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 1
Description of the case study 1
    Preparation 2
    Course of a round 2
Architecture 2
The component model 2
    Modeling the model 2
    Design decisions and discussion 4
    Implementation 5
    Implementation detail,decisions and discussion 7
The component View 8
    Modeling the component View 8
    Implementing the component View 8
The component Controller 8
    Modeling the component Controller 8
    Implementing the component Controller 8
Conclusions 8
Bibliography 8

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*). Players win by achieving one of the following final game states:

- A hand value of 21 with only two cards (*blackjack*)
- A value higher than dealer's without exceeding 21
- A value less than 21 while the dealer busts

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.

Preparation

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.

Course of a round

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:

Table 1.

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 if the player only has his two initial two cards
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 and the first card of the dealer is an Ace and the player only has his initial two cards

If a player busts, they lose the round and have to pay their bet to the dealer

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.

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 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 has multiple attributes and functions.

- `id` to distinguish games
- `state` stores the 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` provides the function `evaluate`. However the logic of evaluating a player is sourced to the `player` class. Another object stored in a game object is the chat.

The `player` has the following attributes.

- `name` stores name of the player
- `state` stores the state of the player
- `insurance` stores wether or not a player chose to take insurance
- `balance` stores the total balance of a player
- `bet` stores the player's bet in a round
- `profit` stores the player's profit 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 are alot of functions that contain the game's logic. Depending on the game's state, a player can take several actions, which are passed over to the `player` class by the API. Following functions are supplied.

- `hit` draws another card for the player
- `stand` the player doesn't want more cards
- `double` doubles the player's bet and draws one last card
- `insurance` the player takes insurance
- `evaluate` determines the player's profit

Alongside the listed functions there are some helpers, which provide logic for finding the next player and joining or leaving a game.

Since the players all play against the dealer, another class `dealer` is required. It has no attributes, but it stores the deck of the game and likewise the players a hand object. Concerning the functions of the class, `evaluateInsurance` is a helper for evaluating a players profit. The function `deal` gives every player two cards and also draws for the dealer.

The `hand` class has the attribute `value` which stores the value of the contained cards. On one hand this class provides helper functions for evaluating a player's hand. On the other hand it supplies the `getOptimalSum` function that calculates the optimal sum of a hand. This is needed because the card Ace can have the value 1 and 11.

The `card` class consists of two attributes.

- `value` stores the value of a card

- `suit` stores the suit of a card

Besides the constructor and setter functions there is a getter function for the value that returns a card's actual value,

Inside the `chat` class, the players' messages are stored.

The last class in the model compoment is `usr`, which has these attributes.

- `name` stores name of user
- `balance` stores balance of user
- `highscore` stores all time highest balance of user

It has no functions other than a constructor and setters.

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

Figure 1. 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.

why we have usr and player

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.

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`. Usually the object's function 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 classes attributes.

Beginning with `game` class that has a the attribute `ID` with a unique value and the following functions.

Table 2.

name	parameter	return value	functionality
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
evaluate	self, caller:integer	void	sets game's state to “evaluated”, calls player:evaluate for all players
latestId	none	double	returns the highest id of all games in the database, needed for new games since the id always gets incremented
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

Secondly the `player` class

Table 3.

name	parameter	return value	functionality
joinGame	gameId:integer, name:string	void	if it is the first player, creates a new game ob-

name	parameter	return value	functionality
			ject 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 than one player, adds leave information to chat
bet	self,bet:integer	void	updates self's bet by given parameter bet, calls player:next
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
draw	self	void	calls deck:drawCard and hand:addCard to add a card to the player, updates the player's hand and the deck
stand	self	void	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
insurance	self	void	updates self insurance to “true”
nextPlayer	self	`XML` element player	determines and returns the next player
isLast	self	boolean	determines and returns wether player is last
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
evaluate	self	void	calculates profit of player and updates self's balance accordingly, also updates user's balance accordingly

dealer

Table 4.

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

deck

Table 5.

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
shuffle	self	XML element deck	uses a random number generator to create a new deck with a random permutation of the cards
drawTo17	XML element hand,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

optimal sum recursive

Implementation detail,decisions and discussion

magic of next

dealer drawing at first

deck is real not generating random cards

profit for front end

how to deal with broke players

problem of trying to replace node multiple time -> work around

double and bust call next() -> problem of evaluate data not in db if it was last player

The component View

Modeling the component View

Implementing the component View

The component Controller

Modeling the component Controller

Implementing the component Controller

Conclusions

Bibliography

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