Communication protocol

The communication during the game is completely handled with String that are serialized JsonObjects.

We build a JsonObject like this:

```
{
        “method”: “$methodName$”,
        “$value$”: “$content$”
                .
                .
                .
}
```

$methodName$ is the actual name of the method that will be called either on the server or the client, and $value$ is the key that will be used on the other side to retrieve the information $content$. There may be more than one key, here are two examples.

From Server to Client

The JsonObject has only one “value” key that holds a JsonObject filled with all the keys or JsonObjects needed.

Example:

Something changes in the model, so the view needs to be updated. In the view there is a method declared like this:
public void updateBoard(String value);

What we do consists in:
- building on the server a JsonObject with all the information that the method on the view needs;
- serializing it with the toString() method (will be called “serializedBoard” for simplicity);
- adding to the serializedBoard to a JsonObject like this:
  ```
  {
          “method”: “updateBoard”,
          “value”: “serializedBoard”
  }
  ```
- serializing this JsonObject with the toString() method;
- sending it to the client.

The client deserializes the String (in other words it just creates the actual JsonObject from the String) and calls the right method.
Now the updateBoard method on the view can do what it has to do with the JsonObject it needs.

From Client to Server

The JsonObject has different keys for every piece of information.

Example:

The user wants to collect the weaponCard with id = 1: the JsonObject will be built as such:

```
{
        “method”: “askCollect”,
        “cardIdCollect”: 1
}
```

The method “askCollect” will be called on the right class of the server with the whole object as a parameter and can use every key to search the information it needs.

This abstraction allowed us to hide the network and simulate a local JsonObject transmission.

RMI

Server:
The class “Presenter”, an abstract class that implements the “VirtualPresenter”.
Two classes, “RmiPresenter” and “SocketPresenter”, extend “Presenter” and implements its abstract methods, only the ones that are devoted to the network.
Client:
The class “ConsoleView” implements the methods of the interface “VirtualView”, which are the methods that we expose to the server using a “callback” method that not only gives the client server the possibility to use client’s methods, but also sets a one to one connection between the ConsoleView of each client and its Presenter.
Thus, every client has a personal presenter that accesses the model which is unique (within the same match).

Socket

Server:
The class “SocketPresenter” extends “Presenter” but, as said before, also has methods that put on the output buffer the serialized JsonObject for the client.
Symmetrically the client sends a String that is actually a JsonObject that can be used after being deserialized.
Client:

The class SocketConnection deserializes the JsonObject as soon as it is processed from the input stream to make it usable.

Initialization

Login phase

```
{
        “method”: “selectPlayerId”,
        “playerId”: “$username$”
}
```

Every time a new user connects to the server, everyone connected receives a notification.
In every moment any user can ask the server to start a new game, but the request is accepted only if there are at least three people logged in, the minimum number of players necessary to play.
If there are too many people waiting and no one asks to start a new game, the game starts automatically.
Since for every query from the user, the client and the server share each other Json objects, to make the description less verbose we will omit that part and pretend the client is the user that “speaks” with the server.

New game phase

```
{
        “method”: “askCreateGame”,
        “gameId”: “$gameName$”,
        “numberOfDeaths”; $intNumberOfDeaths$,
        “frenzy”: “$frenzy$”
}
```

```
{
        “method”: “selectGame”,
        “gameId”: “$gameName$”,
        “character”; “$characterName$”
}
```

Once the game starts, the server will set the player who created the game as the first player.

Game

Full message exchange of the shoot phase

Client to server:

```
{
        “method”: “selectAction”,
        “actionNumber”: $number$
}
```

Server to client:

```
{
        “method”: “completeSelectAction”,
        “value”: $properJsonObject$
}
```

Client to server:

```
{
        “method”: “askActivateWeapon”,
        “cardId”: $number$
}
```

Server to Client:

```
{
        “method”: “updateState”,
        “value”: $properJsonObject$
}
```

Client to server:

```
{
        “method”: “askUseEffect”,
        “line”: $stringWithInformation$
}
```

Server -> executes;

Server to Client:

```
{
        “method”: “updateBoard”,
        “value”: $properJsonObject$
}
```

This is the interaction that goes on during the whole game.