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 XML programming language. It specifies the decions made regarding the architecture, problems and their solution. It contrains references to the the approach that was taken regarding user requests to the server and the respective responses, a class diagramm explaining our program structure. Furthermore, we will give an overall summary and reflection on the practical course and on the development-work that came with it. Finally, a step-by-step run through of our blackjack program will be given with detailed highlighting and explenations of UI and Backend features.

Chapter 1. Design Choices

In the following section the graphical 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 his/her cards and bets and enough room for their names and balances to been displayed. In the middle of the table, good visible for everyone, there is a bigger space for the cards of the dealer and underneath it the conditions for blackjack-wins and insurance are mentioned.

Cards

Each card contains three elements:

- Color
- Value
- Turned

The four kinds of colors are: Spades, Hearts, Diamonds and Clubs. The values range from 2 to Ace. Court cards values equal 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 one or eleven, whichever benefits its owner more. The boolean is called „turned“ and is only relevant for the second card, the dealer gets at the start of every round. This card is the sole card that gets shown upside down. All the other cards 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. 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

Like the cards the design of the chips was also hand-crafted as svg. The chips represent the current bet of each player and their value will be displayed on them. For a nicer appearance, the shown values above 1000 will be divided by 1000 and rounded to one decimal plus K (example: 15.2K). Their color changes with the amount of the bet to simulate a casino-feeling.

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 insight into the game design and how players can interact with the game.

Lobby

For a multi-client based game a lobby is indispensable. It’s the starting point for each player. At the lobby screen there is a choice to make. You can either creat a new game, or join an existing one. To join a running game from the displayed list, you have to enter your name and to click the join button,

of one of the available games. If you wish to creat a new game, you have to additionally define the range of bets for the new table with a minimum and maximum. The new game can then be joined by others. Games are available, when 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 maximise his balance. Regardless every player has to wait for the rest of the table to perform their action, before he can execute his next one. 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 start of each round every player has to place a bet. The amount cannot exceed a players balance and has to match 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 one player after another actually plays out his/her turn until they click the stand button or surpass 21 with the value of their handcards. If they hit, they are provided an additional card, if they double they get another card and double their bet. If the openly shown card of the dealer is an ace the players are given an extra choice: to buy an insurance. If they decide to buy an insurance, they only lose half of their bet when 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. He flips his second card and if his cards value is below 17 he hits cards 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 and their losses collected. Then they can decide if they want to continue or leave the table.

Last of all: the ready check. 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 2. Architecture

MVC Architecture

We designed our application recreating the MVC (Model-View-Controller) design pattern. This is a very widely used design pattern when it comes to full stack projects like this one. It is perfectly suited since we have an XML database in as well as a BaseX server in the backend. The database stores our data and the server handles user requests by retrieving the requested data from from the database and processes it to give the user a response. This response is the data embedded in a UI view that gets displayed on the user's screen. As you can see in the class diagramm below, the controller is the interface between the backend and what the user sees.

Modules

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

Controller

The controller file is the first instance that the user encpunters when sending REST requests to our game. Each function in the controller file is mapped uniquely to a url that triggeres the function call when entered. These functions mostly have no game logic implemented but just rederict to the actual instance's function call. The controller was a very helpful instance when it came to testing, since we could send GET requests through URLs. In the following we will present the few most imporatant methods of the controller module, since most of them only redirect to function calls of other modules, which will be presentet later.

- setup, startingPage

  The 'setup' function is the first one that gets called when startig the webapp. It creates the XML database and redirects to 'startingPage'. This method transforms the lobby to be able to show e.g how many games are available to join and which are closed.

- startGame

  reads in the minimum and maximum bet parameters given by the user, checks them for correctness, and creates a game instance in the database.

- join

  This function is crucial for playing the game itself. When creating or joining a game the user recieves an html page with the a websocket element through which he subscribes to a game instance and gets notified about changes.

- draw

  One of the main functions of the Controller module. Every time the state of the database gets changed, the game is given to this draw function, which transforms it to HTML via XSLT. This HTML is then sent to every user that is subscribed through the websocket element.

Game

One of the main classes besides Player and Dealer. It contains most of the game logic with regards to blackjack rules and non-player-related actions. It is responsible for controlling the states of the game as well as assigning the right to take action at a single player at a time. Having the control over the states, it is also responsible for the decision of which content gets shown to which user, e.g depending on wheather he is the current active player or just spectator.

- createGame, insertGame

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

- setActivePlayer

  This is one of our main function when it comes to the flow of our game. It is always called when a player is done with his/her turn and the next active player is set. It has a lot of logic implemented in it, since we implemented case distinction regarding our game states. 'setactivePlayer' sets the new active player while checking if that player is the last one in the current round. If that is the case it performs different actions that with respect to the current game state and gives the turn back to the first player in the array. These actions are game important for the game logic. One example for that could be that after every player has finished playing against the dealer the game gets evaluated and reset.

- 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 a deck.

- darwCard

  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 the game off. 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 and giving him the Mth and Mth + 1 cards (where M = numberOfPlayers * 2) did not work for us.

- setResult

  This method gets called every time that the round is completed and all the players have played against the dealer. Its job is to update the 'won' tag of each player instance with regards to the outcome of the game. All the cases get evaluated by comparing the card value of the dealer to the card value of each player. The player can either win, lose or even draw with the dealer. In this case we defined that the player only gets his bet back if they draw having a Blackjack card. If they draw without a Blackjack hand the dealer automatically wins.

- evaluateRound

  Like the name of this function suggests, it is responsible for checking the result of the 'setResult' function, e.g. by checking the 'won' element of each player instance and applying the result to the actual outcome of the player's balance. Here we followed the rules implemented in the normal Blackjack game which suggest a payout of bet*1,5 plus the bet itself if the player won with a Blackjack hand and a restoration of twice the bet value if he wins normally.

- isRoundCompleted

  This function returns true if the activePlayer is the active player is the las player in the array. 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 do 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.

- resetTable

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

- assignPositions

  This function is responsible for the rearrangement of the players after a round has completed. It is called after the 'resetTable' method. Since players can leave the game at the end of each round the slots of those players free up. We decided not to leave every player that keeps on playing on his position but to order then so that they are all besides one another starting from the right hand of the dealer. This function Is responsible for rearranging these remaining players.

- 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. It also checks if 5 players are present and sets the availability to join the game to false. In this case other players would be unable to join that game.

- finishRound

  This function is the las function that gets 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-explenatory but we will go more in to detail about them later. Effectively, the 'finishRound' function assigns new positions to the players that want to stay in the game and keep playing a new round, deletes the players that want to exit and frees up the slots for new players that want to join the current game.

Player

This game instance contains all the Blackjack logic related to a single player interacting with the system. It implements all the actions a player can do during a Blackjack game and gives him the freedom to make choices e.g. to decide if he wants to stay in the game or so. Classic Blackjack player-related actions can be setting the amount he wants to bet, hitting, standing or setting an insurance.

- createPlayer

  Just like 'createGame', this method creates a ready-to-play instance of a player with respective ID, name, balance, bet, insurance and position on the table.

- setBet

  This function's job is to check if the bet that was entered by the user is valid, e.g. if he has enough balance or if is bigger or smaller than the maximum and minimum bat that was determined before the game. After that, if the bet was found to be valid, it is added to the player's instance in the database. If the player setting the bet is the last player in the players array, the 'setBet' function triggeres 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 hit 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 probability of winning against the dealer. The method implements this process of doubling the bet of a player in exchange for another card. 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 player can draw the card but the turn is automatically passed to the next player since in any case this means that the player passes 21 and therefore looses this round. In tha case that the cards value is under 21 the player is able to repeat this action untill any of the other two cases is reached.

- setInsurance

  This function is triggered when the player has the possibility of setting his insurance and chooses to do so. Then, to 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 calculation 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 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 a he will get kicked out automatically.

Dealer

Technically, the dealer is part of the players, so we could have modeled his functions within the Player module. Instead, we decided to split the two modules because the dealer plays a big role in the evaluation steps within the flow of the game and therefore required a lot of additional features. One of them would be that the card value of the dealer is the key to evaluating the outcome of the game for all the contestants. Also, we had to change the approach with which we distibuted 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 continuous pseudo-drawing of the next card to draw from the deck and checking if the value is over 16.

- cardValueDealer

Calculates the value of the dealer's hand. Because of the update constraint issues, we did 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 in principle is similar to the 'drawCard' method in the Player module. Since the dealer has no user interaction and therefore hast to decide by himself if he draws another card or not, we implemented this method. It uses the uses the earlier described functions 'cardValueDealer' and 'numberOfDrawingCards'. The first one calculates the base value of the dealer's hand and based off of that the second one looks at the cards on the deck and decides how many cards the dealer will have to draw. When this amount is determined, the dealer draws that amount of cards from the deck and deletes them from it.

- turnCard

  Turns the second card of the dealer that is hidden by default when the cards are dealed 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 gets calles 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 gets notified and updated every time changes get sent to this path.

- getIDs

  Returns all teh IDs of websockets currently connected to the instance.

- send

  Gets 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 single websockets that are subscribed to the affected instance. This is mainly important for pushing the newly updated game so that users can actually see the changes in form of game actions.

- get

  This method returns the key-value pairs from the websocket that map instances together, e.g. the player with a player ID oder with the game

- connect, disconnect

These two methods get called when a client actually connects to an instance or disconnects from it. He 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 siply 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 3. 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 errorneos lines.

Before we fully implemented our GUI, we either tested by simply returning xml documents, which we then checked on their correctness regarding the expected behavior, or simply looked at the current state of the DB.

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.