Master-Project: „Reflexionsunterstützung für virtuelle Rollenspiel-Umgebungen (Serious Games) “

(WS 15/16)

Technical Documentation

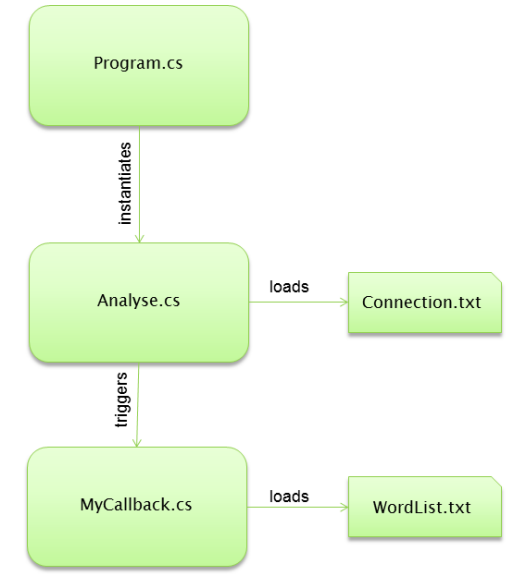
Table of contents:

GAME CLIENT Back-End: Multi-Agent-Architecture Functions

(Author: Markus Mentzel)

Agent Structuring:

Every agent consists of at least three different Classes as you can see on the Graphic below.



Analysis Class:

The analysis Class is the one that is holding the constructor of the Agent. This is the class where the Agent Object is defined. The object provides a connection to the SQLSpaces and reads in other important information if necessary. The decision to which tuple the Agent should react is also defined in this class.

MyCallback Class:

This class provides a call method that is invoked if a tuple of a special form was written into the tuple SQLSpaces. Most of the functionality of the Agent and also the output-tuple is defined in this class.

Program Class:

Responsible for starting the Agent. Every agent is an independent Program running in parallel to the others.

Connection.txt:

Holds the information necessary for getting a connection to the Server.

[Wordlist.txt]

Some agents are searching for special keywords inside the conversation to enrich some properties to the player input. E.g. the rudeness agent reads in a list of swearwords.

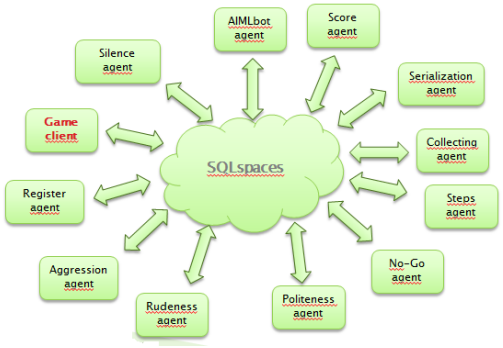
Agent Grouping

Preprocessing Agents:

Some of the agents can be grouped as preprocessing agents. Preprocessing agents react directly to the user input. That means it reacts to the tuple that is written by the game client.

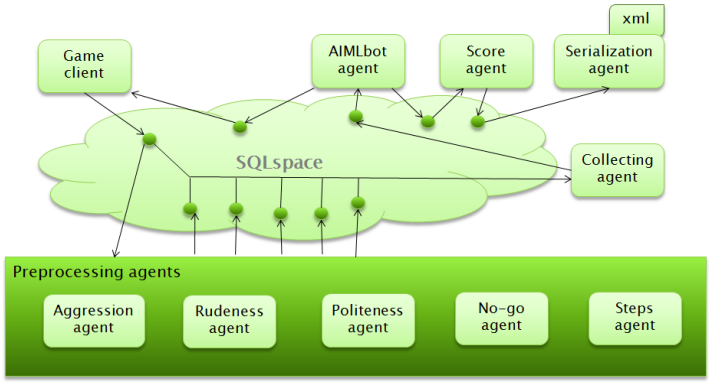
The Output tuple of the preprocessing Agents is taken by the Collecting Agent.

Most of the preprocessing agents are working with word list and analyzing the player input.



The Application is divided into the game client itself and eleven agents. Every Agent is an individual program running in parallel to the others. They are responsible for analyzing the data to provide the different properties that are important for the scoring system but also for the game flow. The communication between the agents and the client is organized using SQLSpaces as a blackboard system. Every agent including the client can read/write tuples from/to the SQLSpaces. They are waiting for an appearing tuple which has the structure they can deal with. After that they can use the information the read tuple provided, enrich data and write a new tuple where other agents might react to.

Because every agent is just answering to a special form of tuple a certain direction is set. We consider a game loop in the graphic below.



After the user has written a message and the game client wrote the affiliated tuple into SQLSpaces, the preprocessing agents read it. Preprocessing agents are mostly analyzing the behavior of a User by mining the text of the messages. The Group of Preprocessing agents is including…

- …the *Aggression Agent* which is analyzing the player input, searching for aggressive Keywords or Symbols that are hold in a text file.

- …the *Politeness Agent* which is analyzing the player input, searching for polite Keywords that are hold in a text file.

- …the *Step Agent* which is counting the overall number of Messages.

- …the *Rudeness Agent* which is analyzing the player input searching for rude Keywords or Symbols that are hold in a text file.

- the *No-Go Agent* which is analyzing the player input searching for phrases that should not be used even if they are neither aggressive or rude. The phrases are hold in a text file.

The results of the preprocessing agents are taken by the *Collecting Agent* to form one big tuple out of it. It contains all the enriched data which can now be used by the *AimlBot Agent*. It uses this information to decide on a fitting bot answer dependent on the aiml-file. It adds the answer to the tuple and writes it into the SQLSpaces. This information is read by the game client that can now display the bot Answer and it can loop again.

But not only the client read the tuple of the AimlBot Agent. Also the *Score Agent* is taking the data to calculate a Score. The tuple generated by the Score Agent is taken by the *Serialization Agent* to save the important Data for the Group Reflection tool into an xml-file.

Not every Agent is part of this loop. For example, the *Register Agent* which is responsible for taking the username and providing a new Session for the user. If the user doesn't answer for a while the *Silence Agent* is triggered. In that case also the AimlBot Agent gets active and writes a Silence Message as bot answer.

Aggression Agent

Input tuple:

```
(Name   | SentenceOpener | Anfrage | Session# | AnswerQuality | ID  )
(String | String         | String  | int      | int   | int )
```

Functionality:

The agent is reading the input tuple. He takes the SentenceOpener and the PlayerInput and searches for aggression.

Aggression is recognized by using …

- … aggressive words that are stored in a text file
- … capslock
- … three or more exclamation marks

If the agent find aggression he sets aggression to true and increases the counter.

We use Counter tuple to get access to the last aggression count.

Counter tuple:

```
(Session# | “Aggressions-Count” | Name   | AggressionCounter)
(int      | String              | String | int              )
```

After that the Agent can write the Output tuple with the new extracted information.

Output tuple:

```
(Session# | ID  | “Aggression analysis” | Aggression | AggressionCounter |
TupleID)
(int      | int | String               |boo         | int               |
int    )
```

Rudeness Agent

Input tuple:

```
(Name   | SentenceOpener | Request | Session# | AnswerQuality | ID  )
(String | String         | String  | int      | int           | int )
```

Functionality:

The agent is reading the input tuple. He takes the SentenceOpener and the PlayerInput and searches for rudeness.

Rudeness is recognized by using swearwords that are stored in a text file

If the agent finds rudeness, he sets rudeness to true and increases the counter.

We use Counter tuple to get access to the last rudeness count.

Counter tuple:

```
(Session# | “Rudeness-Count” | Name   | RudenessCounter)
(int      | String           | String | int            )
```

After that the Agent can write the Output tuple with the new extracted information.

Output tuple:

```
(Session# | ID  | “Rudeness analysis”   | Rudeness   | RudenessCounter   |
TupleID)
(int      | int | String               |boo         | int               |
int    )
```

No-go Agent

Input tuple:

```
(Name   | SentenceOpener | Request | Session# | AnswerQuality | ID  )
(String | String         | String  | int      | int   | int )
```

Functionality:

The agent is reading the input tuple. He takes the SentenceOpener and the PlayerInput and searches for No-go’s.

No-go’s are Phrases you should not use in a customer discussion. These Phrases are stored in a text file.

Rudeness is recognized by using swearwords that are stored in a text file

If the agent finds No-go he sets No-go to true and increases the counter.

We use Counter tuple to get access to the last No-go count.

Counter tuple:

```
(Session# | “No-Go-Count” | Name   | nogoCounter)
(int      | String        | String | int        )
```

After that the Agent can write the Output tuple with the new extracted information.

Output tuple:

```
(Session# | ID  | “No-Go analysis”      | Nogo       | NogoCounter       |
TupleID)
(int      | int | String               |boo         | int               |
int    )
```

Politeness Agent

Input tuple:

```
(Name   | SentenceOpener | Anfrage | Session# | AnswerQuality | ID  )
(String | String         | String  | int      | int   | int )
```

Functionality:

The agent is reading the input tuple. He takes the SentenceOpener and the PlayerInput and searches for politeness.

Politeness is recognized by using polite words or phrases that are stored in a text file

If the agent finds politeness it sets politeness to true and increases the counter.

We use Counter tuple to get access to the last politeness count.

Counter tuple:

```
(Session# | "Politness-Count" | Name   | PolitenessCounter)
(int      | String            | String | int              )
```

After that the Agent can write the Output tuple with the new extracted information.

Output tuple:

```
(Session# | ID  | "Politeness analysis" | Politeness | PolitenessCounter |
TupleID)
(int      | int | String               |boo         | int               |
int    )
```

Steps Agent

Input tuple:

```
(Name   | SentenceOpener | Anfrage | Session# | AnswerQuality | ID  )
(String | String         | String  | int      | int   | int )
```

Functionality:

The agent is reads the input tuple and increases its counter every time

After that the Agent can write the Output tuple with the new extracted information.

Output tuple:

```
(Session# | ID  | "Steps analysis" | StepCounter | TupleID)
(int      | int | String           | int         | int    )
```

Collecting Agent

Input tuple:

```
(Name   | SentenceOpener | Anfrage | Session# | AnswerQuality | ID  )
(String | String         | String  | int      | int   | int )
```

Functionality:

The Agent is waiting for the tuples of the preprocessing agents and for the messagetime-tuple additionally sent by the client holding the time a user needs to answer to the bot.

Waiting tuples:

```
Step-tuple:
(Session# | ID  | “Steps analysis” | StepCounter | TupleID)
(int      | int | String           | int         | int    )

No-go-tuple:
(Session# | ID  | “No-Go analysis”      | Nogo       | NogoCounter       |
TupleID)
(int      | int | String               |boo         | int               |
int    )

Politeness-tuple:
(Session# | ID  | “Politeness analysis” | Politeness | PolitenessCounter |
TupleID)
(int      | int | String               |boo         | int               |
int    )

Rudeness-tuple:
(Session# | ID  | “Rudeness analysis”   | Rudeness   | RudenessCounter   |
TupleID)
(int      | int | String               |boo         | int               |
int    )

Aggression-tuple:
(Session# | ID  | “Aggression analysis” | Aggression | AggressionCounter |
TupleID)
(int      | int | String               |boo         | int               |
int    )

MessageTime-tuple:
(Session# | ID  | “MessageTime analysis” | MessageTime)
(int      | int | String                 | long       )
```

It is collecting all the new data and extends the input-tuple to generate the new output-tuple. The tuple also registers if the answer was sent by the SilenceAgent and sets a Boolean named Silencetrigger.

Output tuple:

(Name | SentenceOpener | Anfrage | Session# | AnswerQuality | ID | rudeness | RudenessCounter | Aggression | AggressionCounter | politeness | politenessCounter | Nogo | NogoCounter | StepCounter | MessageTime | Silencetrigger)

(String|String|String|int|int|int|bool|int|bool|int|bool|int|bool|int|int|long|bool)

AimlBot Agent

Input tuple:

(Name | SentenceOpener | Anfrage | Session# | AnswerQuality | ID | rudeness | RudenessCounter | Aggression | AggressionCounter | politeness | politenessCounter | Nogo | NogoCounter | StepCounter | MessageTime | Silencetrigger)

(String|String|String|int|int|int|bool|int|bool|int|bool|int|bool|int|int|long|bool)

Functionality:

The Aiml uses the different information that it reads from the input tuple, to decide about the response of the AimlBot. The possible answers can be load from the AIML-Files that are stored inside the package. If a user was aggressive, rude or silent the Bot will tell this and repeat the last Botanswer. Otherwise it will choose a new Answer that is leading further through the conversation.

Output tuple:

Tuple for Score:

(Name | SentenceOpener | Anfrage | Session# | AnswerQuality | ID | rudeness | RudenessCounter | Aggression | AggressionCounter | politeness | politenessCounter | Nogo | NogoCounter | StepCounter | MessageTime | Silencetrigger | Response)

(String|String|String|int|int|int|bool|int|bool|int|bool|int|bool|int|int|long|bool| String)

Tuple for client:

(Name | SentenceOpener | Anfrage | Session# | AnswerQuality | ID | rudeness | RudenessCounter | Aggression | AggressionCounter | Response)

(String|String|String|int|int|int|bool|int|bool|int|String)

Score Agent

Input tuple:

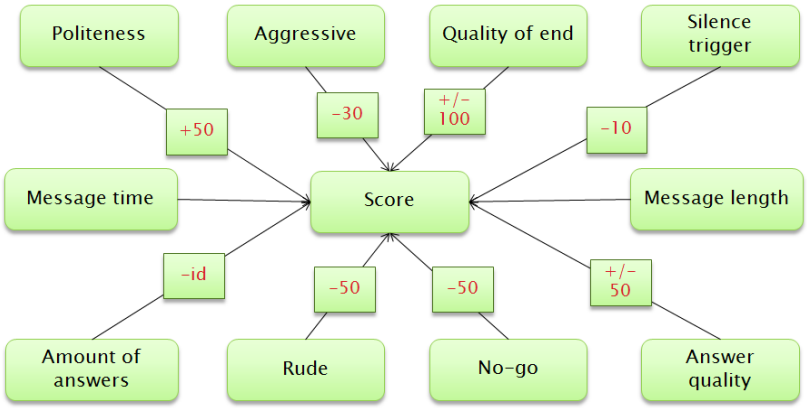
```
(Name   | SentenceOpener | Anfrage | Session# | AnswerQuality | ID  |
rudeness | RudenessCounter | Aggression | AggressionCounter | politeness |
politenessCounter | Nogo | NogoCounter | StepCounter | MessageTime |
Silencetrigger | Response)

(String|String|String|int|int|int|bool|int|bool|int|bool|int|bool|int|int|l
ong|bool| String)
```

Functionality:

The Score Agent is using all this information to calculate a Score.

The Graphic below shows how the score is influenced by the different properties.



After the Calculation the score is added to the tuple and written to the SQLSpaces.

Output tuple:

```
(Name   | SentenceOpener | Anfrage | Session# | AnswerQuality | ID  |
rudeness | RudenessCounter | Aggression | AggressionCounter | politeness |
politenessCounter | Nogo | NogoCounter | StepCounter | MessageTime |
Silencetrigger | Response | Score)

(String|String|String|int|int|int|bool|int|bool|int|bool|int|bool|int|int|l
ong|bool| String|int)
```

Serialization Agent

Input tuple:

```
(Name    | SentenceOpener | Anfrage | Session# | AnswerQuality | ID   |
rudeness | RudenessCounter | Aggression | AggressionCounter | politeness |
politenessCounter | Nogo | NogoCounter | StepCounter | MessageTime |
Silencetrigger | Response | Score)

(String|String|String|int|int|int|bool|int|bool|int|bool|int|bool|int|int|l
ong|bool| String|int)
```

Functionality:

The Serialization Agent is responsible for saving all data that is important for the Group Reflection tool. The information is stored in an xml-file which is used as an exchange format and can be read by the client. In the following you can see how the xml-file is structured.

```
<Root>
    <Message>
        <MessageID>1</MessageID>
        <PlayerName>Player1</PlayerName>
        <SentenceOpener>"Hi, my name is"</SentenceOpener>
        <PlayerInput>"Jamie"</PlayerInput>
        <BotResponse>"I have a problem with my order, Jamie."</BotResponse>
        <Aggression>false</Aggression>
        <Politeness>false</Politeness>
        <Rudeness>false</Rudeness>
        <NoGoSentence>false</NoGoSentence>
        <MessageTime>6801</MessageTime>
        <BotDelay>2000</BotDelay>
        <Answerquality>2</Answerquality>
        <SilenceTrigger>false</SilenceTrigger>
        <ScoreOverall>22</ScoreOverall>
    </Message>

    <Message>
        ...
    </Message>

    ...

    <Result>
        <PlayerName>Player1</PlayerName>
        <ResultQuality>6</ResultQuality>
        <AnswerAmount>5</AnswerAmount>
        <OverallTime>176801</OverallTime>
    </Result>
</Root>
```

Register Agent

Input tuple:

```
(Name   | hasSzenario)
(String | boolean)
```

```
Additional Waiting tuple:

("SessionNumber"|session)
(String |          int)

(Name|troublecounter|complaincounter)
(String| Int | Int)
```

Functionality:

The Register Agent provides a new Session for every Player and would later on decide which szenario the user will have to solve.

Output tuple:

```
(Name|Session|troublemaker)
(String|Int|bool)

("SessionNumber"|session)
( String |          int)

(Name|troublecounter|complaincounter)
(String| Int | Int)
```

Silence Agent

Input tuple:

```
(Name   | SentenceOpener | Anfrage | Session# | AnswerQuality | ID  |
rudeness | RudenessCounter | Aggression | AggressionCounter | Response)

(String|String|String|int|int|int|bool|int|bool|int|String)
```

Functionality:

The Silence Agent is starting the timer that starts every time after the AimlBot Agent is writing a touple. After a certain time, if there was no tuple written by the client which means that the user did not write a message, the Silence Agent will provide a silence tuple. The AimlBot is then reacting to it and write its response.

Output tuple:

```
(Name   | SentenceOpener | Anfrage | Session# | AnswerQuality | ID  |
rudeness | RudenessCounter | Aggression | AggressionCounter | politeness |
politenessCounter | Nogo | NogoCounter | StepCounter | MessageTime |
Silencetrigger)

(String|String|String|int|int|int|bool|int|bool|int|bool|int|bool|int|int|l
ong|bool)
```

GAME CLIENT Front-End: Web Application Functions

(Author: Dorian Doberstein)

- Interface between player and backend
- Contains chat environment and search functionality
- Introduction text in the beginning of the game introduces the player to the scenario and functionality of the game client
- Messages are sent from the client to the backend with the chattextsenden() function, which also assigns the answerquality depending on the chosen sentence opener
- After the backend processed the data, it is sent back to the client and chatfunction() is called
- chatfunction() processes the data for special cases (player was aggressive, rude or the message was caused by the silence trigger), then it sets the corresponding state and displays the fitting sentence openers
- The message time is calculated in the game client (time is measured when chatfunction() is called and when chattextsenden() is called and calculates the difference) and is sent to the backend together with the rest of the data (playername, sentenceopener, playerinput, sessionID, answerquality, MessageID)
- The search buttons in the chat environment call the corresponding functions (e.g. checkname()), which check the playerinput in the search fields
- When the player has found the customer information he can send this information to the customer with the found_customerinfo button which calls the function foundcustomerinfo(), so the function foundcustomerinfo() also sends a message directly to the backend
- When the player leaves the game early by pressing the “End conversation” button, the function gameover() is called which sends a message with answerquality=6 to the backend, which signals the end of the game and the game data is serialized
- If the player choses a sentenceopener with a corresponding answerquality of 3,4 or 5 the game has reached an end-state, is automatically serialized and gamecomplete() is called which hides most of the game functionality except for the window, which contains the chat conversation

Group Reflection Tool Architecture and Functions

(Author: Marco Bäumer)

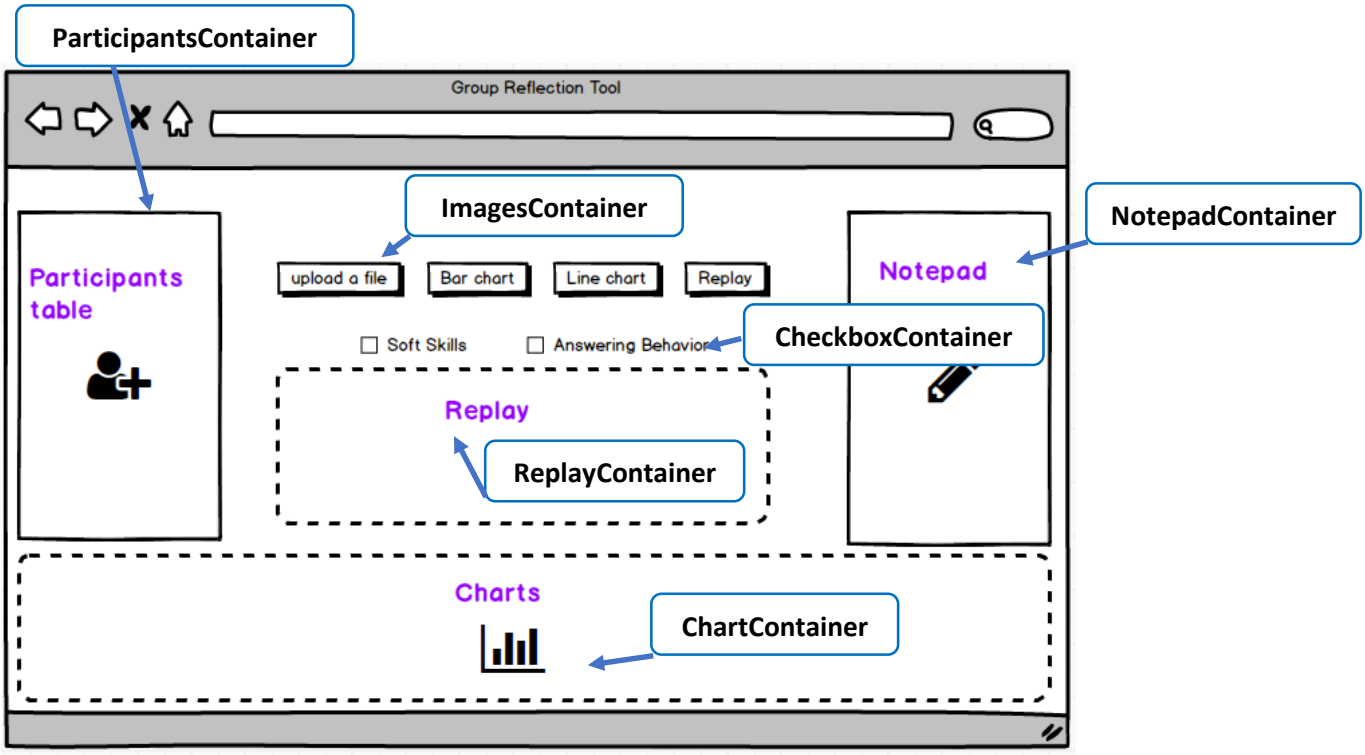


Figure 1. Mockup of the Group Reflection Tool with the site structure of the group-reflection-tool.html

(1) Requirements:

- **Browser: Mozilla Firefox strongly recommended**, mainly because of some css-features and the interpretation of the httpXMLRequest method interpretation for reading the participant files
- **Browser with cookies allowed**
- **Internet connection recommended**, since you have more possibilities to export the charts

(2) General structure of the Group Reflection Tool

File	Purpose
Group-Reflection-Tool.html	Structure
Data center.css	Design
/js/Group-Reflection.js	Program logic

The whole application consists of one html site, in which the unused elements are displayed out. The notepad can deal with HTML code, so the tool´s design implementation respects this. Thus, cases of breaking the separation of layout and design are necessary and will be mentioned.

(3) Description of the program logic located in Group-Reflection.js

Action	Involved methods	Description
a. Open a file		
	open_File_Dialog()	Virtual click on a hidden file input element for opening a browser dialog
	handle_Selected_File(event)	After choosing the file -> XMLHttpRequest for getting access to the xml file as a NodeList-Object Proofs if the player is already loaded -> if not, throw an alert to the user Reads the players messages and results Prints the player´s name and score on the screen Causes a rewriting of the replay (open_Replay()) or the chart (updateChart(„“)), depending on the state of the site from which the file was loaded Calls the notepad for adding the new player
	prepareDataCausedByFirstPlayerMessage(messagesListArray, xmlDocument)	Separates the first message from the rest, necessary since the first message is minor of interest and the graph needs a list containing of the messages to display and to jump in Decreases the message ids since it would be illogic for the user why they begin at 2 instead of 1
b. Showing a chart		
	updateChart(chartType)	Hides the replay container, displays the ChartContainer (openDataCenter()) Calls the draw function of the bar (plot_BarChart()) or the line-chart (plot_LineChart()), depending on the user wanted type If no parameter is given (if the player loads a new file), the method updates the already shown chart type, so it will be auto updated

Action	Involved methods	Description
c. Displaying a chat history **Jump to a specific message from the line chart into the replay**		
	open_Replay()	Hides the ChartContainer Changes the player representation into radio buttons Adds the player overall score
	printReplayData(selectedPlayerName, mID)	Prints the chat of the selected player The method is called for both purposes, simply printing the chat history of a player and for jumping into a specific message from the chart as well The player messages get the same id which they have in the x-axis in the line-chart so they can be jumped in from there Adds the „copy into notepad“ button, which will only be displayed if the user hovers over them If the function is called with a message id (mID) unequal 0, which is the case if the player clicks on a message in the line graph, the message will get a colored background by inline css and the window scrolls to the specific message automatically The needed amount of answers are reduced by 1, because the first one is a program generated message
	measureTrue(i, j, Measure)	Helper method, which gives back - The text for the specific chat ending - The text for the specific answer quality - The measures text if they are true in the player file The measures are colored inline, so they can be printed from the notepad in the pdf later on The sentence opener and the player input are reformatted to the schema: „sentence opener player input“

Action	Involved methods	Description
d. Copy a message into notepad		
	copyIntoNotepad(divId)	Copies the html code of the message into the notepads visible tab Deletes the „copy into notepad button“ before copying to prevent appearing in the later on PDF
e. Generating a PDF document		
	convertToPDF()	Gets the HTML content out of the tab Generates a PDF with the fromHTML-Method from the jsPDF library

Group Reflection Tool : Charts

(Author: Diba Heidari)

extract_data_from_DataBase():

- This function converts the initial 2-dimensional array of [player, messages] to a one-dimensional array of [messages], which is then searched based on the player names and relevant data is extracted.

plot_BarChart():

- The function takes the list of the selected measure which are going to be plotted as the input.
- **answers[n]** is a 2-dimensional array which stores the number of time each of the selected categories has been true for each player.
- The bar chart is plotted using Highcharts and by taking the data from **barData**.

plot_LineChart():

- The function takes the list of the selected measure which are going to be plotted as the input.
- A color and a symbol are assigned to each measure, which will be used to create the scatter charts and generate the tooltip.
- **plotObject.seriesData** is drawn by Highcharts function. The tooltip is generated dynamically by **txtBox**, which has been returned by the **lineChartFormatter()** function.
- The clickability of the chart is defined in **plotOptions**, under **events**.

lineChartObject():

- This function takes the **selectedCategories** and the **selectedColor** arrays as the input to create the chart object.
- **legendItemClick: function(event)** controls all the charts of each player (the line chart and the scatters) by clicking on the player name in the legend (shows/hides all of them) .
- **TrueArray** stores the data to be drawn in the scatters, it stores the points were each specified measure was true (for AnsweringQuality measure, the value is taken into account).

- **LineData** is used to draw the scatters (for the selected measures). To be able to control all the scatters of each player with a single click on the player name in the legend, it has been distinguished if only one measure is selected or more, and based on that the scatter is drawn.

lineChartFormatter():

- This function takes the PlayerName, MessageID, selectedCategories, selectedColor as the input and generates the tooltip text and its format by using some HTML tags and returns **txtTemp**.

Group Reflection Tool: Notepad Editor

(Author: Shaghayegh Abdollahzadegan)

notepadEditor()

- Creates an instance of notepad. It is possible to add or remove the buttons in the editor's panel by using different type of panels as full panel or normal panel. This configuration is also changeable from nicEdit.js file by using the buttonList.

var myNicEditor = new nicEditor()

- Creates a new nicedit object. A single instance of nicEditor contains:
  - 1 or more editor instances (nicInstance/nicIFrameInstance)
  - 1 nicPanel

panelInstance('area')

- Creates an inline content editor with attached control panel on top of the element.

clearNote()

- Clears the notepad area by deleting all li's for tabs

Cookies.set(pName, content, { expires:365})

- Stores the name of the player in a cookie variable. The parameters of the function above are the name of the cookie (pName), the value of the cookie (content), and the number of days until the cookie should expire (expires) which is here been set for 365 days. The function sets a cookie by adding together the cookie name, the cookie value, and the expires string.

Cookies.get(player_name)

- Returns the value of a specified cookie with name player_name.

createTab(pName)

- To create tabs, it uses a list and CSS stylings. The tabs are based on an unordered list with ID #tabs. Each element will be added as a li element with ID #pName to the #tabs list. If the created li is not the first tab, a class inactive will be added to that. Each tab has a class inactive or has no class. The li with no class, is the active one. After adding a new li for the player

pName a text area with ID #tabpName will be created as well to create an instance for notepad editor.

notepadEditRadio(playerName)

- Has the same functionality as notepadEditor() function but is used as an instance creator. The difference is when the replay button is clicked. This button changes the participant table elements to radio button. So the elements ID are different. This unction uses different ID's to create the panel instances.