Assignment 2

Team number: 35
Team members

Name	Student Nr.	Email
Elif Kalkan	2691928	e.kalkan@student.vu.nl
Jay Doerga	2696009	j.a.doerga@student.vu.nl
Jazzley Louisville	2685898	j.j.j.louisville@student.vu.nl
Zefan Morsen	2691045	z.b.morsen@student.vu.nl

Implemented feature

ID	Short name	Description
F1	Start game	Tasks can be tagged via freely-defined labels called tags
F2	Pause/resume	The master should be able to pause the current game played and resume it at any point.
F4	Sequence	The master should be able to create a list of games to play in an order of his choosing.

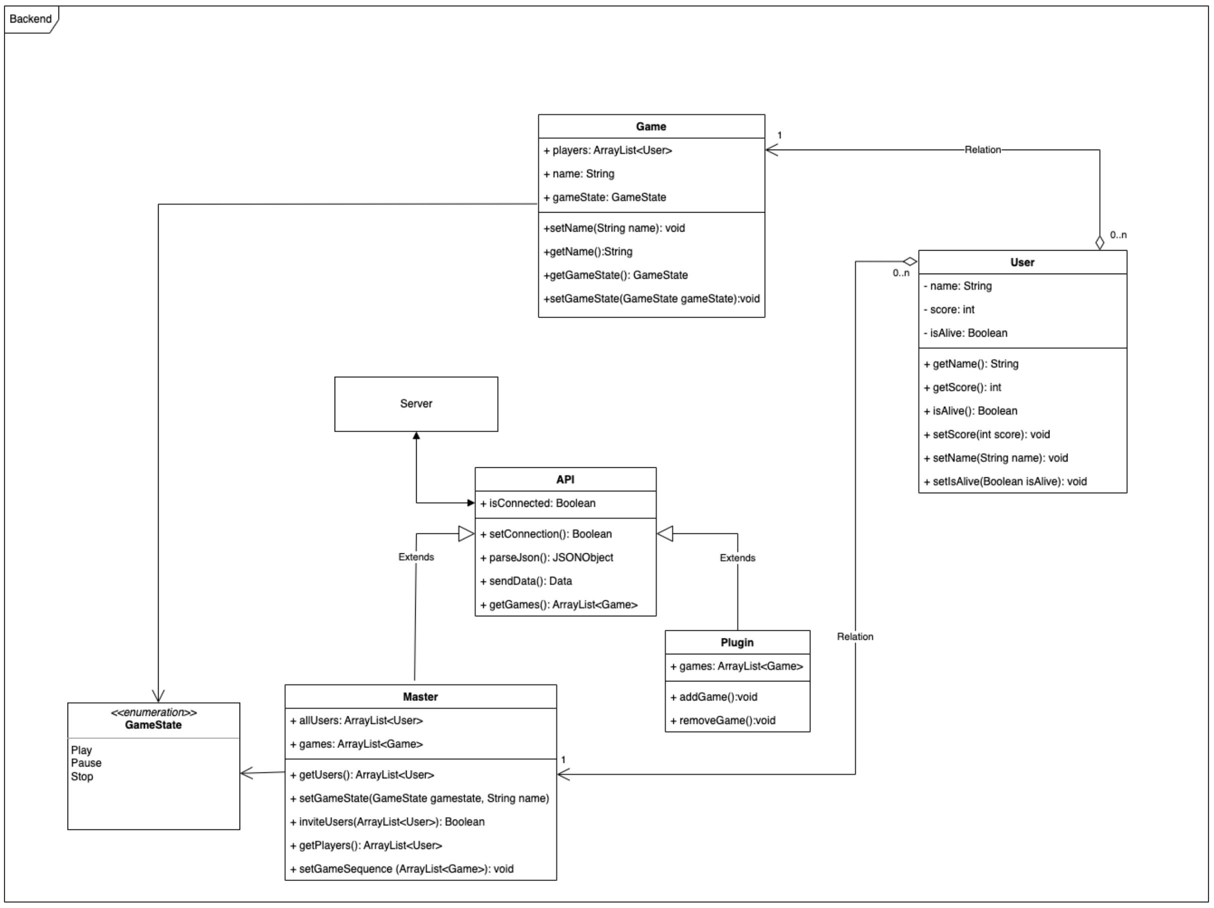
Used modeling tool:

- Draw.io (app.diagram.net)

Class diagram

Author(s): Jazzley, Elif, Jay, Zefan

This chapter contains the specification of the UML class diagram of your system, together with a textual description of all its elements.


Backend
Game
+ players: ArrayList<User>
+ name: String
+ gameState: GameState
+setName(String name): void
+getName():String
+getGameState(): GameState
+setGameState(GameState gameState):void
1
Relation
0..n
0..n
User
- name: String
- score: int
- isAlive: Boolean
+ getName(): String
+ getScore(): int
+ isAlive(): Boolean
+ setScore(int score): void
+ setName(String name): void
+ setIsAlive(Boolean isAlive): void
Server
API
+ isConnected: Boolean
+ setConnection(): Boolean
+ parseJson(): JSONObject
+ sendData(): Data
+ getGames(): ArrayList<Game>
Extends
Extends
Relation
Plugin
+ games: ArrayList<Game>
+ addGame():void
+ removeGame():void
<<enumeration>>
GameState
Play
Pause
Stop
Master
+ allUsers: ArrayList<User>
+ games: ArrayList<Game>
+ getUsers(): ArrayList<User>
+ setGameState(GameState gamestate, String name)
+ inviteUsers(ArrayList<User>): Boolean
+ getPlayers(): ArrayList<User>
+ setGameSequence (ArrayList<Game>): void
1

Brief description about what the class diagram represents
The class diagram shows that our control room is mainly based on the following 5 classes: **Game**, **Master**, **User**, **Plugin**, **API**. These classes are related to each other in certain ways, and some of those relations express inheritance. The relation between the **Master** class and the **API** class, for example, shows inheritance because the **Master** class uses the **API** class to communicate with the server. Furthermore, each class includes the relevant data of specific objects and operations. Each class describes a particular task and includes the relevant attributes and methods to carry out that task. For example, the **Master** class is responsible for keeping track of the users who are still alive and games that can be played. **Master** class is also in charge of inviting the users to the games as well as setting the game state and the game sequence. In addition, this class also includes some methods for retrieving related data for certain requests.

Brief description of the meaning of the most relevant attributes
We have used a number of public and private attributes to design our control room. Our class diagram expresses many attributes that represent the general definition of our control room. **Master** class contains some of the most relevant attributes of our control room such as the attribute *games* which is responsible for creating an array of all the games that the master can choose to start for the players. Another most relevant attribute included in the **Master** class is the attribute *allUsers* which is responsible for keeping track of all the users who are still alive, and therefore can participate in the new games, within an array. The **Game** class also contains some of the most relevant attributes of our class diagram such as the attribute *players* which is responsible for creating an array for all the users who are currently playing the ongoing game. The difference between the attributes *allUsers* and *players* is an important aspect of our control room. The **User** class also has some important attributes such as the attribute *score* which keeps track of the user's scores as well as the attribute *isAlive* which is responsible for indicating the user's status with a boolean variable.

Brief description of the most relevant operations
The most relevant operations are everything that is needed to start and monitor the game. The operations come from the master class and they are:

- getUsers ()
- inviteUsers (array)
- setGameSequence (array)
- setGameState (gamestate, string)
- getPlayers ()

The first operation returns all the players that joined the waiting room. The master then invites the players that will participate in the games that will be played with the second operation. With the third operation, the master chooses what games will be played out of the list with all the games, and sets the order of the games that will be played. With the help of the fourth operation, the master is able to start, pause or stop the game at any time. The master uses the fifth operation to receive a list of all the players that are still alive.

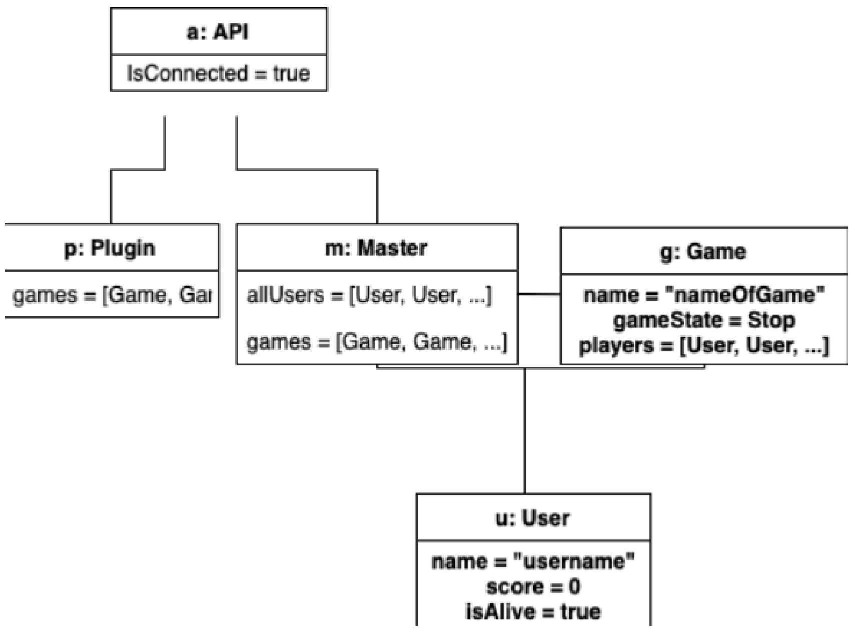
Brief description of the meaning of the most relevant associations involving it
The most important associations are the ones between the master, the API, the user, and the game.

The master has a direct connection with the user because the master needs the information of the user to be able to let the control room work. The master also has a direct connection with the API so data can be sent and received from the server.

The connection between the user and the game class is also important because in the game class the game actually happens and users are needed to play the games.

Object diagram

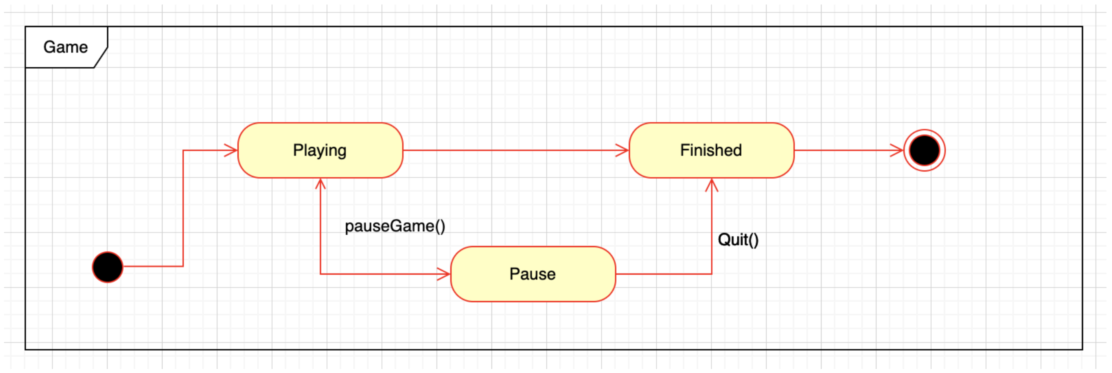
Author(s): Jazzley, Elif, Jay, Zefan



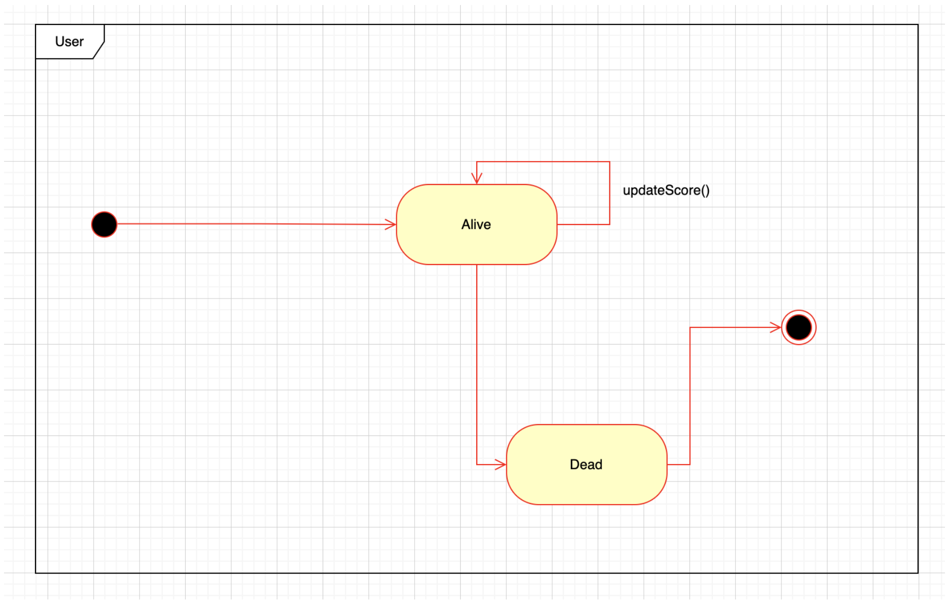
The displayed snapshot is our object diagram of the control room when a game is in action. The object diagram helped us decide which attributes are needed for every class and the relation between them. We see from the object diagram that API is the main object. It's the main object because the API is needed to communicate with the server so we can send and receive the needed data. API has as attribute isConected so we know if a user made a connection. The plugin object is directly connected to the API, because the game developers will need direct access to insert their game into the framework. We also see from the snapshot that the master has the following attributes: allUsers and games. These are needed so the master can invite the players that are going to participate in the games and that he can see all the available games so he can make the game sequence. The master is connected to the game object. The game object has as attribute the name of the game, gamestate, and the players (all the alive users that are participating in the game). The user object has as attribute the name of the user, the score, and isAlive as boolean to see whether a player is alive or dead.

State machine diagrams

Author(s): Jazzley, Elif, Zefan



The state machine diagram above shows some of the states of our game class. The game starts with the state “playing.” After this initial state, the game can either be paused or finished by the master. If the event pauseGame() is triggered the game state changes from “playing” to “pause” and the game is paused. In addition, the event pauseGame() is a two-way event which means the game state can be changed from “pause” to “playing” so that the game can resume after being paused. Furthermore, if the game is in “pause” state, the event quit() can trigger the game state to change to “finished” which means the game ends.

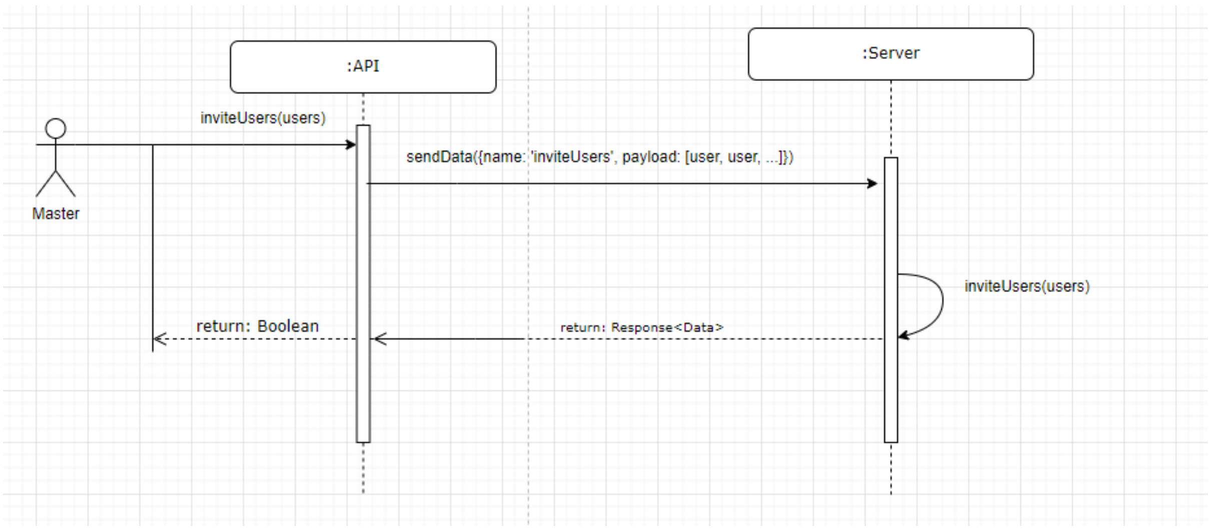


The state machine diagram above shows some of the states of our user class. The game starts with the state “alive.” The looping event updateScore() gets triggered after each game to update the user's scores. If the user is no longer alive the state changes to “dead” and the game ends for that user.

Sequence diagrams

Author(s): Jay, Elif, Zefan

Master Inviting Users to the Game

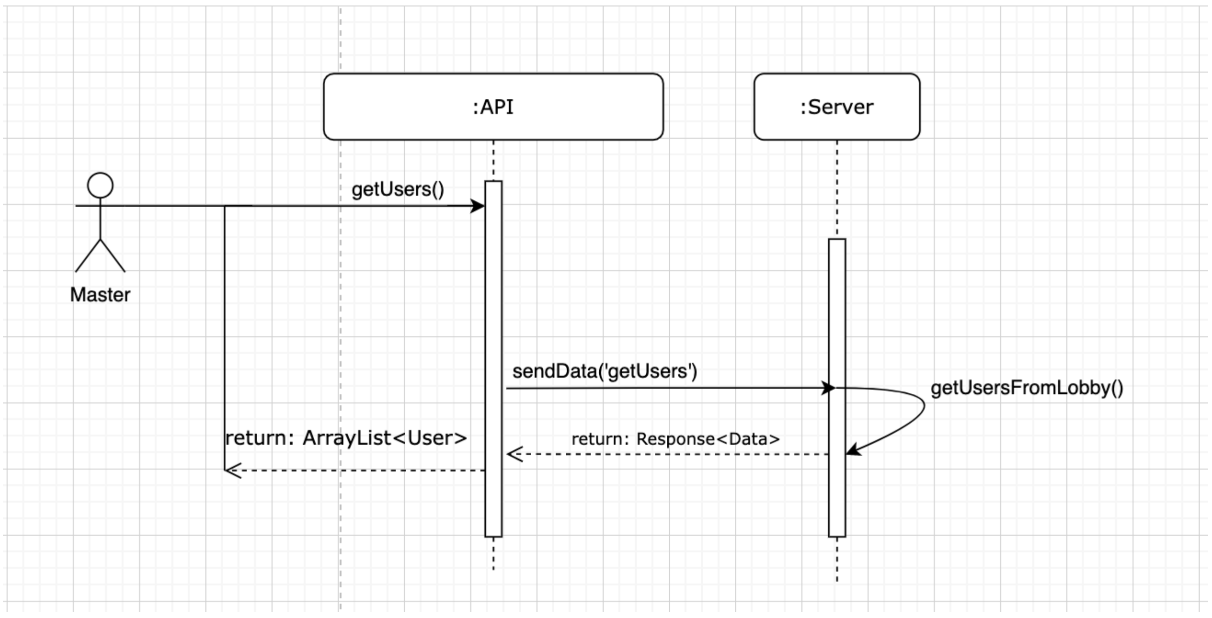


The sequence diagram above describes the process of master calling the chosen users into a new game. This process is mainly performed by the interactions between the **Master** class, the **API** class, and the Server.

In our control room, the master has the ability to either invite every user in the lobby as a player or choose particular users as players for the game that is going to be played. If the master ends up choosing particular players for a game, the users which are not chosen as players for the game wait in the lobby on idle mode.

When the master wants to invite users to a game, the **Master** class interacts with the **API** class through the function inviteUsers(). The function inviteUsers() then triggers the **API** class to interact with the Server through the sendData() function which sends the array of the users chosen by the master to play the game from the **API** class to the Server. After that, the server executes the getUsersFromLobby() function which gets the selected users from the lobby into the game as players and returns the response data to the **API** class. Lastly, the **API** class returns a Boolean which indicates if the users were successfully invited.

Master fetching the available users



The second sequence diagram describes the process of the master fetching the available users. This is performed by interactions between the master class, API class, and the server.

For the master to invite the users to the game, he should be able to see all the eligible users. The getUsers() function is used to see all the users that can be invited to the games. The getUsers() function returns an ArrayList of all the users.

When the function is called, the master class interacts with the API class. The API class then triggers the sendData() function to communicate with the server. The server then calls the getUsersFromLobby() function and it returns an array of all the eligible users. If the server has the data it sends it back to the API class. The API class then returns an array to the master containing all the users. The data is consistently sent back between the master and the server with the API as middleware/parser. All of the data will be sent through a network making use of sockets on a constantly open connection.

Implementation

Author(s): Jazz & Jay

The steps given to us in assignment 2, helped us along the way of how we should model our control room systematically(from class → object → state machines → sequence diagrams). This helped us immensely when approaching the design of the control room.

We followed the structure given to us in the assignment. We started off with the class diagrams looking at which classes would be necessary for the system to function on a lower simpler scale. We added classes such as Master, Game, User, API, etc. We did this systematically by asking ourselves first which classes the master class would need to influence and we adjusted our classes according to the master class. The class diagram provided us with a clearer vision of the system. Because Java is an object-oriented programming language these classes were easy to implement and turn into code.

During the implementation stages, realized that a lot of the classes had redundant functions and some of the classes just did not make sense as part of our implementation. We incrementally changed some functions, removed classes, and even added functions where needed to ensure that we had a proper and clear vision of the system and that this system's diagrams matched the implementation.

After running through the class diagram stage, the other stages were quite simple. All of our other diagrams are based on the class diagram, hence why we decided to work systematically and design the class diagram before designing anything else. The addition, removal, and changing of all the diagrams happened consistently throughout the whole development process.

By creating state diagrams, we were able to narrow down what our system does on the smallest level. This helped us remove a lot of redundancy, the sequence diagram narrowed down the interaction between different components of our system, which helped us realize that there was a lot of unnecessary interaction between different parts of the system.

Key solutions:

1. How do we manage the different states of the game?
    a. By creating an array of strings and mapping the different gamestates to a certain game. It’s vital that we keep a track of what games are paused, playing, or are already finished. So the master has an easier time navigating the system
2. How does a master choose which state a game is in?
    a. By using the following commands
        i. SET GAMESTATE PAUSED/PLAY/FINISHED
3. How does a master exit the game?
    a. By using the following command
        i. EXIT

Main Java Class

Location: src/main/java/softwaredesign/Main.java

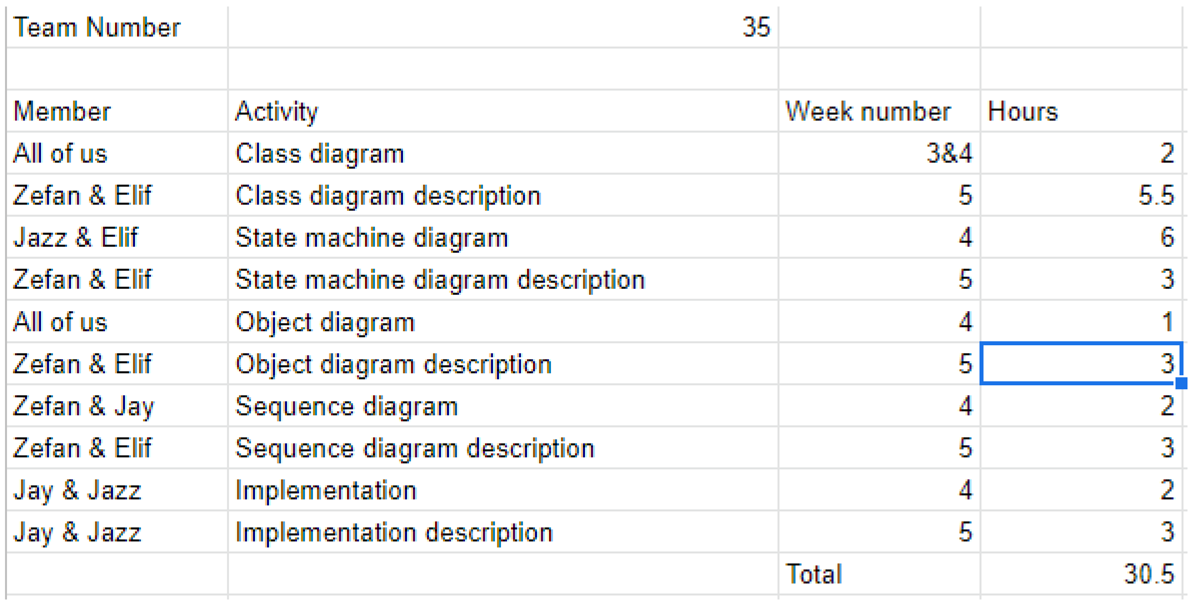
Jar File:

Location: SD-VU-35-SQUIDGAME/out/artifacts/software_design_vu_2020_jar/SquidGame.jar

Video Demo:

https://youtu.be/R04MfC_Cq-M

Time logs



Team Number	35		
Member	Activity	Week number	Hours
All of us	Class diagram	3&4	2
Zefan & Elif	Class diagram description	5	5.5
Jazz & Elif	State machine diagram	4	6
Zefan & Elif	State machine diagram description	5	3
All of us	Object diagram	4	1
Zefan & Elif	Object diagram description	5	3
Zefan & Jay	Sequence diagram	4	2
Zefan & Elif	Sequence diagram description	5	3
Jay & Jazz	Implementation	4	2
Jay & Jazz	Implementation description	5	3
		Total	30.5