XML Technology - WS19/20 - BlackJack Project - Technische Universität München (TUM)

Mr. Ikbal Yesiltas

Mr. Lorenzo Brazzi

Mr. Markus Zuber

Mr. Patrick Reto

XML Technology - WS19/20 - BlackJack Project - Technische Universität München (TUM)

Mr. Ikbal Yesiltas
Mr. Lorenzo Brazzi
Mr. Markus Zuber
Mr. Patrick Reto

Table of Contents

Blackjack XML Web Application

This docbook describes and documents the design, implementation and testing of the BlackJack webapp game written in the XDM programming languages. It specifies the decisions made regarding the architecture, user requirements and their corresponding solutions, and thus captures anything that was relevant to understand and implement the web application, covering the game mechanics and used architectural style through to even low-level essential functions and emerged challenges. Furthermore, we will give an overall summary and reflection on the practical course and on the development process itself, that came with it. Finally, a step-by-step run through of our blackjack program will be provided with detailed highlighting and explanations of the UI and Backend features.

Chapter 1. Technologies

Following Languages and Tools that were used to build this Project.

XML

Since we are dealing with the development of an XML blackjack game, the Extensible Markup Language cannot be dismissed as a major component of our project. It is a markup language that defines a set of rules for encoding documents in a clearly arranged way. Due to its simplicity and generality, it finds its primary usage in the representation of documents as well as data structures, commonly used in web services.

As our specific project is concerned, XML defines all the basic files and structures that are used by different technologies later on. These are realized in form of basic, static XML files that are modified dynamically during the course of the game. Obviously, since we use BaseX, XML builds the Database of our project. All of the data generated by our game is saved in a single XML file composed by a strict structure of elements and attributes that play a role in realizing the logic of our game.

XQuery

XQuery is a functional and querying language primarily designed for the extraction of data, usually in the form of XML, and the interaction between the world wide web and databases. It was developed by the XML Query working group of the W3C and is closely related to XSLT, which transforms the queried data. XPath, which is a subset of XQuery and XSLT, is used for querying and selecting nodes in XML formatted data.

As far as our own project is concerned, the XQuery modules make up the most part of the game's logic as well as the access to the database. Basically, anything that needs to be processed and calculated is performed by XQuery. All the XQuery modules are marked as .xqm files. Each of them store functions that insert/delete into or modify data from the DB.

SVG

SVG is the abbreviation for Scalable vector graphics. It is the technical standard for two dimensional vector graphics and is recommended by the W3C. SVG rendering is supported by all major internet browsers, like chrome, firefox, safari etc. In this project SVG was used for all visual game elements like cards and chips. SVG has a lot of useful features like defining groups (<g/>), where, by giving an name (id) to the group, all elements in this group can be transformed and scaled together. Or <def/>, which doesn't render elements instantly, but stores them, to use them later with <use/>. By combining these two we could evade a lot of redundancy. For example cards have a lot of figures or positions in common (twos and threes only differ by one symbol and the cipher), so by storing the framework, we could easily create the next one. Scaling is also very easy with SVG. We outsourced a lot of standard variables(e.g. player positions, card dimensions) in „general variables“ and referred to them later, so they could be easily adjusted everywhere, by just changing them in one place. SVG makes it also possible to create complete individual graphical objects with <path/>. In this project an example would be the pattern of the chips. With <text/> text can be visualized for every user to see. It can be predefined or flexible, which we made use of, by reading changing balances or bet-values off the data bank. For the buttons we defined an external stylesheet. That made it possible to creat them individually(e.g. color,size,etc.). Each of them has its own functionality and they are important tools for the players to interact with the game. All in all SVG makes it possible to customize the entire GUI.

XSLT

XSLT is a turing-complete transformation language for XML documents. It transforms XML documents into other XML documents as well as into other formats, of which HTML is the widely spread

use case. It is part of the Extensible Stylesheet Language (XSL). The XSLT processor, a piece of software, is responsible for performing the transformation instructions given in the stylesheet. The workflow is as follows: The input for the xsl processor is the xml document and the xslt stylesheet. Then this document is transformed by the template rules defined in the stylesheet. Finally, the output is a new document, that is compliant to the desired structure/style.

For our project, the heavy use of the xsl processor was undeniable. The whole view, i.e. html output for the client, is created by manipulating the necessary data with XQuery beforehand, and then transforming it through the use of the stylesheets into a presentable html document.

HTML

HTML stands for "Hypertext Markup Language”. It is the standard scripting language for writing documents designed to be displayed in a web browser. Web browsers usually receive HTML documents from some kind of web server and then render its elements on to the page. The HTML for itself is just text, but can be styled and animated by other scripting languages like CSS or JavaScript. HTML is similar to XML as it is built by elements. The difference is that HTML elements are mostly predefined and have a specific meaning whereas XML elements can be whatever the author decides to structure his file with. In our Blackjack app we only used HTML passively. This means that we did not create HTML files ourselves. Instead, we used XSLT to transform the XML database to HTML through a XSL file, every time when changes happened to the database that need to be displayed to the user. The style in this case is given by applying both CSS-styling and XSL-styling.

DocBook

The whole documentation for this project was written in DocBook. DocBook is an XML file that has the advantage of allowing transformation into many different formats, e.g. PDF or HTML. Since it is an XML document and already established as a standard for documentation, it has a validation mechanism through DTD. This decreases error probabilities and provides more clarity and uniformity, especially for the transformation process.

BaseX

BaseX is a native XML database management system and XQuery processor, specialized in storing, querying and visualizing large XML documents. Since an huge XML file is used to store data and therefore represents the whole database, BaseX is a document-oriented database. It is specialized on XML documents; therefore, it supports query languages such as XPath and XQuery. It has a command line interface to start the BaseX server itself and both a User Interface and a web interface to interact with the single databases. I.e., BaseX is a web server for xml related web apps. It provides APIs for sending and receiving data, such as RestXQ. BaseX STOMP, a release of BaseX that enables the server to realize a STOMP-communication with the clients, plays a big role in our project since we used the STOMP framework to establish the connections of the clients websockets and server, while also allowing for subscription frames, in which the clients always receive view updates of the DB, very much like the subscribe-publish pattern.

Maven

Maven is a build management and build automation tool from Apache. It is primarily used for Java projects. Since we needed Java to compile compile the complete sources of the main BaseX-STOMP project, we were required to use Maven. Furthermore, Maven comes with a Webserver from Apache called Jetty. We used Jetty to enable multiclient communication.

Chapter 2. Design Choices

In the following section the visual game elements will be illustrated in more detail.

Table

The table consists of simple circles and is laid out for up to 5 players. Every player has an own spot for their cards and bets and is provided with enough space for their names and balances to be displayed. In the center of the table, with good visibility for each participant, there is a larger placeholder for the cards of the dealer. Underneath, the conditions for blackjack-wins and insurance are showcased.

Cards

Each card contains three elements:

- Color
- Value
- Turned

The deck consists of the classic french style playing cards, i.e. there are four kinds of colors: Spades, Hearts, Diamonds and Clubs and the values range from 2 to Ace with the Court cards equaling to 10. The value of each ace is determined individually. Depending on the value of the other cards in each players or dealers hand an ace counts as either one or eleven, whichever benefits the play more. "Turned" is a boolean flag determining whether a card is shown or not, and actually is only relevant for the second card the dealer draws at the start of every round, since its the sole card that is shown upside down. All the others are shown normally. All card designs were put together manually (except for the court cards) and are automatically scaled by changing the width of the cards. This was achieved by first implementing the single symbol and then from there on transforming it to build the other numbers , e.g. simply build the "two" cards by doubling, scaling and positioning the single symbol and use that particular for the "four" cards symbols, which is then used for the "five" cards symbols, etc. This also provides an abundance of redundancy, which was quite important, as we, every now and then, could therefore perform some slight tweaks to our designs very time efficiently. Our deck of cards consists of six standardized decks, which is common for Blackjack. After each round the deck is shuffled, so new random cards can be dealt out.

Chips

Similar to the cards, the design of the chips was also hand-crafted as a svg. The chips represent the current bet of each player with the value being displayed on top of them. For a more appealing visuals, we used colors to differentiate the values into clusters, e.g. values below 100 will be displayed as blue chips, whereas values above 5K are black. These color changes also simulate a casino-feeling and makes the game slightly more immersive.

Possible Chip colors:

- For values 0...100: blue
- For values 101...500: green
- For values 501...1000: purple
- For values 1001...5000: red
- For values over 5000: black

The following sections will give further insights into the game design and how players can interact with the game.

Lobby

For a multi-client game a lobby is indispensable. It’s the starting point for each player. Within the lobby, You, as a player, can either create a new game, or join an existing one. To join a running game from the available games list, you have to simply enter your name and then click on the join button. If you wish to create a new game, you have to additionally define the range of betting for the new table with a minimum and maximum. The new game can be then joined by others. Games are available, if there is still an open slot and not all players have confirmed the ready-check for a new round.

Course of the game

Whenever a player joins a game he is provided 10000 balance. Every player plays for himself against the dealer to maximize their balance. Regardless of their own play, each player has to wait for the rest of the table to perform their actions, before they can execute their own ones. Every player has to realize at least one action for each game state. There are four of them:

- Bet
- Hit / Stand / Double (/ Insurance)
- Continue or Leave
- Ready Check

At the beginning of each round, every player has to place a bet. The amount cannot exceed a players balance and has to be compliant to the minbet/maxbet conditions of their respective table. The bet will be subtracted from the players balance and visualized in form of a chip in front of each player as soon as the betting stage is over.

When all player have set their bets the playing cards are distributed. The players and the dealer receive two cards each, in which one of the dealers card will be shown upside down. Now, sequentially the players actually play out their turn until they press the stand button or surpass 21 with the value of their hand. If they hit, they are provided an additional card, if they double they get another card and double their bet. If the first card of the dealer is an ace the players are given the option to buy an insurance. If they decide to buy an insurance, they only lose half of their bet if, and only if, the dealer hits a blackjack. The cost of the insurance adds up to half of the value of their current bet.

After all players have made their moves it’s the dealers turn, who flips the second card and if this card ´s value is below 17 another hit has to be committed until that’s not the case anymore. Afterwards, the outcome will be calculated for each player individually. Their winnings (and possible insurances) will be payed out. Then they can decide if they want to continue or leave the table.

Last but not least: the ready check. By now the round cycle has been finished and a new one starts. This is the phase, where new players can join the table. If all players are ready, the next round will start with a new betting-stage.

Chapter 3. Architecture

MVC Architecture

We designed our application around the widely used Model-View-Controller design pattern, which finds its usage on a number of webapp scenarios like this project, because of the high cohesion and loose coupling it provides. It can be perfectly tailored to our needs, since we use a XML database as well as a BaseX server in the backend. The MVC is mainly used for separating concerns and abstraction, which in our case would be the main logic of the game (Model), the corresponding UI (View) and the handling of the requests (Controller). As you can see in the class diagram below, the controller represents the interface between the backend model and the user`s view (client Browser). Especially the testing has been significantly made easier by a sophisticated usage of the controller.

Our application follows a classic client-server architecture with following characteristics:

- The client runs in a web browser.
- The server runs in a web server (Jetty)
- Client and server communicate through HTTP requests and responses.

We used the RestXQ API to facilitate XQuery as a server-side processing language, i.e. to interpret and map the HTTP Requests from the client (GET and POST) to the XQuery functions of the Server. To do so, we simply annotated the functions in the controller with %rest:path("somePath") and %rest:HTTP-method. E.g. %rest:path("bj/setup") and %rest:GET for interpreting the an incoming setup request from the client. But since each function can have parameters, those annotations alone were not sufficient. We needed a way to provide parameters, especially the gameID or playerID, within the request url. RestXQ provides such an option by simply adding the variable within the request url, enclosed in bracelets. For instance, %rest:path("bj/continue/{$gameID}/{$continue}") stores the gameID and the boolean information to be used for the function setContinue, that it is mapped to.

Our whole application server has only one response. That is, the server responses with the visual transformation of each user request that alters the game instance in the DB. We realized that by using the update:output and web:redirect methods provided by BaseX.

Modules

The following sections will describe the modules used to implement the functionality of the Blackjack game.

Controller

The controller module is the first instance in the chain of operations. As the name suggest, it controls the whole flow of the application by sending/receiving REST requests. Each function in the controller is mapped to an unique url that calls the corresponding function of the model or creates a new view of the changed model. The controller was a very helpful entity when it came to testing, since we could send specific "Test" GET requests through URLs. We will present the most important "controller-exclusive" methods.

- setup, startingPage

  The 'setup' function is the first one that gets called when startig the webapp. It helps the user to install the game by simply typing in the url in the browser, hence no DB instantiation or manual connection steps are needed. I.e. it automatically creates the XML database and redirects to 'startingPage'. This method also transforms the lobby via XSL to be able to show e.g how many games are available to join and which are closed.

- startGame

startGame ensures that the min and maxBets, given by the user, are compliant to the rules, e.g. gt0, min lt; max, etc.

- join

  Through this function´s url the client will subscribe and connect to the channel with the corresponding game and player ids and therefore will consistently be updated, each time the draw function is called.

- draw

  Represents the View of the MVC style. Every time the state of the database is changed, this function is called, so that it can transform the game XML into HTML via XSLT. This HTML is then sent to every user that is subscribed through the websocket element.

Game

The core class of the model, which consists of the game, player, and dealer module. It contains most of the game logic in regards to blackjack rules and non-player-related actions, such as shuffling the deck, determining winners, paying out, etc.. It is responsible for determining the states of the game as well as assigning players as active/inactive.

- createGame, insertGame

  These two functions are responsible for creating a new game instance, if a player wants to create a new game, and insert that game instance into the XML database.

- setActivePlayer

  This is the core function/mechanics of the whole system. Besides giving play access rights to the next active player, it handles every end of a state (call by the last player in the sequence), expect for the betting state, since this one is regulated by the controller). I.e. when each player has played their round, it calculates winners and losers, after that, gives each player their deserved payouts and finally it resets the table, deletes players that wanted or had to, assigns the players to their correct seat position and puts the state into "ready", thus ending the cycle, so that new players can join again. After handling each ending of a state, this function always sets the first player as active again. So that the new state can be applied to each player.

- popDeck

  This function is essential for the process of dealing out cards to the players. It's main functionality is very simple: it deletes the first card from the deck.

- drawCard

  Mainly used alongside 'popDeck' since we want to give the player the first card from the top of the deck.

- dealOutCards

  This method is responsible for giving each player and the dealer 2 cards respectively, to start off the game. It draws the card from the deck with respect to the position of each player by retrieving the Nth and Nth-1 card from the deck where N = playerPosition * 2. The dealer always gets the 11nth and 12th card from the deck since there are never more than 5 players. This is rather simplistic, but it did work quite well for us.

- setResult

  This method is called every time when the "play" round is completed and all the players have played against the dealer. Its job is to update the 'won' tag of each player instance respecting the outcome

of the game. All the cases are evaluated by comparing the card value of the dealer to the card value of each player. The player can either win, lose or draw with the dealer.

- evaluateRound

  Like the name of this function suggests, it is responsible for checking the result of the 'setResult' function, i.e. by checking the 'won' element of each player instance and applying that to determine the player's resulting balance. Here, we followed the rules implemented in the standard Blackjack game which suggest a payout of 3 to 2, if the player won by Blackjack, and a return of the placed bet plus the same amount added, if it is a normal win.

- isRoundCompleted

  This function returns true if the activePlayer is last player in the sequence. This is important for changing the game state.

- shuffleDeck, setShuffledDeck, getDeck

  The 'shuffleDeck' function retrieves a set of 6 card decks (standard in online blackjack is between 6 and 8 decks) and shuffles the cards inside them to a single deck, which is modeled as a 'cards' element, that will be used for the whole game. That deck is then inserted in the right game instance through the function 'setShuffledDeck'. We also implemented a function 'getDeck' to make it easier to retrieve the deck instance from the database since this is a procedure that is needed by various other functions.

- finishRound

  This function is the last function that is called when all players have played and a new round is ready to be started. The method itself has 3 function calls: resetTable, assignPosition and prepareGame. The names of these methods are pretty self-explanatory but we will go more in to detail about them later. Effectively, what it does is to assign new positions to the players that want to stay in the game and keep playing a new round, delete the players that want to exit and free up the slots for new players that want to join the current game.

- resetTable

  This method closes the round by deleting all the players that don't want or cannot affort to play the next round, resetting all players' won tags to set back to false (standard) and deletes all the cards from de hands of players and dealer.

- assignPositions

  We defined most our functions in a sequential manner, i.e. for our implementation design it is necessary that there are no "gaps" between player positions, since every function (especially the draw function) assumes that the player within the sequence position i is also the player with the same named tag "position" = i. Hence, we implemented this function, that simply rearranges player position after some left the game.

- prepareGame

  This method is called right after 'assignPositions'. Its purpose is to prepare the game instance to be ready for a new round. This includes setting the state to the initial game state ('ready'), if there are still players left, otherwise it changes the state to "deleted". It also checks if 5 players are present and sets the availability to join the game to false. In this case no other player would be able to join that game.

Player

This module implements player based actions, such as betting, leaving, hitting, doubling, drawing cards, etc., as well as, calculation functions, e.g. to determine the score of the hand.

- createPlayer

  Just like 'createGame', this method creates an default instance of a player with respective ID, name (only non-default value), balance, bet, insurance and position on the table.

- setBet

  This function's task is to check if the bet that was entered by the user is valid, e.g. if they have enough balance or if the bet is bigger or smaller than the maximum and minimum bet. Subsequently, if the bet was found to be valid, it is added to the player's instance in the database, otherwise they simple have to type it again, but are notified by the log box in the upper right corner of the screen. If the player setting the bet is the last player in the sequence, the 'setBet' function triggers the change of the game state to the playing state and calls the process of dealing out the cards for everyone.

- stand

  A player can click on the stand button at any time during his turn to pass the turn to the next player. This function implements just that by simply calling 'setActivePlayer'

- double

  This is another Blackjack-specific action that the player can choose if he sees a high chance of winning against the dealer. The method implements this process of doubling the bet of a player in exchange for another card, i.e. after doubling it delegates to hit(). It respects the bounds of minimum and maximum bet by setting them as the player's bet if double of the bet would be too big or too little.

- hit

  With this function the user is able to draw a new card. First, it calculates the value of the cards the player already has. If that value has already passed 21, an error message will be displayed and the user will not be able to draw the card. If the player's cards value is equal to 21, the turn is automatically passed to the next player, since anything other than that makes no sense. In the case of the cards´ value being under 21, the player is able to repeat this action until any of the other two cases appear.

- setInsurance

  This function is triggered when the player has the option of buying insurance and chooses to do so. Then, from the player's balance, half of the current bet is subtracted since he has to pay that amount to realize the insurance. After that, the insurance element of that player's instance within a specific game is updated to true in the database.

- cardValueOfPlayer

  This function calculates the Score of the cards in the players´ hand. All cases besides the Ace are trivial, but since A can be either 1 or 11, a more sophisticated approach was chosen. First, the function counts the number of Aces in ones hand, followed by calculating the score of cards without aces. Finally, by using a left fold, the function counts aces as 11, as long as the final score does not go beyond 21. Since this function is called after each card is drawn, the calculations always return a valid assignment for the Aces!

- drawCard

  As the name suggests this function pops the card at the top of the deck and inserts it into the active player's hand.

- setContinue

  Player continuation function. This method processes the decision of the player to either keep playing or to leave the game. It also checks if the balance of that specific player is lower or equal to zero so that they will get kicked out automatically.

Dealer

Even though the dealer shares similar functionality as the players, it is a different entity to the game, thus it needs its own module, especially because of the additional functions that are needed to correctly evaluate the game and, since the dealer only plays, when all players have finished their turns, so it shouldn`t be within the same module. One of them would be, for example, that the card value of the dealer is the key to evaluating the outcome of the game for all the contestants. Or we had to change the approach with which we distributed the cards to the dealer since there is no user interaction an the dealer has to pick his cards by himself.

- numberOfDrawingCards

  Tail recursive function that determines how many cards the dealer has to draw until the score is above 16. This is done by continuously "pseudo-drawing" of the next card to draw from the deck and checking if the value is over 16.

- cardValueOfDealer

  Calculates the value of the dealer's hand. Because of the update constraint issues, we had to re-write the similar method from the Player module. In this module it takes an entire game element instead of the game ID.

- newCardValue

  This function is needed because the card Ace can be either 1 or 11. When this is called, the dealer has drawn a new card, thus the score of the hand must be recalculated in order to correctly assign the value for the Ace, which is 11, if not beyond 21, and 1 otherwise.

- drawCard

  This is another function that is ,in principle, similar to the 'drawCard' method in the Player module, that needed to be specifically adjusted to the dealer, since we implemented the "dealer has to hit 16" type of BJ game, so that the dealer has to automatically draw according to the current circumstances of the game. Therefore, we altered the old method with some tweaks tailored towards the dealer`s needs. It uses the the previously described functions 'cardValueOfDealer' and 'numberOfDrawing-Cards'. The former calculates the base value of the dealer's hand and based on that, the latter decides how many cards the dealer will have to draw. When the number is calculated, the dealer draws that amount of cards from the deck and deletes them from the stack.

- turnCard

  Turns the second card of the dealer that is turned by default when the cards are dealt out to all the participants.

- play

  Models the dealer's turn. Combines the actions of turning his second card and, if necessary, drawing some more cards.

Websocket

This module is essential for the realization of our multi-client application. It contains methods that map a single client to a specific game and player through a websocket.

- subscribe

  This method is responsible for the mapping a websocket, which represents the client, to the current game thorough a url that contains the game ID and player ID. This method is called when a player creates or joins a game, so that this player is subscribed to all the changes that happen within that instance in the database. This player is notified and updated every time changes are sent to this path.

- getIDs

  Returns all the IDs of websockets currently connected to the server.

- send

  Is called when changes happen in the backend, e.g. due to player action processing, that update the database and therefore need to be pushed to the individual websockets that are subscribed to the affected instances. This is important for pushing the newly updated game to the clients` browsers, so that users can actually see the changes to the table.

- get

  This method returns the key-value pairs of the websocket that store information about their subscription. E.g. application, gameID, or playerID.

- connect, disconnect

  These two methods are called when a client actually connects to an instance or disconnects from it. They can connect by creating or joining a game or disconnect by either deciding to leave the game at the end of the turn or by simply closing the window where the game session is played.

Helper

This module has only one method: 'currentTime'. It returns the exact time in which the method was called. This method is very helpful to us since we insert events into a notification box and supply the timestamp of this notification.

Chapter 4. Testing

We tested our implementations by simulating specific requests and test urls that were defined in the controller module. Through this, we could simply then insert the test url command in our browsers to see which results were yielded. In case of errors, the baseX compiler, most of the time, provided a convenient error message with the corresponding erroneous lines.

Before we fully implemented our GUI, we either tested by simply returning XML documents, which we then checked for their correctness regarding the expected behavior, or simply looked at the current state of the DB, in case of updating methods. Also to speed things up, we made use of automated tests via python scripts.

Throughout this project, we made use of the classical software testing phases. I.e. we first did test the individual functionality of the modules (unit testing), then tried to integrate them to other components in our system to see if they do not alter their behavior. Finally, after integrating, we always tested the system as a whole by trying to simulate a round of BJ until the next error shows itself, or to fully check if all changes work according to our expectations.

Chapter 5. Challenges

Technical and Logical Challenges that we came across.

XQuery Update Constraints

This is easily the biggest Challenge we faced during this project. XQuery, by its core, is a purely query and functional language. That means the standard procedures used in imperative or Object Oriented Programming languages, such as Java, cannot be fully applied in this use case. Fortunately, BaseX provides the so called Update Facility, that extends the Xquery language by expressions that allow for changing Instances in the Xpath Xquery Data Model. I.e. we can:

- Insert a node
- Delete a node
- Replace a node (or its value)
- Create a modified copy of a node

However, at first we still tried to use this as a sort of imperative language, which proved to be a fatal mistake. Update Expressions are stored in a pending update list. This means that all the update expression in there are performed, regardless of their order. But our game logic required the opposite, for instance, before we set the Results of the players, the dealer had to make his play, to determine the winners of the round. Only using update expressions lead to either logical mistakes similar to the one mentioned earlier or to complete errors, e.g. trying to modify a node, that should have been inserted beforehand. To bypass this problem, we made use of the paradigm of functional programming, i.e. try to bind the logic in a more mathematical style. Instead of trying to update the same Game over and over again (which only enlarges the pending list), we passed a copy of the game element as an input, performed all procedures on the copy, and returned the modified copy as an output. That way, all update functions within the copy statement are executed before delivered within the output. Coming back to our previous example, we then modified the dealer:play function such that it returns the game element instead of updating the DB. Subsequently, this modified copy of the game is given to the setResult function, which can now safely determine all winners correctly because the dealer already made the move.

Browser View

Sometimes when an Error regarding the browser view occurred, we tried to change it another browser, which solved the problem but ironically, provided a new one. It was very interesting to see how each Browser renders HTML code differently. However, we could not determine why exactly this happens and found out that even though all Browsers more or less provide the correct View, the Safari Browser is the most reliable one for us.

WebSocket Programming

Actually, there are no issues with the clients connecting to and disconnecting from the server. Strangely, if the first player leaves the game the clients browser should, as it does for the remaining players in the table if they leave, just show the table as a spectator. Logic wise, however, the procedures are correct and the connections are established or cut accordingly.

Coronavirus Pandemic

Because of the ongoing Coronavirus it was not possible for us, as a team, to physically meet in our most urgent/final phase of the project. The communication via Zoom and Skype was a big help, but it does not come close to the real thing. Our entire work environment being in our comfortable homes,

was holding us back more than we thought. Nonetheless, with current techincal advancement, it was still feasible to implement our project in the scope, we intended to achieve.

Chapter 6. Development

Development

In the beginning of this course, we declared which team member wants to take care of which main responsibility. Thus, 2 members were responsible for the front-end, and the other 2 for the back-end. This allowed our team members to mainly specialize, and therefore be more efficient in coding and developing. E.g. while the back-end team implemented the stand function, the front-end could implement the buttons at the same time, so that the final test could be more intuitive. But of course, there was a lot of cross sectional work, too, because often times we had to identify which part of the system, the GUI or Logic, did not work correctly, so the synergy of front and back end was very important for us.

Development Environment

All of our members used the IntelliJ IDE, because it covers all aspects of the XML Stack. It has a fully fledged support of the XQuery language as well as XSL, HTML, CSS and also version control via git. This allowed our team members to always be up to date, clearly see changes someone made and overall improve communication and workflow.

Basically, the full stack could be implemented solely via intelliJ, which came in very handy for us, because of our sectioning of the workflow in front- end back-end.

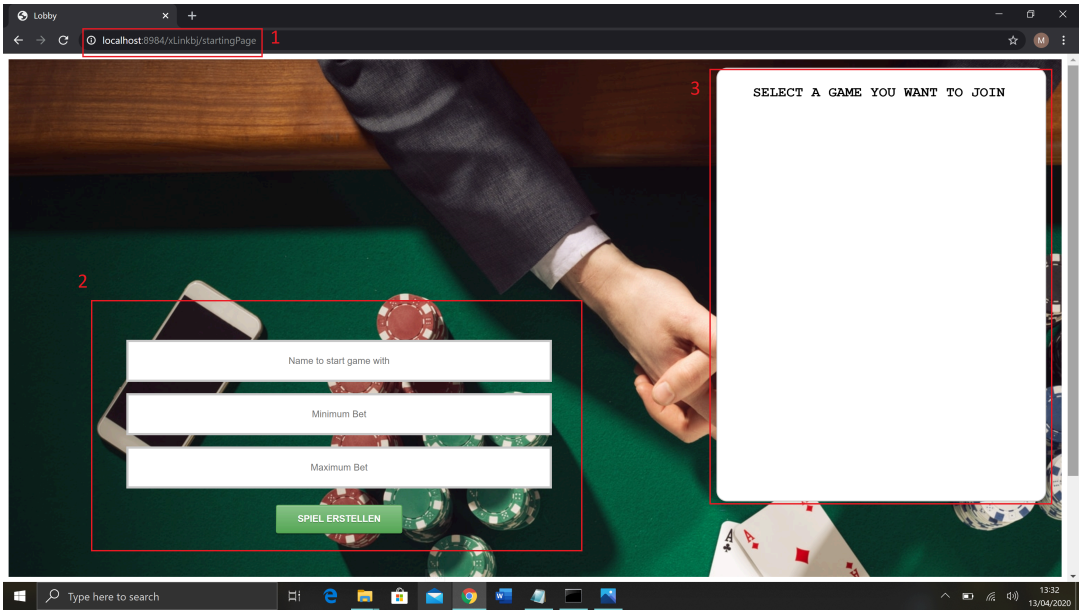
Reflection

Overall, the structure of the course was very thoughtfully organized, while also providing very helpful coaching. Even with the current limitations caused by the corona virus, the instructors showcased their flexibility by adapting very quickly, through utilizing software such as zoom. This allowed us to stay within the schedule.

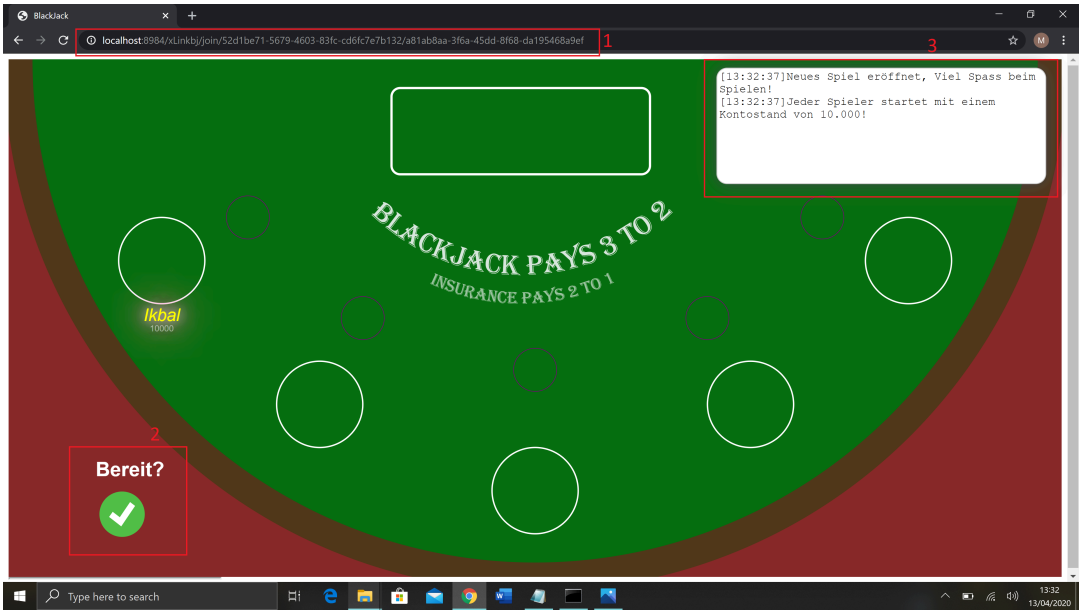
In conclusion, this was a very interesting experience, to be able to implement a full web app from scratch, using solely the XDM Languages, which forced us to broaden our horizon and thinking patterns. The fact that the semantics of the app was based on BlackJack made it even more interesting. Prof. Brüggemann-Klein and Mr. Ulrich were always ready to help and provide further materials in case you wanted to go further in details and functionality. Basically, the whole experience can be summed up as flexible, interesting, and challenging.

Chapter 7. Visual Playthrough

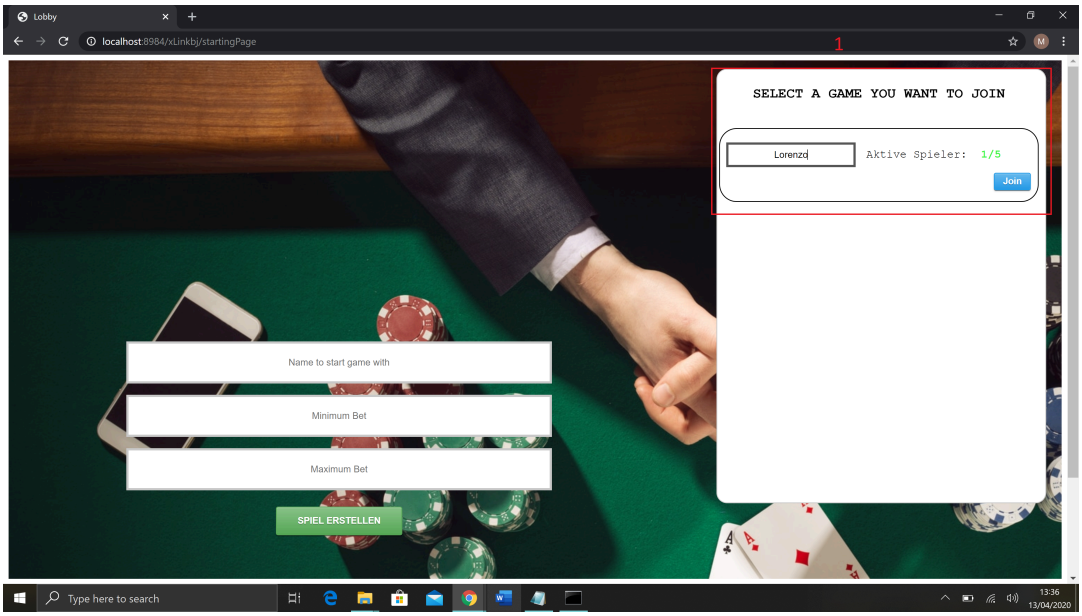
Description of the UI and flow of the Game with all its states.



(1) After inserting the url xLinkbj/setup, the DB is initialized and redirected to xLinkbj/startingPage which prints the lobby. (2) If an User wants to, here they can create a game by inserting the needed values. (3) Since no game has been created, yet, this section remains empty.

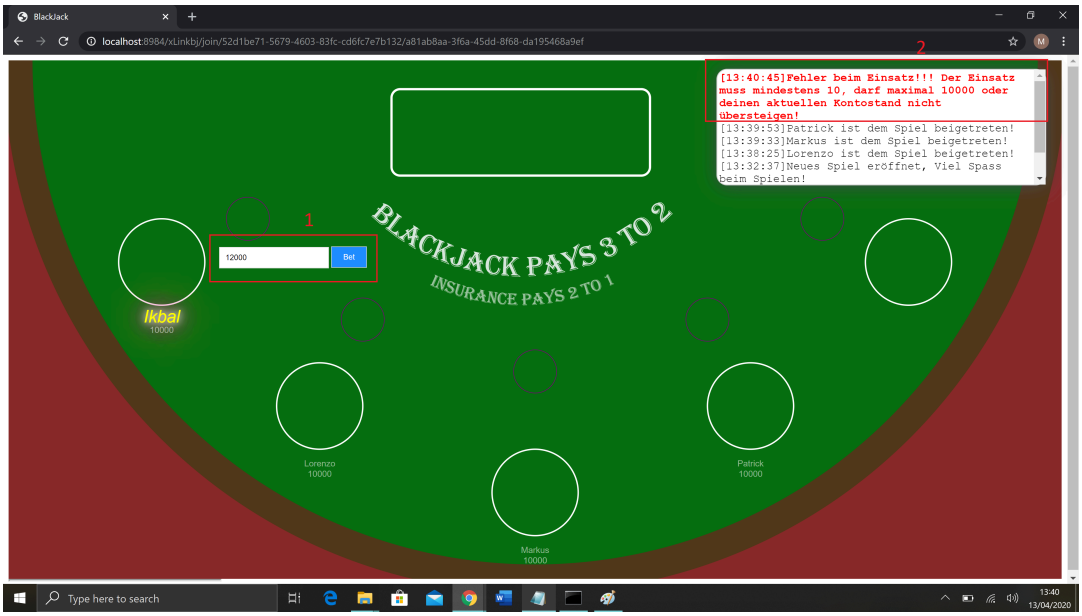


Suppose Ikbal has created a game with minBet = 10 and maxBet = 10000. After he pressed on the Start button, following Screen appears: (1) Join Url shows that the client has subscribed to the game with the specific ids. Now this browser will, as long as it stays connected, receive updates on the view (2) Ready Button to handle the ready state (3) Notification Box, that logs all game related statements.

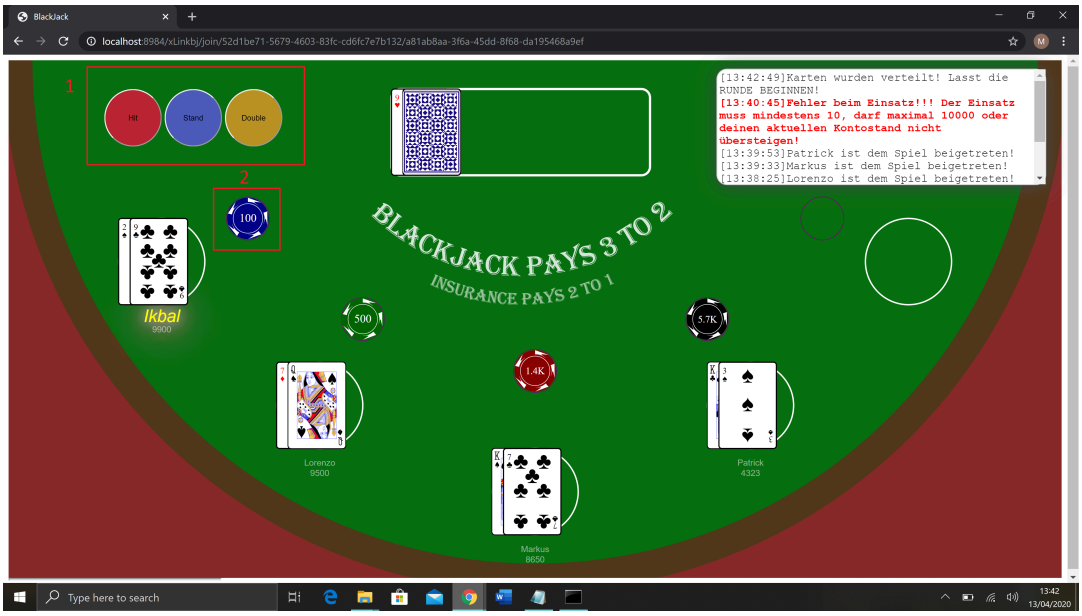


Now if a second Player, let´s say Lorenzo, wants to join the created game, he can simply click on Join in the lobby. (1) The game is currently in the ready state and thus available, containing only one Player.

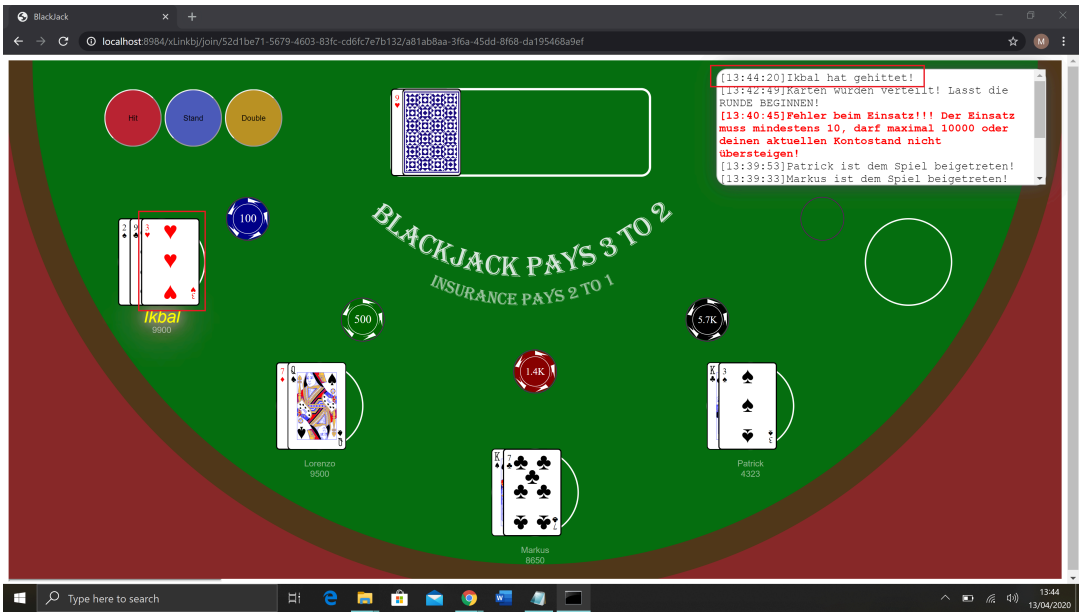
After he has joined, this will be returned. Note, how he, as the non-active player, can see who plays actively at the moment.



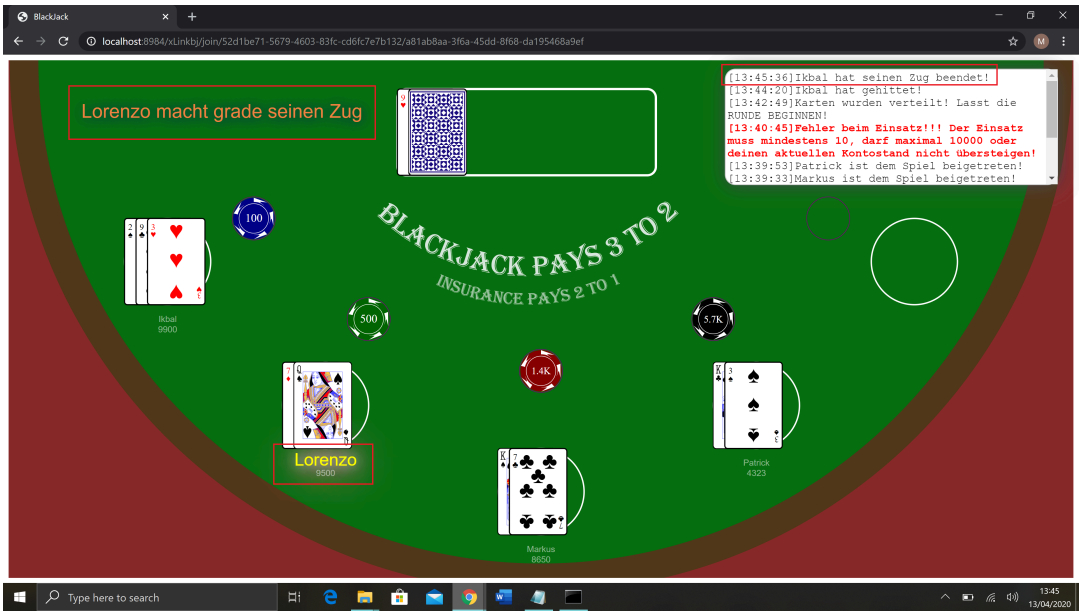
After the players Joined the table and clicked on the ready button, the betting stage is opened. (1) Input Box for Bet (2) Error Message if wrong Bet



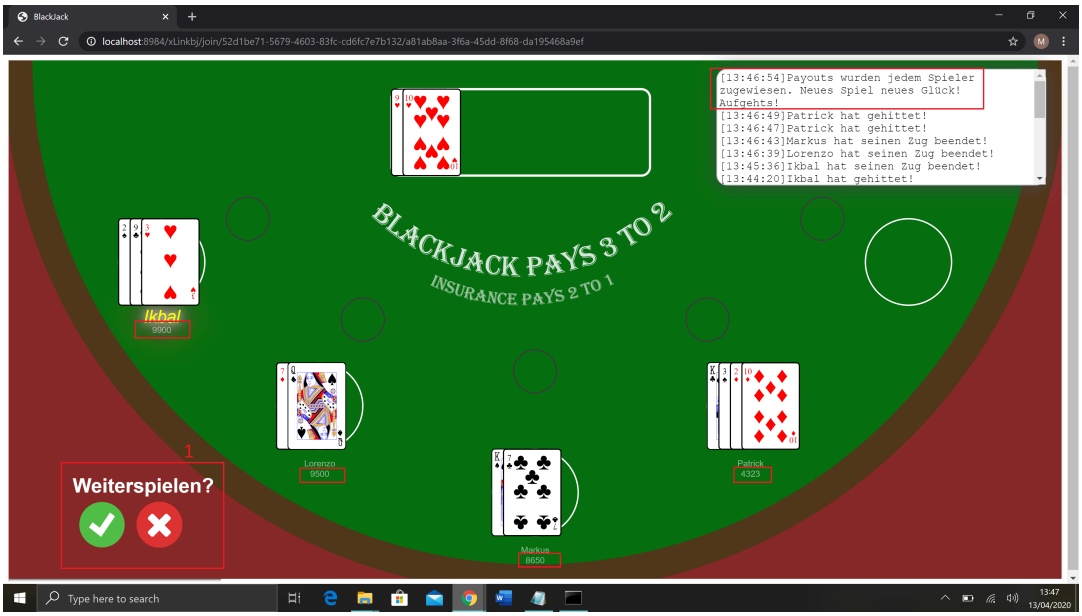
After everybody has placed their bets, the cards are dealt out and the playing state emerges. The active Player has now multiple options to choose from. Hit, Stand and Double. (1) Player Actions (2) Chip with the bet amount and dynamically color adjustment



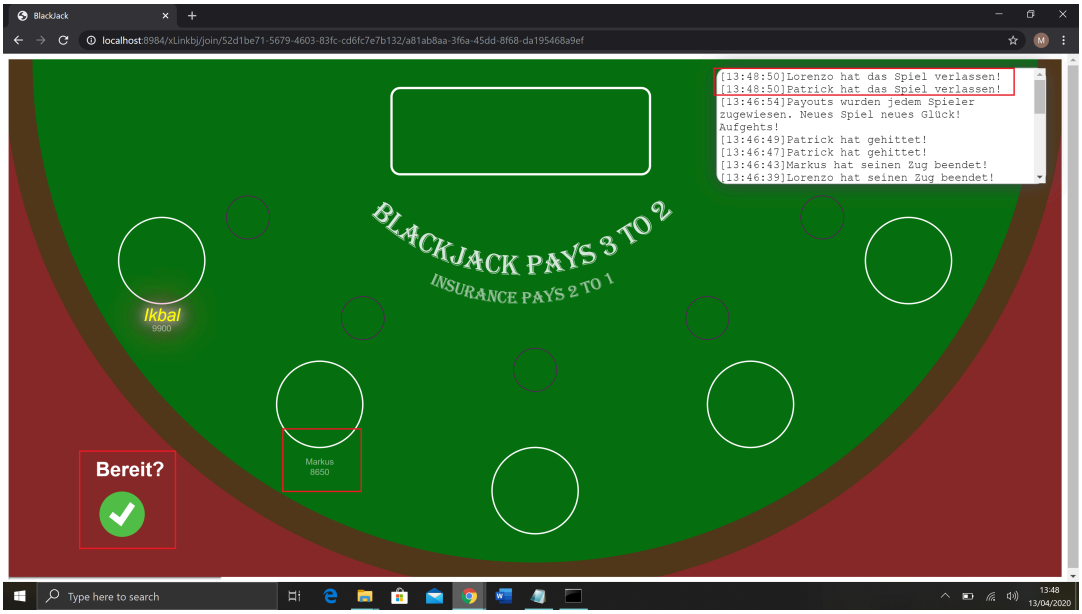
Player Hits and gets a new card.



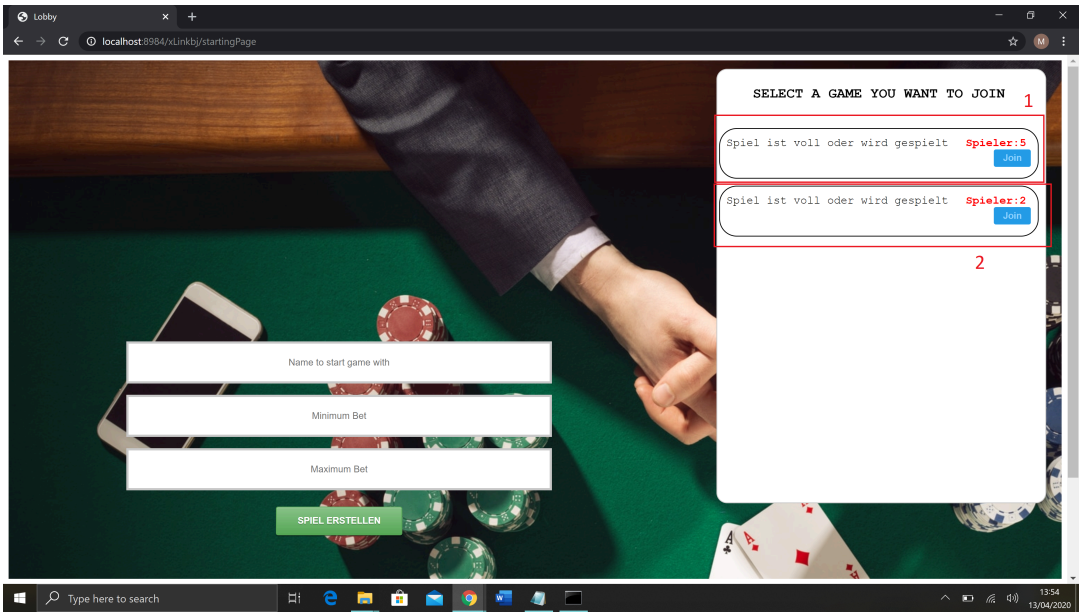
Player presses on Stand and the active Player rights are passed on to the next player in line.



After the last player plays their move, the round is automatically evaluated. Each balance is adjusted according to the evaluation and we arrived at the "continue" state. (1) Option to choose whether or not to continue playing



Suppose Lorenzo and Patrick have left the game. Then the seats are rearranged, such that Markus` seat is now the next in line. Also, the cycle is finished and we are at the ready state again.



Now that we are at the ready state again, the game becomes available again. (1) Games that are full cannot be entered (2) Games that are in a state other than ready cannot be entered