Software Engineering Final Project
A.A. 2018/19

Team #06: Aspesi, Battiston, Carabelli

Contents

1 Design choices

1.1 Fat Server, thin Client

We decided to take the *thin client* approach, letting the server manage everything aside from taking the user's input.
Our client has no game logic, it just displays the state of the game board provided by the server as a bare-bone version of the model. These classes reside in the `clientmodel` package.
As explained in 5.1, after successfully logging to the server, the client waits indefinitely for server requests until it receives an `endGame` notification.

1.2 Weapons, Powers and base actions

We chose to take a functional approach to the modeling of weapons and power cards actions, as well as base actions like shooting, running, picking up loot etc.
Every action is defined as a λ-function, with a unique `ID`, defined in `ActionLambdaMap`. Actions are assigned to the correct instance of the card by placing its lambda `ID` in an attribute.
Following the tutor's suggestion, we decided to extend the functional approach to the application of cards effects on the Player, by providing a public method which takes a `PlayerLambda` functional interface as an argument.
The `PlayerLambda` instructs the Player instance on how to apply effects to itself. A small set of useful and frequently used `PlayerLambda`s are collected in the `EffectsLambda` class.

1.3 Feasibility of user chosen actions

We decided to only provide users with actions they can carry out, so that we would never have to manage wrong inputs.
This feature is achieved with the creation of `FeasibleLambdaMap`. Its function is very similar to `ActionLambdaMap`, but provides a convenient way to check if an action can be executed in a particular state of the match.
Thanks to this, we can make sure to only suggest feasible actions by sending only whose which pass the feasibility test.

2 Functionalities

2.1 Socket and RMI connections

We implemented both connection options. They can be used interchangeably since `Server`, `Connection` and `Client` interfaces (used to implement network communications) have one implementation for each technology.

2.2 Console and Graphical UI

We implemented both User Interface options. They can be used interchangeably since they both implement the `UserInterface` interface.
The user can choose its preferred interface as described in 4.

2.3 Multiple Games

The server can run many games simultaneously. When a player connects, he's added to a lobby with other players who chose the same number of skulls.
When the lobby reaches three players a timer starts. If the timer runs out or the fifth player connects, the match starts. This behavior of the match is described in 6.2.
Other players who choose the same number of skulls will be placed in a new, empty lobby and will be able to play their match even if the first one hasn't finished yet.
The timer for the start of the game, the minimum number of players per match and other parameters of the server can be configured as described in 4.

2.4 Persistence

While the server executes matches, at the end of every turn, the server saves their states in `.adr` files, a JSON formatted representation of their whole state. These files are deleted when the match ends, since they're not needed anymore.
If the server unexpectedly shuts down before all active matches end, the first new instance of the server will load `.adr` files as joinable matches alongside new empty lobbies.
Users who were playing those matches can reconnect with the same nicknames to be instantly added to the old game's lobby. When all players reconnect, the match restarts from the last saved turn.

3 Code structure

The structure of the sources folder follows the Model View Control pattern used to divide classes by their function.

- **`model`**: represents the state of the game in a single match. It includes players, cards, points and more
- **`controller`**: logic of every routine executed by the server and client for:
  - Connection and log in
  - Management of already bound connections
  - Advancement of matches by executing turns routines
- **`view`**: interface between the game's logic and the user
- **`clientmodel:`** reproduction of model classes used to send data to clients. They are designed to only share data which is relevant to the receiving client. Every class' name starts with its model counterpart, followed by the `View` suffix.
- **`exceptions:`** exceptions used throughout the project

4 Configuration

Server

The server (`AM06_Server.jar`) can be configured by placing a `config.json` file in the execution folder.
Here is the content of an example configuration file:

```
{
    "startMatchSeconds": 60,
    "playerTurnSeconds": 180,
    "minPlayers": 3
}
```

The values in this snippet are the default ones, used by the server when no `config.json` is found.

Client

Clients (`AM06_Client.jar`) are fully configured using startup flags. The client can be started up with a Command Line interface (no flag) or with a Graphical User Interface (`-g`).
When executing the Client with a GUI, JavaFX libraries for the correct OS have to be provided by placing the `javafx-sdk-11` folder in the same folder where `AM06_Client.jar` is placed.

AI Client

An *Artificial Intelligence* mode (`AM06_AI.jar`) has also been built for debug and “single player” purposes. It can be used to instantiate “bot” players.
To start an AI player in GUI mode the `-g` flag is needed. Just like `AM06_Client.jar`, JavaFX libraries are needed in this case.
AIs normally use Socket connections. They can be configured to use RMI connections with the `-rmi` flag.
If this client is executed on machines other than the server’s machine, the server’s IP can be specified with `-ip [Server’s IP]`.

5 Execution flow

5.1 Client

The client can be started up in CLI or GUI mode by using the corresponding flags.
Both user interfaces will prompt the user to choose if he wants to connect using RMI or Socket connections. After this, the user has to input the server’s IP.
When the client successfully connects, it start to indefinitely listen for requests from the server.
At first, the server will send requests to log the user in by following the Login routine.
After asking the nickname to prevent name collisions, we ask for the desired number of skulls. We decided to create a different lobby for every choice to avoid forcing users to play very long matches if they don’t want to.
At the end the user will be correctly placed in a match, and the client will wait for the start of the match (6.2) to show the game’s map.
The following requests will be game state updates or requests for actions, when the player’s turn comes.
When the game ends, the client will show the winning user and every other user’s ranking, with the number of points they scored during the game.

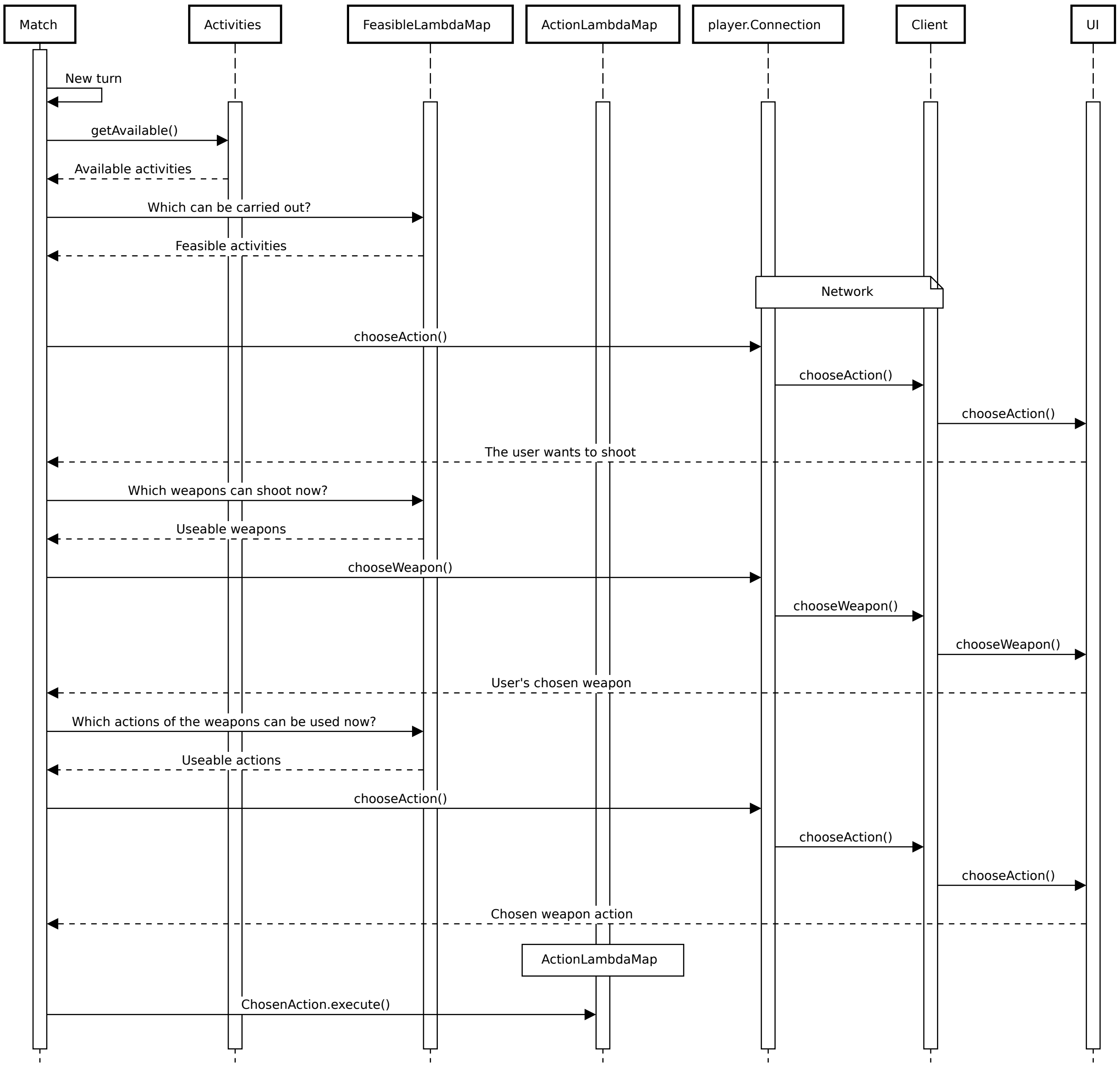
5.2 Server

Right after starting up, the server start to listen for RMI and Socket connection requests.
To achieve games persistency, if the `matches` folder contains `.adr` files, these are loaded to let users reconnect.
When a new connection is found, the Login routine is executed, and the client is added to the correct match (6.2).
If all conditions to start the match are met, the server starts the first turn.

Game logic

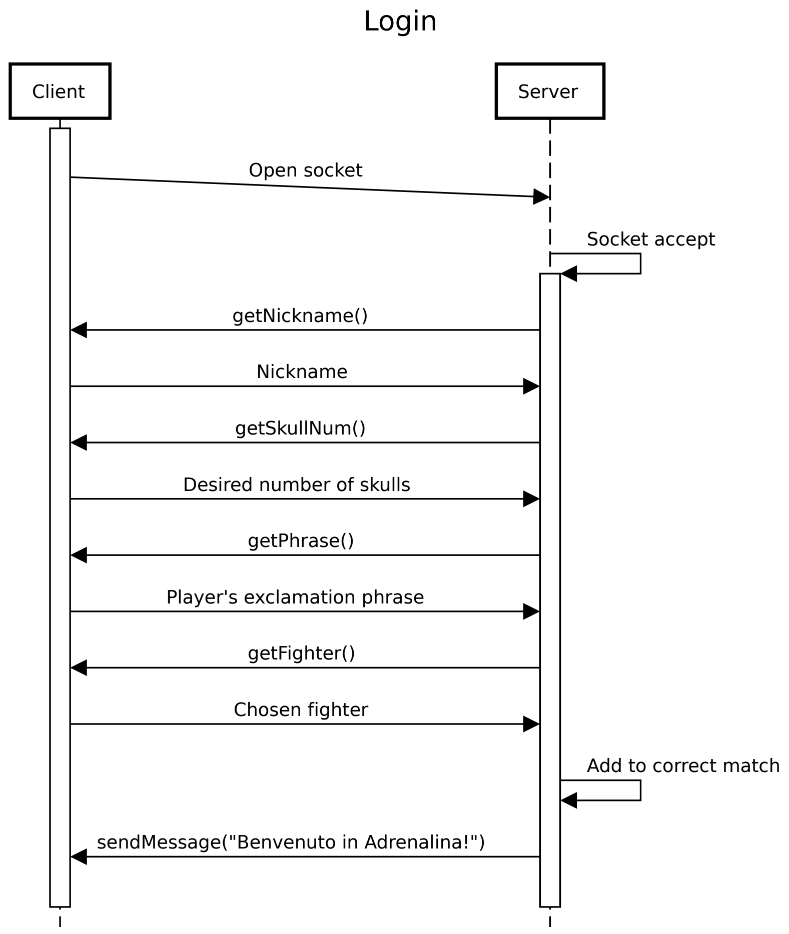
At every turn, a player has to choose which actions he wants to execute to advance in the game.
As an example, the following diagram shows the execution flow for one of the most complex actions of the turn (shooting with a weapon):

Shoot

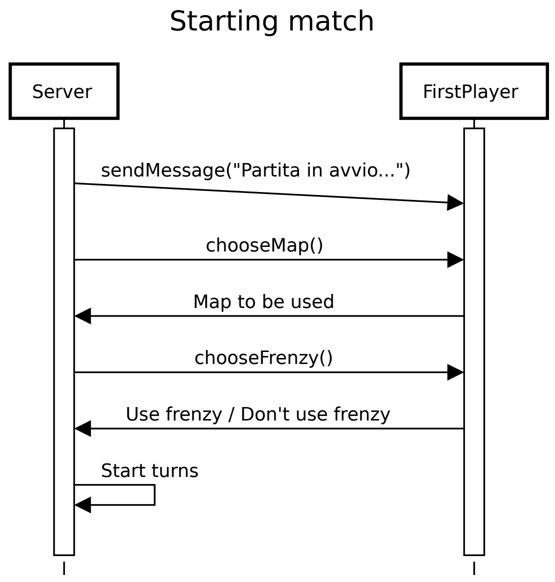

Match
Activities
FeasibleLambdaMap
ActionLambdaMap
player.Connection
Client
UI
New turn
getAvailable()
Available activities
Which can be carried out?
Feasible activities
Network
chooseAction()
chooseAction()
chooseAction()
The user wants to shoot
Which weapons can shoot now?
Useable weapons
chooseWeapon()
chooseWeapon()
chooseWeapon()
User's chosen weapon
Which actions of the weapons can be used now?
Useable actions
chooseAction()
chooseAction()
chooseAction()
Chosen weapon action
ActionLambdaMap
ChosenAction.execute()

6 Relevant communication routines

6.1 Login routine



6.2 Starting Match


Starting match
Server
FirstPlayer
sendMessage("Partita in avvio...")
chooseMap()
Map to be used
chooseFrenzy()
Use frenzy / Don't use frenzy
Start turns