Assignment 3

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

Format: establish formatting conventions when describing your models in this document. For example, you style the name of each class in bold, whereas the attributes, operations, and associations as underlined text, objects are in italic, etc.

Summary of changes of Assignment 2

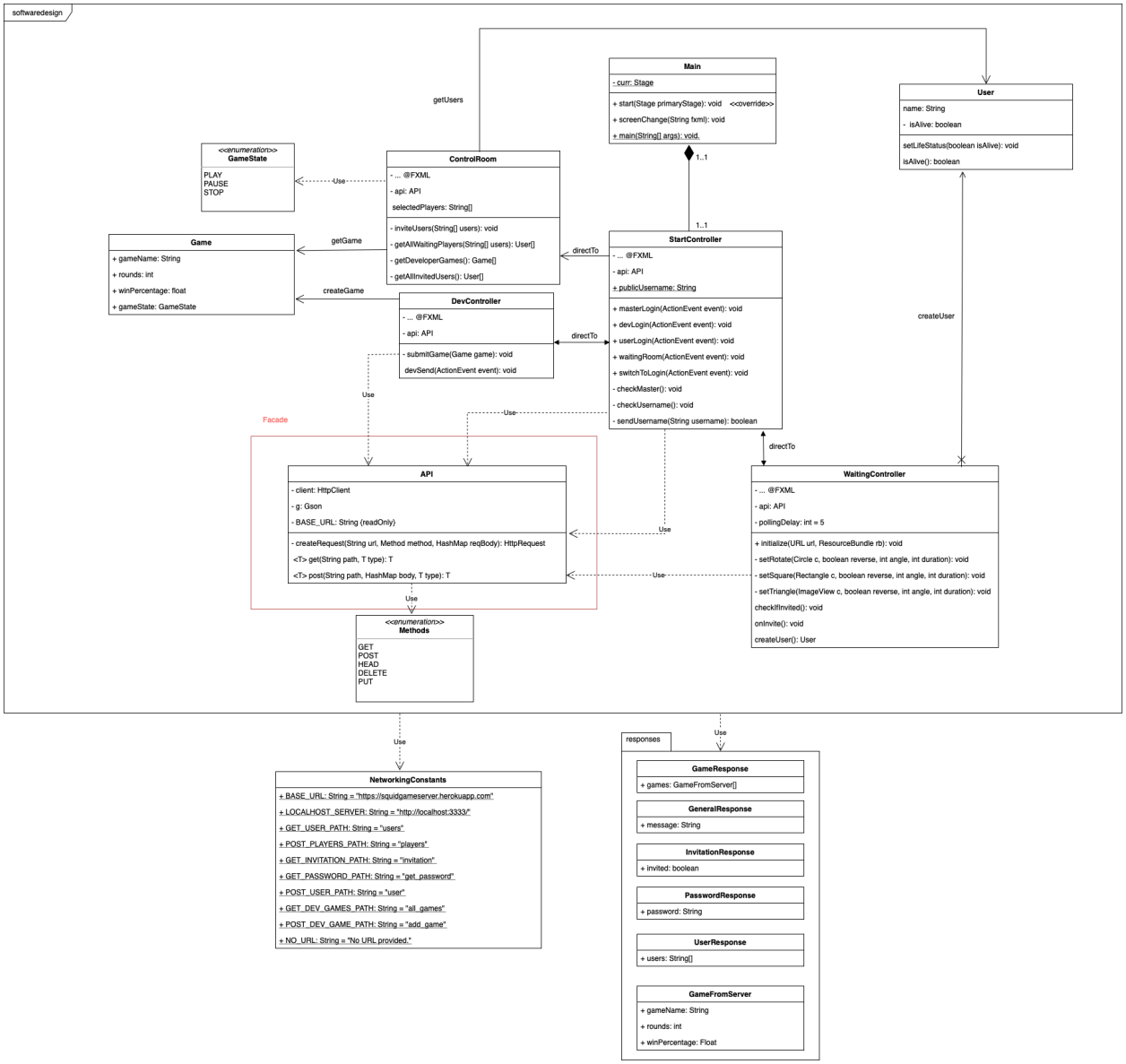
Author(s): Jazzley

Type	Issue	Solution
Implementation	1) The implementation is highly different from the UML diagrams. The latter should have been adjusted accordingly after you have realized the redundancy	1) Changed the UML diagrams to reflect the current system we have implemented. We did this by going over our system again and seeing where the differences were, furthermore the previous system was a bare-bone solution to get a better grasp on what it is we wanted to implement. Along the development timeline we discovered that we sometimes needed more classes,

		different design patterns etc.
Class diagram	1) Missing multiplicities 2) The master class should not be responsible for keeping track of the users who are still alive, the game should do this 3) The Game class should contain an array of the users who participated in the game. Every User object in this array saves its own state(whether alive or dead). The array of Users in Master is not needed. 4) Why does the master need to keep track of games that can be played of that array of games already exists in Plugin?	1) Added the mulitplicities between classes within our class diagram, to reflect our system better 2) Created a separate class called user to keep track of the user status 3) Decided to keep track of the users who played in the master class. 4) Removed the plugin class, since it became obsolete. Revamped what our master keeps track of.
State machine diagram	1) More elaboration	Added the necessary explanation which was initially missing, also elaborated more on the user diagram, to show the flow of the user and the possible states the user can find theirself in.
Sequence Diagram	1) Missing reasoning	Combined the two earlier sequence diagrams into one, since at the initial startup the master does not issue a fetch command manually, it is however fetched automatically as soon as the actor opens up the master console.

Application of design patterns

Author(s): Jay

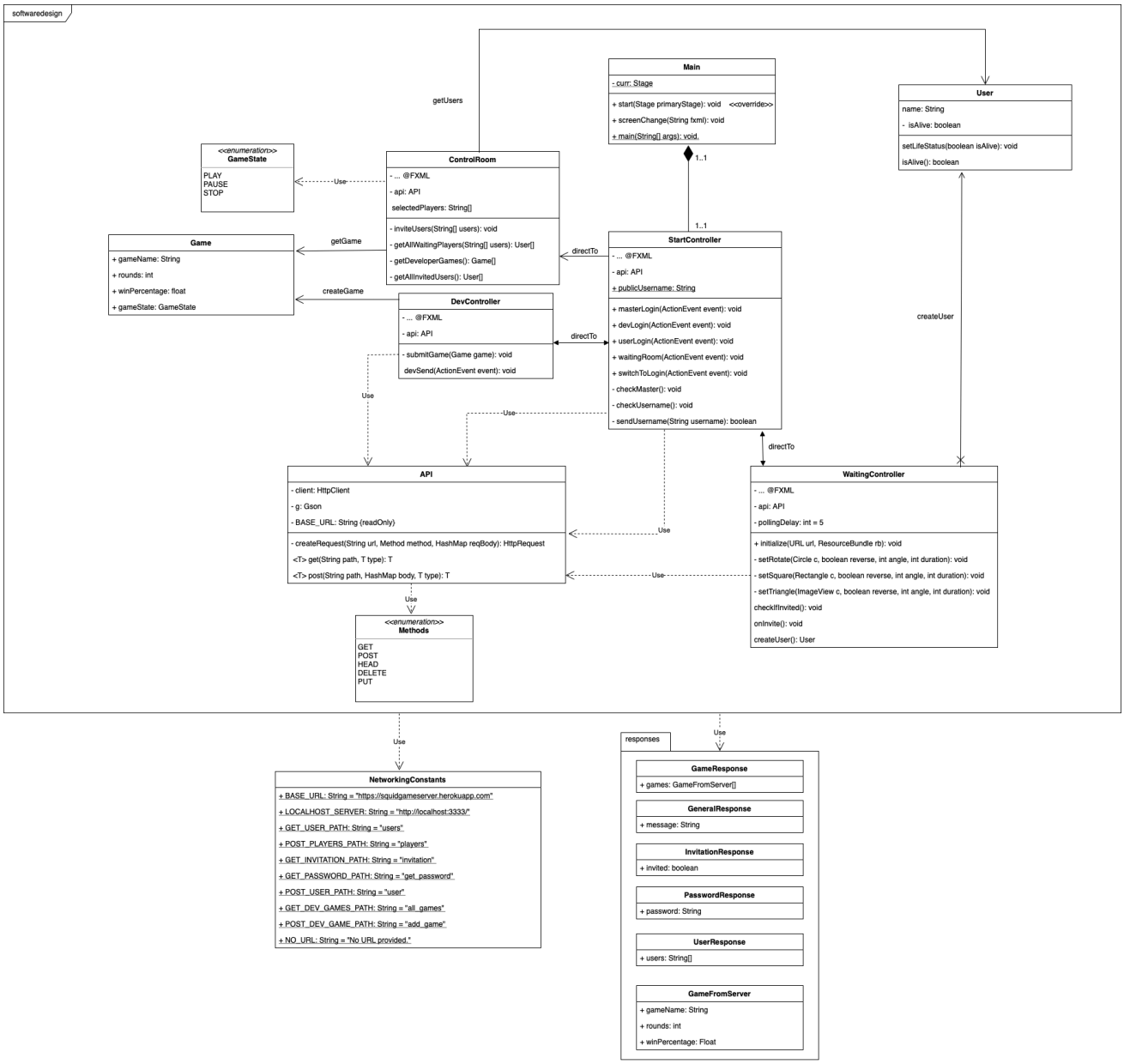


	DP1
Design pattern	Facade
Problem	Sending API requests is complex in Java which uses a lot of packages and other objects.
Solution	A clean class which makes it easy to send an API request with the maximum of 1 call.
Intended use	We intended to use the API requests for inviting users, to set a user on the waiting list, adding/fetching games etc. There are a lot of api requests in our application, because we wanted to transfer the logic over a networking system.
Constraints	For very complex api requests, we are not able to use the class. But we did not need any of these requests so practically, for our use, there were no constraints
Additional remarks	The API class did indeed fix the complexity of API requests which made the most important part of our intended use realized.

Class diagram

Author(s): Jay & Elif

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



Brief description about what it represents:
The class diagram represents the structure of our whole application. We can observe that our class diagram consists of controllers which are implemented classes for our gui, helper packages and extra classes for the base logic of our application. The controllers consist of fxml fields which are useful to retrieve data from the active user. Examples of these are input boxes and button events. We had to make the base classes for the minimum logic that we wanted to have in. These were features like networking, keeping track of certain games and users. That is why we have the *API*, *Game* and *User* classes. The API class takes a lot of detail away which made sure that we could send HTTP requests with only one method. This abstraction increased our speed in developing the application. Last but not least, we also implemented some helper classes. There were a lot of networking constants which could suddenly change to something else. That's why we put the networking constants in one class. The responses package made sure that we always have the same response from the server. This also clarifies what sort of response we get back from the server.

Brief description of the meaning of the most relevant attributes:
Our class diagram expresses many attributes that represent the general definition of our system. We have used a number of public and private attributes to design our system. ControlRoom class contains some of the most relevant attributes of our control room such as the attribute *getDeveloperGames* 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 ControlRoom class is the attribute *getAllWaitingPlayers* which is responsible for keeping track of all the players who are still alive, and therefore can participate in the new games, within an array. The attribute *selectedPlayers* is another important attribute that is responsible for creating an array for all the users who are currently playing the ongoing game. The User class also has some important attributes and one of those attributes is the attribute *isAlive* which keeps track of the user's status and therefore is responsible for indicating the user's life status with a boolean variable.

Brief description of the meaning of the most relevant operations:
The most relevant operations are the operations that are needed to start and monitor the game. Below are the most important operations of our system:

- getallWaitingPlayers()
- inviteUsers (array)
- selectedPlayers ()
- setGameSequence (array)
- setGameState (gamestate, string)

These important operations reside in our ControlRoom class. The operation getAllWaitingPlayers() is responsible for returning all the players that are in the waiting room. The master then invites the players that will participate in the games that will be played with the operation inviteUsers(array). The operation selectedPlayers() returns all the users who accepted

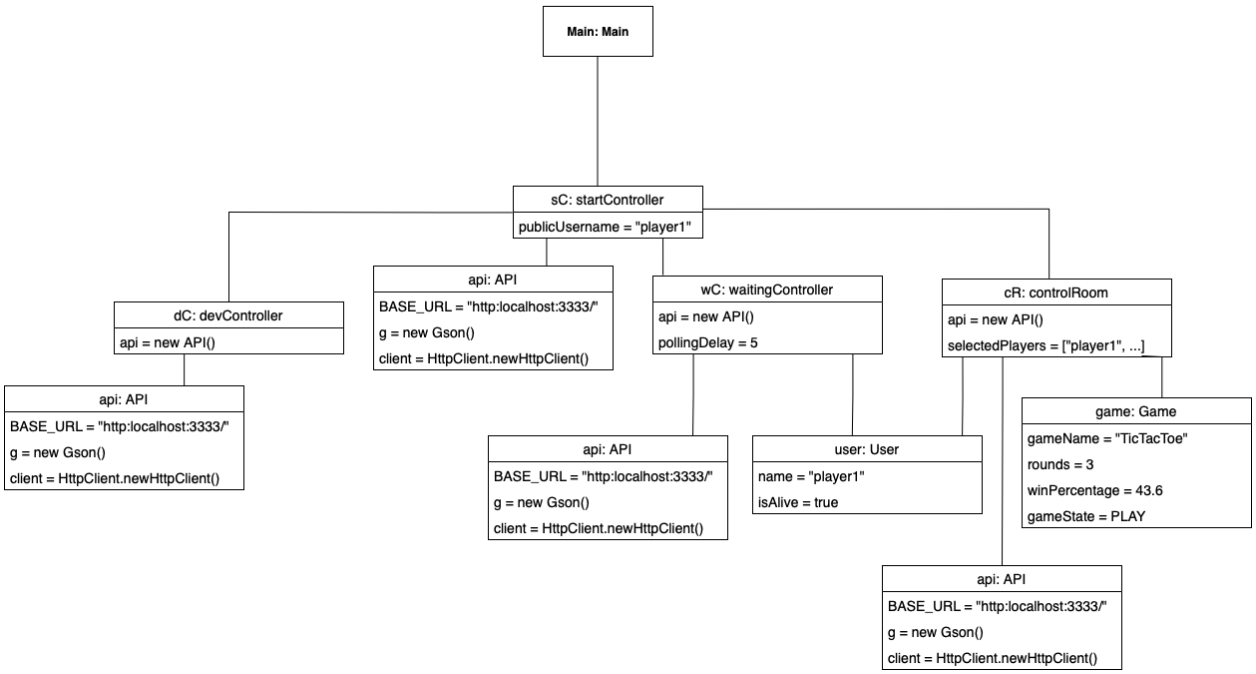
the invitation of the master and therefore now the players of the ongoing game. The operation setGameSequence() lets the master choose what games will be played out of the list with all the games, and sets the order of the games that will be played. The operation setGameState() is responsible for letting the master choose to start, pause or stop the game at any time.

Brief description of the meaning of the most relevant associations involving it (each association can be described only once in this assignment):
One of the most relevant associations in our system is between the ControlRoom and the Game. That is because ControlRoom needs the information within Game to function properly. When the master, for example, wants to start a new game the master should be able to see all the games and choose the one that he would like to start. Another relevant association in our system is between the ControlRoom and the User. That is because ControlRoom also needs the information of the User to function properly. When the master, for example, wants to start a new game he should be able to see the user status.

Object diagram

Author(s): jay

Example: Running State

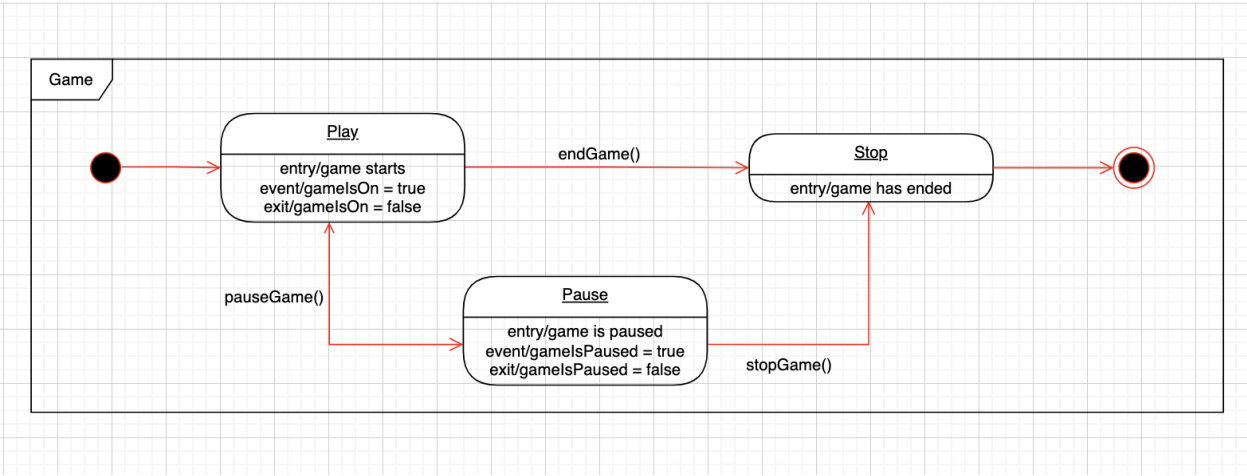


Our object diagram represents our system during the execution of our application where there are some users waiting and playing our game. We can observe multiple API classes which are used for HTTP requests to make sure that everything is sent correctly over a network. This makes it a multiplayer gaming effect where the ‘Master’ (controlRoom) has an impact on every user and game. This is also represented in our object diagram, because the control room has a

link with Game, User and API. The waitingController should only be able to either create a user or not. This is why it is connected to the User object. devController, which is used to add games, might be a bit eccentric. It creates the question to wonder why there is no game connection. This is because the devController only sends the games to the server, but in the controlRoom it will get instantiated. All the pages are accessible through our homepage, which is why our controllers (placed logic for pages) are all connected to the startController.

State machine diagrams

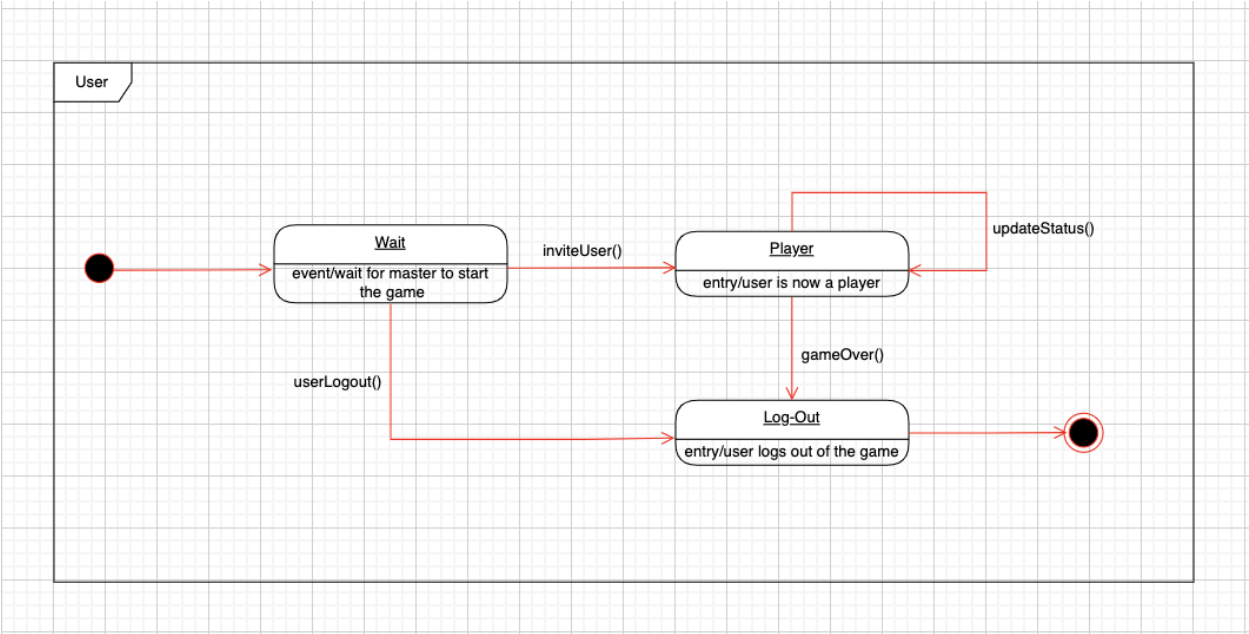
Author(s): Elif



The state machine diagram above shows some of the states of our game class. The game starts with the state “play.” When the game enters this state game starts and the variable gameIsOn equals true.

After this initial state, the game can either be paused or stopped by the master. When the variable gameIsOn equals false the state “play” is exited. If the event endGame() is triggered, the game state changes to “stop” and the game ends.

If, however, the event pauseGame() is triggered the game state changes from “play” to “pause” and the game is paused. 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. When the game enters the pause state, the game is paused and the variable gameIsPaused equals true. When the variable gameIsPaused equals false, the pause state is exited. If the event pauseGame() is triggered, the game state changes to “play” and the game resumes. If, however, the event stopGame() is triggered, the game state changes to “stop” and the game ends.



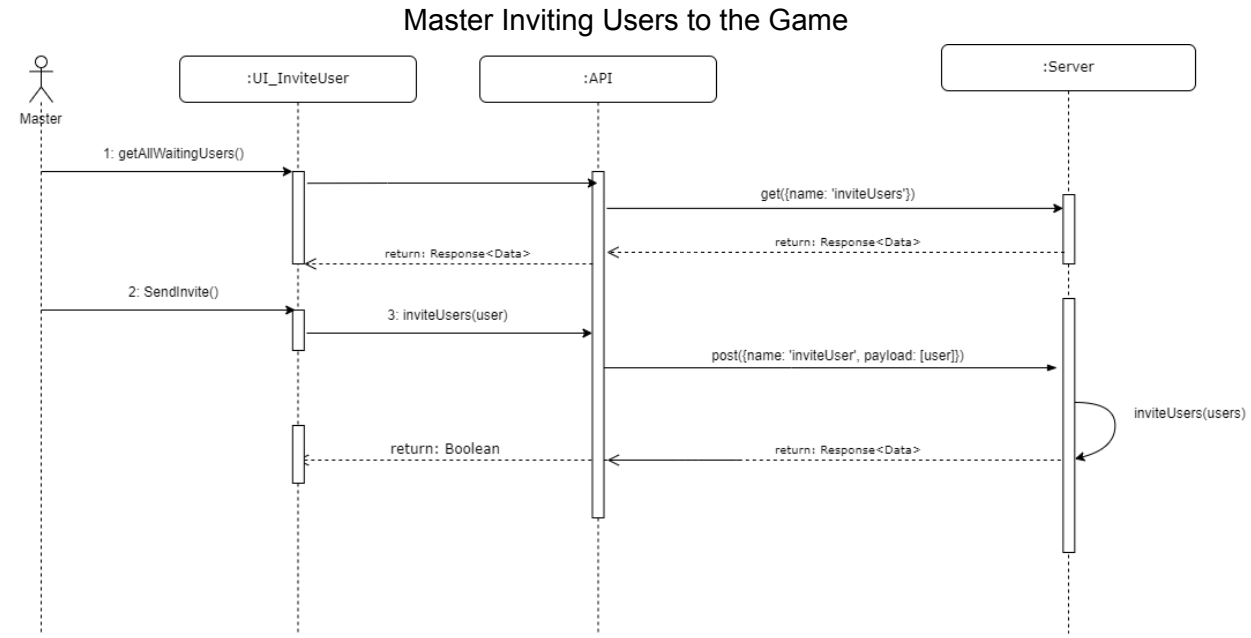
The state machine diagram above shows some of the states of our user class. The user starts with the state “wait.” When the user enters this state, the user waits for master to state the game.

After this initial state, the user might get an invitation from the master or choose to log out of the system. If the event userLogout() is triggered, the user enters the state “log-out” and logs out of the system. If, on the other hand, the event inviteUser() is triggered, the user enters the “player” state and becomes a player in a game.

The event updateStatues() continually checks and updates the player statues to see wheter the user is still alive in the game. If the user is not alive anymore, the event gameOver() is triggered and the user enters the state “log-out” and logs out of the game.

Sequence diagrams

Author(s): Jazz & Zefan



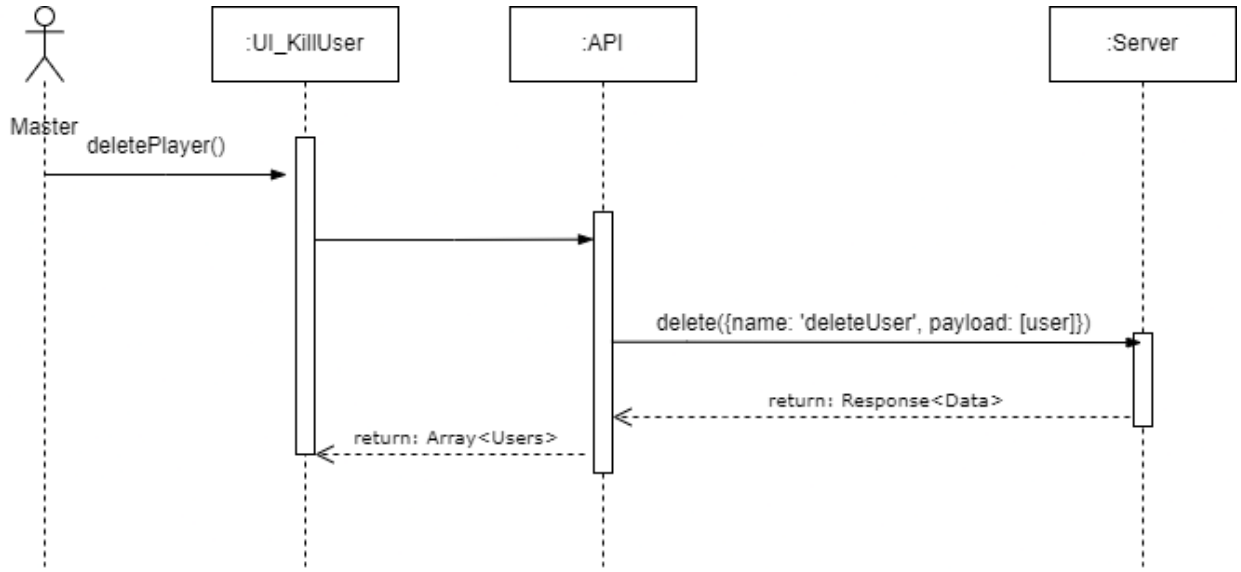
The sequence diagram above describes the process of master calling the chosen users into a new game. This process is mainly performed by the interactions between the **Master** class, the **API** class, and the Server.

In our control room, the master has the ability to invite players one by one from a list displayed on the gui. After choosing certain players to play his version of ‘a game of squid’, the users which were not chosen as players remain idle in the lobby waiting until the master decides to invite them to the game.

Initially when loading up the GUI the master automatically triggers the getAllWaitingUsers() function which in turn sends the data through our **API** for post-processing the java String into JSON data, the **API** then goes ahead and triggers a GET request, as a response to this GET request the server returns and ARRAYLIST of users currently waiting in the lobby for the master to choose from.

When the master wants to invites users to a game, the Master class interacts with the **UI_InviteUser** object. By choosing a user the SendInvite() function is triggered, the **UI_inviteUser** object then proceeds to call the function inviteUsers(). The function inviteUsers() triggers the API class to send a POST request to the server. The server then goes ahead and sends back response data in the form of JSON, this JSON is parsed by our server. Lastly, the **API** class returns a Boolean which indicates if the user was successfully invited.

Master killing a player



The ‘master killing a player’ sequence diagram is an interaction overview diagram that gives an overview of the process that the master(super user) sequentially goes through when killing a player. This diagram consists of 3 object lifelines with one actor i.e. UI_Killuser, API, Server and Master in that order.

When the master chooses to kill a player, they select a player from the UI_KillUser, click on kill user, which calls the deletePlayer() function. After the function has been called the API translates this function call’s payload into JSON so this can be parsed by the server, the API then makes a delete request to the server. The server then proceeds to remove the killed user from the Server’s database. After the Server has successfully removed the entry from its database the server sends response data (an array) in the form of JSON, this data is translated by the API into java code which in turn removes the player’s visibility from the GUI.

Implementation

Author(s): Jazz & Jay & Zefan

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, I 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 games 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 doing the following
        i. BY CHOOSING A GAME FROM THE CURRENTGAMES TAB THEN CLICKING THE PAUSE BUTTON TO TOGGLE A PAUSED OR PLAYING STATE(INITIALLY ALL GAMES ARE PAUSED).
3. How does a master exit the control room?

a. By doing the following
    i. CLOSING THE SYSTEM

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://www.youtube.com/watch?v=Umk_919U7J8

Time logs

<Copy-paste here a screenshot of your time logs - a template for the table is available on Canvas>

Team Number: 35			
Member	Activity	Week Number	Hours
Jay & Jazz	Design patterns	6	4
Jay & Elif	Class diagram	6	3
Jay & Elif	Class diagram description	6	2
Jay	Object diagram	6	3
Jay	Object diagram describtion	6	1
Elif	State machine diagram	7	3
Elif	State machine diagram description	7	1
Jazz	Sequence diagram	7	3
Jazz	Sequence diagram description	7	1
Jay, Jazz, Zefan, Elif	Implementation	8	12
Zefan & Jazz	Implementation of GUI	8	10
Jazz & Jay	Implementation description	8	2
		Total	45