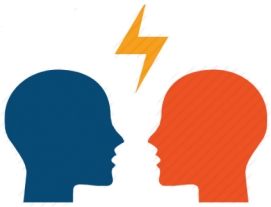
OBJECT-ORIENTED SOFTWARE ENGINEERING



DEBATE IT

DESIGN REPORT

GROUP 1B

21301294 **YAĞIZ GANİ**

21502938 **ÇAĞATAY SEL**

21200846 **AHMET SARIGÜNEY**

21501109 **YASİN BALCANCI**

SECTION 01

17.04.2018

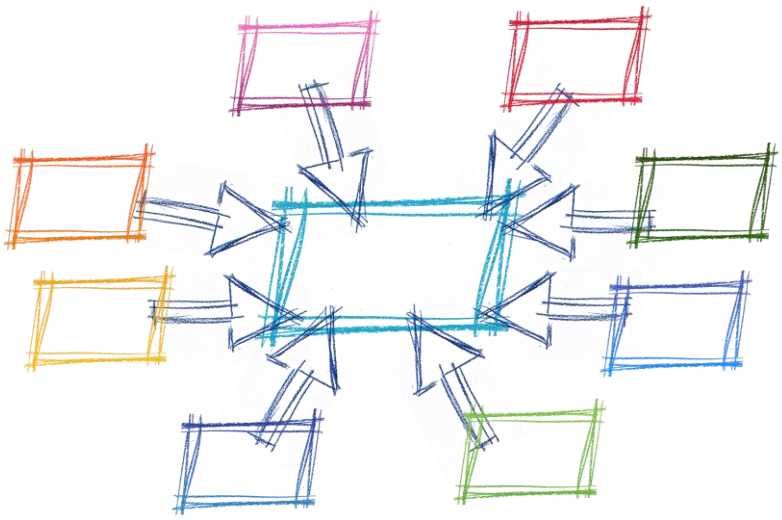
INTRODUCTION

This section explains why we come up with such a design and the tradeoffs that were made during the design.

- **Purpose of the system**

  The purpose of "Debate it" is to provide people an environment where they can practice their debate skills in an enjoyable environment.

  Our system also aims to provide its user with the replays of debates that other users participated in so that even the users that only want to read about the ideas of other people can utilize our application.



- **Design goals**

In order to achieve the purpose of our system we have set some design goals and tried to address them in our design.

⇨ Portability

In order to broaden the range of environments that our system can support, we have designed a generic server that is based on requests and responses.

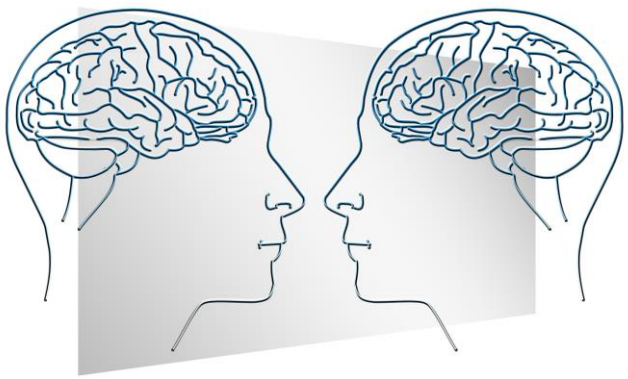
Although the system is supporting android devices at the moment, it can support ios devices, PCs in the future with the help of adapters as a result of our design.

⇨ High performance

In order to increase the server efficiency and meet our network constants which we determined in the analysis stage, we designed our server and client to minimize repetetive data transfer through network. Instead of sending whole objects, client and server sends only the necessary information that is enough to recreate objects in recipient.

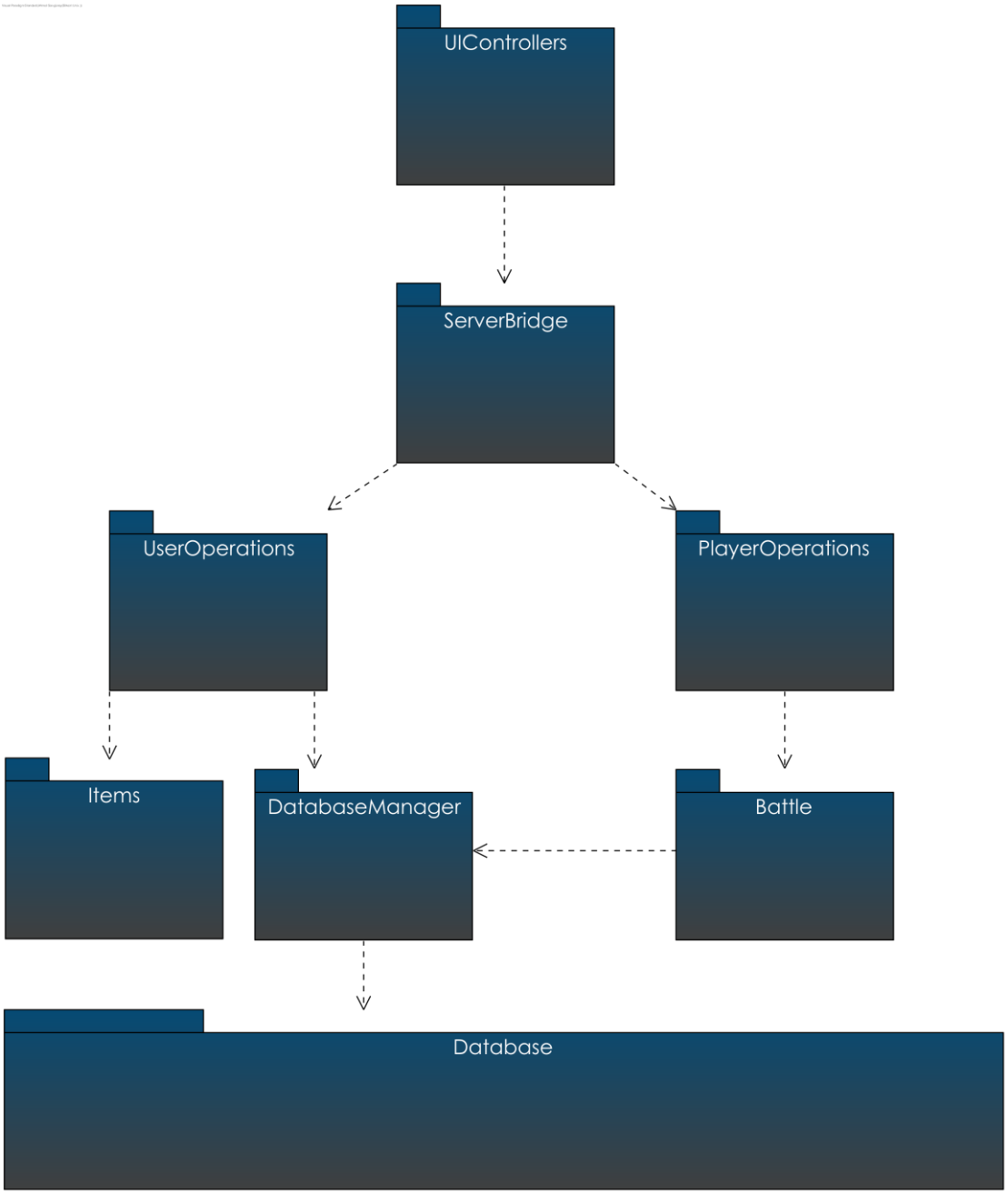
⇨ Ease of use

To achieve an easy to use system, we have designed our client interfaces in a simple way. We have also added a remember me option to login screen to make login process more easy for users.



SOFTWARE ARCHITECTURE

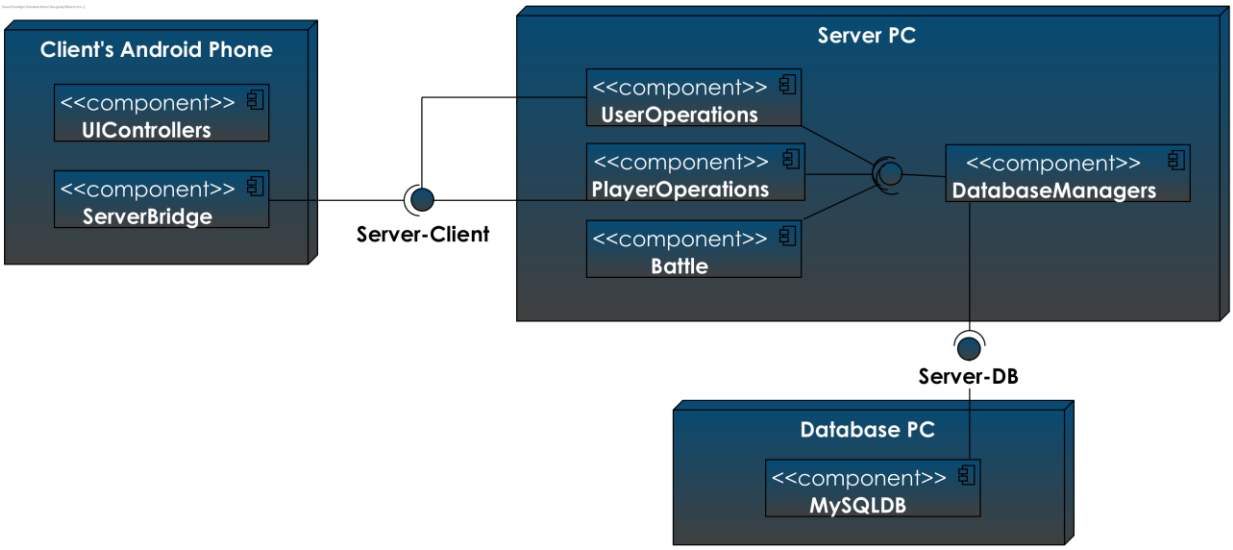
- **Subsystem decomposition**



We have followed three layered architecture while designing the subsystems. Our first level contains UI controllers which is responsible for user interactions with the UI. Second layer contains control subsystems. Our third layer is database layer. Second layer communicates with database via database managers.

We have also followed client-server architecture in order to meet multiplayer gameplay requirement. This architecture serves to our purpose because server machine is able to play the game and inform multiple client by utilizing multithreading.

- **Hardware/software mapping**



Our system has three different types of hardware. One of them is the server which will be deployed in a windows machine and will be implemented in JAVA language with the latest jdk. Another machine will have a mySQL database. The database and server connection will be done by using Java Database Connectivity(JDBC) application programming interface.

Third hardware in our system is the client devices which are mobile phones that run Android OS. Our system will support Android KitKat and later android devices. Server and client machines will be in the same local network communicate with "CilentBridge" and "ServerBridge" subsystems. These subsystems will utilize Socket Programming in order to establish the connection.

- **Persistent data management**

Data management is a crucial part for our system. Database machine will be responsible for storing and managing all kinds of datas. A mySQL server will be deployed on database machine. Users have to register to our system and sign in to use our system. Necessary information such as username and password will be stored in this machine. The database will also keep other user related informations such as inventory, past debates, record of all the past debates so that users can view or replay these past debates and keep voting.

Client machines will keep some of information regarding their users. These informations will be user options and their username and password so that users will not need to type their username and password every time they try to log in.

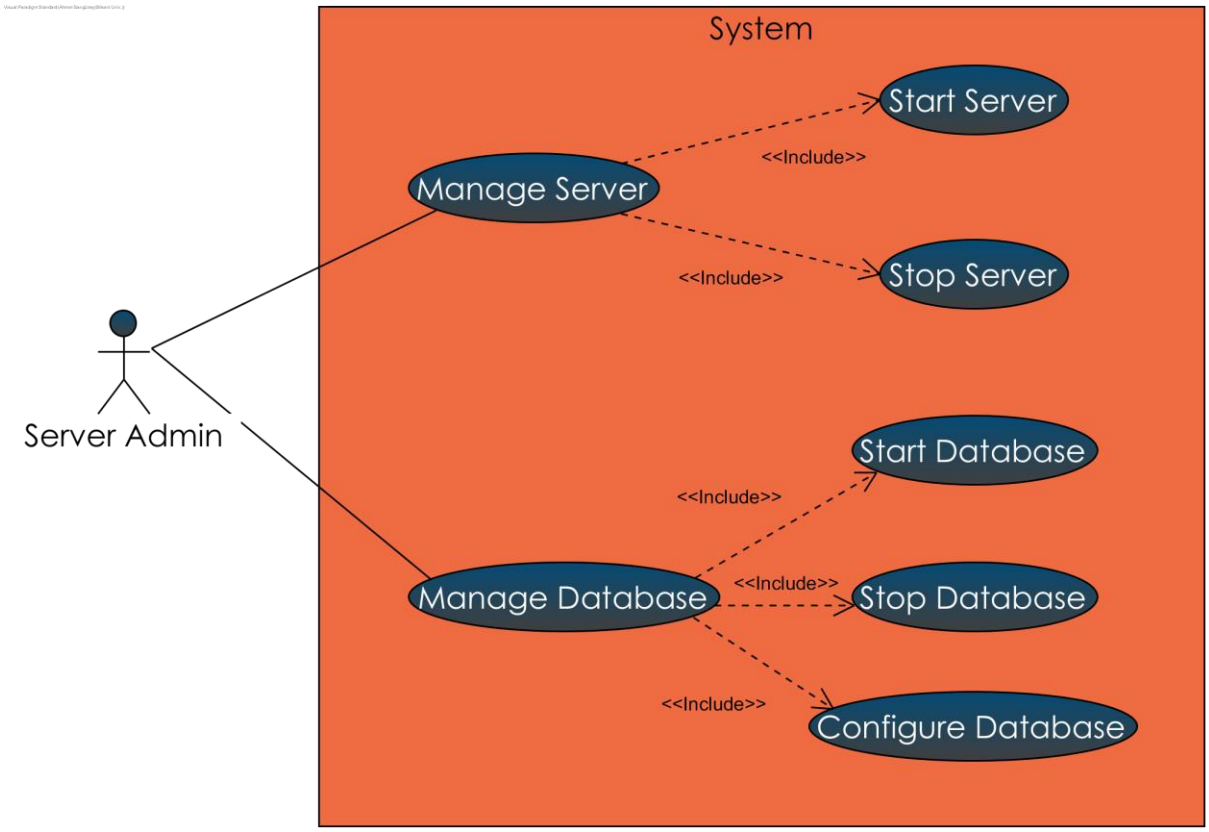
- **Access control and security**

We have two types of actors in our system and these are user and system administrator. These two actors will have different access to system which can be seen from the access matrix below.

	User	Inventory	Battle	Debate	Market	Settings	Server	Database
A user	login() logOut() viewProfile() editProfile()	view ()	join () play () quit()	view ()	buyItem()	change()	request()	
System Admin							start() terminate()	start() terminate()

- **Boundary conditions**

There are several boundary condition of our system which are managed by system administrators.



⇨ Starting

The user must have the application that was installed via provided .apk file and must be connected to the same LAN with the server.

⇨ Login

The user must have an ID-password matching, pre-registered account in order to login.

⇨ Logout

The user must be logged in order to log out. It can be logged out by tapping the "Log out" button in the options menu.

⇨ Shutdown

Users are able to quit the game by double-tapping "back" button in main menu.

⇨ Exception handling

Server will handle connection exceptions during gameplay by disconnecting the player who is having connectivity problems. Howeer, server and database does not designed to address problems caused by excessive amount of connections to these devices.

SUBSYSTEM SERVICES

- **Database management subsystem**

  - ⇨ "ItemManager" class

    Makes us able to get item objects according to their IDs from the database.

    - *Item[ ] getBuyableItems(int userID):* Takes user ID as parameter and returns the items that that User has not got in its inventory.
    - *Item[ ] getUserItems(int userID):* Takes user ID as parameter and returns the items that has been bought by that User.
    - *void storeUserItem(int userID, int itemID):* Takes user ID and item ID as parameters and records that item to the users inventory.

  - ⇨ "DebateManager" class

    Provides finished debates, which are stored in the database, as debate objects, according to their IDs.

    - *Debate getDebate(int debateID):* Takes debate ID as parameter and returns the unique Debate object that that ID belongs to.
    - *void storeDebate(Debate dbt):* Takes a Debate object as parameter and records it to the database.
    - *Debate[ ] getUserDebates(int[ ] debateIDs):* Takes an array of debate IDs as parameter and returns the array of Debate objects that the IDs correspond to.

⇨ "UserManager" class

Records the users that sign up to the database also provides the signing in. Checks whether a username is in use or not and provides user objects according to their usernames.

- *boolean signUp(String username, String password):* Takes username and password as parameters and tries to create a new User object in the database according to that. Returns if it managed it or not.
- *User signIn(String username, String password):* Takes username and password as parameters and checks whether they match with a User object from the database, if so returns it, otherwise returns null.
- *boolean checkUsername(String username):* Takes a username as parameter and checks whether it matches a User object from database or not and returns corresponding boolean.
- *User getUserDetails(String username):* Takes username as parameter and returns corresponding User object from the database.

o **Server models**

⇨ "UserHandler" class

This class is responsible for handling the clients request. It will start listening to client when the user opens the applications. It will use manager classes to retrieve data from the database and send the data retrieved data to user.

- **Constants**

  public final int REQUEST_INVENTORY = 0

  public final int REQUEST_PLAYED_DEBATES =1

  public final int REQUEST_PAST_DEBATES = 2

  public final int REQUEST_INVENTORY=3

  public final int REQUEST_BUYABLE_ITEMS = 4

- **Attributes**

  private ItemManager itemMan

  private DebateManager debateMan

  private UserManager userMan

- **Methods**

  public void answerUserRequest(int requestID): This method takes a request id as parameter and retrieves the necessary data using the manager. It then send the date to client.

⇨ "PlayerHandler" class

This class is responsible for listening to players during battles. It will also inform the player about the current status of the game.

- **Attributes**

  Player player

- **Methods**

  public String listenPlayer()

  public void updateBattleStatus(Debate db, int stage): This method will send the debate object and the current state of the battle to client so that client can update its ui.

  public void sendCurTimeToClient()

⇨ "Player" class

This class represents the user during a debate battle. Player objects contain the state of a user and the decisions they made during battles.

- **Constants**

  public final int SIDE_NEGATIVE = -1

  public final int SIDE_POSITIVE = 1

  public final int SIDE_SPECTATOR = 0

  public final int VOTE_YES = 2

  public final int VOTE_NO = 3

- **Attributes**

  private int playerID

  private String username

  private Avatar selectedAvatar

  private Title selectedTitle

  private Frame selectedFrame

  private int side:

  private ArrayList<Argument> arguments

  private int vote

  private volatile int consecutiveGamesWatched: Number of games that a player was not selected as debater will be recorded so that side selection algorithm could give priority people who haven't been able to debate for a while.

  private volatile int gamesPlayedInSession: Number of games that a player participated in as a debater will be recorded so that side selection algorithm could give priority to people who have played less.

- **Methods**

  Setter/Getter methods.

⇨ "BattleThread" class

This class is responsible for the dynamic structure of gameplay. Battles will be held in new thread in the server so that server can support more than one battle to be held at some time.

- **Attributes**

  private Debate currentDebate

  private PlayerHandler[] playerHandlers

  private int remaningTimeInMinutes

- **Methods**

  public void run() : This is the method that will run when the thread is executed.

  private boolean startInitialArgumentStage() : This method will initialize first stage of the game. If one of the debaters leave before the stage is complete, it will return false and the debate will be closed.

  private boolean startCounterArgumentStage(): This method will initialize second stage of the game. If one of the debaters leave before the stage is complete, it will return false and the debate will be closed.

  private boolean startAnswersStage():This method will initialize third stage of the game. If one of the debaters

  leave before the stage is complete, it will return false and the debate will be closed.

  private boolean startConclusionStage(): This method will initialize last stage of the game. If one of the debaters leave before the stage is complete, it will return false and the debate will be closed.

  private void generateNewDebate(: This method will generate a new debate when the current debate of battle is closed.

private void closeCurrentDebate(): This method will close the debate when conclusion stage is complete and when a debater leaves prior to finish.

private void designateSides(): This method will assign one player to positive side, one player to negative side and remaining two players to spectator side based on their side selections. This method will give priority to players who weren't selected as debater in the past debates so that every user will be able to play as debater.

⇨ "BattleTimer" class

This class is responsible for counting the timer and sending current battle time to clients.

- **Attributes**

  private volatile int timer

  private volatile boolean running

  private volatile boolean counting

  private volatile ArrayList<PlayerHandler> pH

  private BattleThread bT

- **Methods**

  public void run(): This method starts counting when startTimer method is called and count from timer to 0. When time reaches zero, it changes the current stage of battle to next stage.

  public void startTimer(int timer): Starts timer with given timer value in minutes.

  public void stopTimer(): Stops counting.

  public void terminate(): Terminates timer thread.

  public int getTimer(): Returns current time

⇨ "ServerBridge" class

It represents the client side of the data transferred between the server and client. This class sends request to the server via predefined unique request IDs and takes data in return. It can also forward data to user interface classes.

- *String[] request(int requestID, ArrayList<Serializable> requestParameters):* This method starts an asynctask which sends requestId and requestParameters to server.
- *void startListeningToServer():* This method starts an asyctask which establishes the communication with server. This asynctask listens server until client disconnects.
- *void disconnectFromServer():* This method closes connection to server.

o **Client models**

⇨ "User" class

- *private int userID:* Represents user ID
- *private string username:* Represents user name
- *private string password:* Represents password.
- *private int points:* Represents the points that the user has in his own account
- *private Avatar selectedAvatar:* This attribute will be used to keep the user's current avatar object.
- *private Title selectedTitle:* This attribute will be used to keep the user's current title object.
- *private Frame selectedFrame:* This attribute will be used to keep the user's current frame object.

- *private ArrayList<Item> inventory:* ArrayList of Item objects that represents the inventory of the user.
- *private ArrayList<Integer> pastDebateIDs:* This attribute is for holding the ID's of the past debates in the game.
- *private ArrayList<Integer> votedDebates:* It shows the vote history of the user.
- *public void buyItem(int itemID):* User can buy items by entering the ID of the item selected.
- *public void changeAvatar(int itemId):* User can change his/her avatar by entering the ID of the item selected.
- *public void changeTitle(int itemId):* User can change his/her title by entering the ID of the item selected.
- *public void changeFrame(int itemId):* User can change his/her Frame by entering the ID of the item selected.

⇨ " Item" class

Item class is the superclass of "Avatar", "Frame" and "Title" classes

- *public void changeFrame(int itemId):* User can change his/her Frame by entering the ID of the item selected.

⇨ "Expression" class

This class has an integer type of atribute as an "expressionID" that represents the each unique expression object.

⇨ "Avatar" class

This class has an integer type of atribute as an "avatarID" that represents the each unique avatar.

⇨ "Title" class

This class has an integer type of atribute as an "titleID" that represents the each unique title.

⇨ "Argument" class

In order to broaden the range of environments that our system can support, we have designed a generic server that is based on requests and responses.

- *private int sentTime:* The time argument sent.
- *private String argument:* Argument statement.
- *private int stage:* The stage argument sent in.

⇨ "Idea" class

In order to broaden the range of environments that our system can support, we have designed a generic server that is based on requests and responses.

- *private int ideaID:* Each idea object has a unique id.
- *private String statement:* Each idea has statements.
- *private int category:* Each idea has a category like philosophy, military, history or health, etc.

⇨ "Debate" class

- **Attributes**

private Idea idea: Idea of the debate.

private ArrayList<Player> players: Players in the debate.

private int debateID: Debate id.

private int debateLength: Length of the debate in terms of seconds.

private int yesVotes: Yes votes for the debate.

private int noVotes: No votes for the debate.

private int stage1Length: Length of the stage 1 in terms of seconds.

private int stage2Length: Length of the stage 2 in terms of seconds.

private int stage3Length: Length of the stage 3 in terms of seconds.

private int stage4Length: Length of the stage 4 in terms of seconds.

- **Methods**

public void addPlayer(Player player): This method adds a player to the debate.

public void removePlayer(Player player): This method removes a player from the debate.

public void addArgument(Argument argument): This method adds the argument to the debate.

- **Controllers**

⇨ "RegisterActivity" class

Displays one text box for the username, one for the password and one for the confirmation of the password, also two buttons one of which directs user to login page while the other one registers. Listens to the actions of register page.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "LoginActivity" class

Displays one text box for the username one for the password and two buttons one of which directs user to register page while the other one logins. It also listens the actions of login page.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "MainActivity" class

Displays the username of the logged user and user's past debates.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "InventoryActivity" class

Displays the items of the user and listens the actions of the inventory page.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "MarketActivity" class

Displays all items and listens the actions of the market page.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "MarketActivity" class

Displays all items and listens the actions of the market page.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "BrowseBattleFragment" class

Displays a screen until four available users exist on this page.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "BattleMenuActivity" class

Displays four players in the battle, the selected items, expression and sides of them. Listens to the arguments and display arguments from the past stages. Listens to the final voting.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "PastDebatesActivity" class

Displays all the finished debates by the time that each debate was votes, the category and the topic of each. It listens the specific debate selection and displays options for replaying the debate or seeing conclusion directly. It also listens the choice and displays the debate accordingly and lastly listen again the vote.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

⇨ "OptionsActivity" class

Displays "Sound on/off" and "Logout" options and listens the actions of this page.

- *boolean receiveAndUpdateUI(String data):* Takes a data string and updates the UI. Returns if it updated the UI or not.

- **Views**

This system contains xml files that will be attached to controller classes.

LOW-LEVEL DESIGN

- **Object design trade-offs**

  - ⇨ Security vs response time

    Because all the objects will be kept in database, not in the clients, there will be some extension at the response time exchange for security improvements.

  - ⇨ Cost vs security

    In order to reduce time required for development and cost, we have not implemented user input filtering which exposes the database to sql injections.

  - ⇨ Usability vs functionality

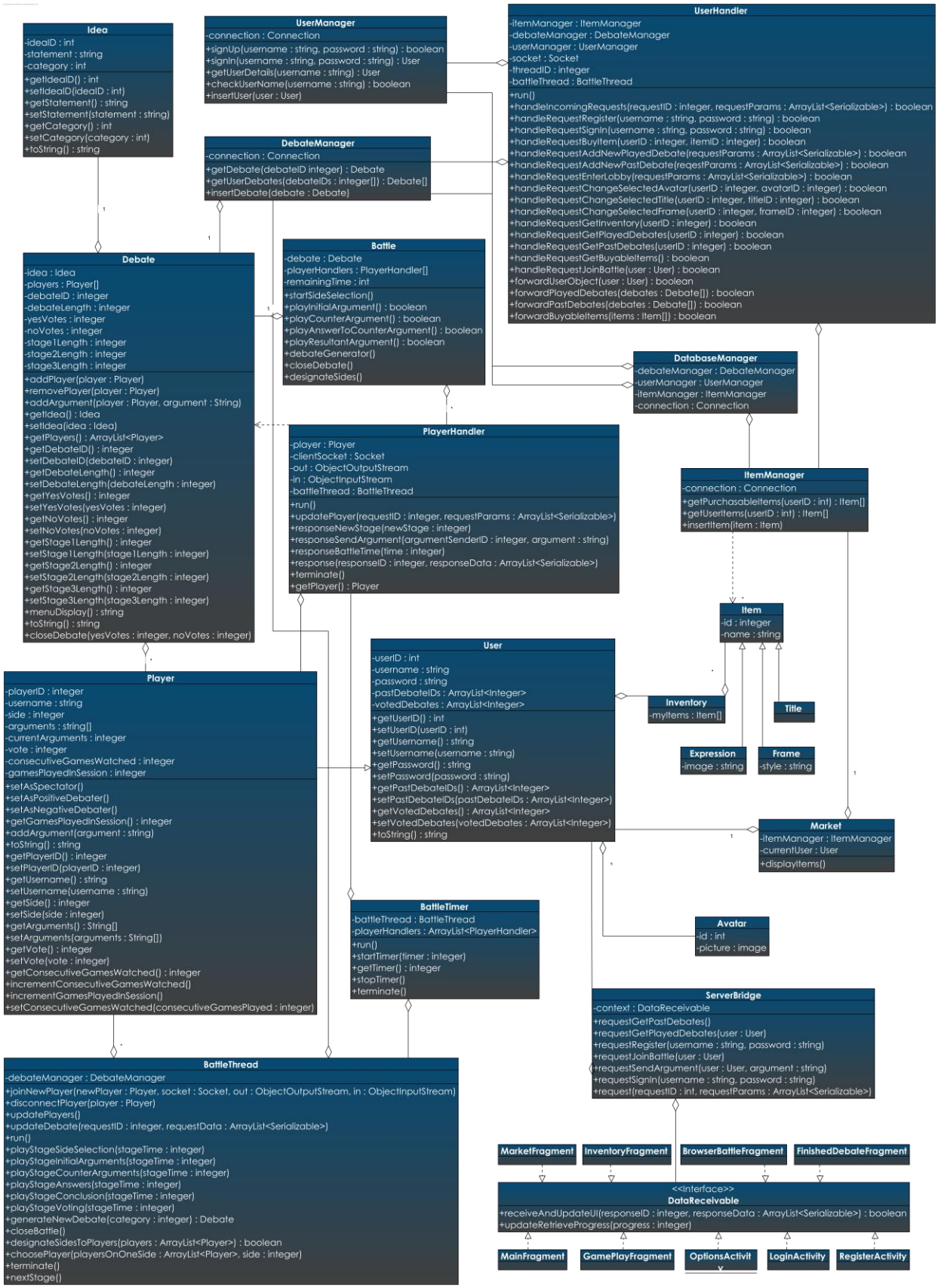
    While designing, we have decided to make expression that are send by players during the gameplay to be fixed for all users. With this design, users will not be able to buy new expressions from market. However, all the users will be able to use all expressions without buying anything.

  - ⇨ Low-cost vs robustness

    Our system does not implement any mechanism to deal with excessive amount of client connections. Therefore, the server machine and database is expected to go offline since they are ordinary laptop computers which are not designed to be server. But they serve as low-cost servers.

- **Final object design**

  Diagram below shows the class diagram that is planned to be implemented.


Idea
-ideaID : int
-statement : string
-category : int
+getIdeaID() : int
+setIdeaID(ideaID : int)
+getStatement() : string
+setStatement(statement : string)
+getCategory() : int
+setCategory(category : int)
+toString() : string
UserManager
-connection : Connection
+signUp(username : string, password : string) : boolean
+signIn(username : string, password : string) : User
+getUserDetails(username : string) : User
+checkUserName(username : string) : boolean
+insertUser(user : User)
UserHandler
-itemManager : ItemManager
-debateManager : DebateManager
-userManager : UserManager
-socket : Socket
-threadID : integer
-battleThread : BattleThread
+run()
+handleIncomingRequests(requestID : integer, requestParams : ArrayList<Serializable>) : boolean
+handleRequestRegister(username : string, password : string) : boolean
+handleRequestSignIn(username : string, password : string) : boolean
+handleRequestBuyItem(userID : integer, itemID : integer) : boolean
+handleRequestAddNewPlayedDebate(requestParams : ArrayList<Serializable>) : boolean
+handleRequestAddNewPastDebate(reqestParams : ArrayList<Serializable>) : boolean
+handleRequestEnterLobby(requestParams : ArrayList<Serializable>) : boolean
+handleRequestChangeSelectedAvatar(userID : integer, avatarID : integer) : boolean
+handleRequestChangeSelectedTitle(userID : integer, titleID : integer) : boolean
+handleRequestChangeSelectedFrame(userID : integer, frameID : integer) : boolean
+handleRequestGetInventory(userID : integer) : boolean
+handleRequestGetPlayedDebates(userID : integer) : boolean
+handleRequestGetPastDebates(userID : integer) : boolean
+handleRequestGetBuyableItems() : boolean
+handleRequestJoinBattle(user : User) : boolean
+forwardUserObject(user : User) : boolean
+forwardPlayedDebates(debates : Debate[]) : boolean
+forwardPastDebates(debates : Debate[]) : boolean
+forwardBuyableItems(items : Item[]) : boolean
DebateManager
-connection : Connection
+getDebate(debateID integer) : Debate
+getUserDebates(debateIDs : integer[]) : Debate[]
+insertDebate(debate : Debate)
Debate
-idea : Idea
-players : Player[]
-debateID : integer
-debateLength : integer
-yesVotes : integer
-noVotes : integer
-stage1Length : integer
-stage2Length : integer
-stage3Length : integer
+addPlayer(player : Player)
+removePlayer(player : Player)
+addArgument(player : Player, argument : String)
+getIdea() : Idea
+setIdea(idea : Idea)
+getPlayers() : ArrayList<Player>
+getDebateID() : integer
+setDebateID(debateID : integer)
+getDebateLength() : integer
+setDebateLength(debateLength : integer)
+getYesVotes() : integer
+setYesVotes(yesVotes : integer)
+getNoVotes() : integer
+setNoVotes(noVotes : integer)
+getStage1Length() : integer
+setStage1Length(stage1Length : integer)
+getStage2Length() : integer
+setStage2Length(stage2Length : integer)
+getStage3Length() : integer
+setStage3Length(stage3Length : integer)
+menuDisplay() : string
+toString() : string
+closeDebate(yesVotes : integer, noVotes : integer)
Battle
-debate : Debate
-playerHandlers : PlayerHandler[]
-remainingTime : int
+startSideSelection()
+playInitialArgument() : boolean
+playCounterArgument() : boolean
+playAnswerToCounterArgument() : boolean
+playResultantArgument() : boolean
+debateGenerator()
+closeDebate()
+designateSides()
DatabaseManager
-debateManager : DebateManager
-userManager : UserManager
-itemManager : ItemManager
-connection : Connection
PlayerHandler
-player : Player
-clientSocket : Socket
-out : ObjectOutputStream
-in : ObjectInputStream
-battleThread : BattleThread
+run()
+updatePlayer(requestID : integer, requestParams : ArrayList<Serializable>)
+responseNewStage(newStage : integer)
+responseSendArgument(argumentSenderID : integer, argument : string)
+responseBattleTime(time : integer)
+response(responseID : integer, responseData : ArrayList<Serializable>)
+terminate()
+getPlayer() : Player
ItemManager
-connection : Connection
+getPurchasableItems(userID : int) : Item[]
+getUserItems(userID : int) : Item[]
+insertItem(item : Item)
Item
-id : integer
-name : string
User
-userID : int
-username : string
-password : string
-pastDebateIDs : ArrayList<Integer>
-votedDebates : ArrayList<Integer>
+getUserID() : int
+setUserID(userID : int)
+getUsername() : string
+setUsername(username : string)
+getPassword() : string
+setPassword(password : string)
+getPastDebateIDs() : ArrayList<Integer>
+setPastDebateIDs(pastDebateIDs : ArrayList<Integer>)
+getVotedDebates() : ArrayList<Integer>
+setVotedDebates(votedDebates : ArrayList<Integer>)
+toString() : string
Inventory
-myItems : Item[]
Title
Expression
-image : string
Frame
-style : string
Player
-playerID : integer
-username : string
-side : integer
-arguments : string[]
-currentArguments : integer
-vote : integer
-consecutiveGamesWatched : integer
-gamesPlayedInSession : integer
+setAsSpectator()
+setAsPositiveDebater()
+setAsNegativeDebater()
+getGamesPlayedInSession() : integer
+addArgument(argument : string)
+toString() : string
+getPlayerID() : integer
+setPlayerID(playerID : integer)
+getUsername() : string
+setUsername(username : string)
+getSide() : integer
+setSide(side : integer)
+getArguments() : String[]
+setArguments(arguments : String[])
+getVote() : integer
+setVote(vote : integer)
+getConsecutiveGamesWatched() : integer
+incrementConsecutiveGamesWatched()
+incrementGamesPlayedInSession()
+setConsecutiveGamesWatched(consecutiveGamesPlayed : integer)
Market
-itemManager : ItemManager
-currentUser : User
+displayItems()
BattleTimer
-battleThread : BattleThread
-playerHandlers : ArrayList<PlayerHandler>
+run()
+startTimer(timer : integer)
+getTimer() : integer
+stopTimer()
+terminate()
Avatar
-id : int
-picture : image
ServerBridge
-context : DataReceivable
+requestGetPastDebates()
+requestGetPlayedDebates(user : User)
+requestRegister(username : string, password : string)
+requestJoinBattle(user : User)
+requestSendArgument(user : User, argument : string)
+requestSignIn(username : string, password : string)
+request(requestID : int, requestParams : ArrayList<Serializable>)
BattleThread
-debateManager : DebateManager
+joinNewPlayer(newPlayer : Player, socket : Socket, out : ObjectOutputStream, in : ObjectInputStream)
+disconnectPlayer(player : Player)
+updatePlayers()
+updateDebate(requestID : integer, requestData : ArrayList<Serializable>)
+run()
+playStageSideSelection(stageTime : integer)
+playStageInitialArguments(stageTime : integer)
+playStageCounterArguments(stageTime : integer)
+playStageAnswers(stageTime : integer)
+playStageConclusion(stageTime : integer)
+playStageVoting(stageTime : integer)
+generateNewDebate(category : integer) : Debate
+closeBattle()
+designateSidesToPlayers(players : ArrayList<Player>) : boolean
+choosePlayer(playersOnOneSide : ArrayList<Player>, side : integer)
+terminate()
+nextStage()
MarketFragment
InventoryFragment
BrowserBattleFragment
FinishedDebateFragment
<<Interface>>
DataReceivable
+receiveAndUpdateUI(responseID : integer, responseData : ArrayList<Serializable>) : boolean
+updateRetrieveProgress(progress : integer)
MainFragment
GamePlayFragment
OptionsActivit
y
LoginActivity
RegisterActivity

- **Packages**

  Our system has four packages that will be implemented by developers. First of them is the "SharedModels" package which includes classes of entity objects that are transferred between server and client. Both server machine and client devices make use of this package. Second one is the "Server" package. It is contains classes that handle multiple user connections, battle management. Third one is "DatabaseManagement" package which includes classes that communicates with database and insert or remove items from database. Last package is "Client" in which UI controllers are located.

  Our system also uses the application programming interface of Java database connectivity(JDBC).

- **Design patterns**

  While designing our system, we decided to follow three layered architectural style and server-client architecture. We choose three layered architecture because it serves our purposes by separating UI controllers and other control subsystems.

  In our design, we followed two design patterns in order ease development process. First of them is adapter pattern. This pattern is used to adpat the JDBC application programming interface to our system. The classes in "DatabaseManager" serve as adapters of JDBC api to our system in which we deal with debates, users and items.

  Second pattern that we have followed is façade pattern. We used this pattern to provide an interface for programmers to access to database. "DatabaseManager" class serves as the façade class for the "ItemManager", "DebateManager" and "UserManager" classes.

  "DatabaseManager" puts together all of the methods of this class and makes necessary connection to database.

CONCLUSION

- **Design patterns**

A "replay" mode will be added to the application. All users will be able to see the record of the past games. They will be able to see the players' strategies and actions to win the debate during gameplay.

- **Glossary**

Idea: A controversial statement which will divide users into two sides.

Avatar: A small image that will be shown in the debate screen with the nickname of user.

Frame: A border that will be around user's nickname.

Title: It will shown before user's nickname in his profile and debate screen.

Expression: A certain expression such as "Wow" or "gg" which users will be able to use during debate battles.

- **References**

[1]"The Premier Online Debate Website | Debate.org", Debate.org, 2018. [Online]. Available: http://www.debate.org/. [Accessed: 17- Feb- 2018].

[2]B. Bruegge and A. Dutoit, *Object-oriented software engineering*. Harlow, Essex: Pearson, 2014.