes:**

This class has no attributes since this class is responsible for changing game settings. However, we will keep information of our players and settings in database so we will not add any attributes to this class.

**Constructors:**

**public SettingState( GameStateManager gsm ):** This is the constructor of this class and this constructor will call super(gsm) so that gsm of our States will be equal to gsm of Piece12.

**Methods:**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State. In this state getScores method will be invoked and by looking it’s return value leaderboard table will be drawn.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**public void getSettings():** This method’s purpose is fetching user settings from database and apply those settings to the game.

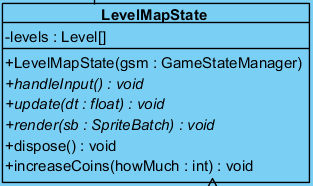
**public void setSoundEffects( boolean isOn ):** This method will change sound effect settings on database first then it will apply to the game. boolean isOn parameter represents the new setting.

**public void setMusic( boolean isOn ):** This method will change music settings on database first then it will apply to the game. boolean isOn parameter represents the new setting.

**public void setNotifications( boolean isOn ):** This method will change notification settings on database first then it will apply to the game. boolean isOn parameter represents the new setting.

**LevelMapState Class**

This class has a certain amount of levels in it and shows user a map of them. User can reach the PlayState by choosing a level at this State. Also, after finishing a level PlayState goes back to this state and player earns coins of that level at this State.



**Attributes**

**Level[] levels:** Level map draws a map of levels by using this array of levels.

**Constructor**

**public LevelMapState(GameStateManager gsm):** This constructor creates a LevelMapState and it will call super(gsm) so that gsm of this State will be equal to gsm of Piece12.

**Methods**

**public void handleInput():** This method will process user input and take actions according to that as handleInput method in State class. It will differ since user interface is different in each State so that processing user input and taking actions will be differently implemented.

**public void update( float dt ):** This method will be called in the update method of the GameStateManager class. So this method will be in a loop and purpose of this method is the catching user input by calling handleInput whenever an input occurs or rendering screen by calling render method whenever it needs.

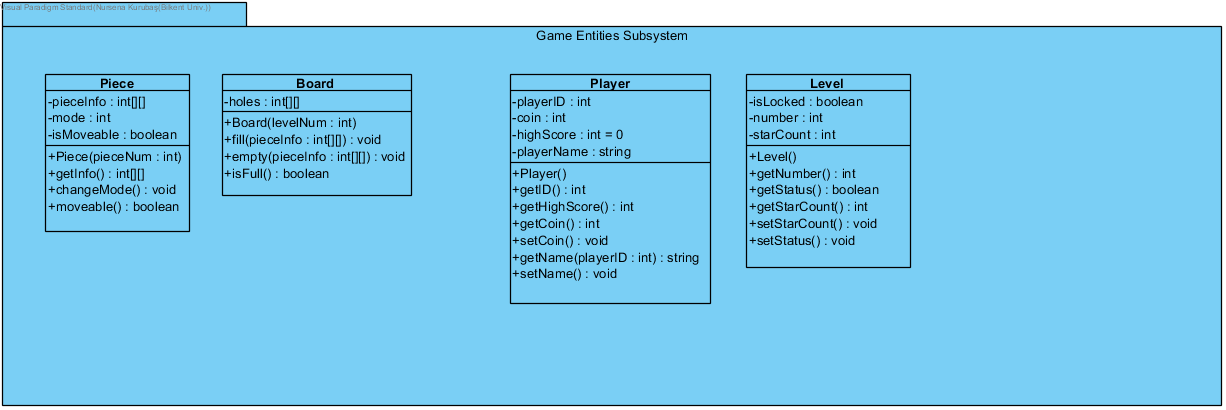
**public void render( SpriteBatch sb ):** This method will render gui elements on the screen and will be called by update method so that it can render every time when change occurs. Since every State has different gui elements this method will differ in implementation on every State. In this state getScores method will be invoked and by looking it’s return value leaderboard table will be drawn.

**public void dispose():** This method will delete unnecessary gui elements when a class is no more current class. So by doing that we will prevent memory leaks and our game will work faster.

**public void increaseCoins(int howMuch):** This method increase player’s coins if player wins a game, according to the input parameter howMuch, which depends on the finished level.

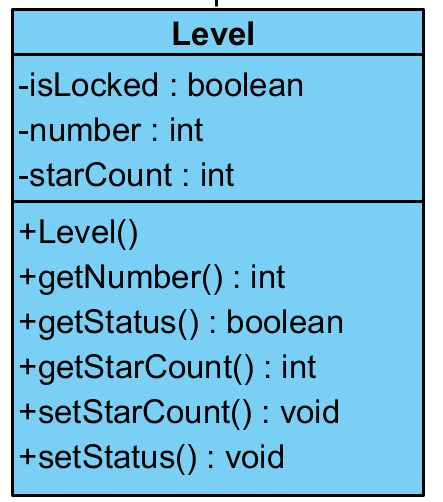
**3.3. Game Entities Subsystem**

This subsystem consists of our game entities, which are non-state classes that will be used as if they are solid objects in the game and has no direct relation with game management.



**Level Class**

Each game mode consists of many levels in it, which are shown at LevelMapState. Levels will be arranged from easy to hard but this Level Class is mainly for Level Map, at the moment they don’t have a game in them as an attribute or a method. We represent them to user like each level has a game in it, but in fact, they will only be buttons that connect LevelMapState to PlayState.

****

**Attributes**

**boolean isLocked:** Since the player cannot play a harder level without achieving the easier ones all levels comes after the current one will be locked. If isLocked is true the level will be shown as grey and with a lock on it. The button of a locked level cannot be pressed. Except the current level of player, all the previous levels will be unlocked that means isLocked will be false and they will shown colorful with stars on them. Current level will also be colorful but no star will be shown since user has not earned any yet.

**int number:** Each level has a number that shown in level map starting from 1 to the end. We also use this attribute to determine the beginning board of the game at PlayState.

**int starCount:** This attributes represents the star count of a level which can be from 0 to 3. Locked levels and the current level that user still playing has no stars yet so this attribute has 0 as the value. Other levels that player successfully finished has 1 to 3 stars according to the success rate of the player.

**Constructors**

**public Level(int levelNum):** This constructor takes information from database to create the level by using the number attribute of the level.

**Methods**

**public int getNumber():** This method returns the *number* attribute of the current Level object.

**public int getStarCount():** This method returns the value of *starCount* attribute of the current Level object

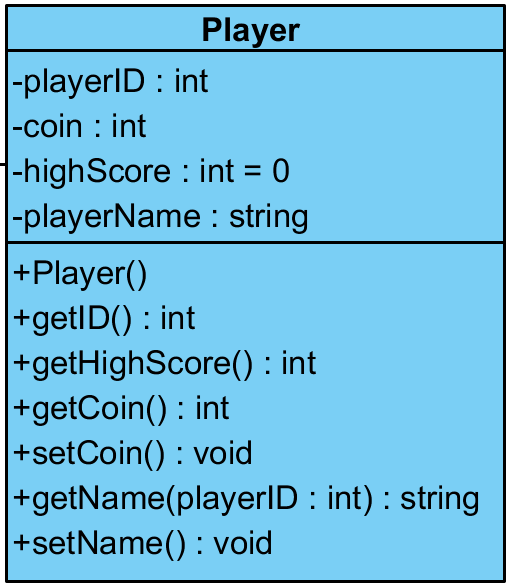
**public boolean getStatus():** This method returns the value of *isLocked* attribute of the current Level object.

**public void setStarCount():** This method changes the *starCount* attribute according to the information it takes from database.

**public void setStatus():** This method changes the *isLocked* attribute according to the information it takes from database.

**Player Class**

This class represents the initial user of the game. We save the attributes of players in a database. At each application launch, we get the informations of initial user from that database table and arrange the game accordingly.



**Attributes**

**int playerID:** Each player has a private ID given by the system to track the player’s initial game in database.

**int coin:** This attribute keeps the amount of coins that player earned through the game playing.

**int highScore:** This attribute keeps the high score value of the player

**String playerName:** Each player has a nickname that they choose at the beginning that is kept in this attribute. In multiplayer mode, players will be able to see the opponent player’s name.

**Constructor:**

**public Player():** This constructor creates an instance of a player with attributes token from database.

**Methods:**

**public int getID():** This method returns the initial Player’s ID.

**public int getHighScore():** This method returns the initial Player’s high score.

**public int getCoin():** This method returns the initial Player’s coin amount.

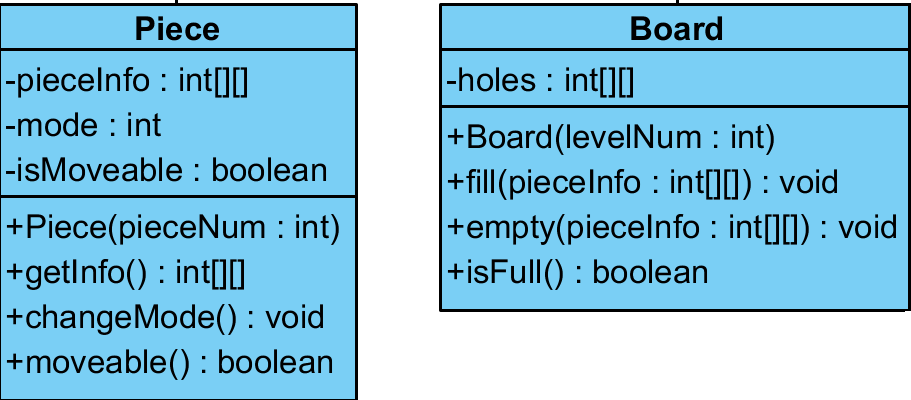
**public String getName(int playerID):** This method returns the initial Player’s name from database by using the player’s ID.

**public void setCoin():** This method sets the coin amount of the initial player.

**public void setName():** This method sets a playerName for the Player object. User will not be able to use this method in game, this method is for the constructor of Player class.

**Piece Class**

This class is a representation for each 12 pieces of the game. Users will be able to drag, turn and place the pieces on the board. Each piece in the game has special shapes and colours and we consider to use LibGDX framework for them. PlayState will have 12 Piece instances in it. This is one of the major objects of our game.



**Attributes**

**int[][] pieceInfo:** This attributes keeps an info about the shape of the piece. For example in a 2x2 array 1’s mean there is full 0’s mean empty and unity of full places shapes the piece.

**int mode:** This attribute keeps the information about the rotation of the piece. It can take values from 0 to 3, they mean normal state, turned 90 degrees right, turned 180 degrees, turned 90 degrees left, respectively.

**boolean isMoveable:** This attribute determines if the piece is moveable. At start of some levels, some pieces are placed on the board as default ones and since they determine the level they are not moveable. We only keep pieces that player can change the places as moveable.

**Constructor**

**public Piece( int pieceNum):** This constructor creates a Piece object by taking some information from database using the pieceNum parameter.

**Methods**

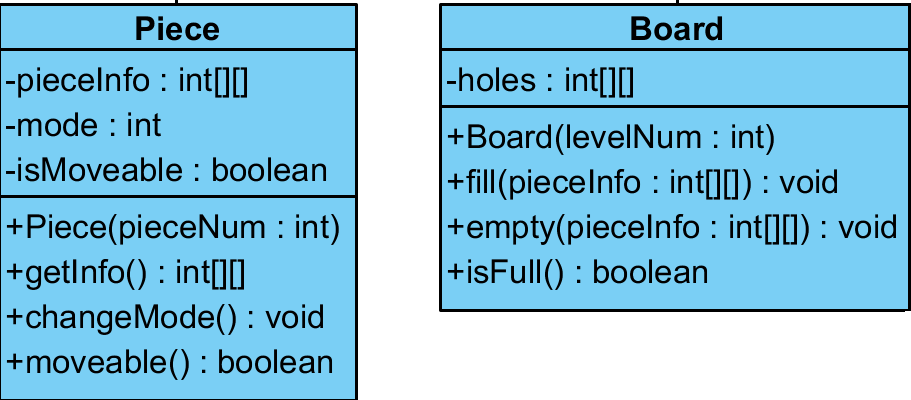
**public int[][] getInfo():** This method returns the *pieceInfo* attribute of the Piece object.

**public void changeMode():** This method is used to turn the piece 90 degrees right, which means it increases it increase the attribute *mode* by one and if *mode* is 3, then it makes it 0 again.

**public boolean moveable():** This method returns the *isMoveable* attribute of the Piece object.

**Board Class**

This class represents the board of our game. Filling the board with the pieces is the aim of the game. PlayState class has one special Board instance in it. This is the other major object of our game.



**Attributes**

**int[][]holes:** This attribute keeps the track of the emptiness of the holes on the board and it considers the board as a 2D array. 1’s represent full holes and 0’s represent empty holes on the board.

**Constructor**

**public Board(int levelNum):** This constructor creates an instance of Board and according to the levelNum parameter it determines the initial state of the holes on the board with the information it takes from database.

**Methods**

**public void fill(int[][] pieceInfo):** This method is used to place a piece on the board. pieceInfo gives the shape of the piece and the holes fill accordingly.

**public void empty (int[][] pieceInfo):** This method empties the board according to the removed piece’s pieceInfo which means setting according holes to 0.

**public boolean isFull():** This method returns true if all holes’ values are 1 and false if there is a hole with value 0.

**3.4. Detailed System Design**

The complete detailed system design is shown next page. This design is composed of the union of three subsystems: Game Manager System, Game States System and Game Entities System.

