Mancala

Praktikum: XML-Technologie

01.07.2016

Table of Contents

Mancala, also Manqala or Mancala, is a very popular ancient game with numerous variations. This count-and-capture game can be played by two people, with the aim of winning the most "seeds" on the "field".

Introduction

Type Board game

Number of players Mostly two

At the beginning of the game, a **Mancala-Playboard** is needed. The board will be seperated into 2 rows. Each player has 6 small **"houses"** on their side, and a big **"store"** - which also called **Kalah** - where the stones are to be collected and stored. In this lab course, we chose the Kalah-version, which uses 48 "seeds" equally distributed in 12 houses.

Playing rules

Game objective : The objective of the game is to capture more seeds than the opponent.

- Player begins by choosing a house at his option and starts distributing the seeds around the board until there is no seed left in his hand. The seeds should be dropped one-by-one and in counterclockwise direction. The opponent's Kalah should be skipped.
- If the last seed is dropped into the player's Kalah, he got an extra turn.
- If the last seed falls into an empty house owned by the player, player is allowed to capture all the seeds of the opposite house and store them in his own Kalah.

- The player who still has seeds on his side of the board when the opponent's houses are empty captures all of those.
- The game ends where one player has no seeds left. The player with the most seeds wins the game.

Technologies

In this lab course, there were many of XML-Technologies used.

- SVG: Scalable Vector Graphics, helps describing two-dimensional graphics for the Web with support for interactivity and animation.
- XSLT, XSL: EXtensible Stylesheet Language is a part of XSL, which is the stylesheet language for XML. XSLT helps transforming XML Documents into other XML Documents.
- XQuery is to XML what SQL is to database tables. XML Data will be queried thanks to XQuery.
- BaseX: scalable, high performance XML Database engine which allows querying and storing XML on the net.

Diagrams

Use Cases and Use Case Diagram The mancala game can be splitted up in the following usecases:

- Initially the players want to be able to start a new game of mancala.
  - The players also want to have the possibility to discard the current game and restart it.
- Each player wants to be able to make specific moves.
- TODO The players want to be able to win the game and see how many games they have won so far.
- TODO If the players decide to continue the match later, they want to be able to save and load the game.

Architecture

The following diagram show the architecture of our implementation of the Mancala game. The architecture can be divided in three components, which are also a representation for the MVC (model view controller) pattern.

- *Database*: The database represents the storage for one gamestate. That means one current instance of the game is saved in the database.
- *Client*: The client is the frontend of the application, i.e. the user interaction interface. Any action from the players will be forwarded as a RESTful request to the server.
- *Server*: The server receives the requests (actions) from the client and then acts as a controller by deciding if a new gamestate should be created or the existing gamestate (currently saved on the database) should be edited. Independently from that decision the new gamestate will be stored in the database and a visual representation of this gamestate is being returned to the client.

Sequence

Each of the following sequence diagrams describe a chain of events that is triggered when a player interacts with the system. These events are actions between the four entities:

- *Client*: The client is the user interface, which serves the user as visual representation of the application. If a user interacts with the system, the interaction is aimed at a component shown by the client.
- *Controller*: The controller represents the application controller. It forwards requests from the frontend to the backend and it triggers updates of the client, if new data are available.
- *Game*: The game is implemented as the server of the application. Different frontends (Client and Controller) can interact with this server, i.e. the game. Therefore, it also acts as a controller but its scope is only a instance of the mancala game.
- *Database*: This entity stores the active gamestates. A gamestate is an instance of the current game. Each game always has one gamestate that can be saved.

Create game

TODO MICHI In order to start a new game the player can press the "New game"/"Start over" button or one of his houses.

1. *click*: The user clicked on the according objects shown by the Client.
2. *newGame()*: The client evaluates the click, for example, on the "New game" button. Then the client sends the newGame() request to the controller in order to instanciate a new game. Now the Client is waiting for a response of the controller.
3. *newGame()*: The controller received the newGame() request from the client and forwards it to the server, i.e. the game.
4. *createGameState()*: After receiving the request for a new game the game creates a new instance of the mancala game with the initial settings.
5. *setGameState()*: The just instanciated gamestate is immediately sent to the database where it is being stored.
6. *gamestate*: Now the game sends the gamestate to the controller as response for the newGame() request from the controller (Nr. 3).
7. *updateGameStateView()*: The Controller generates the SVG which can represents the current gamestate.
8. *gameStateSVG*: The just generated SVG is now sent to the client as response for the newGame() request (Nr. 2).
9. *refreshView()*: Finally the Client refreshs its view and the user can now see the game with the initial settings.

Select a house

TODO MICHI In order to make a move, a player can click in the browser on a house and trigger the intended move.

1. *click*: The user clicked a house to start the game.
2. *clickedHouse(int)*: The client evaluates the click to be a click on a house with a specific index. Then the client sends the clickedHouse(int) request with the regarding index as parameter to the controller in order to trigger the player move. Now the Client is waiting for a response of the controller.
3. *clickedHouse(int)*: The controller received the clickedHouse(int) request from the client and forwards it to the server, i.e. the game.

4. *getGameState()*: The server now requests the current state of the game from the database.
5. *gamestate*: The database returns the current gamestate.
6. *updateGameState()*: After receiving the current gamestate the server evaluates the move which was triggered by the player and updates the gamestate appropriately.
7. *setGameState()*: The just updated gamestate is immediately sent to the database where it is being stored.
8. *gamestate*: Now the game sends the gamestate to the controller as response for the clickedHouse(int) request from the controller (Nr. 3).
9. *updateGameStateView()*: The Controller generates the SVG which can represents the current gamestate.
10. *gameStateSVG*: The just generated SVG is now sent to the client as response for the clickedHouse(int) request (Nr. 2).
11. *refreshView()*: Finally the Client refreshs its view and the user can now see the game with the initial settings.

Classes

The following list explains the classes shown in the class diagram.

- *Player*: The Player class represents one of two players. It has an ID to differentiate between the players and each player has a winCount, which contains the amount of won games. Thereby it has the method to increase the winCount after a won match and the method to reset the winCount if the game session has came to an end.
- *Players*: The Players class is the controller of two Player classes. It remembers which players turn it is and provides the according methods to set/get/toggle the player turn.
- *GameObject*: The GameObject is an abstract class which works as a superclass for the House and the Store class. To identify the different GameObjects they contain the property ID. Furthermore, each GameObject tracks the amount of seeds in it and provide the appropriate methods to get/set/increment the current seedCount. The Store and the House classes has up to now the exact same attributes and methods as their superclass.
  - *House*: The houses represents the holes on the board in which the seeds will sit in.
  - *Store*: The store is the slightly bigger house, which stores all the seeds a player won. Therefore, each player has only one store.
- *Layer*: The Layer class is just a wrapper class for the store and the houses of one player. Since a player has six houses and one store it consists exactly of six houses and one store. Furthermore, the Layer class contains the attribute position, which is an enumeration of either "top" or "bottom".
- *Board*: The board is the contoller class for the mancala board. It consists of two layers - one top and one bottom layer. The board is able to get a hosue or a store by its ID so that the appropriate method (inSeedCount) can be called if a player clicks on a house. The board also wraps getting a house/store and calling its incSeedCount by the handler method clickedHouse, which is called if a player clicks on a house. Finally the board contains the resetBoard method, which sets the board to its initial setup.
- *Game*: The game is the top level controller of one instance of a game. It consists of the Board and the Players class and after calling the contructor the currentGame property is instanciated. Additionally it has the method to reset the current game.

Implementation

Filestructure

The following list shows the files, which were used to implement the basic functionalities of XQuery for mancala.

- *initial_gamestate.xml*: Contains all initial elements (initial values for a game to start) shown in the class diagram with according mulitplicity.
- *Static.xml*: Combines and transforms current gamestates from server with static measurement input values to create an SVG file that displays the Mancala board. Contains a number of measurement input values (such as horizontal/vertiacal distance between houses, distance between stores and nearest houses etc), so that the Mancala board can be automatically created (see "Transform.xsl").
- *mancala.xqm*: Contains all methods described in the class diagram.

OOP, namespaces and naming conventions

In order to realise the architecture represented in the classdiagram we introduced multiple namespaces, to simulate class-scopes. Each namespace represents a class and each method of that class is defined inside that namespace. Every class is in its own file and is separated into a public and a private section. Similar to 'self' in Python, the $this-reference of the current object is passed as the first parameter of every method.

Naming conventions

Public methods are named as is, private methods are prefixed with an underscore. Getter do not have the prefix 'get', setter are prefixed with 'set'. A method returning a boolean is prefixed with 'is' (e.g. seedCount, setSeedCount, isRowEmpty).

Index DB

TODO: node manipulation by reference

TODO: IndexDB for games: At first startup, the index-db has to be initialized.

Single Node Updates

As XQuery updates are gathered and applied at once the end of the XQuery script, nodes cannot be updated multiple times. In addition, updated values are not visible until the next query. We deal with this problem in two ways: 1. Precompute values. This increases the complexity of the seed distribution, but guarantees that no two updates on the same nodes are executed in one query 2. To issue an update statement and access the same updated values in a GET request, a forward statement is used. The called method updates the database and forwards the client to another method, where the updated data is accessible.

FLOWR vs. Updating-statements

XQuery FLOWR statements cannot be used in conjunction with updating statements because of the delayed-updates issue described above. By marking a function *updating*, FLWOR statements cannot be used inside the function scope. To mitigate this issue, updating functions use getters on composite objects instead of local *let* variables. As a consequence let statements are inaccessible, forcing us to

use getter methods for every time a variable would be used. Recursion is used as a replacement for loops, most noticable in the algorithm distributing the seeds.

Graphical User Interface (GUI)

The project's homepage delivers the main GUI, which is an SVG file as shown below:

The shown Mancala board contains:

- 12 houses, 6 for each player. Player 1 owns the upper houses, player 2 the lower.
- 2 stores, 1 for each player. Player 1's store is on the left and player 2's on the right.
- Game buttons, such as:
  - *New Game*: creates a new game instance without affecting players' scores.
  - *Reset Scores*: creates a new game instance and reset ALL players' scores.
- The text in the middle indicates current game status, for example: which player is playing or if one player has won. Players' turns are hinted by a typical color and a blinking triangle in the corresponding store. In case of winning, the status text in the middle will point to the left or right depending on which player has won.
- Each house is a clickable object. However, the current player can only select his own stores that have positive (> 0) number of stones. In addition the shown digits, the corresponding amount of stones is also displayed as a set of dots, each of which represents one stone.

An example of a gamestate after several turns:

An example of a gamestate, in which player 1 has won:

The main process of storing and transforming data from server to GUI as an SVG file can be described in the following steps:

1. *SVG outline definition*: The board's "skeleton" is defined by *Static.xml* stored in the server. This XML file contains very few static measurements that draws the GUI's layout, such as width, height, distance between objects etc. Based on these values, the rest can be automatically computed in the later stage, which hence ensures the model's flexibility and easiness for later adjustment. Styles for these objects (e.g. houses, stores, seeds etc.) are also stored and can be adjusted anytime. In this step, the most important part is to organize required tags in a meaning structure.
2. *SVG data source*: Game data (such as number of seeds in each house, whose turn it is etc.) is store in the server and can be retrieved by a RESTful request. The gamestate fetched can then be processed and displayed on the board. The gamestate is seperated from the static measurements to save storage space and increase request processing time. The data source must conform to the SVG outline.
3. *SVG transformation*: *Transform.xsl* combines both SVG outline and data source into one single SVG file, which is displayed by default on the project's homepage (i.e. http://localhost:8984/). Based on the two XML sources, the XSL file calculates, creates, aligns and assigns styles to all objects shown on the board. Some of the techniques that are used to implement the transformation are:
   - <*defs*> and <*use*> tags to reuse objects
   - <*if*> or <*choose*>, <*when*> and <*otherwise*> tags to handle different data states
   - <*call-template*> and <*template*> tags to enable recursion with parameters over repeated work

TODOs

Warning

- Eine Beschreibung der Requests und Responses zwischen Client und Server nach dem REST-Prinzip und ihre Abbildung auf Queries mit restXQ.
- Eine Beschreibung der Benutzeroberfläche für die verschiedenen Stadien des Spiels mit den jeweiligen Interaktionsmüglichkeiten.

Future Work

The game could be improved by adding the following additional features

Play against the computer

The player has the possibility to choose between playing against a real opponent or against the computer. When a game against the computer is selected, the player can chose the difficulty level of the game. There are 3 options available - easy, medium, difficult.

Allow multiple pairs of users to play simultaneously

Currently, the mancala application can only save the state of one game which limits it to allowing only two users to play. In order to extend the service and support a multiple number of games simultaneously, each game state should be preserved separatelly.

User Ranking

Allowing multiple games at the same time asumes a bigger number of users. Adding a user ranking and statistics would provide the players with a better overview of their own performance as well as of their competitors. The statistics could be integrated into each user's profile, showing the amount of games he/she won or lost. A hall of fame listing the best 5 players of all time could be an additional motivation for the users.

User Network

Building a user network allows the inegration of many additional features which can improve the user experience. As mentoined above, each use will have a profile that shows his statistics. The profile will also be visible for other players. Additional features could include challenging a user, chat as well as offline messaging.

Animation

Adding animation and special effects to the game play would also have a positive impact on the user experience. An example for such special effects is animating the movement of the seeds or adding sound effects when a seed get into the user's store.