XForms Multiclient Blackjack Documentation

Daniel Meint <d.meint@tum.de>

Felix Hennerkes <ga38hom@mytum.de>

Janik Schnellbach <janik.schnellbach@tum.de>

Maximilian Karpfinger <maximilian.karpfinger@tum.de>

Table of Contents

Introduction

Description of the case study

We implement the popular casino game Blackjack as a multi-client web application. Globally, there are a variety of rulesets with slight differences. We seek to present the most universally accepted variation, commonly found in Las Vegas casinos.

Up to five *players* compete not against each other, but separately against the *dealer*. The objective of each player is to draw cards and maximize the sum of their respective values (*hand value*) without exceeding a sum of 21 (*bust*).

Card values

The game is played with a single deck of french playing cards. Number cards are worth their value, e.g. the seven of hearts is worth seven points, face cards (Jack, Queen, and King) are worth ten points, and an Ace can be counted as either one or eleven, depending on what is more favorable in a specific

situation. A hand containing an Ace counted as eleven is referred to as a *soft* hand, because the value of the Ace will change to a one to prevent the player from busting if he was to draw another card and otherwise exceed 21. A card's suit is irrelevant in Blackjack.

Course of a Round

Blackjack is round-based and each round consists of multiple stages. We distinguish between the betting, playing, and evaluation phase. After the end of a round, the next round can be initialized by any participating player. Players can leave the table at any time.

Betting

Before the actual playing begins, players place their bets. Our version uses US-dollars as currency and does not limit players in the amount they want to bet. It does, however, prohibit them from playing without betting at all. We will refer to this stage of the game as the *betting phase* throughout the following sections.

Playing

At the beginning of the *playing phase*, each player is dealt two cards face up. The dealer receives one exposed card that everyone can see and one hidden card.

The dealer subsequently asks each player, going clockwise around the table, whether they want to improve their respective hand by drawing additional cards. Each player has the following options:

Figure 1. Possible player decisions.

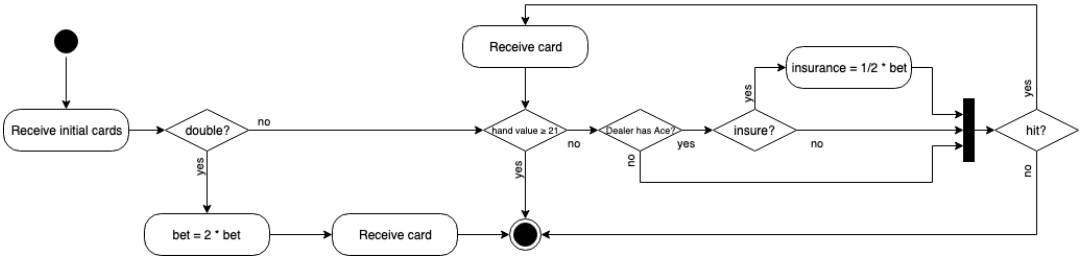


Table 1. Playing options explained.

Action	Description	Condition
Stand	Ends the player's turn	Always possible and automatically performed when the player busts
Hit	The player receives another card from the deck	Available if the player's hand value is less than 21
Double	The player doubles his bet and receives exactly one more card before finishing his turn	Available as the player's first action
Insurance	The player puts a side-bet worth half his initial bet on the outcome that the dealer has a Blackjack	Available if the player's hand value is less than 21, the first card of the dealer is an Ace, and the player only has his initial two cards

Playing double or insurance further assumes that the player has a sufficient balance to cover the increased bet size.

After all players have finished their turn, the dealer plays in a predetermined manner: He draws cards as long as his hand is worth less than 17 points and must stand on a soft 17 or better.

Evaluation

Once the dealer completes his turn, each player's hand gets *evaluated* against the dealer's hand. As described above, the goal of each player is to get a hand total closer to 21 than the dealer without busting. Concretely, a player wins by achieving either of the following final game states:

- A value higher than dealer's without exceeding 21
- Any value less than 21 while the dealer busts

Winning players get paid even money, i.e. 1:1 on the initial bet. If they win with *blackjack* (hand value of 21 with two cards), they receive one and a half times their bet (3:2). Furthermore, in case a player has decided to place the insurance bet and the dealer has indeed blackjack, this bet is payed out 2:1. A player winning a $5 insurance bet receives back the $5 plus an additional $10 from the bank, for example.

If player and dealer have equal hands and did not bust, they tie, and the player's bet is returned (also called *push*).

In any other case, e.g. the player busting even if the dealer busts as well, the player loses their bet.

Architecture

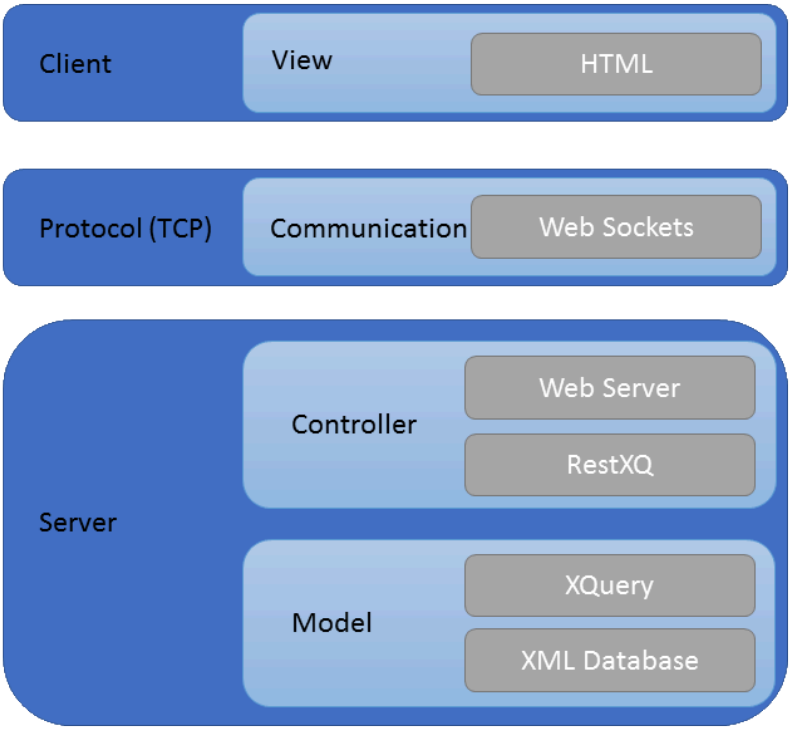
As our Blackjack game is implemented as a web application, the basic architecture resembles a client-server model, where the client (front end) runs in a web browser and directly communicates with the server (back end). Due to the simplicity of our application and the utilization of XStack, we resided from using more sophisticated architectures, such as Client-Server-Broker or Client-Server-Dispatcher.

The server side Business Logic and Persistentcy Layers are realized using the XQuery Processor of BaseX and a XML database. The content is rendered on the server side using a XSLT processor.

Communication using only HTTP requests and responses is not bidirectional. The client can send stateless requests that are answered by the server with a response. The server is not able to send requests to the client, as no active connection is maintained after fullfilling a request. This kind of communication is sufficient in web applications where the business logic is only based on actions of the client. Applications, where the server should be able to update the content of the client without any requests, requires a communication that is fully duplex. Examples for such applications include streaming services, voice over ip and online multiplayer games. A bidirectional communication can be realized using web sockets. Here a Transmission Control Protocol (TCP) connection between the client and the server is established and maintained until the client or server closes the connection. Therefore, the server is able to send new content to the client without incomming requests until the connection is closed. BaseX supports Web Sockets, utilizing the Jetty's WebSocket servlet API. As we decided to implement a distributed version of the game, allowing multiple clients to play together, we made use of the websockets support of BaseX.

Further, we complied with the Model View Controller (MVC) architecture in order to enable seperation of concerns and increase the maintainability of the application by decoupling the data representation from the data access. In MVC the system is seperated into 3 different subsystems: the model representing the data access, the view for the data presentation and the controller as a mediator between the data presentation and access. Implementations of MVC can be categorized as Push and Pull variants. In the Pull variant, the data is retrieved by the view from the model, while in the Push variant the model updates the view after changes to the data. In our application, every player should see the current state of the game. Therefore, we chose the push variant. The resulting architectural structure of our web application can be seen below.

Figure 2. Model of the Resulting Architecture.



The component model

Modeling the model

In a MVC-architecture the model has the control of the application's logic. Furthermore it is resposibel to store data in a database. In order to maintain data, several objects with unique functionality are needed. All these collaborating objects provide different attributes and methods. In the following the data model will be described at first. Thereon the design decisions will be discussed. The second part consists of the implementation of the game, followed by a discussion on implementation details and what problems were faced.

The model database consists of a list of games. For that a class `Game` is needed which stores every information about the instance of a blackjack game. A game object needs an attribute to store its' id and another attribute to store the current state of the game. Regarding `Game` functions, their main purpose is constructing,initializing and resetting game objects. Because every game object also stores all the attending players and a dealer, the functions essentially execute operations on all players. In addition to the basic functions, `Game` must provide some function to evaluate the game. However the logic of evaluating a player should be sourced to the `Player` class. Another object stored in a game object is the chat.

The `Player` will need some attributes to store meta data like name, balance, bet and profit. Furthermore this class has to hold a player's state and wether a player chose to take insurance in a round. One more object stored inside the player is the players hand, which holds the cards. As to the `Player` functions, contiguous to the constructors and setters, there will be alot of functions that contain the game's logic. Depending on the game's state, a player can take several actions, which will be passed over to the `Player` class by the API. Alongside the playing functions, some helpers which provide logic for finding the next player and joining or leaving a game are needed. Finally logic to evaluate a player must be integrated.

Since the players all play against the dealer, another class `Dealer` is required. It does not require any attributes, but it must store the deck of the game and likewise the players a hand object. Concerning the functions of the class, it should provide a helper function to help determining a player's profit,

depending on their insurance state. More logic to give every player two cards and also draw for the dealer will be required.

The next class we need in our model is `Hand`. On one hand this class will provide helper functions for evaluating a player's hand. On the other hand it will supply some function to calculate the optimal sum of a hand. This is needed because the card Ace can have the value 1 and 11.

The `Card` class will provide two attributes to store a card's value and suit. Besides the constructor and setter functions there will be a getter function that returns a card's actual value,

Inside the `Chat` class, the players' messages should be stored.

The last class in the model compoment is `Usr`. It has no functions other than a constructor and setters. We store the usr's in another database “users”.

The model is represented in the following class diagram. Helper functions and setters/getters/constructors are omitted for readability.

Figure 3. Class diagrams for component Model.

Design decisions and discussion

Continuing the section about the modelling of the model, all the design decisions will be presented and discussed.

One of the decisions was to implement something that takes care of the players' and the dealer's cards, because we did not like storeing single card instances inside the mentioned classes. We came up with `Hand` class that deals with holding cards for the players and the dealer which has the benefit of getting someones cards summed up value easily as well as evaluating a player's hand. Another great advantage of implementing this class is that the feature “split” could be added to our solution without the need to change other classes. In the real worlds game of blackjack players can decide to split their initial hand which allows them to play with two hands at one time. Without the hand class, some other ideas would be needed to enable this feature. For example one could add another attribute to the `Card` class which indicates to which hand it belongs.

Because we want to store the global balance of a player permamently, we introduced the `Usr` class. This way we can easily look up the player's highest achieved balance for the highscore list in the lobby.

We chose to have the `Deck` as an extra class so that we can simply store all of the game's deck's cards in one single instance, instead of having all of the cards inside the `Dealer` or `Game` object. This grants a better structure in the model and simplifies functionality like shuffling the deck or drawing cards. When thinking about extending the game, this class can be helpful for example when one wants more than one deck. (in some games of blackjack each player has their own deck)

So why do we need `Dealer` instead of just having another player object? At first sight both classes may not differ alot when looking at what players and the dealer actually do in a game of blackjack. However, by closer inspection the dealer has alot of differences. Starting with the attributes, a dealer does not need any information stored like a name or balance. The dealer does not place a bet each round either. The only resemblance is that both objects store a hand object. Comparing the functions of the classes, the dealer has no choices to make so none of the player's functions that are needed for a player's decision are required. Without the `Dealer` we would need another way to store the cards included in the game. Altogether a dealer object is too unequal to just use another player object instead.

Another question is How to deal with players that lost all their money? We decided that those players get set to spectating mode, where they can reset their balance. ???

As an extra feature, it was decided to implement a chat for the players. One reason for that is that we settled for the multiplayer version for local networks. With the chat the players can communicate with each other which makes the game more vivid. Moreover when players leave or join the game a

small information text will be inserted into the chat so other players will know that someone joined/ left their game.

In conclusion our main idea of modelling the model was to prevent writing the same code multiple times and segregating code by it's functionality. Instead of having a few huge classes with a big range of functions we rather wanted to have more but smaller classes that are not too complex. This way the model stays alot more readable and it makes it more adjustable and integrating extensions easier.

Implementation

The compoment model has the responsibility of holding data. Since we use an object-oriented approach, all implemented classes provide their own functionality and attributes. All of the compoment model's functions are written in `XQuery` and all the data is stored as `XML` elements. So each class contains data which is then combined in our database for BaseX. The database also lets us query functions via `XQuery`. This programming language has a functional approach and was made to run queries on `XML` databases. One of the most importan highlights of the language are `FLOWR-expressions`. The name `FLOWR` contains the initials of “for-let-order-where-return”. For comparison, this is equivalent of the “select from where” queries of `SQL`. The object's function usually have a first parameter `self` which indicates on which object the code should be executed. Because most of the classes have setters, getters and constructors and the implementation of these functions do not differ in comparison to other classes we won't describe the implementation for every single class. In fact all the setter methods return a new object by calling the classe's constructor with alternated parameters. Regarding the constructors they all create `XML` elements depending on the classe's attributes.

Beginning with `Game` class that has the following attributes and functions.

Table 2. Game attributes

name	type	description
id	double???	to distinguish games
state	string???	stores the state of the game

Table 3. Game functions

name	parameter	return value	functionality
evaluate	self, caller:integer	void	sets game's state to “evaluated”, calls player:evaluate for all players, uses caller to determine what action the last player took
latestId	none	double ???	returns the highest id of all games in the database, needed for new games since the id always gets incremented
play	self	void	sets game's state to “playing”, sets all players besides the first player to inactive, the first one to active and finally calls dealer:deal
reset	self	`XML` element of game	sets game to state “betting”, basically calls the classes constructor with all the players that have a balance > 0

Table 4. Player attributes

name	type	description
balance	integer???	store the balance of player
bet	integer	stores bet of a player
insurance	string???	stores wether or not a player took insurance
name	string???	stores name of player
profit	integer	stores profit of a round
state	string???	stores state of player

Table 5. Player functions

name	parameter	return value	functionality
bet	self,bet:integer	void	updates self's bet by given parameter bet, calls player:next
double	self	void	updates the player's bet to two times of the amount of bet, calls game:evaluate if player is last otherwise player:next
draw	self	void	calls deck:drawCard and hand:addCard to add a card to the player, updates the player's hand and the deck
evaluate	self	void	calculates profit of player and updates self's balance accordingly, also updates user's balance accordingly
hit	self	void	calls helper player:draw, gives player another card and if they bust calls player:next or game:evaluate if they are the last player
insurance	self	void	updates self insurance to “true”
isLast	self	boolean	determines and returns wether player is last
joinGame	gameId:integer, name:string	void	if it is the first player, creates a new game object with the new player, else just inserts the new player and adds an information to the chat about the player joining
leave	self	void	deletes self and calls player:next if self was active and there is more

name	parameter	return value	functionality
			than one player, adds leave information to chat
next	self	void	sets self state to “inactive” and if there is a next player:nextPlayer to “active” else checks if game's state is “betting” then calls game:play otherwise game:evaluate
nextPlayer	self	XML element player	determines and returns the next player
stand	self	void	calls player:next

The dealer has no attributes.

Table 6. Dealer functions

name	parameter	return value	functionality
deal	self	void	firstly calls deck:drawTo17 for self, afterwards deals two cards for each player by adding cards of the deck to hand and removing them from the deck
evaluateInsurance	self,playerIsInsurance:integer,bet:integer	integer	calculates a player's bonus if depending on wether they have an insurance and if the dealer got a blackjack

The deck class has no attributes.

Table 7. Deck functions

name	parameter	return value	functionality
drawCard	self	(XML element card,XML element deck)	calls deck:getCard to get the topmost card and calls deck:removeCardAtIndex to remove topmost card to return the new deck
drawTo17	hand:XML element hand,deck:XML element deck	(XML element hand,XML element deck)	recursively draws cards from given deck to given hand until hand's value >= 17
removeCardAtIndex	self,index:integer	XML element deck	calls helper deck:removeCard that finds and removes the card at given index and returns a new deck without the card

name	parameter	return value	functionality
shuffle	self	XML element deck	uses a random number generator to create a new deck with a random permutation of the cards

Table 8. Hand attributes

name	type	description
value	integer	stores the sum of the hand's card's value

Table 9. Hand functions

name	parameter	return value	functionality
addCard	self,card:XML element hand	XML element hand	adds the given card to given hand and returns new hand
evaluate	self,toBeat:integer	string	checks if player beats the dealer and returns a string accordingly
evaluateToInt	self,toBeat:integer	integer	checks if player beats the dealer and returns an integer accordingly, needed for player:evaluate
getOptimalSum	values:sequence of integers???	integer	recursively calculates optimal sum in favor of player, replacing aces with value 1 if the sum of values > 21

Table 10. Card attributes

name	type	description
suit	string???	stores the suit of the card
value	integer	stores the value of the card

Table 11. Card functions

name	parameter	return value	functionality
equals	card1:XML element card,card2:XML element deck	boolean ???	checks if value and suit of both cards are equal
getNumericalValue	self	integer	returns cards value, 10 for picture cards, 11 for Aces

Table 12. Usr attributes

name	type	description
balance	integer???	store balance of user
highscore	integer???	store the all time highest balance of user

name	type	description
name	string???	stores name of user

Table 13. Usr functions

name	parameter	return value	functionality
win	self,amount:integer	void	updates user's balance and replaces user's high-score with balance if balance > highscore

The chat has no attributes.

Table 14. Chat functions

name	parameter	return value	functionality
addMessage	self,msg:string ???	`XML` element chat	adds given message to chat and returns new chat

Implementation detail, decisions and discussion

While implementing the model, we faced a few problems and some of the constraints of `XQuery`. In this section some examples will be illustrated where we had to rethink our attempts of implementing database updating logic. Also some interesting parts of our code will be discussed.

The first problem we faced was that one cannot replace the same node in one return statement more than one time. Because we wanted a comfortable version of the game that requires as few input from the user as possible, we made functions like player:hit automatically call player:next when the player busted. That way the busted player stands and the next player can play without any further user input needed. So in our first attempts of implementing the dealer draw function, we ran into the mentioned problem, because if the last player calls hit (which replaces the node of the deck) and busts at the same time, the dealer draw function was called which also tries to replace the deck. A fast work around was found by not updating the deck in the dealer draw function since after the dealer has played the round is over and we don't need the deck anymore. However we decided to have the dealer draw in the begin of a round to avoid running into this problem in the future.

As to the second problem, it has the same root as the first problem. Since player:double automatically calls game:evaluate if the player was the last one, the database was not updated yet so evaluate would still operate on data pre the player:double call. This means that the last player's bet is not doubled and their hand is their initial hand. One solution could be to have another game state like “toEvaluate” after all players have played and the game is ready to be evaluated. However atleast one user input to have the game evaluated would be needed which interrupts the game flow. This problem also occurs when the last player busts. Our final solution was to add a parameter `caller` to game:evaluate so that the functions knows which action was the last that happend. In case that `caller` equals 0 the last player decided to stand so no updates to the database were made. When `caller` equals 1 the last player busted, so we need to make his profit negative. The last case is when `caller` equals 2, this means the last player doubled so we need to add one card to their hand and doubling their bet before evaluating the player.

For the `Deck` implementation we decided to create a “realistic” since we didn't like the idea of generating random cards each time a draw occurs. Randomly generating cards would either face the problems of possibly having duplicate cards or needing to implement some code to check if a generated card already exists and repeat generating a card until it is no longer a duplicate.

Like mentioned before, we want a comfortable experience for the player by always ending one's turn automatically if they choose to double or bust. To achieve this we implemented the function player:next, player:nextPlayer and player:isLast. Together these functions check if a player is the last one

and help setting the next player to active otherwise. This can be used in both game states where input of each player is required (betting,playing).

The component View

Modeling the component View

Implementing the component View

The component Controller

Modeling the component Controller

The controller has the task to mediate between the data representation (view) and the data access (model). Therefore, the controller has to handle all requests of the view concering the game, but also the establishment and teardown of the connection.

Implementing the component Controller

Possible requests from the client are: bjx, signup, login, logout, setup, games, highscore, delete, join, leave, draw, bet, hit, stand, double, insurance, new round and chat. These are either forwarded directly to the model or answered with static html files. The requests bjx, signup, login, logout, highscore are used for the account control and the displaying of the highscores. The remaining requests are used to implement the game functionalities, including the group chat.

Collaboration

The implementation of the web application was done as a group project with a group size of 4. Git was used as a distributed version control system. Further, the issues functionality of git was used to track and assign tasks and problems. Different branches were utilized to test different possible implementations.

Regular meetings in person at the Informatik Faculty of the TUM and voice ocer ip chats (TS3) were arranged using the messenger whats app regulary to coordinate the collaboration and progress. Visualizations of git concerning the issues and commits can be seen below.

Figure 4. Git Overview (Master).

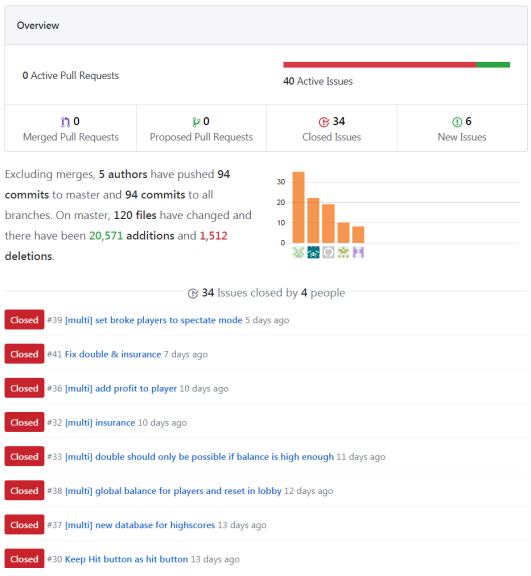


Figure 5. Code Frequencey (Master).

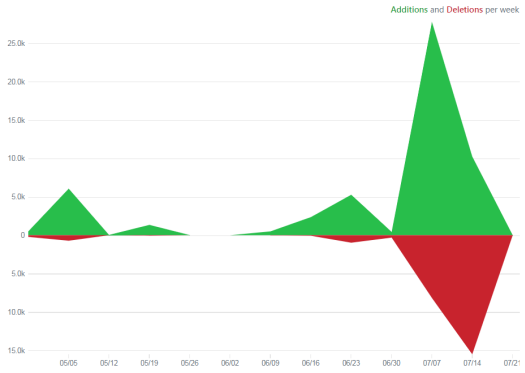


Figure 6. Contributions (Master).

Conclusions

Bibliography

[BaseX] BaseX homepage. http://basex.org. Last accessed on 2019 June 16 .