CS 319 - Object-Oriented Software Engineering

System Design Report

Your Story

An Online Messaging Portal for Text Based RPG Games

Group 22

Ali GÜNEŞ

Cevat Bariş YILMAZ

Erin AVLLAZAGAJ

Ertuğrul AKAY

Course Instructor: Uğur DOĞRUSÖZ

Contents

Introduction

“Your Story” is a game which has completely different gameplay from the other games. is a text-based multiplayer role-playing game, which will be implemented in Java. The game will is designed for desktop computers. There are different scenarios and different kinds of characters in the game. The purpose of the game is having fun by chatting with the other players in order to develop a story and acting like a character you always wanted to be.

The detailed information about the content of the game will be presented in the overview section. This report contains detailed gameplay information and the use cases of the project in the upcoming sections. In the first section we are discussing the purpose of this system and what features will it bring better than any other similar implementation. The second following section is where we describe the goals of our design and all the tradeoffs and our picks. In this section we discuss six goals of our design based on the non-functional requirements. We will also discuss what our choices conflict with and why our decision was better that its counterpart. This section is then followed by the architecture of our software also. In this section we will be discussing about the components and the architecture type we chose as well as detailed description of all the components and their inner classes. All the schematics below are created with Visual Paradigm.

Purpose of the system

This system is created to bring together a group of story writers to develop a story their way and get to know other interesting people in the world. This game’s purpose is to help the players develop their imagination and collaborate to create a great story or even connect with other people sharing the same talent and interest.
This game is different from many similar game because the user is more valued to his writing and creating skills rather than just valued by how fast they click the mouse. Another difference to other games is that the ending of the game is not decided on when the time ends or the “castle is captured”, the ending is let free for the player to decide democratically. The same way is for starting the game or kicking out toxic players.

Design goals

Reliability

Our software is designed by keeping in mind that it shall not be crashed under any user's input. During the design we consider any possible state of our objects such that no conditions can lead to any serious crash. We use exception in any potential buggy piece of code. The exceptions will then reroute the program to normal operating way, sometimes without user's consent. The game will respond in determined time if the internet connection exists, if the objects are instantiated, if a lobby is currently in game, if user doesn't exist in database[1] or any other exceptional case that is determined in Analysis Report for each use case of the software. We determined the bugs that can be occurring and determined how can we handle these bugs. Most of the bugs that can occur are because of the game not actually being in real-time. Since the client will be refreshed every 5 seconds the lobbies might have started the game and still appear in the home page as lobbies waiting for players and many other scenarios. After we finish the project, we will possibly release a beta version to get a better idea of any unexpected bugs that might occur while running the program and fix them, this will increase the reliability of the software.

Robustness

Since our software is going to be written in Java we assume Robustness is a feature we get for free. Since Java runs on all the Operating Systems without crashing we are given a great head-start when choosing this language rather than C or C++. Since the program is online, the program should handle the bugs that occur due to loss of connectivity to the database. The client should be able to handle the connection problem and omit any changes if already logged in. Also, the program should wait for connection to the database to reestablish and disallow any change to the data the user wants to make. When reconnected the client will update its data to server's not vice-versa. If the user is not logged in the client can't let the player see the main view. It will notify the user when connection is established. It will be able to communicate using the JDBC which will take care of the protocol. On top of it is the library Erin had developed to simplify the connection.

Modifiability

Most of the system's functions are easily modifiable when the coupling of the subsystems are kept at minimum. In "Your Story", functions of a class do not enormously affect their sub or parent classes, as the data in the database must not be changed unwillingly. In order to do this, abstract classes will be used in both user interface and in the other layers. The close architecture also makes it easy to modify its parts such that the dependent classes will be easily modified once the layer they depent from is substituted or modified. This architecture type is discussed in more details in the second part of this paper.

Usability

Since the user interface is one of the most crucial factors for players of the Your Story, buttons and other interactable objects will be kept basic. Simple instructions will guide the players to do their aim. Rather than doing chain interactions, (like popping a page after a page in player's' interface, every single step of their actions will mostly be kept in a single page, and all of these steps will have an explanation written on them. With this method players will not be stucked to understand the program.

Portability

Portability is one of problems of software development, since today there are lots of different device types, operating system and versions , having a stable performance between all these environments for the softwares is not that easy. To partially overcome this situation, the development language will be Java which can be executed on different platforms. So that game will be playable for all operating systems that able to run JVM[2].

Another advantage of using Java is potential opportunity to port the game for Android environment. Since language used in Android applications is also Java, just by changing the first layer of our architecture, which contains the user interface classes, we can turn the game into an Android app. Moreover, since we using a closed architecture it will not cost us much engineering time.

Finally, using SQL to manage data in our database will let us to migrate any other relational database since it is the standard query language between relational database management systems.

Rapid developments

Rapid development is a key issue to our project. Since we need to release the final product by a specified deadline around beginning of January, before the end of Fall semester of the academic year 2016-2017 we need to be fast at developing and releasing this product. The time to write the code and debug possible bugs is approximately one month, that's why we might release a copy for beta testing by half of the development stage. To help us achieve this rapid development we will be using a MySQL driver for Java called JBDC topped by multi-purpose database communication library that Erin had written before for other applications. This design goal was more as a requirement rather than a choice, because it made us remove the Server written in Java which would be as a middleware between client and database and which would update the client in real time rather than make it query the database every 5 seconds.

Trade-offs

Portability VS Efficiency

To achieve the portability goals, we had to cut some of the functionality of the game. The first trade we made is using Java, which can not be converted to an 'executable-like' shape. Therefore all users have to setup JVM to their computer before playing the game. Also because of existence of a virtual environment between software and operating system, the execution time will be longer than compared to other languages. However since the game won't be that complex, this trade-off won't hurt us much.
The second trade is using a relational database system to manage database. While using a relational database lets us to migrate between different systems because of their standardised structure and language, SQL, we can move between different databases any time. However the downside of relational databases is their efficiency. Since getting the data as we want requires many usage of 'join' command which causes slower performance, relational databases are not fast as object oriented databases.

Rapid Development VS Functionality

By choosing a rapid development we lost a lot in functionality. The game was first decided to have a server and a client sitting in two different machines. The server and the client would

communicate and share data with each other. This would make help us drastically reduce traffic in our database and even have the server and database run on different machine which would increase the computational power of both by many times. What made us retreat from this decision was the fact that the server and the client needed a protocol to communicate (so we needed to design a protocol) and also the socket programming is something new to the majority in our group, not to mention handling connection issue with keep-alive[3] packet transmission through time intervals and encrypting the traffic for secure data exchange. So now the client just queries the database every 5 seconds in a different working thread which updates all the dependant objects if any new data comes. The security of the communication is handled by the JDBC driver.

Definitions, acronyms and abbreviations

[1] Database: A system that is efficient for storing or retrieving data
Socket: A term used for all the network setup needed to
[2] JVM: Java Virtual Machine a virtual machine that converts its program instructions into native instruction of the processor it runs in.
[3] Keep-Alive: Network packets sent at a specified interval to let the server know the client is still connected

Software Architecture

Subsystem Decomposition

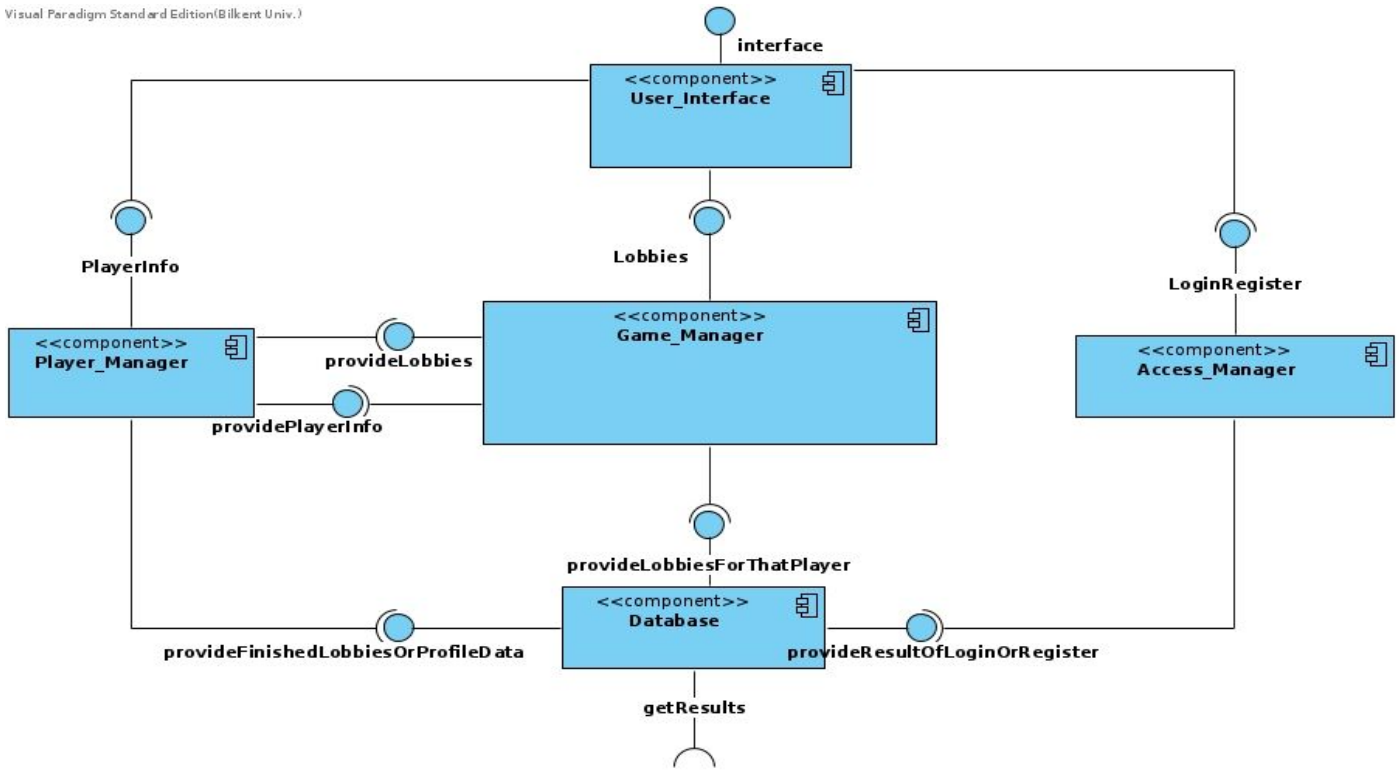


Figure 1.A : Component Diagram of "Your Story".

First layer

The diagram shows all the components and it represents three layered architecture style of our project, "Your Story". The top layer consists of the UI Classes. This layer is responsible for creating the user interface and is designed to offer high usability on the front-end. It takes user input from the keyboard and the mouse, and sends it to managers in second layer which handle the business logic. It will then be updated with any new data the second layer provides. This layer can change in the future to be able to work for Android's UI.

Second layer

In the second layer of architecture, we have the components which handle the business logic. All three of them, are managing and interpreting the data coming from the database in the third layer as well as from the UI components of the layer above. These components are also responsible for UI, when new data comes, or even inserting new data to the database taken from the UI, like chat messages that the user sends.

Third layer

In the third layer, we have the component that handles and simplifies the connection to the database. This one is responsible for feeding new data to the client and provide the required authentication data for the registration and login of a player. It will also insert new data to the database like registering a new user, adding some chat text, updating user's data etc. As mentioned above if there is no internet connection the query(ies) left unprocessed will be considered undone and won't be completed unless the user retriggers such query. The third layer will notify for loss of connection and the second layer will halt and transmit that information to the first layer which will freeze and let the user know the connection is lost and they should wait for it to reestablish.

Reasons for choosing this architecture

In the process of selecting the suitable architectural structure of our project, we considered our design goals and came to the conclusion of using a three layered architecture, for the following reasons:

- It provides a complex structure that properly handles the database communication in regards to the non-functional requirements that were discussed in the analysis report.
- It creates low coupling since all of the classes fit nicely to this architecture and logic of the whole system.
- Since we are able to handle the operations for registration/login of a player and the main game functions in the same level of layer, the data that is required can be stored or taken from a single database, which is in the layer below. This also fits to our design goals (our chosen trade-off), which is Rapid Development over Functionality.

Type of architecture

For this project, we choose the closed architecture over the open one, because one of our design goals is to have the game be modifiable. This will help us to add a server in the near future. So, in that case, we only need to substitute the third layer and modify the second layer, thus the UI layer will stay the same. Or if we need to make this game run in Android devices we only need to modify the UI layer and every business logic will remain intact. This architecture will make it faster to extend the project in the future to enhance it quickly and push the new update quickly to keep up with the market's demands.

Hardware/Software Mapping

Your Story, is a game that is currently being developed in the last version of Java Development Kit (SE 8U112) and can be played across Mac, Linux, Solaris, Windows 7, 8, 8.1 and 10 since these support latest Java updates. As the most basic needs to use our application, one should have a keyboard for user input, mouse for user interface interactions with a computer. On a side note for the keyboard, only English characters will be accepted as an keyboard input, because of programme does not support any other languages.

Since there is an online text messaging and login system with a huge database, full internet connection is required throughout the usage of the application. We do not connect this database to a server since there is not enough time for development, but updating it every 5 seconds so that the player can see the new messages on their monitor. That's why, we have another machine to host the database of Your Story. This machine will be active 7/24, to handle the operations that the players need to complete before and when playing.

Persistent Data Management

An external database will be used to maintain data flow between clients fluently. All clients will be updating its data every 5 seconds and will be able to modify the data in the database by themselves. Data passed between clients through the database will include dynamic contents like chat messages, lobby informations and profile infos. Other static contents like background pictures will be stored in client-side.

Additionally all clients will be able to directly reach the database, there won't be any server-side script to handle requests coming from clients. Therefore while having a faster connection to the database, we won't be doing any extra work for server-side scripts. However our trade-off here will be uncontrolled accesses to the database.

Finally, database will have MySQL database management system and since MySQL is a relational database, SQL language will be the used to query the data and do insertion, update, deletion, etc.

Access Control and Security

"Your Story" will require a connection to the database. The username and password of the database will be hardcoded inside the core of the software. Anyone who can decompile "Your Story" can view its source code and thus get the username and password that server for connecting to the database. This will allow remote code execution in our database from anyone which is a huge issue and we are aware of that. However we will do our best to hide it to the large public, and possibly implement a recognition algorithm that will make the database unresponsive to remote connection other that from the official verified client. This issue happened because we omitted the java server as an intermediate of the database and client due to having rapid development as a design goal. Other than that the connection to the database is secure and allows no eavesdropping. This is implemented by JDBC driver. Once the client is connected to the database and authenticated he can get all the personal data about himself, but no personal data, like password, of other players will be disclosed. The only data a player can view is other players' profile.

Boundary Conditions

"Your Story" is an online program, so it can be problematic to lose the connection to the database while playing the game. Database connection loss blocks sending messages/votes and receiving messages in the game, and participating to lobby out of the game. While in the lobby, if you lose the connection, you have been kicked out of the lobby. If it happens while in the game, you are just given the warning about loss of the Internet connection, but the game will

still continue without you. Also, your messages cannot be saved to database, so if you reload the page, your message will be disappear.

We do not have any data loss problems because our database will be saved in another device. If the program is closed during the game, you can open the program and connect the lobby again without any loss of data. The problems that can be occur while running the game is all about database connection problems and we have these boundaries to be able to gracefully handle such problems.

Subsystem Services

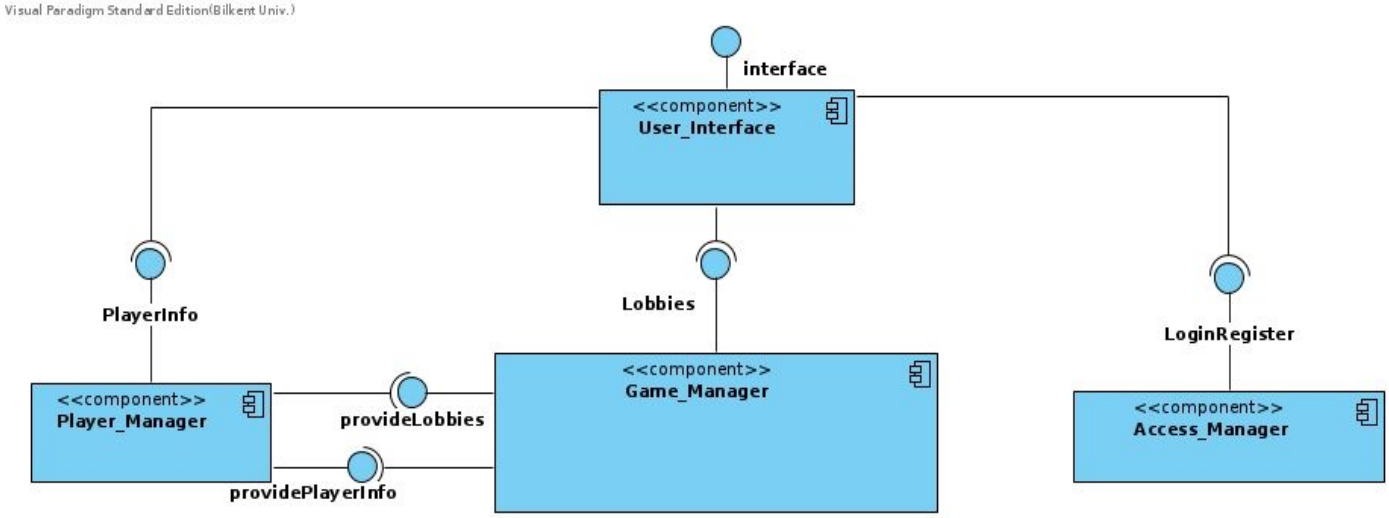


Figure 2.A : Component Diagram Layer 1-2 of "Your Story".

User_Interface Component

The User_Interface component provides a user friendly environment for players. This component updates the view for player everytime there is an interaction occurring between layers. It also gets hardware input from the user, and sends it to the managing layer. For register and login process, when it gets the username and password, it will send these information to access manager to validate it. After the validation process is done, the UI will

change depending on the validity of the user credentials. After that, if it is valid, it will call the player manager to generate the home page.

Player_Manager Component

Player_Manager component requests the data of that particular logged in account. The data is something like the profile picture, description, finished games, unfinished games and all the visible ongoing lobbies where the player can participate. It interacts with the Game_Manager component to create lobbies which will be ascribed to that player. It will also update the view, any time it gets new information.

Game_Manager Component

Game_Manager component handles all the features of the game. It is used to create lobbies and send that data to the database. It will handle everything happening in the lobby like voting, choosing a character and seats to the database. It also sends the data of newly created like the name, story chosen again to the database. This component also responsible sync all data between clients for each player action in lobby.

Access_Handler Component

Access_Handler takes the input from UI and validates that the credentials are correct and then returns the result to the UI. To validate these credentials it connects to the database component and verifies the credential by querying the users' table if the input exists. Then if the table replies with an empty set then that username and password combination doesn't exist. If validation was successful it will get the id of user.

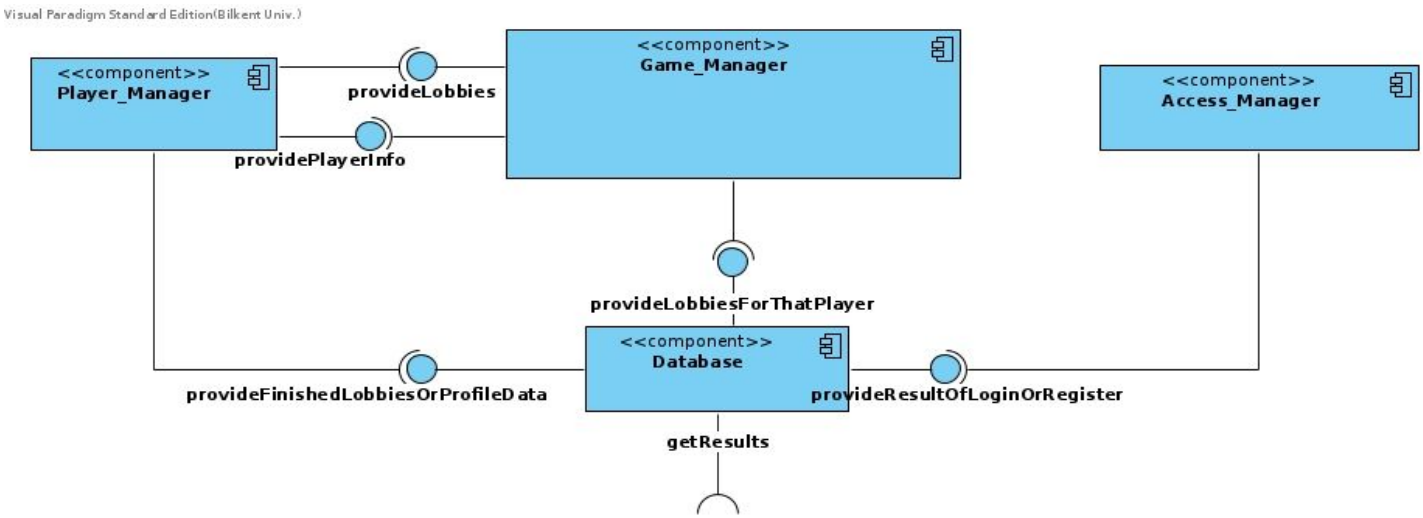


Figure 2.B : Component Diagram Layer 2-3 of "Your Story".

Database Component

Database component handles the connection between client and database securely. It will be able to change or add data to database while also provide the needed data for the usage of second layer. This is going to be achieved by sending queries to the database and receiving the result set from it. Second layer depends on the information retrieved from the database component and the UI layer depends on the data of the second layer. So this again demonstrates how we implemented the close architecture and its dependencies. Such component might change in the future to connect to a server through sockets, so even its naming might change to something more suitable at that time.

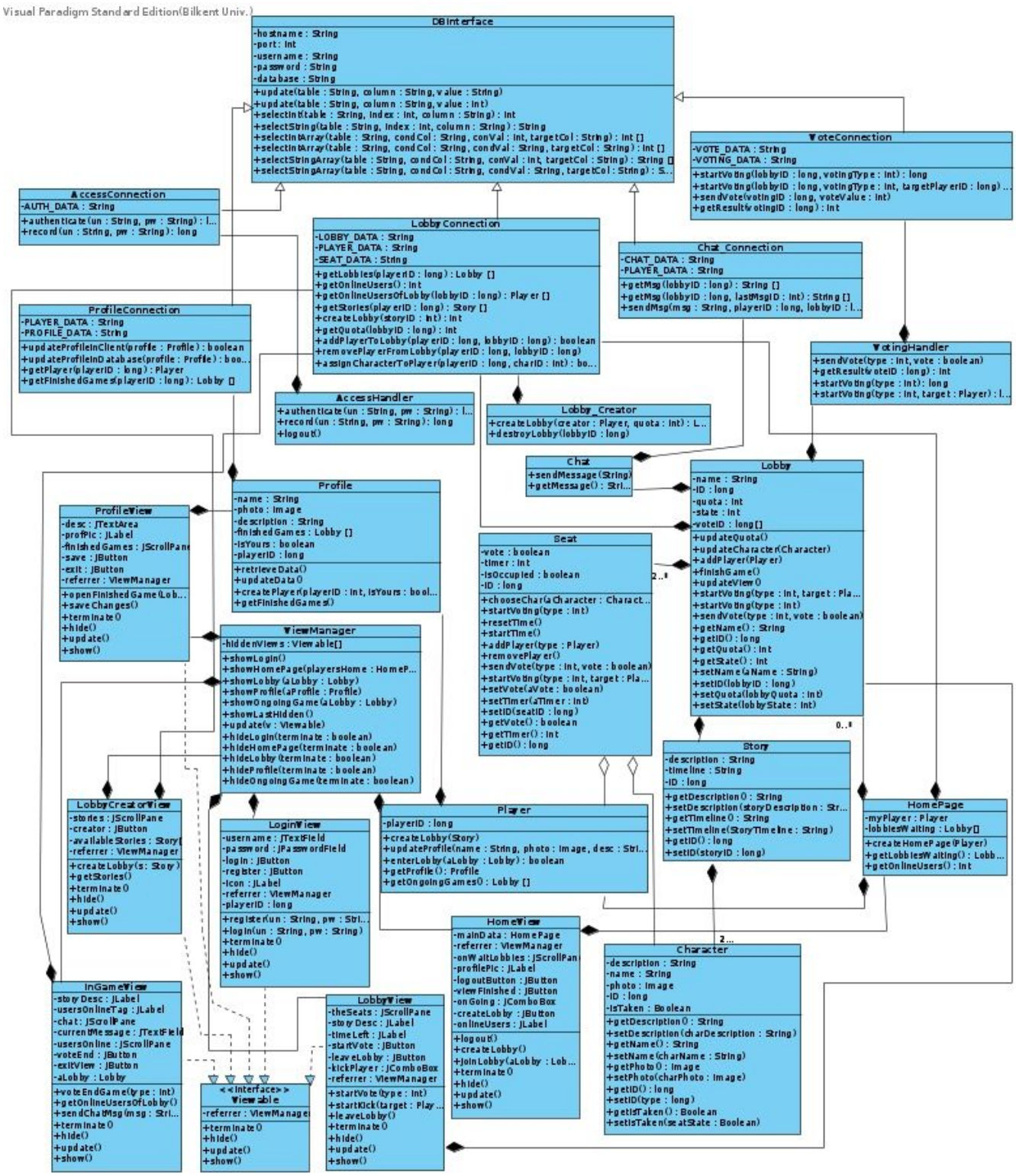
Low-level design

Object design trade-offs

In Your Story's three layered architectural design, having a server that is connected to our third party database and the application would decrease the required amount run time power and RAM of our game. Yet, due to the time constraint for coding and doubling our work by writing a reliable from scratch, our team decided on not having a server in Your Story's three layered architectural design. To solve the problem of updating the information in real time, we decided on a different approach.

Your story updates all of the information coming from the other users and database in a time interval. Once every 5 seconds, the "update" methods in our low-level design will automatically collect the new information and send it to the user interface. By this way, even though we increase the work done by the application, we do not need a complex server to handle the operations.

Final object design



Packages

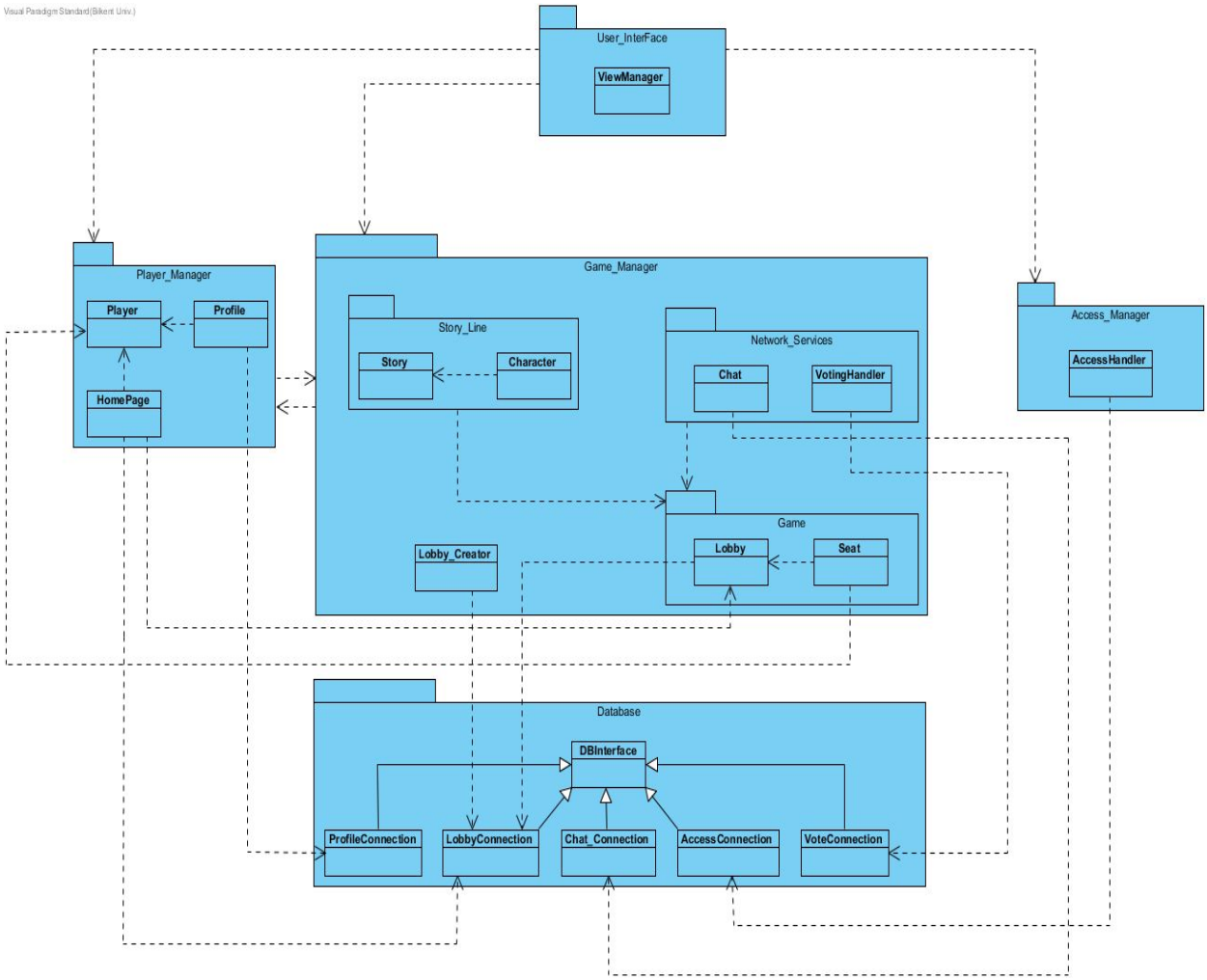


Figure 4.A : Package Diagram of Your Story

Layer 1: User_Interface

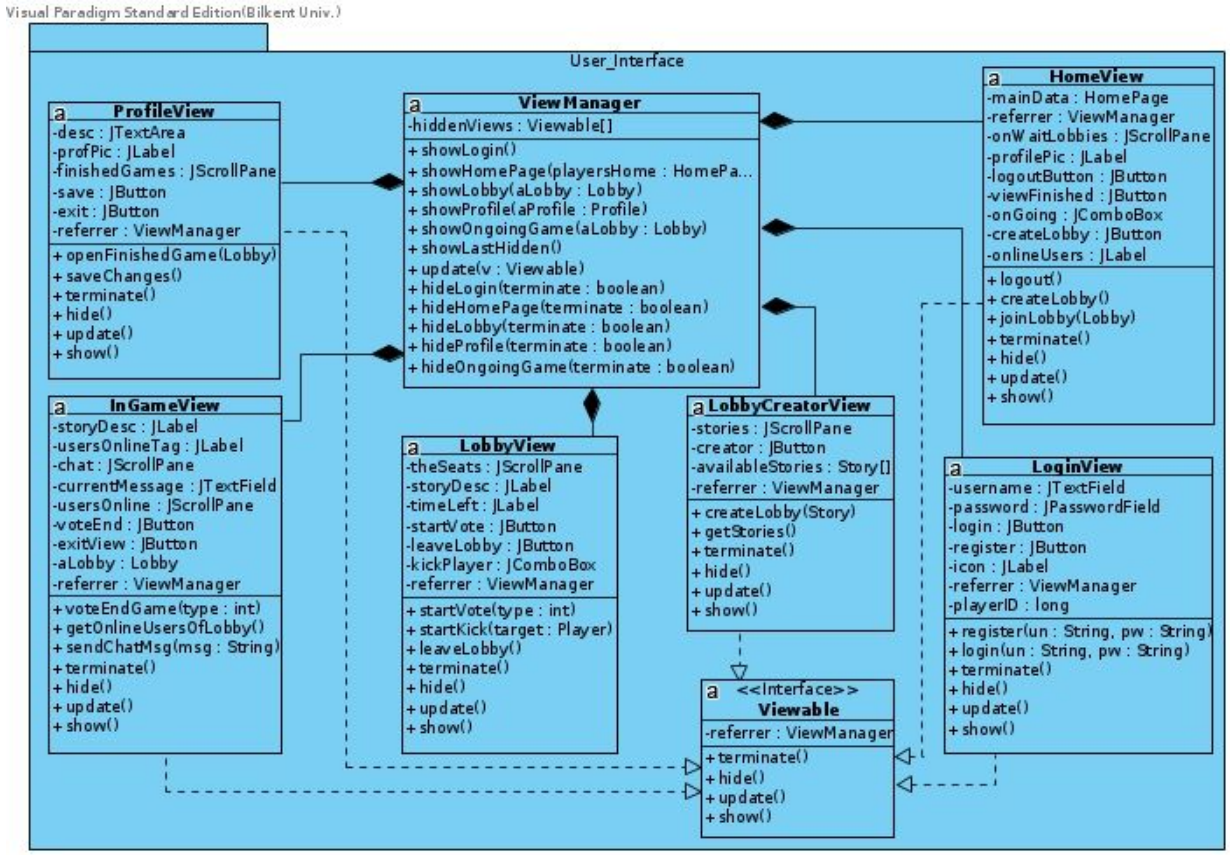


Figure 4.B : User_Interface diagram

The User_Interface package is designed in a good way to allow any change in UI without having to change much in the other classes. It is designed keeping in mind that the UI can totally change. The package will have a head we called ViewManager which will take care of displaying or hiding any view. Each of the views will also have a referrer pointer which will refer them back to the ViewManager and let them call the functions of ViewManager. The terminate, hide and update functions are same for all of them and they must have in order to function properly. So in any case if they were to change the same functions must be implemented. This will be constrained by an interface called Viewable that must be implemented by any UI component. This will in support to rapid development as well as increasing maintainability of the code in the future given that we can switch to JavaFX.

Layer 2: Player_Manager

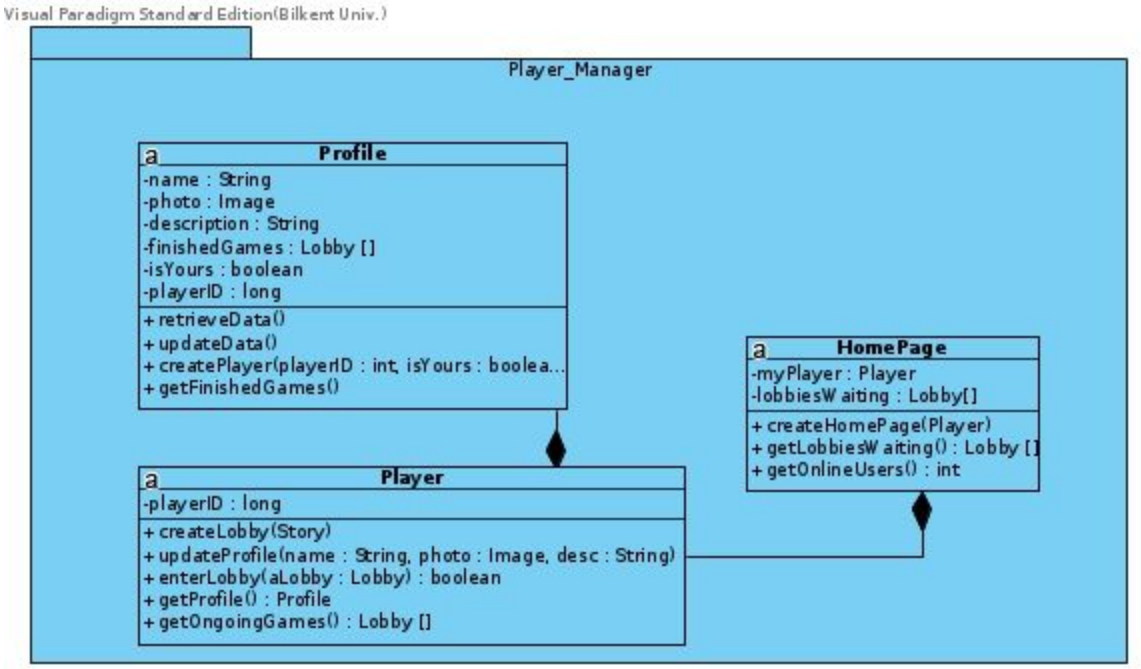


Figure 4.B : Player_Manager diagram

Player Manager component is responsible for managing the data of players and handling user actions outside of the lobbies (In that case seat class is responsible for the actions of users.) It creates lobbies and holds lobbies that associated with players, hold profiles and updates them while also creating the home page for players by getting the lobbies that will be shown in homepage.

Most of actions in Player Manager requires database connection to either insert or select data, therefore it's in a close relationship with Database Manager to fulfill its functions. Also since user actions outside of lobbies represented by Player class, most of methods in View Manager will include Player Component's functions.

Since Player Manager is part of the middle layer in the design, its main role likes a bridge between database and user interface while managing the data passes through two sides.

Layer 2: Game_Manager

Game_Manager Package includes the core classes of Your Story. Most of the use cases are dependent on this package which also has three subpackages. Online messaging, creating a lobby, joining a lobby and voting for a game to start or voting to kick a player cases are handled within this package.

Layer 2: Story_Line (Subpackage of Game_Manager)

The Story_Line package includes the various stories that are stored in the system's database. These stories are stored as String type, and whenever a lobby is created the lobby creator is allowed to choose a story, which will be taken from the database. Every story has it's own description, timeline and characters which are also stored in the database under the related story. Characters has their own name, ID, image and a description. When a player creates a lobby, or when another player joins a lobby, they are prompted to choose a character from the storyline. The options that are available to the player changes according to the "isTaken()" instance. If a character has already been taken, it will not be shown as an option for a new player in the lobby.

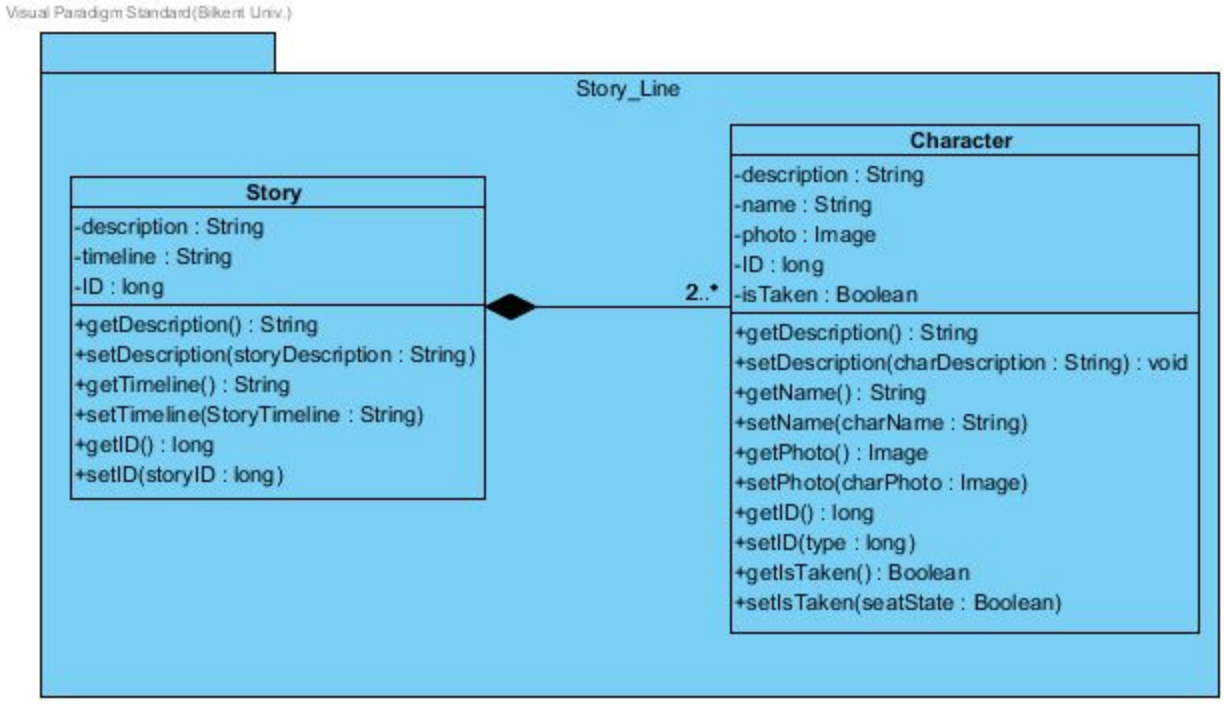


Figure 4.E : Package Diagram of Story_Line

Layer 2: Network_Services (Subpackage of Game_Manager)

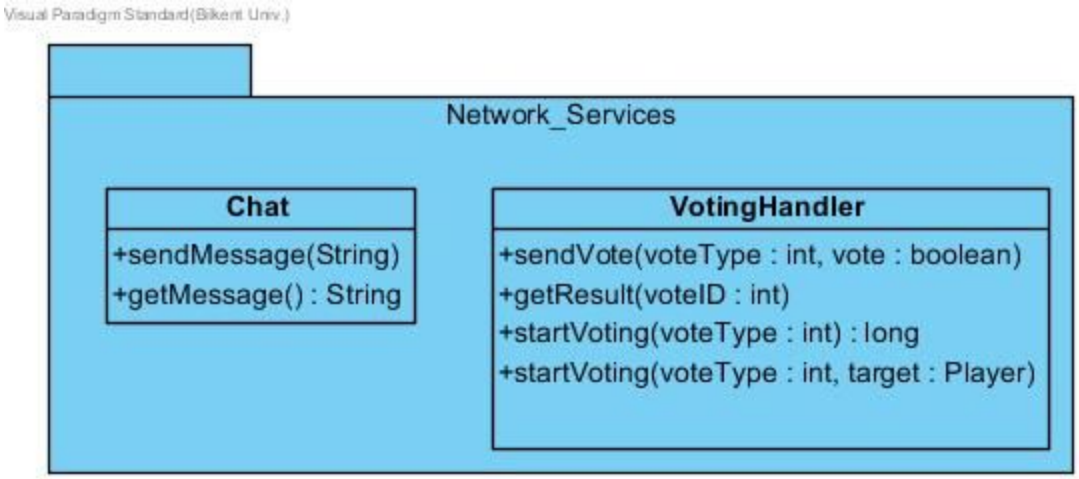


Figure 4.F: Game subpackage diagram

Network_Services Package handles the database-player interactions for messaging and sending votes. The main use case of Your Story, “Developing a story by messaging each other as a group” is done by Chat class. It has only two operations, “sendMessage()” and “getMessage()” . “SendMessage” sends the user’s message to the database, and “getMessage()” operation gets the updated messages from the database. When there is a new message coming from the updated database, it will be shown in the player’s lobby interface as a new message. We put Chat and VotingHandler class in a subpackage to create low coupling, since both of them are related with network part of our design.

Layer 2: Game (Subpackage of Game_Manager)

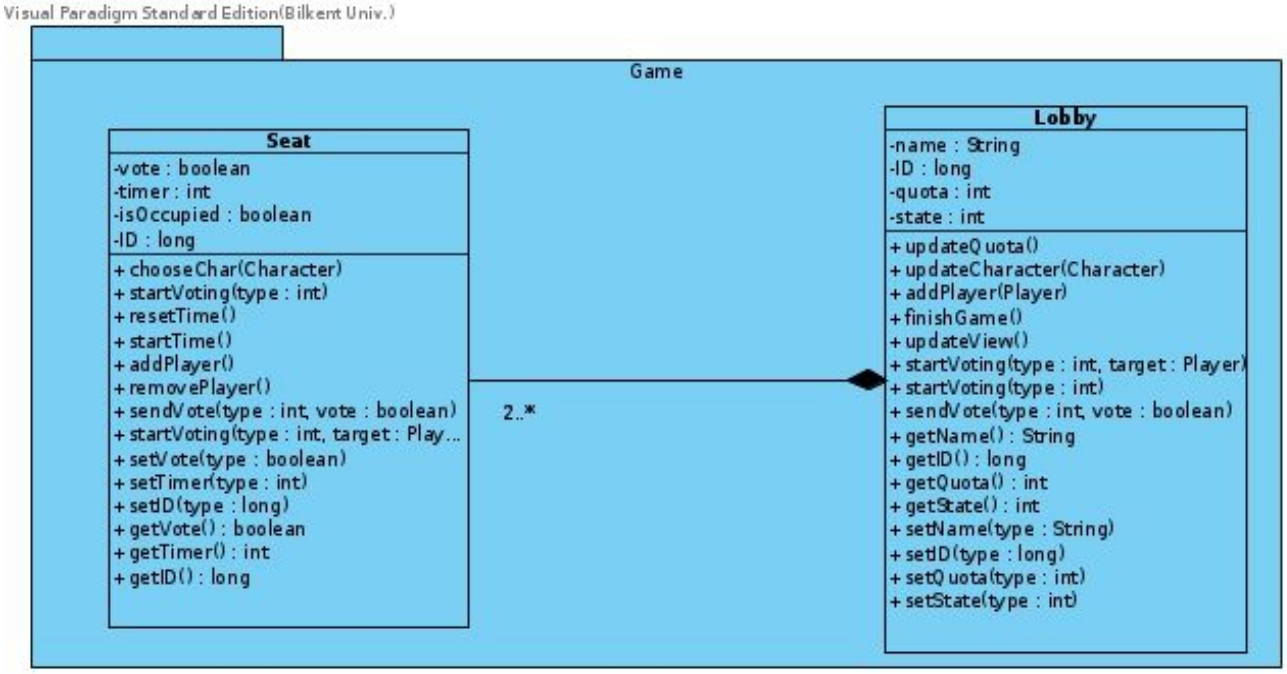


Figure 4.G: Game subpackage diagram

This layer is the one responsible for handling all the game features like the lobby and the seat and all the respective functionalities. Lobby is basically the heart of the game and the Seats is just to reduce the complexity of Lobby. This component is designed to be efficient and supporting the multiplayer nature of the whole game. Due to rapid development the package is limited to having the Lobby communicate to its own table in the database via the third layer. It will query the table every 5 seconds and the data will be updating the Lobby object and Seat object. This component is designed to be extensible and easily modifiable. The naming of the exposed methods of this component are as such so that in the future when implementing the server the same methods are going to be called and do the same thing as before. So when we build the Sockets the names still make sense. This involves having nothing like “queryTheNewChat()”.

Layer 2: Access_Manager

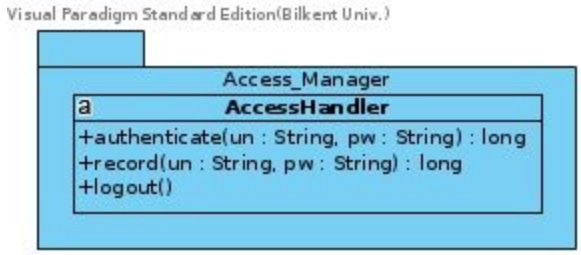


Figure 4.H: Game subpackage diagram

Access manager package is responsible for logging in and signing up. Logging in procedure checks whether the user is exists or not by typed username and password. It searches the username in the database and then if it exists, then checks the password. For the correct entries, access manager loads all data of the user. This package is designed with only 1 class that because there is no other class that can do similar functionality. It satisfies the low coupling with the other packages.

Signing up procedure searches the given username and password in the database. If either username or password already exists, then it is not allowed to sign up to the system and gives a warning to the user. Unique username and password is required. For a successful signup, required database adjustments is done.

Layer 3: Database

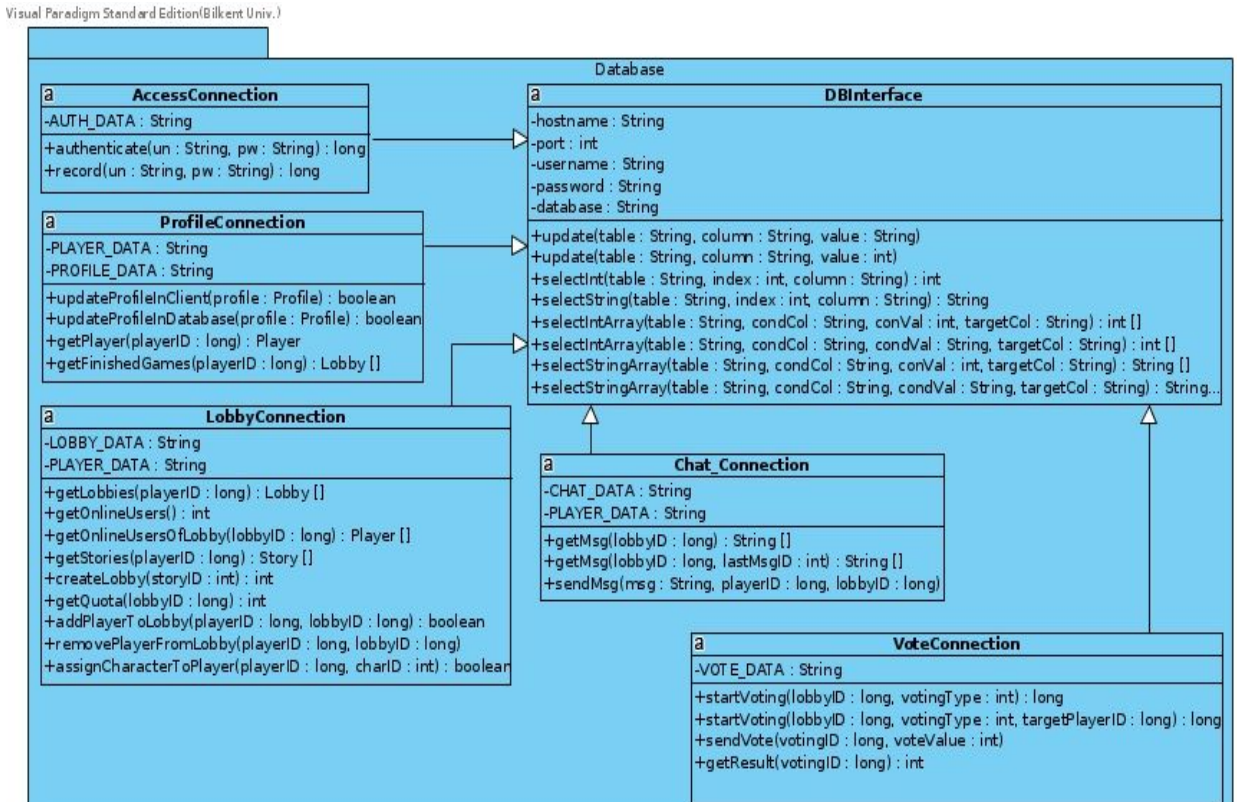


Figure 4.I: Game subpackage diagram

This package is designed to handle all the connection to the database in a secure and simple way. This layer is composed of six classes in a simple hierarchy. On the top is the DBInterface class. It is the only one that directly uses the JDBC library and has the username and password in it. This is part or handling security. By having the credentials in plain text only in a single place in the classes then it's easier to hide them. The other classes like ProfileConnection, VoteConnection, etc just extend the DBInterface and use so create a connection in the same database. The difference is the table they communicate with. The classes all are logically packed in one place since all relate to the parent class and have similar usage, so high coherence is what made us pack all of them together.

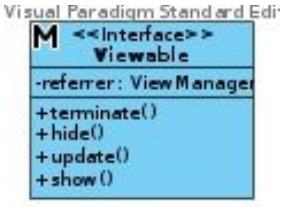
It uses SQL to modify database as the client requested. It includes classes that each one serves to all different instances of a specific class. Each class has different methods according to its client while parent class has some fundamental methods that can be used by all the subclasses.

Since the clients might make requests that blocks each other, Database Management System checks for validity of each request before executing it and making sure that if two requests change the same variable then this will be handled so that one statement is executed at a time.

Class Interfaces

User_Interface classes

Viewable <<Interface>>



private ViewManager *referrer* : This attribute is actually a pointer to the ViewManager class and lets the other Viewable classes call functions from the ViewManager. It simplifies the communication and limits it to the exposed methods of the classes. It makes it easy for all the UI to hide, show or destroy each other in case the classes are changed.
public void terminate(): This function is implemented by all the UI classes below and its job is to terminate the view class by calling the respective UI following it or just closing the whole program. Calling the following UI is done by having a pointer to ViewManager class which all the view will have and can call its functions. We decided to call it referrer like the ones used in PHP.

public void *hide*(): This method will not completely remove the instance of the view created before but it would just hide it so that another view is visible. It is mostly used when viewing the profile when we can completely disappear the home page and let the user read the profile and then when closing the profile user can go back to the home page we just hid via the referrer. This function is designed to increase efficiency Java lacks with the tradeoff of using a few Mb of RAM.
public void *update*(): This method is called automatically within each Viewable object. It gets new data from the database every time interval the developer of the class decides it's good. When it gets new data then the view will update and the user will see the change. It is not a

real-time implementation but close to it. When changed to socket connection then the update will be called whenever the socket gets some data which can be .1 second, 5 minutes, an hour or whatever. It will be way more efficient than querying the database and way more secure. But this method is designed for being extended in the future so it doesn't have a specific name like queryDatabase().

public void *show*(): This method will show the lastly hidden view or any view. In case of LoginView, when user logs in it will show the homepage. In case of ProfileView it will show the lastly hidden Viewable.

ViewManager

private Viewable[] *hiddenViews*: This array will be working like a stack the bottom is at 0 and stuff will be added at the top. It will hold all the Viewable object that are going to be hiden and when the showLastHidden() is called the "stack" will pop the last Viewable element and show call its show() function.

public void *showLogin*(): This function will be the first one to be called when the program starts. This do what its name says; it will show the login UI. It is thought to be called at the constructor of the ViewManager class.

public void *showHomePage(playersHome : HomePage):* This method will be called when the player first logs in successfully. It will take a HomePage object which contains the Player data and the Lobby data and the player's respective profile and everything needed for a homepage described by the analysis report.

public void *showLobby* (Lobby *aLobby*): This method will be called when the user decides they want to connect to a Lobby which is in the waiting state. It will show the Lobby with all the players inside and all other needed data like the story description, characters etc.

public void *showProfile* (Profile *aProfile*): This function is used to display any user's profile. It will display "save" button to whether the profile is of the logged in user or some other user's accordingly.
public void *showOngoingGame* (Lobby *aLobby*): This method will display the UI of a game which is currently going on. It will have the chat, users online and everything of that UI.
public void *showLastHidden()*: This is the method used to display the last one of the UI which were hidden from the used and in the stack. It will call their show() method and remove them from the stack.
public void *update*(Viewable *v*): This method is used to call the update function of any Viewable object (UI) asynchronously in case it's needed. Like in the case of updating the profile data, etc.
public void hideLogin (boolean terminate),
public void hideLobby (boolean terminate),
public void hideHomePage (boolean terminate),
public void hideProfile (boolean terminate),
public void hideOngoingGame (boolean terminate): All these methods do what their names say, they also have a boolean parameter we called terminate. If terminate flag is up then it will terminate completely the respective view, else it will just hide it and push it on top of the stack

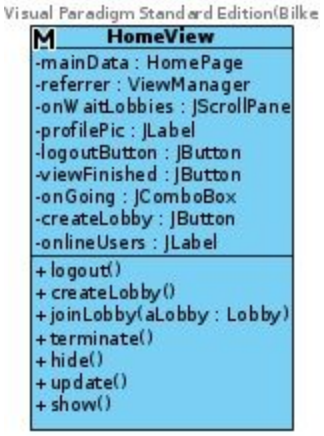
LoginView

private JTextField *username*: This is a text field of javax swing library that will hold the user's input for the username
private JPasswordField *password*: This is a text field of javax swing library that will hold the user's input for the password
private JButton *login*: This is a button that user can click to login

private JButton *register*: This is a button that user can click to register. As in the analysis report login and register will be in the same view
private JLabel *icon*: This is a label that will hold a static image of the icon of “Your Story”
private ViewManager *referrer*: This is a pointer to the main class and allows LoginView to call its functions.
private long *playerID*: This is a field that stores the ID of the person that is logged in the program. This facilitates gathering all the information like profile page etc after the player logs in.

public void *register*(String *un,* String *pw*): This functions will be called when user presses the register. It adds the credentials to the database and automatically logs in the user to be efficient.
public void *login*(String *un,* String *pw*): This functions will be called when user presses the login. It checks the credentials with the database and logs in the user.

HomeView



private HomePage *mainData*: This object hold all the lobbies and all the information about the logged in player. Basically all the data that the home page displays.
private JScrollPane *onWaitLobbies*: This object holds the lobbies that are currently waiting for players.
private JButton *viewFinished*: This button is to view the recently globally finished games.
private JComboBox *onGoing*: This is a dropdown list that gives all the games the user is currently participating at.
public void *logout()*: This function is used to log the player out. It will destroy all the instances where the player exists and reset the game to the initial state.

public void *createLobby()*: This function is called when the player clicks the create button and will open the lobby creation dialogue.
public void *joinLobby(*Lobby *aLobby)*: This function is called when the player clicks on a lobby. It will make the player join a lobby and will call the respective window whether the lobby is in game or waiting.

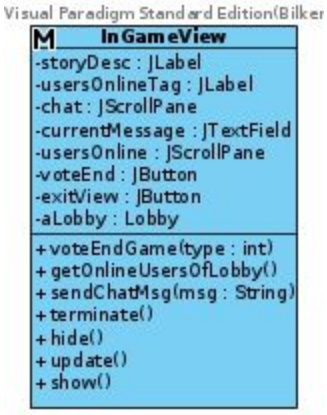
LobbyView

private JScrollPane *theSeats*: This will be a scrollable view that shows the seats that are available or taken for a current lobby in waiting state.
private JLabel *storyDesc*: This is just a label that will make sure the player gets to see the description of the story he/she is about to commit to.
private JLabel *timeLeft*: This is for a newcomer to see how many seconds he has before being locked to the game. This is to make sure the ones who come want to stay and not get bored. When time runs out the player has to stay in the lobby
private JButton *startVote*: This button is used to start a vote for starting the game or not.
private JButton *leaveLobby*: This button allows a player to leave the lobby if they find it boring if the time hasn't run out yet. Else the user has to stay and the button won't be shown
private JComboBox *kickPlayer*: This is a dropdown menu with all the players who have already committed to the game. It is used to allow the user pick a player they want to kick.

public void *startVote(*int *type)*: This method is called when a player starts the voting to start the game.
public void *startKick(*Player *target)*: This method is called when a player starts the voting to kick another player.

public void *leaveLobby()*: This method is used by the player when he clicks the button to leave. It will remove him/her from the lobby, empty that seat and unlock the character they chose

InGameView



private JLabel *storyDesc*: This is just a label that will make sure the player gets to see the description of the story he/she is playing in case he forgets, this happens when player has joined multiple games and takes part in all of them.
private JLabel *usersOnlineTag*: This is just a static label that indicates what the view below it is about. This one will write something like “Online users”.
private JScrollPane *chat*: This is a scrollable view that keeps all the messages players of that lobby have sent so far.
private JTextField *currentMessage*: This is a text field that lets the player write a message to send it to other people
private JScrollPane *usersOnline*: This is a scroll pane used to inform the player of who are actually online at the moment so that he knows what to message.
private JScrollPane *voteEnd*: This is a button that allows user to start a vote to end the game.
private JScrollPane *exitView*: This is a button that will allow the user to exit the lobby and return to the main page.
private Lobby *aLobby*: This will keep the data of the lobby so that the view knows what lobby it is about to show.

public void *voteEndGame(*int *type)*: This method start a vote to end the game. It is designed such that if an error occurs when passing a type it will not start the vote. Just for more security

public void *getOnlineUsersOfLobby()*: This method will be called to get the online users of that lobby. It is called every 5 seconds to assure efficiency in resources.
public void *sendChatMsg(*String *msg)*: This method sends the message the player types in textbox to the database so that others can synchronize to that.

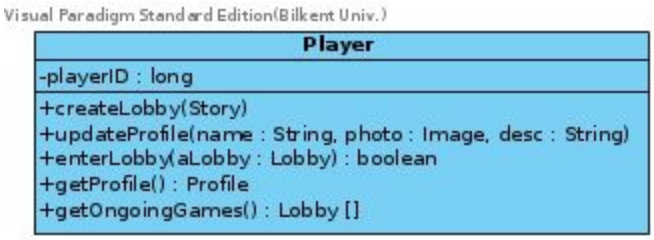
LobbyCreatorView

private JScrollPane *stories*: This view will show to the player who wants to create a lobby all the possible stories in our database.
private JButton *creator*: This is a button that will activate once the user has selected the story they want to have in their lobby. It will create a new lobby instance and add that entry to the table.
private Story[] *availableStories*: This array will hold the stories that it will display to the player.

public void *createLobby* (Story *s*): This will instert a new entry lobby in the database, create a Lobby instance, hide the current view and show the newly created lobby's view.
public void *getStories*(): This method will get the stories from the database and store that in memory to show it to the player.

Player_Manager Classes

Player



private long player*ID*: Unique id number of player. This will be the primary index in the database, therefore all actions that related to database will be using this unique number that created by the database.

public Lobby *createLobby* (Story *story*) : Creates a lobby with given story. It will get the data given by user interface and create the lobby in the database via database connection classes.
public Boolean *enterLobby* (Lobby *aLobby*) : Adds user into the given lobby if it has enough quota, it returns a boolean that shows if action was successful or not. This boolean will be returned by database classes that first will check if there's still an empty seat, or is the game on the state that still let new players to join.
public Profile *getProfile*(): Returns profile of the user. This profile object will include all information about that user, name, photo and the descritipion.
public void *updateProfile*(String *name*, Image *photo*, String *description*): Updates the user's profile with values. New values will be passed to the database and will be stored there.
public Lobby[] *getOngoingGames*(): Returns the ongoing games of player via database classes.

Profile

private String *name*: Name of the player that profile belongs to.

private Image *photo*: Photo of profile.

private String *description*: Description that player uploaded to profile.

private Lobby[] *finishedGames*: All finished games of player.

private boolean *isYours*: Parameter that shows if profile belongs the user in client.

public void *retrieveData*(): Retrieves the data of profile. This will sync the variables of object with the data in the database, therefore before showing a profile to the users, this method will make it certain that the updated data will be showed to the user.

public void *updateData*(): Updates the data of profile page in database with current values. This is the opposite of the retrieveData method, so this one will sync the data in the database with the current values of variables in an Profile object.

public Player *createPlayer*(int *ID*, boolean *isYours*): Creates a player object with given values. This will be useful method when getting new player information comes from the database, by this method an new player instance will be created with given unique ID. The other information about the players will be taken with Profile class via ID variable. isYours parameter shows if new player object will be representative of current user of the client. If so, some actions will be available for the user, like updating profile data, joining a lobby.

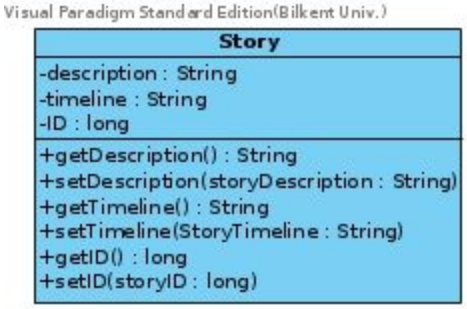
public Lobby[] *viewUnfinishedGames*(): Returns unfinished games of player that profile belongs to.

HomePage

private Player *myPlayer*: Player that home page will be created for. This object needs its isYours boolean to be set true.
private Lobby[] *lobbiesWaiting*: Array of lobbies that needs more players to start. This will be used by user interface classes to show active lobbies to show the games user can enter.
public void *createHomePage*(): Retrieves the data about lobbies from database. This is the method that sync the lobbiesWaiting array with the updated information in database.
public Lobby[] *getLobbiesWaiting*(): Returns the array of all the lobbies waiting to start a game.

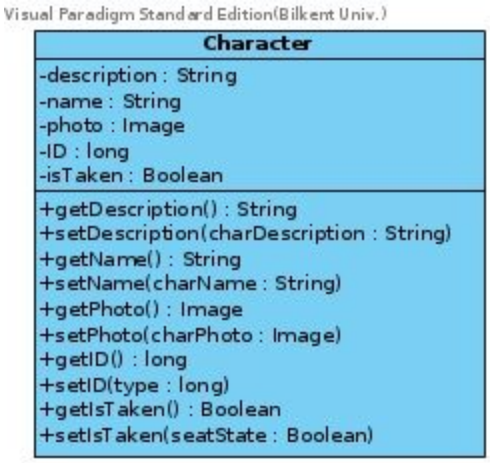
Game_Manager Classes

Story



private String *description*: A short explanation of a story. It includes a summary of what is happening around this timeline, who are the characters, what is the main issue and from which point the players can develop their stories.
private String *timeline*: Timeline includes a short description of the world, year and place.
private long *ID*: Specific identification number of the story, which will be used to assign a story to a defined lobby.

Character



private String *description*: A short explanation of the character. It contains the background of this specific character, which is also dependent on the story.

private String *name*: Name of the character.

private long *ID*: Specific identification number of the character, which will be used to assign to a seat in a lobby.

private Image *photo*: A .png illustration of this character's face.

private boolean *isTaken*: Indicates if this character is taken by a seat or not. A taken seat cannot be assigned to another seat.

Chat

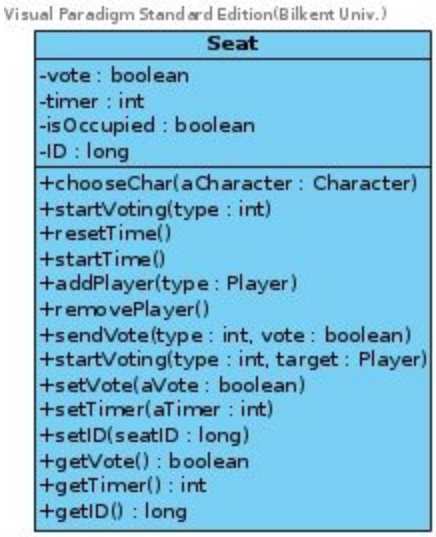
public void *sendMessage(String)*: Sends the message of a seat to the database as a string.

public String *getMessage*(): Gets new messages and updates the current number of messages in the lobby.

VotingHandler

public void *sendVote(*int *voteType; boolean vote):* Sends a vote to the database, with indicating it's type (voteType = 0 is "start the game" voting, voteType = 1 is "kick the player" voting, voteType = 2 is "finish the game" voting) and the vote as boolean.
public int *getResult(*int *voteID)*: Gets the result of voting from VoteConnection. Depending on the voteID, a behaviour is decided for this voting.
public long s*tartVoting(*int *voteType):* Sends the lobbyID and voteType to the VoteConnection This method is used to start the game that is currently inactive in this lobby.
public long s*tartVoting(*int *voteType,* Player *targetPlayer):* Sends the lobbyID , voteType and the player to be kicked to the VoteConnection.

Seat



private int *timer*: This integer will hold the timer of the seat, the time it has before it locks the player in.
private boolean *isOccupied*: This will hold the state of a seat, whether a player is linked to that seat or not.

private long *ID*: This will hold the id of the seat in the seat table.

public void *chooseChar(*Character *aChar):* This method is called to choose a character. It will connect a player entry to the player table with a character entry in the character table.

public void startVoting*(*int *typeofVote):* Sends a player's intention to start a voting to the lobby. Voting type depends on the value of typeofVote: For instance voteType = 0 means "start the game" voting, voteType = 1 means "kick the player" voting and voteType = 2 means "finish the game" voting.

public void *resetTime():* Resets the timer of character selection screen.

public void *startTime():* Starts the timer of character selection screen.

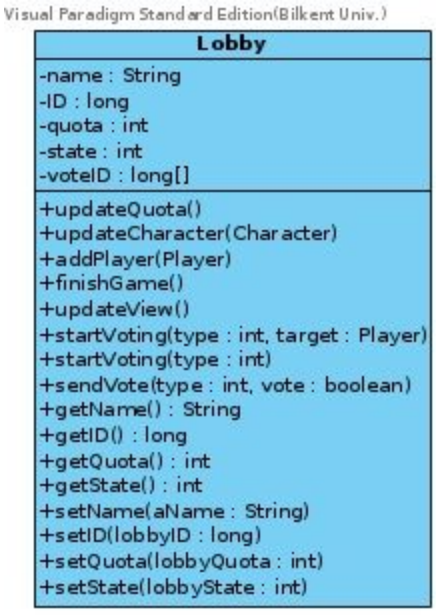
public void addPlayer*():* Seat assigns a Player to itself. **Post-constraint** : After assigning, isOccupied variable of the seat should be *true*.

public void *removePlayer():* Removes the assigned Player from itself. **Pre-constraint:** Seat should be occupied in order to remove the Player.

public void *sendVote(*int *voteType,* boolean *aVote):* This will send the vote of a player in the database and will store it there to be calculated later.

public void *startVoting(*int *voteType,* Player *targetPlayer):* This method is used to start the voting session. It will be called from a player to start the vote to kick another player.

Lobby



private String *name*: A short name of the lobby, which is decided by the creator of lobby.

private long *ID*: Specific ID of the Lobby, in order to be able to separate it from the other lobbies in the database.
private int *quota*: The remaining number of empty seats a lobby can have.
private int *state*: Indicates the state of lobby as an integer. The detailed design of states are explained in the state diagram of lobby.

public void *updateQuota*(): Updates the quota depending on whether a seat has been occupied or not.
public void *updateCharacter*(Character *aCharacter*): Updates the state of a character, as it is taken by a user or dropped by a user.
public void *addPlayer*(Player *aPlayer*): Assigns a player to an empty seat when a player request it. **Precondition**: It cannot be done if there is no empty seats, or the game has already started.
public void *finishGame*(): Finishes the game if the result of “Finish the Game” voting is true.
public void *updateView*(): Updates the lobby view every time a player has entered the lobby, or has existed the lobby.
public void *startVoting(*int *typeofVote,* Player *targetPlayer):* Sends the type of vote and player who is voted to be kicked to the VotingHandler.
public void *startVoting(*int *typeofVote):* Sends the type of vote to the VotingHandler. For now, voteType = 0 is for “start the game” voting, voteType = 1 is for “kick the player” voting and voteType = 2 is for “finish the game” voting)
public void *sendVote(*int *typeofVote,* boolean *vote):* Sends the type of voting and the resulted decision (vote) of a seat to the VotingHandler.

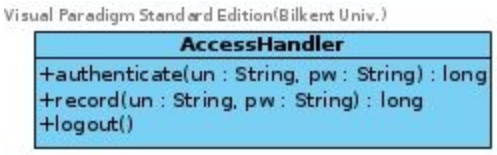
LobbyCreator

public Lobby *createLobby(*Player *creator,* String *name,* int *quota):* This method is used for creating the lobby. This takes the name, creator player and quota as parameters, and creates the lobby object inside itself and returns this lobby object.

public void *destroyLobby(*long *lobbyID):* This method destroys the selected lobby if the game is over by the result of voting of the players.

Access_Manager Classes

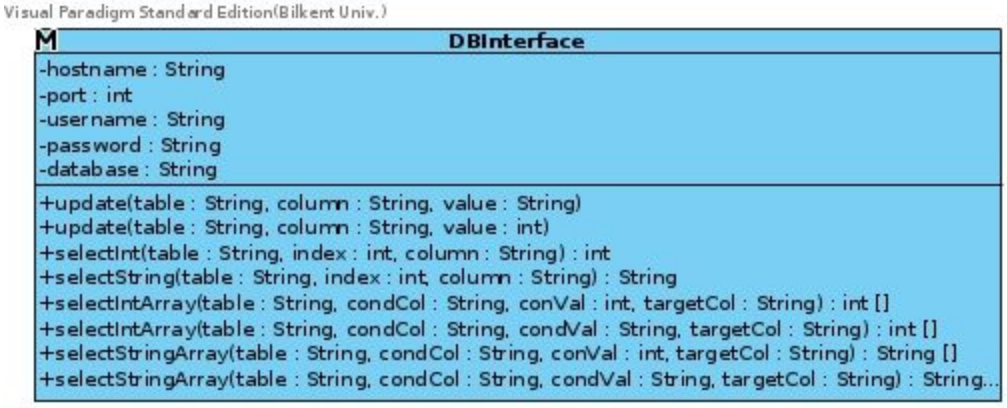
AccessHandler



public void *authenticate(*String *un,*String *pw):* This method is used to log the user in the game. It will be using the connection to the database. It will return the player's id taken from the database if the username and password inputs exist or proper, or -1 if they don't exist.
public void *logout():* Returns true if the user is able to disconnect successfully, false otherwise.
public long *record*(String *un*, String *pw*)*:* This will get the username and input and record that data in the database and will authenticate the newly created user to the game, but returning their id from the database.

Database Classes

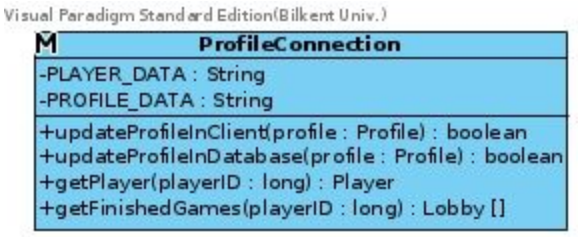
DBInterface



private String *hostname*: Hostname of the database that will be used during connection.

private int port: Port that will be used during connection.
private String *username*: Username that will be used during connection. This is a static username that's user of database, not related to user in the client.
private String *password*: Password of the database user. This with the username will be hidden as much as possible
private String *database*: The database name that will be used to store all the tables
public void *update*(String table, String column, String value): Updates a table with the given values.
public void *update*(String table, String column, int value): Updates a table with the given values.
public int *selectInt*(String table, int index, String column): Returns an integer from the database according to given address.
public String *selectString*(String table, int index, String column): Returns a string from the database according to given address.
public int[] *selectIntArray*(String table, String condCol, int condVal, String targetCol): Returns an array of integers that satisfies given condition from the database.
public int[] *selectIntArray*(String table, String condCol, String condVal, String targetCol): Returns an array of integers that satisfies given condition from the database.
public String[] *selectStringArray*(String table, String condCol, int condVal, String targetCol): Returns an array of strings that satisfies given condition from the database.
public String[] *selectStringArray*(String table, String conditionColumn, String conditionValue, String targetColumn): Returns an array of strings that satisfies given condition from the database.

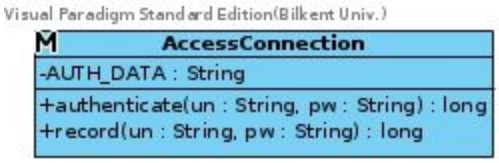
ProfileConnection



private String PLAYER_DATA: Variable that holds the name of player table in database.
private String PROFILE_DATA: Variable that holds the name of profile table in database.

public boolean *updateProfileInClient*(Profile *profile*) : Updates the profile object with the values in the database. Returns a boolean that shows if action was successful or not.
public boolean *updateProfileInDatabase(* Profile *profile*) : Updates the values in database with the values of given profile object.
public Player *getPlayer(* long *playerID*) : Gets the values of given index in the player table and returns a player object with that values.
public Lobby[] *getFinishedGames*(long *playerID*) : Gets the lobbies from database that still ongoing and given player is part of.

AccessConnection



private String AUTH_DATA: Variable that holds the name of player table in database.

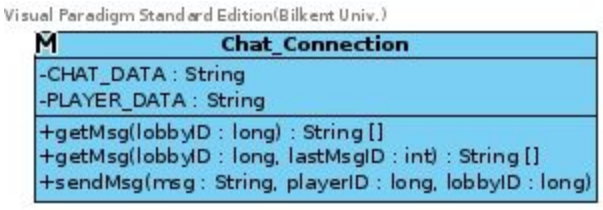
public long *authenticate*(String *un,* String *pw*) : Returns the unique id of user with given username and password. If no record is found it will return -1.
public long *record*(String *un,* String *pw*) : Adds a new row to the Player and Profile tables according to given values and returns the index number of the new row in player table.

LobbyConnection

private String LOBBY_DATA: This holds the name of the lobby table.
private String PLAYER_DATA: This holds the name of the player table.
private String SEAT_DATA: This holds the name of the seat table.

public Lobby[] *getLobbies*(long *playerID*) : Returns the lobbies of the player with given id.
public int *getOnlineUsers*() : Returns the number of currently online users in the whole game.
public Player[] *getOnlineUsersOfLobby*(long lobbyID) : Returns the array of online users of given lobby.
public Story[] *getStories*(long *playerID*) : Returns the array of stories in database, playerID is passed to not allow unauthenticated users get the stories of the database.
public int *createLobby*(int storyID) : Creates a lobby in database with given story id and returns the index value of new row.
public int *getQuota*(long lobbyID) : Returns the remaining quota of the given lobby.
public boolean *addPlayerToLobby*(long playerID, long lobbyID): Adds the given player to the lobby. Returns true if action was successful, false otherwise.
public void *removePlayerFromLobby*(long playerID, long lobbyID): Removes the given player from the lobby.
public boolean *assignCharacterToPlayer*(long playerID, int charID): Assigns character to the given player. Returns true if actions was successful, false otherwise.

ChatConnection

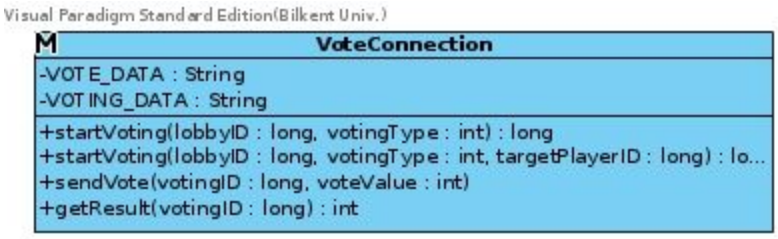


private String CHAT_DATA: This hold the name of the chat table
private String PLAYER_DATA: This holds the name of player table

public void *sendMsg*(String *msg,* long playerID, long lobbyID) : Adds a new message row with given lobby and player values.
public String[] *getMsg*(long lobbyID) : Get all the messages of the given lobby.

public String[] *getMsg*(long lobbyID, int lastMsgID) : Get the messages sent after given message id.

VoteConnection



private String VOTE_DATA: This holds the name of vote table's name

public long *startVoting*(long *lobbyID,* int votingType) : Creates a new voting row with given lobby and voting type and returns index of new row.
public long *startVoting*(long *lobbyID,* int votingType, long targetPlayerID) : Creates a new voting row with given lobby, voting type and target player id and returns index of new row.
public int *sendVote*(long *votingID,* int voteValue) : Adds a new row to the Vote table with given values.
public int *getResult*(long *votingID*): This gets the number of votes thrown for that particular voting and decides whether the vote passed or failed, it is returned as an int. 0 for not finished yet, 1 for passed and -1 for failed.