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*). 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, prohitibt 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	Availble 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

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

Another 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

Design decisions and discussion

why we chose to have object hand

why we have usr and player

Implementation

deck is real not generating random cards

optimal sum recursive

Implementation decisions and discussion

dealer drawing at first

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 .