CS 319 - Object-Oriented Software Engineering

Design Report

Instructor: Eray Tüzün

Project Name: The Wall

Group 3E

Erdem Ege Maraşlı - 21602156

Ayça Begüm Taşcıoğlu - 21600907

Alperen Koca - 21502810

Hammad Malik – 21600468

Ensar Kaya - 21502089

Contents

1. Introduction:

1.1 Purpose of the system

Our aim is to develop a 2D strategy game named “The Wall”, which is inspired of the board game “Walls & Warriors”. The main objective of the game is splitting friendly and enemy units by using the designated walls. The units will mostly be regular knights. However, there will also be some additional units with additional features. The designated walls are same as the walls in “Walls & Warriors” and players are expected to use mouse to drag these walls to the game board and split the enemy and friendly units. Basically, we add some new features and play modes into the Walls & Warriors game such as changing the icons of game objects, challenge mode which is a play mode with a time limit and 3 different map from different difficulty levels, campaign mode which is a play mode with a story line aims to rescue the Princess, developer mode which is a play mode with 2 different options such as creating a brand new map or play a map made by different users. The game will be a desktop application and can be played by using mouse merely. This report consists of an overview of “The Wall”, description of gameplay and game objects. Then, the report specifies functional and non-functional requirements. The system models namely use case, dynamic, object class and sequential diagrams will also be presented. In addition, there will be some example screen mock-ups.

1.2 Design Goals

It’s important to show up design goals of the system in order to clarify the quantities which we are going to focus on during this project. We explained most of our

system's nonfunctional requirements in the analysis stage. Important design goals of our system are described below.

End User Criteria:

Ease of Using:

We have to provide good entertainment to the player who is going to play our game, since we are implementing a game. Thus, the player should not experience any difficulties during the usage of our system which will make his/her experience poor. In perspective of this, we designed our user interface user friendly as much as possible such as a 7 years old kid can play our game without having any kind of problems caused by our user interface. We are going to take only mouse input from user. Click, select, drag and drop operations are going to use. This will also make our game user interface easy to use.

Ease of learning:

Most of the users of this game are not ought to have information about how the game is played, how can somebody win or lose or what is the differences between game modes. We are going to implement a "How to Play" button to inform users about these kinds of issues. Also our game of logic is very simple that almost all the users can easily understand.

Maintenance Criteria:

Extensibility:

When creating any software, extendibility and reusability are always important considerations. As it is often observed that there are continuous updates to all famous applications. Thus, “The Wall” will also be implemented such that it can be extended and updated with time according to the needs and feedbacks received by users.

Portability:

Portability is a crucial thing for a software development because it provides the application can reach wide range of user. So, we determined to implement our system in Java because its Java Virtual Machine provides platform independency.

Modifiability:

In our “The Wall” game, we constructed our system as a three-tier architectural decomposition since it’s more suitable for our project design. We are going to use JDBC and database that’s why we choose this architecture.

Performance Criteria:

Response Time: In our “The Wall” game we don’t need to update the screen regularly. All we need to do is update screen when an action comes up such as selecting a game object, dragging a game object or dropping a game object. So, our game does not need any FPS limit but it must response instantly when an action comes up.

Trade Offs:

Performance vs. Memory consumption: We want to keep performance high while consume less memory. Actually we don’t want to make our game performance too high

because we don't need it. Our game will update itself only when a player makes a move. On the other hand, we don't want to consume too much memory because we don't need to increase memory consumption to increase our performance.

Easy of Using & Learning vs. Functionality: We want our game easy to learn and use. That's our priority against functionality. We don't want to confuse our user with complex functions. We want to increase usability instead of functionality.

1.3 References

Inspired by the board-game Walls and Warriors:

https://www.smartgames.eu/uk/one-player-games/walls-warriors

1.4 Overview

In this section, we represented the purpose of the system, to make it more interesting and entertain able for the user. We want to provide ease of use and ease of learning for the user along with portability, high maintainability and high performance.

2. Software Architecture

2.1 Overview

In this part, we are going to decompose our "The Wall" game system into maintainable subsystems. By dividing a big system into smaller subsystems will reduce complexity of the

system and it is going to prevent overwriting of same subsystems and components. Also, we tried to decompose our system in order to apply three-tier architectural decomposition on our system.

2.2 Subsystem Decomposition

In this section, our system is divided into almost independent part to show how our game organization works. We constructed our system as a three-tier architectural decomposition because it's more suitable for our project design which we are going to database systems.

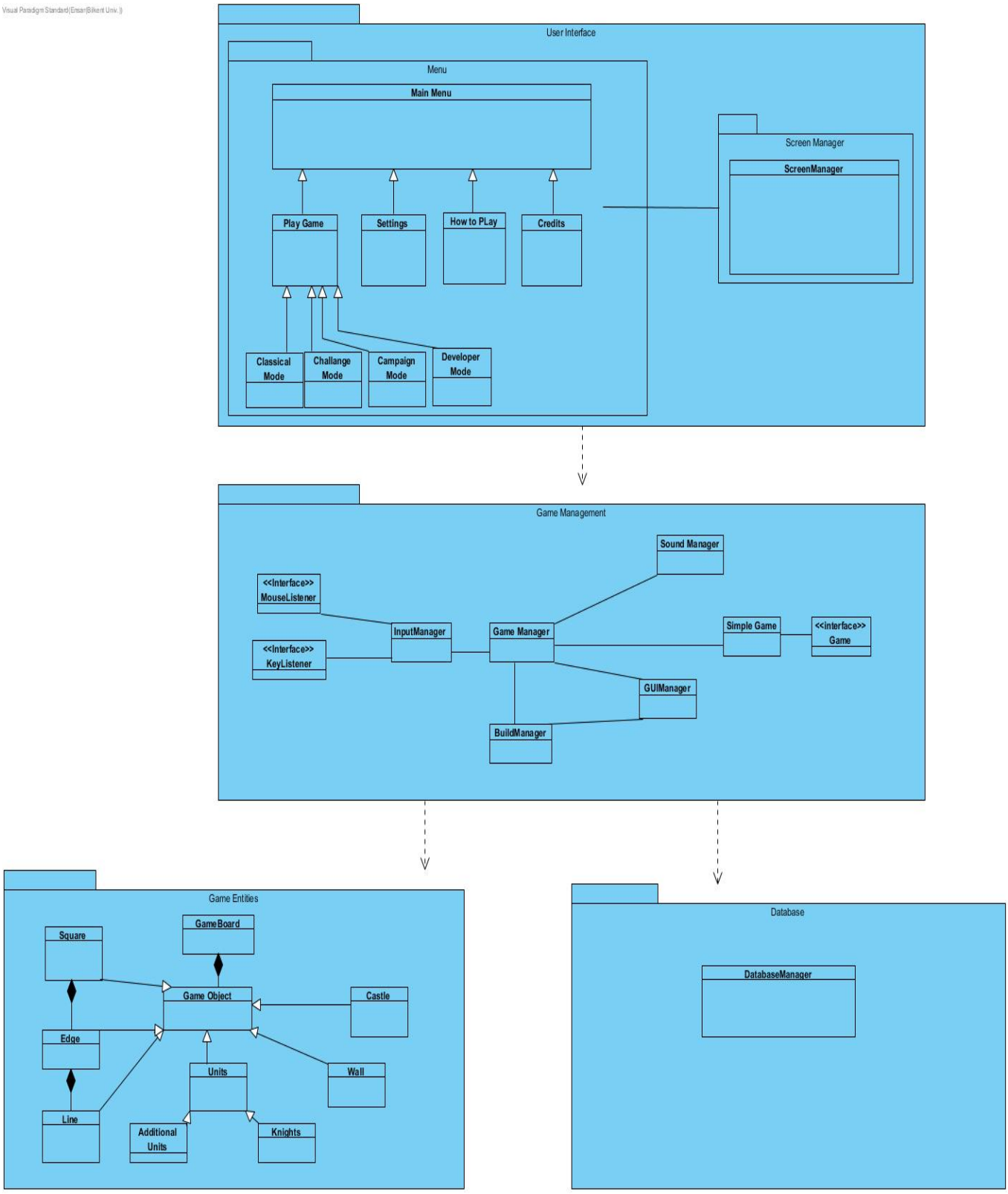
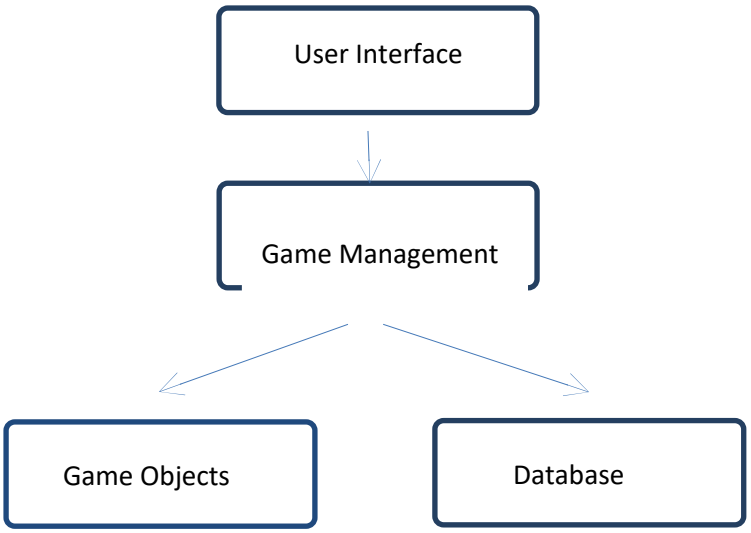

Visual Paradigm Standard(Ensar(Bilkent Univ.))
User Interface
Menu
Main Menu
Play Game
Settings
How to PLay
Credits
Classical Mode
Challange Mode
Campaign Mode
Developer Mode
Screen Manager
ScreenManager
Game Management
Sound Manager
<<Interface>> MouseListener
<<Interface>> KeyListener
InputManager
Game Manager
Simple Game
<<Interface>> Game
GUIManager
BuildManager
Game Entities
Square
GameBoard
Game Object
Castle
Edge
Units
Wall
Line
Additional Units
Knights
Database
DatabaseManager

Figure 1-(Basic Subsystem Decomposition)

2.3 Architectural Styles

2.3.1 Layers

Our The Wall game system decomposition consists of three four layers such as User Interface, Game Management, Game Objects and Database. These four layers have a hierarchy between each other. Our top layer is User Interface as expected since its responsible for the interaction with user. Game Management layer is the following layer. It is actual game engine for our project which get information from Game Objects and Database and fills the User Interface layers background. We have 2 bottom layers since we are going to implement almost 2 different games one is online and the other is offline. First of them is Game objects layer which contains all the necessary objects for our game. Finally our last layer is Database which will store the necessary information for our online game on a local server.



2.3.2 Three-tier Architectural Decomposition

Three-tier architectural decomposition is mainly similar to Model View Controller architectural style. The difference is three-tier is also has base layer for database systems. Other than we are going to have our model, view and controller classes separately. By doing these divisions from a big system to smaller subsystems, we are planning to decrease complexity of our project. Basically, we isolated the main game engine

2.4 Hardware / Software Mapping

During the implementation process of the game “The Wall”, Java programming language will be used, which was developed by the corporation Oracle. We will use the latest Java development kit during the implementation process(Java SE DK 8u191).

In terms of Hardware configuration, “The Wall” mostly requires a basic computer mouse in order to enable users to play and traverse the game. A keyboard is also required as the users are expected to type their names and password for authentication processes. Therefore, it can be said that user’s devices hardware should include both mouse and keyboard.

The game will be implemented and to be able to run on Java. Therefore, a computer with operating system and java system will be adequate. The computer should be able to compile and run the files with the extension of “.java”. Java is independent of platform, namely Java is operable independently from the operating system. Therefore existence of the compiler and any operating system is adequate to run the game.The game will have “.txt” structures for the personal data’s(username and

password).Therefore the operating system has to support ".txt" extended files.While the implementation process, we will use 2D graphics libraries of Java. It does not require and additional software rather than Java environment to be rune. In terms of the hardware, most of the computers are able to run 2D graphics.

2.5 Persistent Data Management

The game of "The Wall" enables users to share their own maps with the game community. Therefore, a server database is required to let users upload and download the desired maps. The maps will be hold and saved in text format (.txt), and these data's will be persistent in the server. The server will also store the usernames and their passwords in the database. We will have a server database consisting of the maps and user's data's.

2.6 Access Control and Security

"The Wall" will enable users to play the base game without any internet connection. However, to share and download the designed maps created by community, internet connection is required.

The users have to login into their accounts in order to access the online features. The users are expected to remember their passwords and usernames for the sake of security. We are planning to help the users having struggles while logging in(forgot password option). The usernames and password will not be shared with anyone else and these data's will be stored in the server database.The maps created by users will be stored in the local memory until the users want to share it with the community. Therefore, server will not access these data's without the permission of the users. The other data's to be used in the game

(sounds,images,maps) will be stored in the local hard drive in order to make the game playable without the internet connection.

2.7 Boundary Conditions

Initialization

"The Wall" will not require any installation of the game. Therefore, ".exe" or some other extension formats will not be used. We are planning to make the game executable with the format of ".jar".

In order to initialize the online features, authentication from the server is required. The users will enter their login information's and the server will try to match these information's with the respective data's in the server database.

Termination

The game "The Wall" can be terminated by using (clicking) the "quit game" option that is in the main menu or by using the top right button imaging cross. The termination of the game will also terminate the connection between the user and the server.

The users can also terminate their connection with the server by logging out. This enables them to continue the game in local database.

Error

If an error occurs while initializing the game such that it will not launch, the game will not start.

If a user will not be able to login to the server, user will be asked to re-enter the login information or to be redirected to the “forgot password” option. If the problem is because of the internet connection, users will be informed about the error.

If an error occurs while playing the game, most probably the progress of the last game will be lost. However, the progress of the “campaign mode” will be held in the local hard drive and users would continue from the last level they have come.

If an error occurs while sharing and/or downloading data from the server database, the users will be informed.

3. Subsystem Service

3.1 Detailed Object Design

The detailed object design diagram will provide a better understanding of the subsystem and their interactions with each other. This diagram will help us to figure out when we design and implement our project.

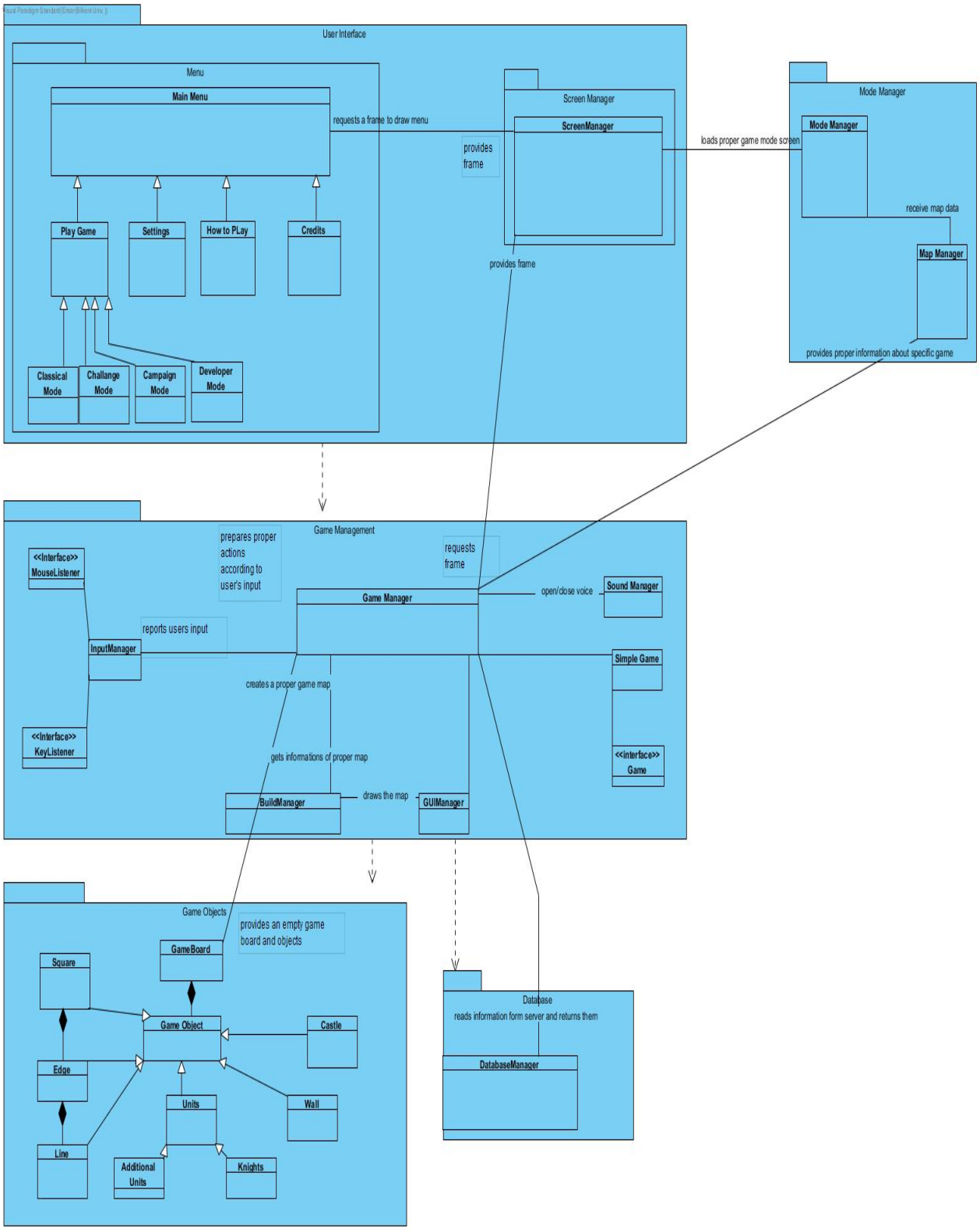
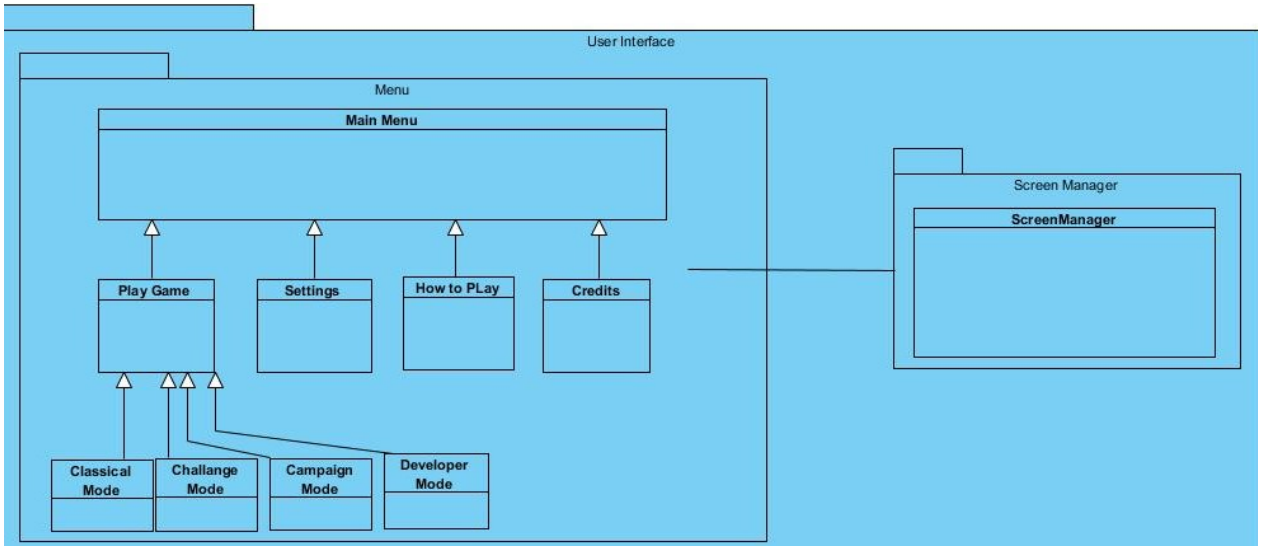
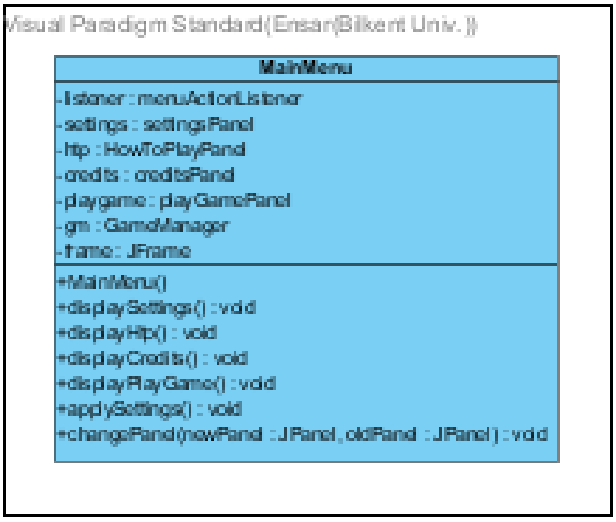


Figure 2-(Detailed Subsystem Decomposion)

3.2 User Interface Subsystem



Menu Class



Attributes:

private menuActionListener listener : This attribute is used for detecting user actions in the main menu.

private settingsPanel settings : Used for constructing the settings button on main menu screen.

private HowToPlayPanel htp : This attribute is used for constructing the “How to Play?” button on main menu screen.

private creditsPanel credits : This attribute is used for constructing the credits button on main menu screen.

private playGamePanel playGame : This attribute is used for constructing the “Play Game” button on main menu screen.

private GameManager gm : This attribute is used for specifying the game to be played

private Jframe frame : This attribute denotes to the main menu frame

Constructor:

public MainMenu() : Initializes the main menu of the game

Methods:

public void displaySettings() : This method displays the Settings button.

public void displayHtp() : This method displays the “How to Play” button.

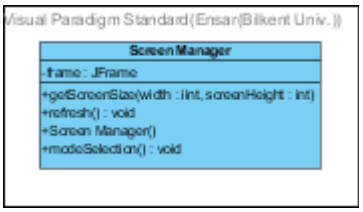
public void displayCredits() : This method displays the Credits button.

public void displayPlaygame() : This method displays the “PlayGame” button.

public void applySettings() : This method applies the changes made by the user in settings.

public void changePanel(newPanel : JPanel, oldPanel : Jpanel) : This method is being used to change the panels on the screen. “newPanel” is being replaced with “oldPanel”

ScreenManager Class



Attributes:

private JFrame frame : This attribute is the frame that the application is built on.

Constructor:

public ScreenManager() : Initializes the ScreenManager for the first time, which will arrange the screen to be displayed on the application.

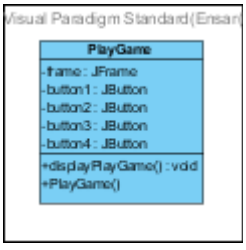
Methods:

public void setScreenSize(width : int, screenHeight : int) : This method sets the sizes of the screen. “width” refers to the width of the screen and “screenHeight” refers to the height of the screen.

public void refresh() : This method refreshes the screen.

public void modeSelection() : This method detects the mode selection of the user.

PlayGame Class



Attributes:

private JFrame frame : This attribute is the frame that user is going to select the mode to play the game.

private JButton button1 : This attribute is the button for classical mode.

private JButton button2 : This attribute is the button for challenge mode.

private JButton button3 : This attribute is the button for campaign mode.

private JButton button4 : This attribute is the button for developer mode.

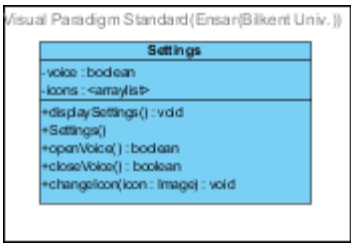
Constructor:

public PlayGame() : Initializes the play game screen for the first time.

Methods:

public void displayPlayGame() : This method displays the "PlayGame" menu on the screen.

Settings Screen



Attributes:

private boolean voice : This attribute indicates whether the voice is opened or not. True means the voice is opened while false means that the voice is closed.

private <arraylist> icons : This attribute is the array of images that users might prefer instead of the default images.

Constructor:

public Settings() : This constructor initializes the Settings for the first time.

Methods:

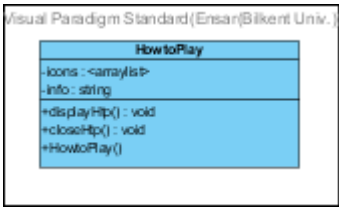
public void displaySettings() : This method displays the existing settings on the screen.

public boolean openVoice() : This method opens the voice of the game. If the operation is done successfully, returns true.

public boolean closeVoice() : This method closes the voice of the game. If the operation is done successfully, returns true.

public void changeIcon(icon : image) : This method changes the icon of the users' units in the game. The given image becomes the image of the units.

HowToPlay Screen



Attributes:

private <arraylist> icons : This attribute contains the icons that will yield messages about how to play.

private String info : This attribute is the message that will be yielded after triggering the respective icon.

Constructor:

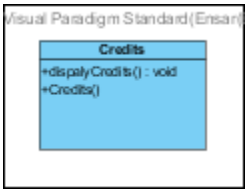
public HowToPlay() : This constructor initializes the "HowToPlay" option, which will help the users about the rules of the gameplay.

Methods:

public void displayHtp() : Displays the "HowToPlay" screen.

public void closeHtp() : Closes the "HowToPlay" screen.

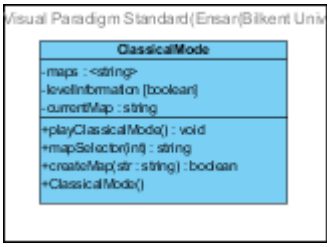
Credits Screen



Methods:

Public void displayCredits(): This displays the credits, which include the names of the developers, on the screen

Classical Mode Screen



Attributes:

private <string> maps: It keeps all the maps for classical mode.

private boolean levelInformation: It keeps the available maps information in an array. Because at the beginning only the first level is open and the others are locked. When player passed a level than next level will not be locked anymore.

private string currentMap: It keeps the information of the map which is going to play.

Constructor:

public ClassicalMode() : It creates an instance of ClassicalMode class.

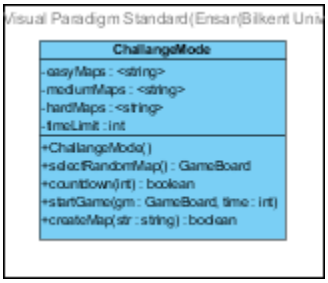
Methods:

Public playClassicalMode():void : It will interact with the GameManagment class, starts a classical mode game.

Public mapSelector(int:int):string : It will select the map in the maps arraylist and returns it's information for createMap(str:string) method.

Public createMap(str:string): It takes a string as a parameter and interact with GameManagment class to create the map which will be play.

Challenge Mode Screen



Attributes:

Private easyMaps<string>: It will store easy difficulty maps for the challenge mode

Private mediumMaps<string>: It will store medium difficulty maps for the challenge mode

Private hardMaps<string>: It will store hard difficulty maps for the challenge mode

Private timeLimit:int : It is the time limit for the challenge mode which we will predetermine.

Constructor:

Public ChallangeMode(): It'll create an instance of ChallangeModeclass.

Methods:

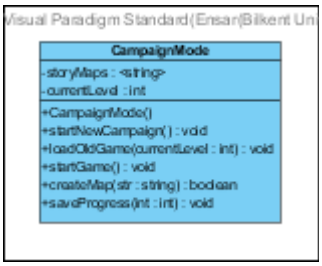
Public selectRandomMap():string : This method will select one map information from the arraylists above and returns the map information with a string.

Public countdown(int:int):Boolean : This method will start the countdown for the game and will return false is countdown is reached.

startGame(gm:GameBoard,time:int) : This method will take prepared game boards and time limitation. It will interact with GameManager class and starts the game.

createMap(str:string):Boolean: This method will take the output of selectRandomMap() method and creates a full game board with all units.

Campaign Mode Screen



Attributes:

Private storyMaps:<string> : This attribute will store the map data's of our campaign mode.

Private currentLevel:int : This value will be equal to the last passed level number.

Constructor:

Public CampaignMode(): It'll create an instance of CampaignModeclass.

Methods:

Public startNewCampaign():void : This method deletes all the progress of player and start a brand new campaign from the beginning.

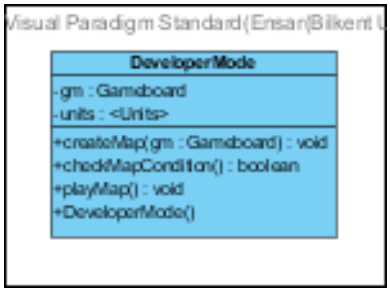
Public loadOldGame(currentLevel:int) : This method helps user the continue campaign mode from his last passed level

Public startGame():void : This method basically works inside the startNewCampaign() , loadOldGame() methods and it starts the game.

Public createMap(str:string):Boolean :This method creates a Game board with given parameter string. Returns true if the process is successful.

Public saveProgress(int:int):void : This method saves the progress of player.

Developer Mode Screen



Attributes:

Private GameBoard gm : This attribute provides an empty GameBoard.

Private <unit> units : This is the array list of all the possible units in the game

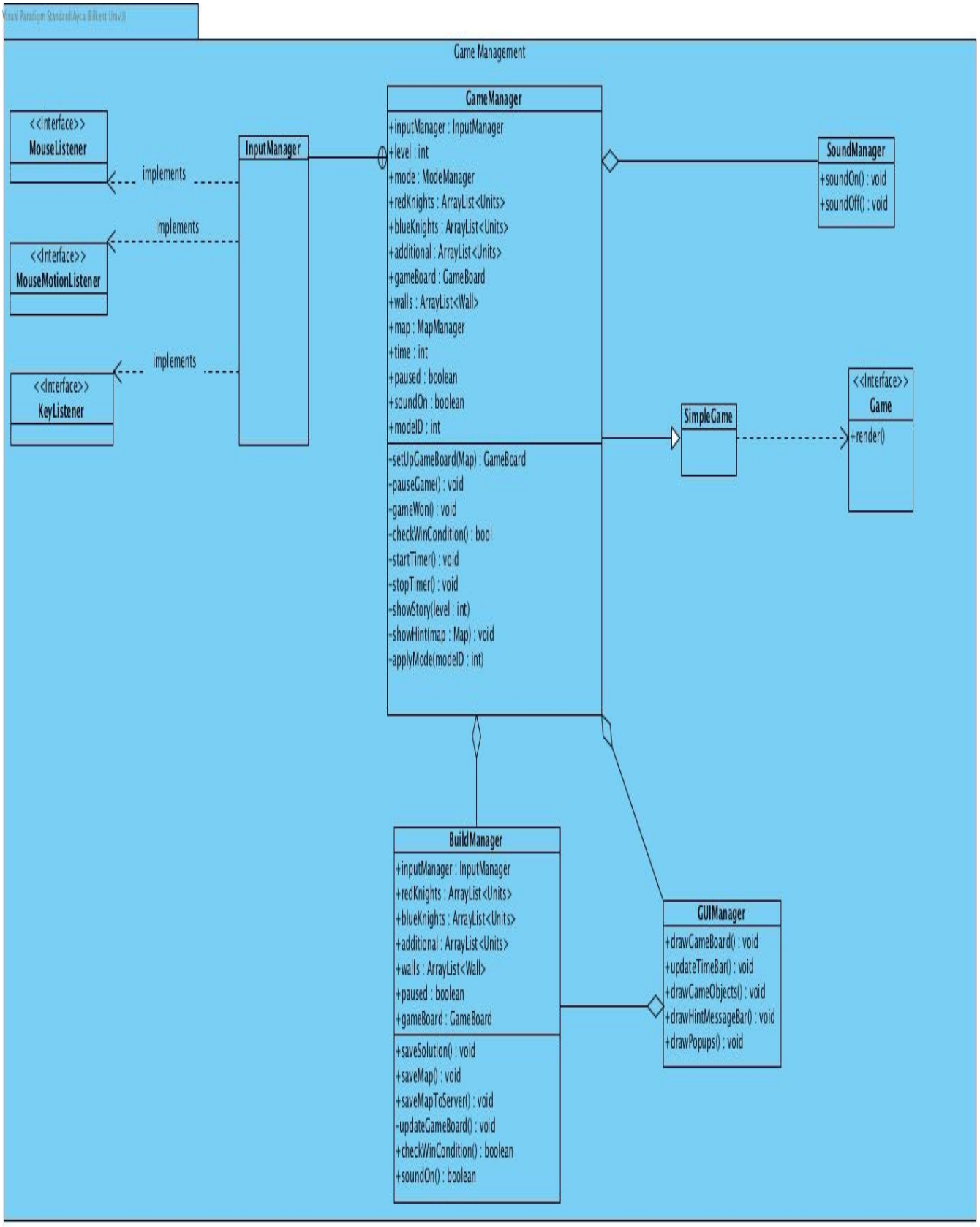
Constructor:

Public createMap(GameBoard gm) : Used to initialize and create a Map using the GameBoard type as input.

Methods:

Public boolean checkMapCondition(): Used to check if the created Map obeys the requirements.

Public void playMap(): Used to play the created map, provided that it fulfills the requirements.



3.3 Game Management Subsystem:

Figure 2 – Game Management Sub-System Interface.

GameManager Class

This class is the main control system class of the game modes classical, challenge and campaign.

It gets map data and mode data from ModeManagement subsystem and user interface subsystem.

Attributes:

private inputManager inputManager : This object gets input from user according to mouse methods.

private int level: This variable contains the level number for campaign game mode. If mode is not campaign it is equal to “-1”.

private ModeManager mode : This object contains the selected game mode options.

private ArrayList<Units> redKnights : This array list contains the red knights for the game which is taken from map.

private ArrayList<Units>blueKnights : This array list contains the blue knights for the game which is taken from map.

private ArrayList<Units> additional : This array list contains the additional units for the game which is taken from map.

private ArrayList<Wall> walls : This array list contains the walls for the game which is taken from map.

private GameBoard gameBoard : This object is empty game board at the beginning. After setUpGameBoard() method called gameBoard object fixed according to map.

private MapManager map: This object contains map data which is taken from mapManager in Mode Management subsystem.

private int time : This variable contains time for the challenge mode. If mode is challenge mode this initialized as 0 at the beginning, otherwise it initialized as -1.

private boolean paused : This variable checks if game is paused or not.

private boolean soundOn: This variable checks if game sound is on.

private int modeID : This variable contains the game mode type. 1 for classical mode, 2 for challenge mode, 3 for campaign mode and 4 for build mode.

Methods:

public void setUpGameBoard(Map map) : This method setups game board according to given map which is taken from mapManager in Mode Management subsystem.

public void pauseGame() : This method pauses the game. Makes paused = true and asks for the pause screen.

public void gameWon() : This method called when game is finished and won. If game is done this calls the main menu screen.

public boolean checkWinCondition() : This methods checks the win conditions. If all conditions satisfied it returns true and calls the gameWon() method.

public void startTimer() : This method starts the timer for challenge mode.

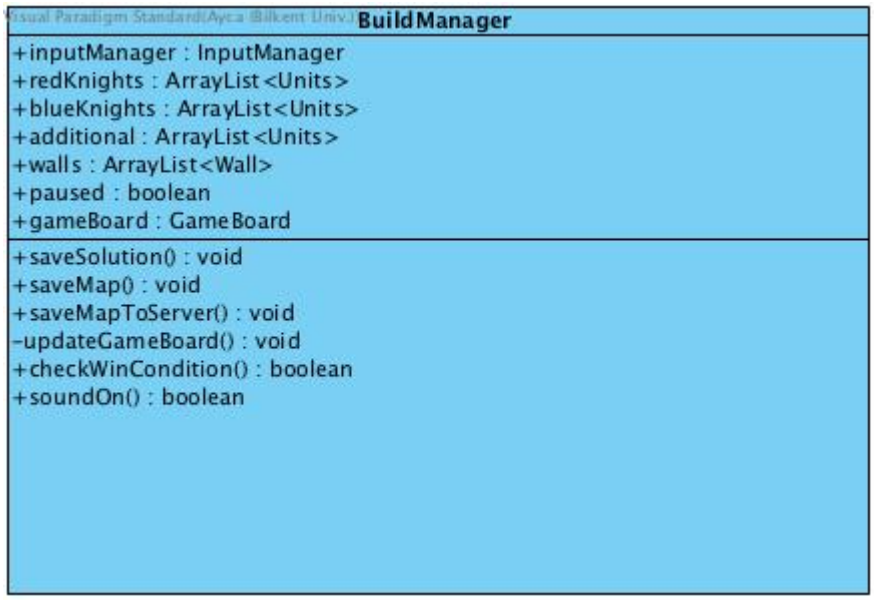
public void stopTimer() : This method stops the time for challenge mode if gameWon() called.

public void showStory(int level) : This method shows the related level's pop up messages which is saved in mode object if mode is campaign mode.

public showHint(Map map) : This method gets the position of the 1 wall from the solution of the map and place that wall into the game board.

public void applyMode(int modeID) : This method setups the game according to the selected mode.

BuildManager Class



This class is the main control system class of the game mode for build mode. User creates a new map on this controller class and saves the new map into the server.

Attributes:

private inputManager inputManager : This object gets input from user according to mouse methods.

private ArrayList<Units> redKnights : This array list contains the red knights for the new map.

private ArrayList<Units>blueKnights : This array list contains the blue knights for the new map.

private ArrayList<Units> additional : This array list contains the additional units for the new map.

private ArrayList<Wall> walls : This array list contains the walls for the new map.

private GameBoard gameBoard : This object is empty game board at the beginning. User fills the game board with his or her actions.

private boolean paused : This variable checks if game is paused or not.

private boolean soundOn: This variable checks if game sound is on.

Methods:

public void updateGameBoard() : This method updates the game board according to actions of user.

public void saveSolution() : This method saves the new map's solution which is provides by user.

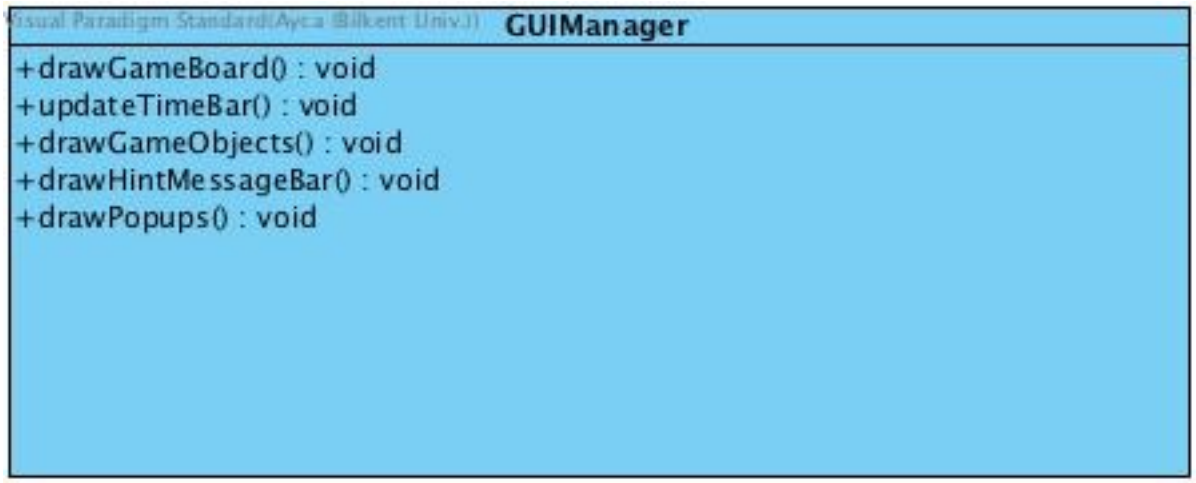
public void saveMap() : This method saves the new map which is provided by user to the MapManager.

public void saveMapToServer() : This method saves the new map to the server if user wants.

public boolean checkWinCondition() : This methods checks the win conditions when user's new map is ready. If all conditions satisfied it returns true and calls saveMap().

GUIManager Class

This class controls and updates the graphical user interface according to the main control systems which are GameManager and BuildManager.



Methods:

public void drawGameBoard() : This method draws the game board according to data from one of the main controllers which are GameManager and BuildManager.

public void updateTimeBar() : This method redraws the time bar every second if game mode is challenge mode. Time data provided by GameManager.

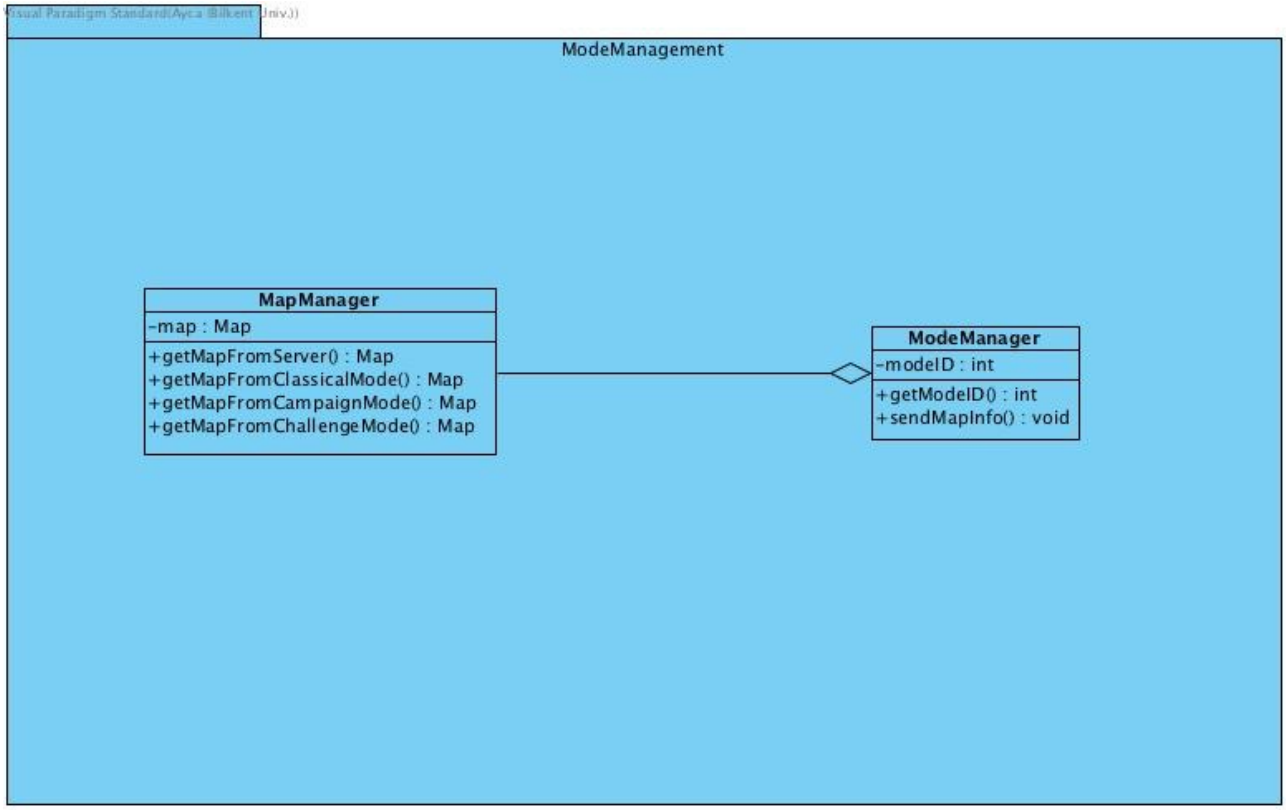
public void drawGameObjects() : This method draws the game objects according to data from one of the main controllers which are GameManager and BuildManager.

public void drawHintMessageBar() : This method draws the hint message when user asked for a hint.

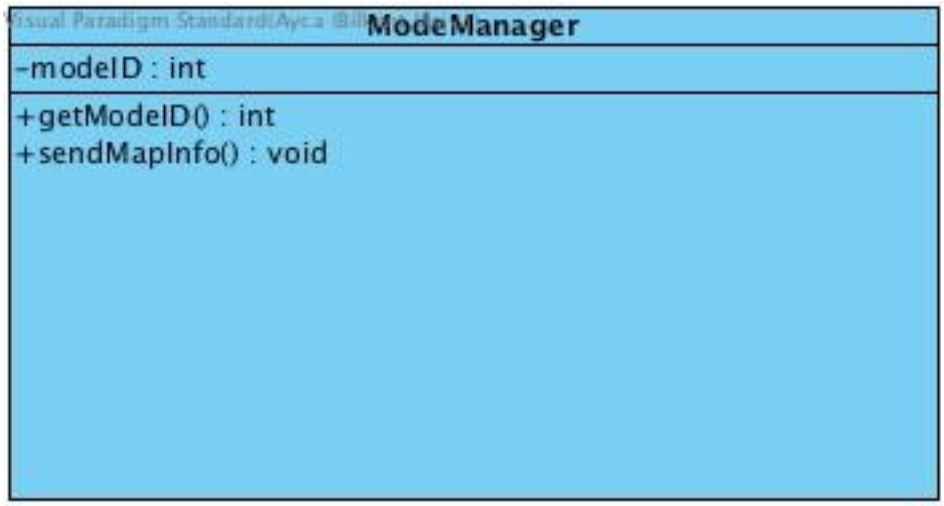
public void drawPopups() : This method draws the pop up messages which is related with the story. This method called in campaign mode.

3.4 Mode Management Subsystem

This subsystem provides mode and map data to our controllers which are in the Game Management subsystem. Mode and map data provided by the selection of user in the menu.



ModeManager Class



This class gets the mode data from the system and provides mode data to the mapManager. Provided map and mode data will be sent to the main controllers which are in the Game Management subsystem.

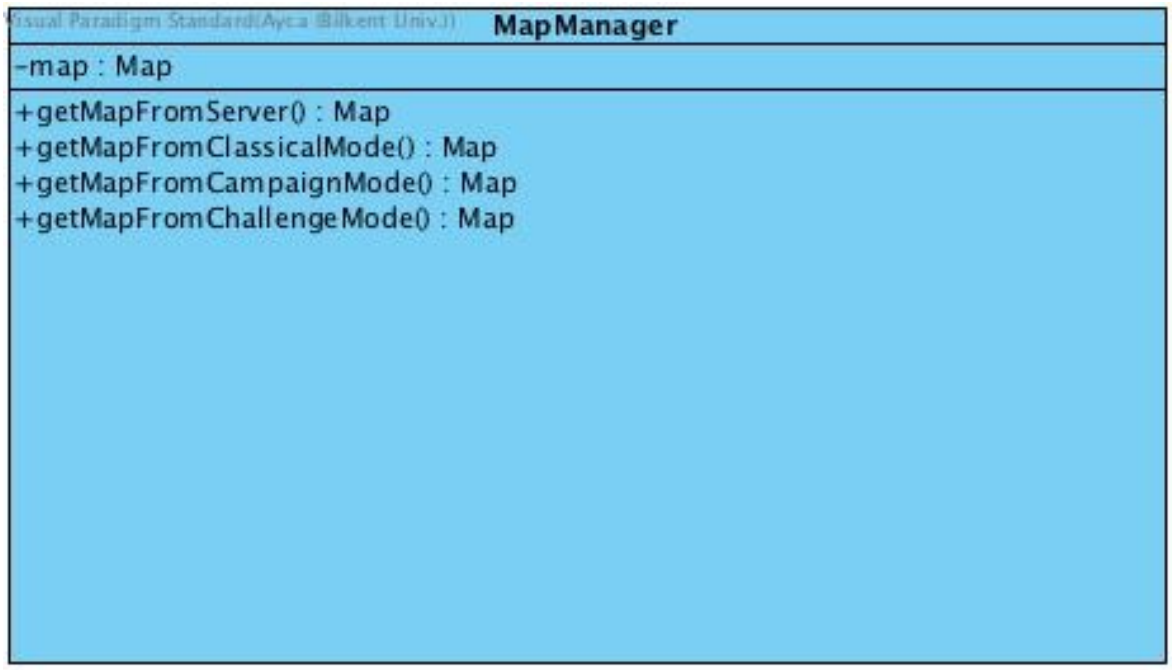
Attributes:

private int modeID : This variables contains the mode information which is taken from user interface on mode selection.

Methods

public void getModeID() : This method gets the mode ID from the user interface on mode selection.

public void sendModeInfo() : This method sends the modeID into the MapManager to provide a map for selected mod.



MapManager Class

This class gets the mode data from ModeManager and level data from user interface and provides the desired map for the main controller subsystem which is Game Management subsystem.

Attributes:

private Map map : This variables contains the desired map data.

private int levelID: This variable contains the levelID which is taken from user interface.

private int modeID : This variable contains the modeID which is taken from ModeManager.

Methods

private void getDesiredMap(int ModeID, int levelID) : This method calls other getMap methods according to the modeID.

public Map getMapFromServer(int levelID) : This method gets the map data from the server by related parameter.

public Map getMapFromClassicalMode(int levelID) : This method gets the map data from the classical mode by related levelID.

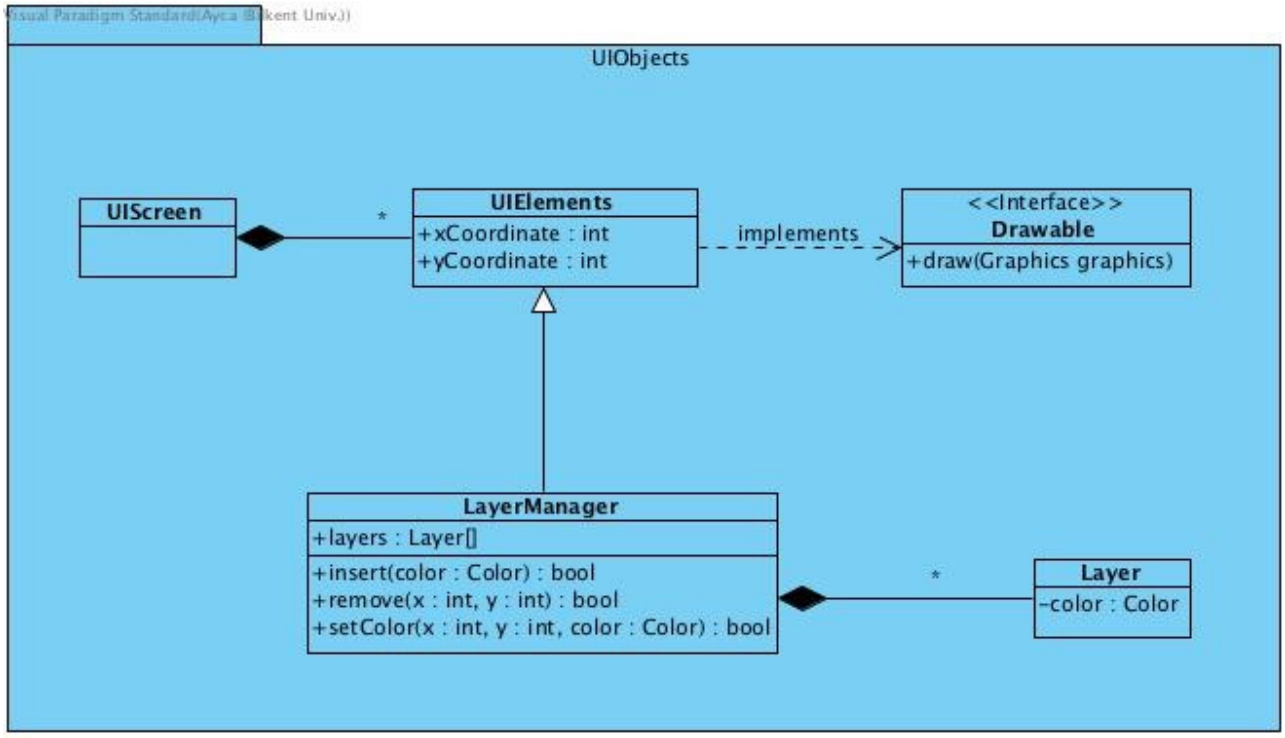
public Map getMapFromCampaignMode(int levelID) : This method gets the map data from the campaign mode by related levelID.

public Map getMapFromCampaignMode() : This method gets a random map data from classical mode.

public void sendGameInfo(): This method send the informations which are levelID, modeID and map to the our main controller subsystem which is Game Management subsystem.

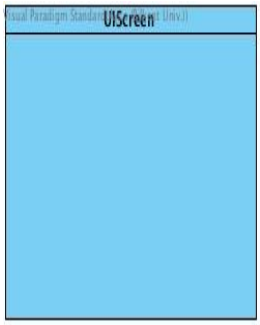
3.5 UIManagement Subsystem

This subsystem is for creating user interface objects.



UIScreen Class

This class will keep UIElements.



UIElements Class

This class will keep the attributes of the basic user interface objects such as their x and y coordinate. There will be various UIElements that they will be represented by UIScreen.

Attributes:

private int xCoordinate: x coordinate position of the user interface object.

private int yCoordinate: y coordinate position of the user interface object.

LayerManager Class

This class will keep a list of Layers. It is an instance of an UIElements.

Attributes:

Layers[ ] layers: list of layers.

Methods:

private bool insert(color: Color):

A method to add a layer, returns true if the process succeed else returns false.

private bool remove(int x, int y):

A method to remove a layer with specified coordinates, returns true if the process succeed else returns false.

private bool setColor(int x, int y):

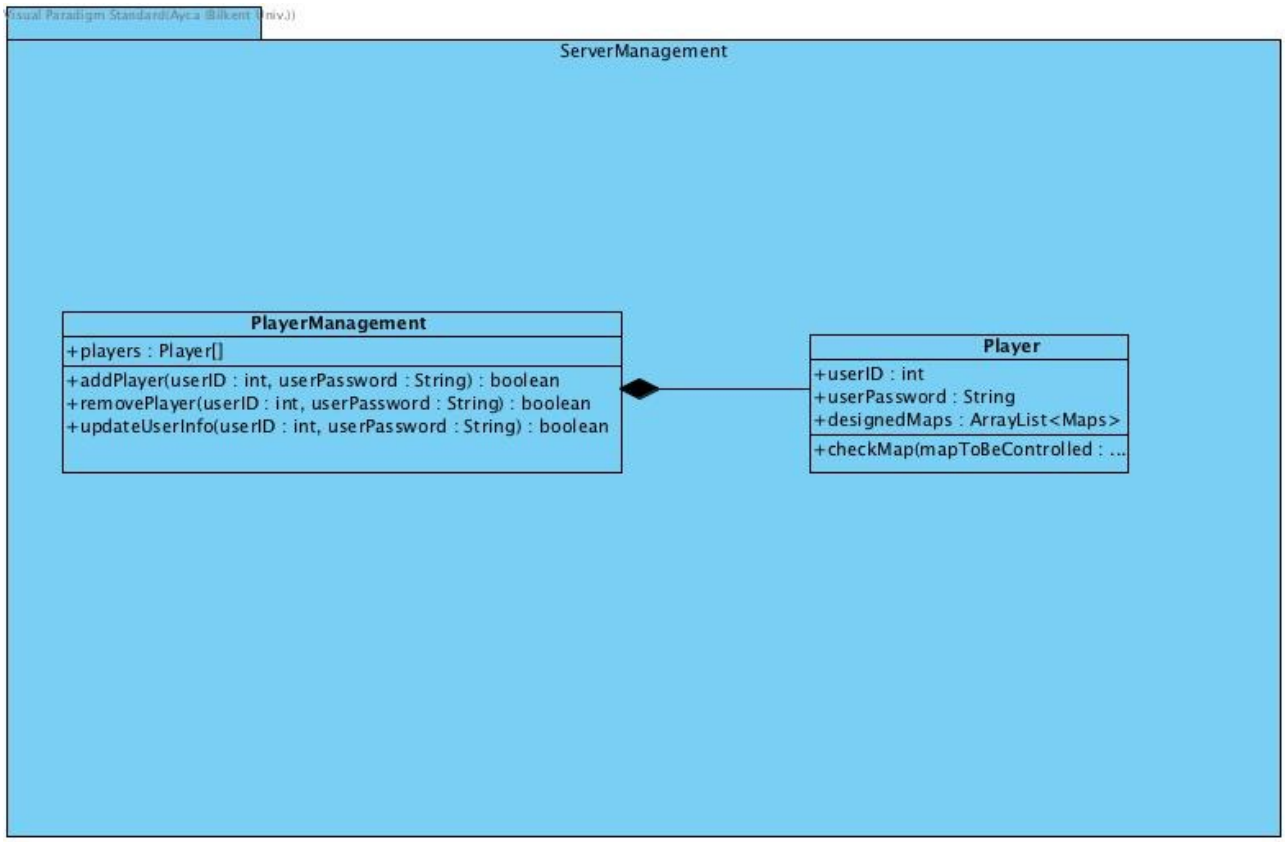
A method to set the color of desired layer which is specified with its coordinates(x and y), returns true if the process succeed else returns false.

Layer Class:

A class for basically a rectangular layer which is an instance of UIElements.

Attributes:

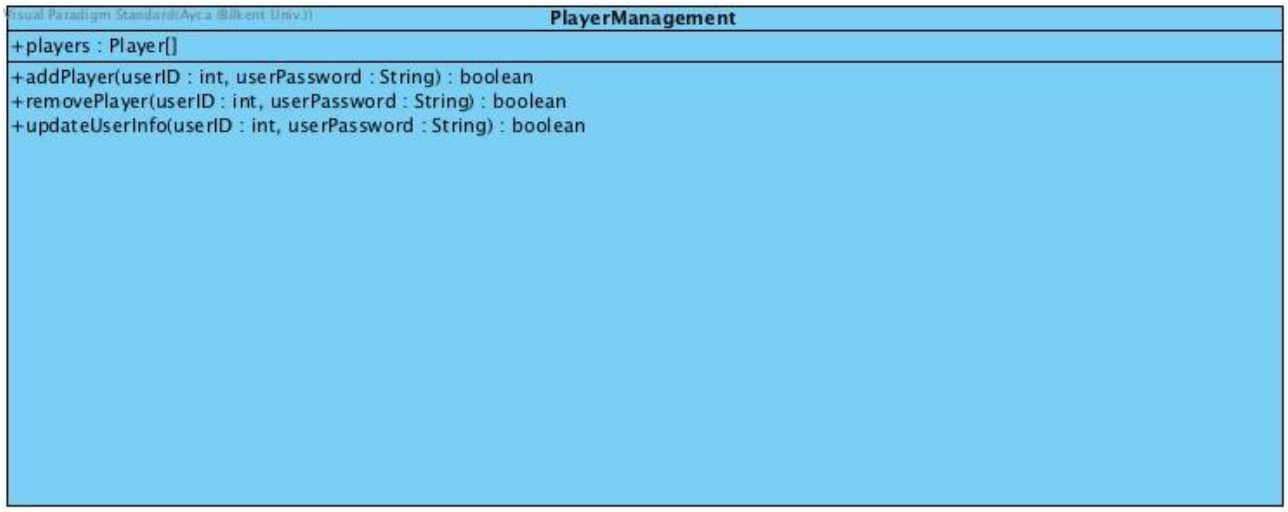
Color color: Color of the layer.



3.6 ServerManagement Subsystem

This subsystem is for creating a player accounts server. There will be a player list and every player can add their valid designed maps to the server if they wish, in developer mode.

PlayerManagement Class



This class will keep a list of players.

Attributes:

private Player[] players: keepsa list of player accounts.

Methods:

private bool addPlayer(int userID, int userPassword):

A method to add a player account, returns true if the process succeed else returns false.

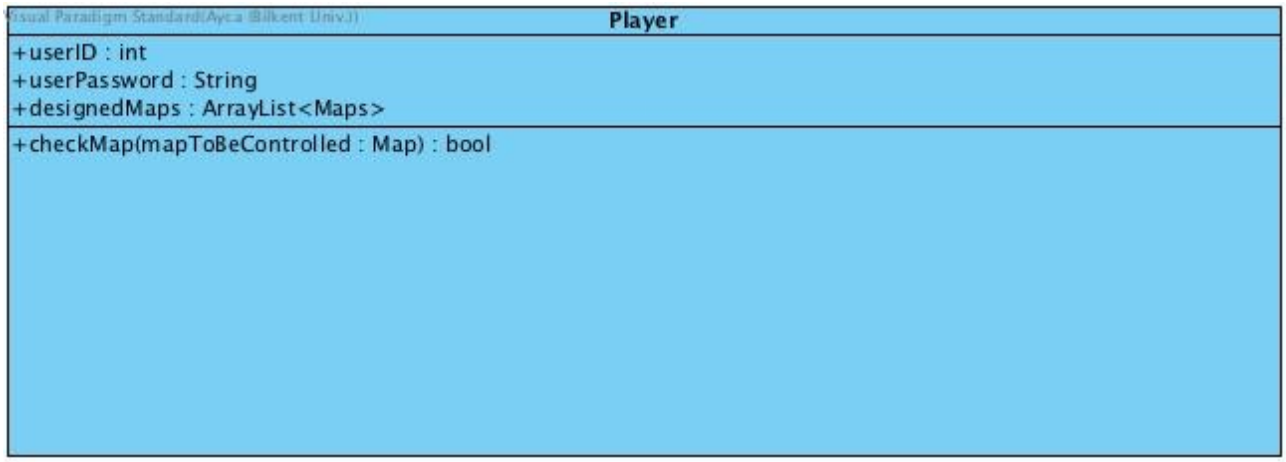
private bool remove(int userID, int userPassword):

A method to remove a player account with specified features(id and password will be requested), returns true if the process succeed else returns false.

private bool updateUserInfo(int userID, int userPassword):

A method to update userInfo, returns true if the process succeed else returns false.

Player Class



This class is used to define a certain player. His user ID and password, along with any designed maps he/she created.

Attributes:

private int userID: a special number to identify user account.

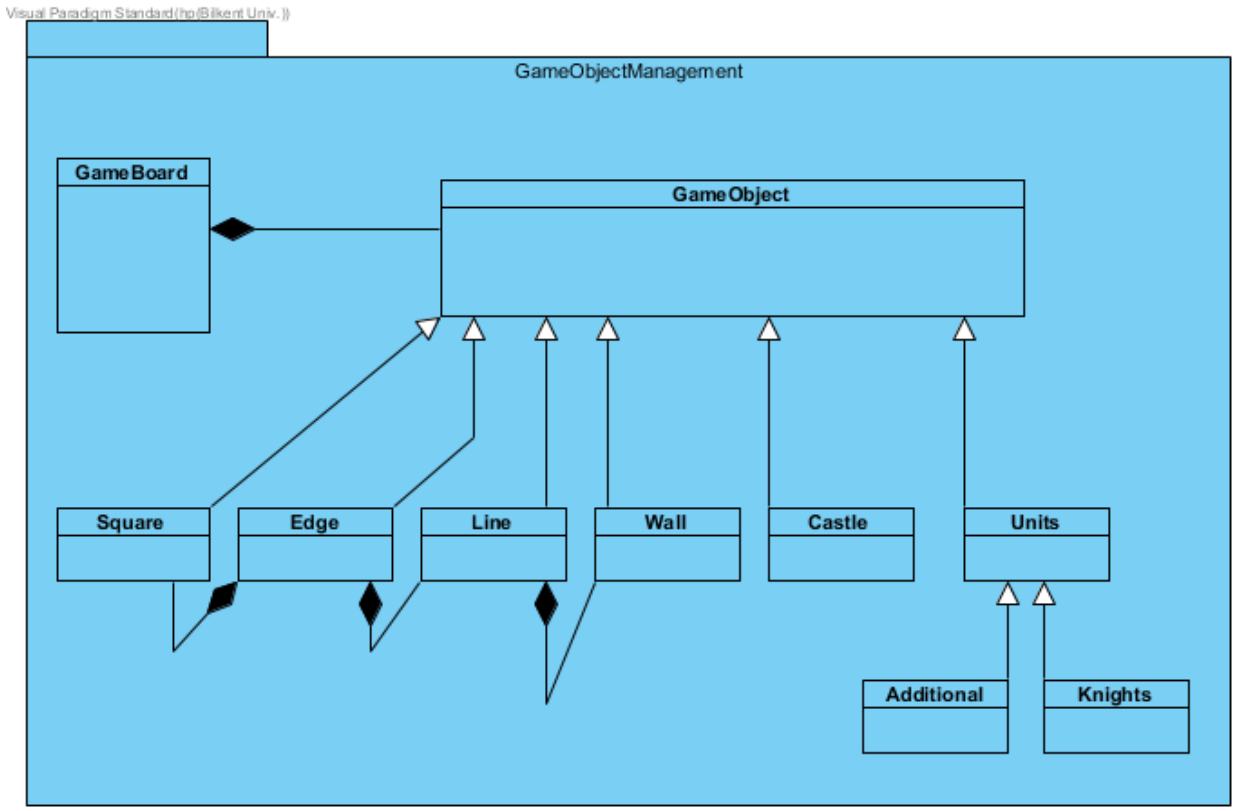
private int userPassword: a special string which is created by user to log in to his/her account.

private ArrayList<Map> designedMaps: a map list for user, if he/she create a valid map then he/she can save it in this list.

Methods:

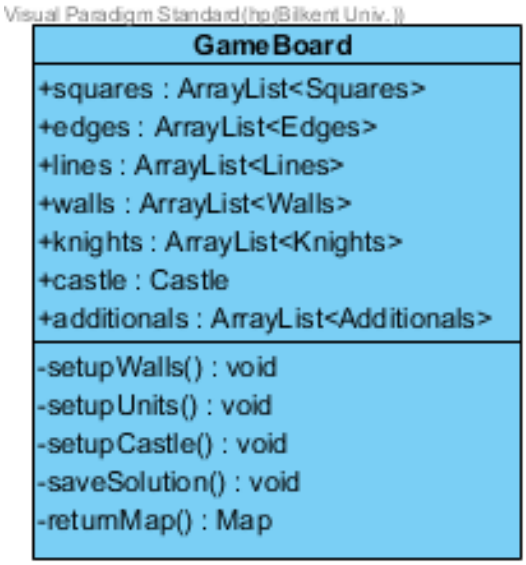
private bool checkMap(mapToBeControlled: Map): A method to check given map, returns true if the appropriate solution for designed map is provided, else returns false.

3.7 Game Object Management Sub-system:



In the Game Entities sub-system, all the game entity classes are grouped together with their relations. We have the following classes: GameBoard, GameObject, Square, Edge, Line, Castle, Wall, Units, Knights and Additional. These classes and interfaces are explained in this section.

- **GameBoard Class:**



Attributes:

ArrayList <Squares> squares:	The collection of all blocks which make up the game board.
ArrayList<Edge> edges:	The collection of all the edges of the squares on the game-board.
ArrayList<Lines> lines:	The collection of the lines on the game-board.
ArrayList<Knights> knights:	The collection of the Knight objects which are placed on the game-board.

ArrayList<Walls> walls: The collections of the walls which are placed on the game-board.

ArrayList<Additional> additionals: The collection of additional objects placed on the game-board.

Methods:

Public void SetupWalls(): Instantiates all the walls onto the game-board.

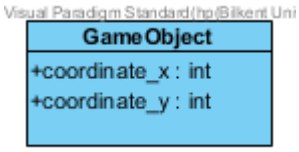
Public void SetupUnits(): Instantiates all the units i.e Knights or additional units onto the game-board.

Public void SetupCastle(): Instantiates the tower-piece onto the board.

Public void saveSolution(): Saves the solution i.e the winning condition, for a particular challenge.

Public Map returnMap(): This method returns the game-board created inside this class.

- **GameObject Class:**

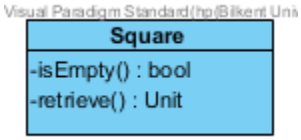


Attributes:

private int coordinate_x: the x coordinate of any object on the 2-D game-screen.

private int coordinate_y: the y coordinate of any object on the 2-D game-screen.

- **Square Class:**



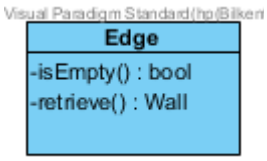
This class represents one square block out of the many squares blocks on the 2-D game-board.

Methods:

public bool isEmpty(): checks whether a particular block has any objects placed on it or not.

public Unit retrieve(): returns the unit (Knight or Tower-piece) which is placed on a particular square block.

- **Edge Class:**



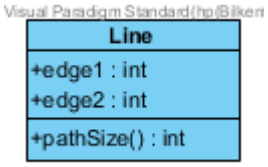
This class represents a particular gap between two square blocks on the 2-D game-board, in which walls can be placed.

Methods:

public bool isEmpty(): checks whether a particular space between the respective squares is empty or not.

public Wall retrieve(): returns the Wall which is placed inside a specific gap on the game-board.

- **Line Class:**



This class is used to help sketch a 2-D line onto the game board to help create a distinct wall.

Attributes:

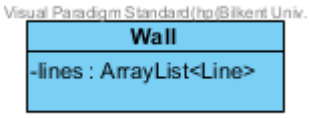
int edge1: The starting point of the line.

int edge2: The ending point of the line.

Methods:

public int pathSize(): The length of the edges which make a certain path.

- **Wall Class:**

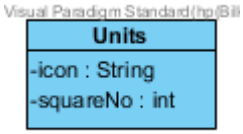


This class uses the Line class to create walls and thus, defines a particular Wall object.

Attributes:

Line lines[]: An array of the straight lines which are used to make one particular Wall, by unique placement.

- **Units Class:**



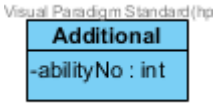
This is the parent class of the Knights and Additional objects. It defines these objects.

Attributes:

String icon: The visible pictorial representation of the Knights or additional objects.

int squareNo: It keeps track of the specific square on which the object is placed on.

- **Additional Class:**



This class defines the additional objects not originally part of the board game "Walls and Warriors." They are extra objects we decided to add in the story mode of the game.

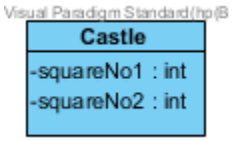
int abilityNo: The conditions and actions which a specific additional unit has to obey in order to win the game.

- **Knights Class:**

This class defines the Red or Blue Knights which are units of the board-game.

int abilityNo: The conditions and actions which a specific Knight has to obey in order to win the game.

- **Castle Class:**



This class defines the Tower-piece from the original game "Walls and Warriors" by specifying its location on the game-board.

int squareNo1: The first square on which the Castle i.e the tower piece is placed on.

int squareNo2: The second square on which the Castle is placed on.