Mancala

Praktikum: XML-Technologie

29.07.2016

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".

1. Introduction

Type Board game

Number of players Mostly two

The core element of the game is the **Mancala-Playboard** and it includes 6 small **"houses"** and one big **"store"** for each player. The houses are lined up in two rows and each house holds 4 seeds at the beginning of the game. The goal is to collect as many seeds as possible in the store, which is also called **Kalah**. There are different versions of the game, but for this project we chose a Kalah-version, which uses 48 seeds in total, equally distributed in the 12 houses.

2. Playing rules

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

- A player begins by choosing a house on his/her side and then distributes 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.

- If the last seed is dropped into the player's Kalah, he gets an extra turn.

- 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.

3. Technologies

In this lab course, a diversity of XML-Technologies was used.

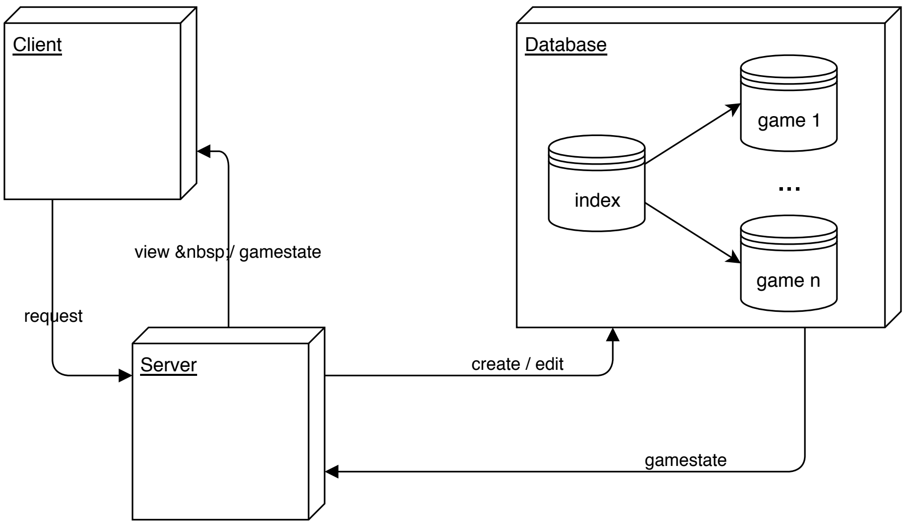
- SVG: Scalable Vector Graphics, helps describing two-dimensional graphics for the Web with support for interactivity and animation.
- XSLT, XSL: EXtensible Stylesheet Language is a part of XSL, which is the stylesheet language for XML. XSLT helps transforming XML Documents into other XML Documents.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

4. Diagrams

4.1 Architecture

The following diagram shows the architecture of our implementation of the Mancala game. The architecture can be divided in three components

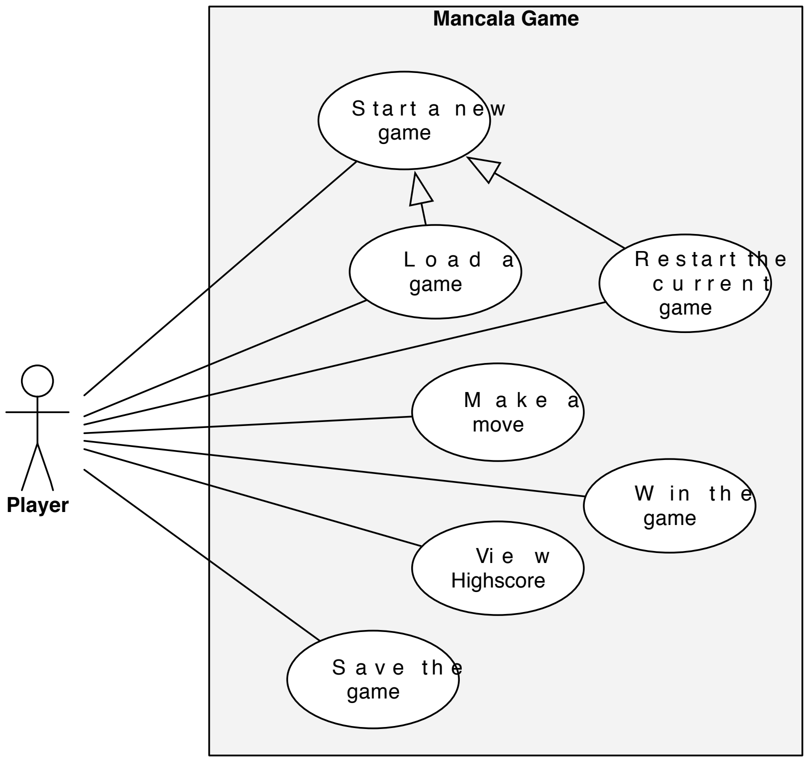
- *Database*: One index database is used to track all games. Each game is represented by a dedicated database, named *game-{id of game}* which stores the state of the respective game. This allows multiple games to be played simultaneously.
- *Client*: The client is the front end of the application, i.e. the user interaction interface. Any action made by the players triggers a request to the server.
- *Server*: The server receives the requests (actions) from the client and forwards these actions to the model layer containing the business logic. The modified gamestate is saved back to the databases and returned to the client for display.



4.2 Use Cases and Use Case Diagram

The mancala game can be splitted up into following use cases:

- 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.
- The players want to be able to win the game and see how many games they have won so far, and with which scores.
- If the players decide to continue the match later, they should be able to resume their game.



4.2 Sequence

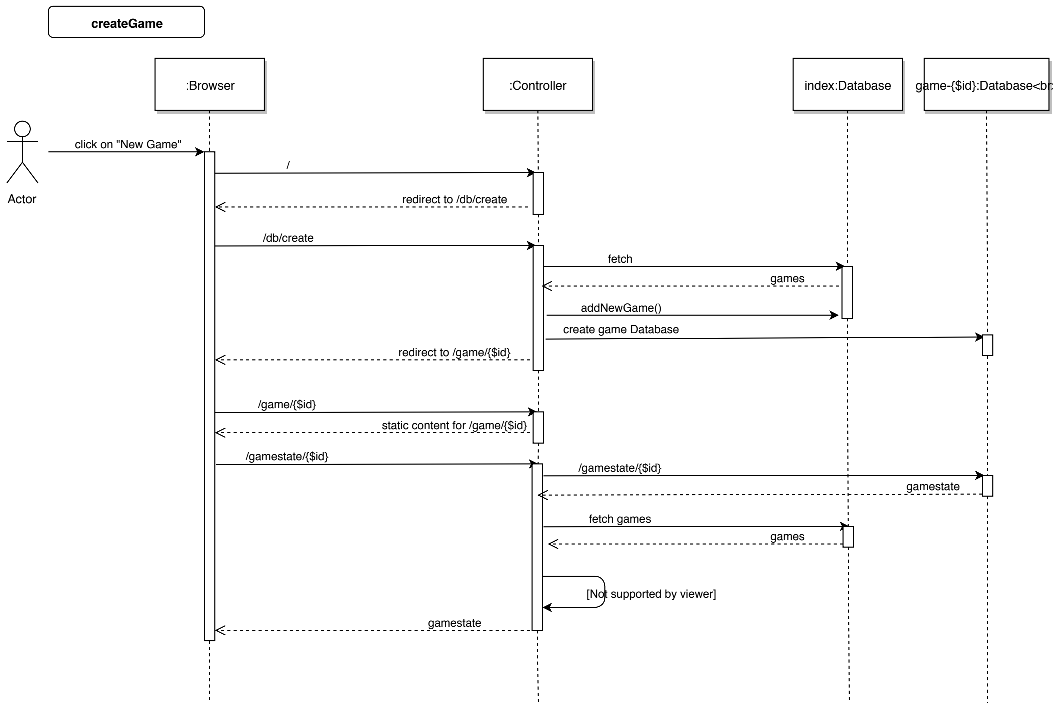
Each of the following sequence diagrams describes a chain of events that is triggered when a player interacts with the system. These events are actions between the four entities:

- *:Browser*: The browser 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 browser.
- *:Controller*: The controller represents the application controller. It forwards requests from the frontend to the backend and it triggers view-updates, if new data are available.
- *:Game*: The game contains the public interface to the model layer. The controller forwards all actions and requests to the game, which updates the gamestate.
- *index:Database*: This entity stores the references to the saved gamestates. A gamestate is an instance of the current game. Each game always has one gamestate that can be saved.
- *game-{$id}:Database*: This database represents one game, i.e. one gamestate.

Create game

When the player accesses / a new game is created. To start a new game, the player can press the "Create New" button.

1. When the client accessses /, the controller forwards to */db/create*
2. The call to */db/create* generates a new game-id from the index database and stores it. This id is used to create a new game database and the client is forwarded to */game/{game-id}*.
3. A call to */game/{game-id}* returns the combined xsl and svg file. As the xsl fetches the gamestate from the game database, the *select* attribute of the *gamestate* has to be modified to contain the game-id of the current game.
4. When evaluating the xsl, the xsl fetches the *gamestate* with the current *game-id* (*/gamestate/{game-id}*) and transforms the gamestate into the svg.



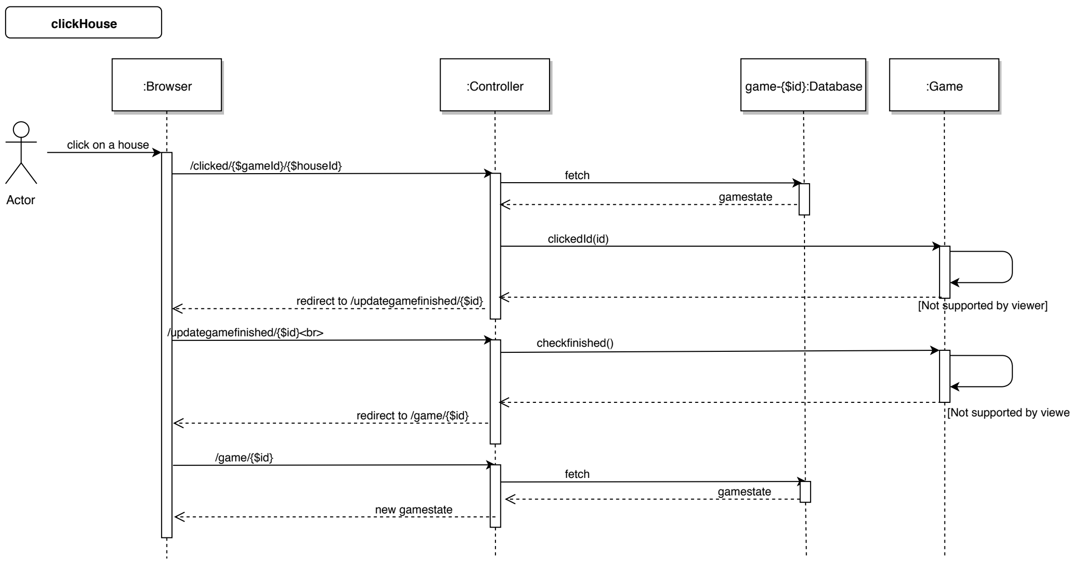
Select a house

In order to make a move, a player has to click on one of his/her houses in the browser.

1. *click*: The user clicked a house to make a move.
2. */clicked/{$gameId}/{$houseId}*: Each house has a specific ID, which is sent by the browser to the controller in order to trigger the player's move. It is to be noticed that only clicks on a house are valid and trigger a request. The Browser then waits for a response from the controller.
3. *fetch*: The controller receives the request for a player's move from the Browser and then fetches the current gamestate from the corresponding game database. The database for this game immediately returns the current gamestate for the game.
4. *clickedId(id)*: The controller calls clickedId(id) on the game, which then modifies the gamestate accordingly, by distributing the seeds.
5. */updategamefinished/{$id}*: After receiving the success response from the game, the controller now sends the response to the initial request from the Browser (1). This response tells the Browser to send another request, which serves to determine if the current game is ended and updating the game's extra attributes, like the wincount.
6. *checkfinished()*: This is the call to trigger the described updates, because the gamestate is handled by the game and not the controller.

7. */game/($id)*: Again the Controller redirects the Browser after receiving the success response from the Game. This redirect requests the updated gamestate from the Controller.

8. *fetch*: The Controller requests the updated gamestate as it is now stored in the database. This is then returned to the Controller and then forwarded to the Browser which now can update the view so that the player see the result of his action.



4.3 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 come to an end.

- *Players*: The Players class is the controller of two Player classes. It remembers which player's turn it is and provides the appropriate methods to set/get/toggle the player turn.

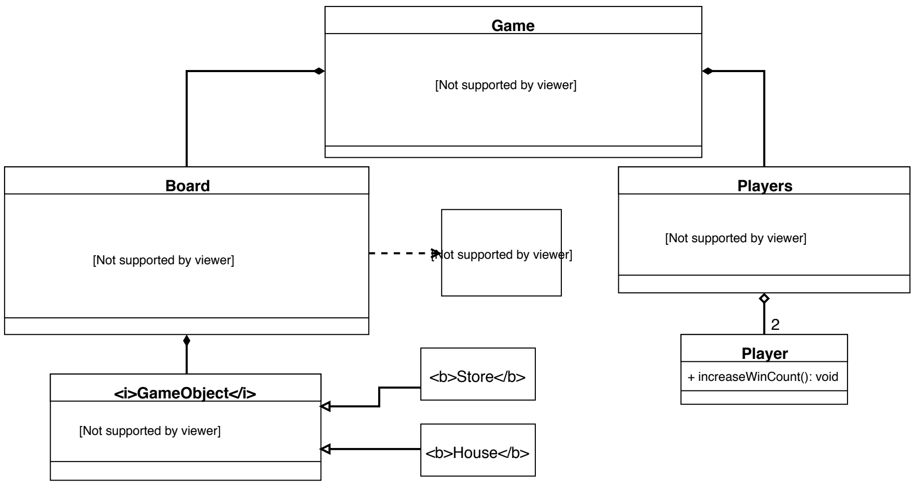
- *GameObject*: The GameObject is an abstract class which works as a superclass for the House and the Store class. A property ID is responsible for identifying different GameObjects. Furthermore, each GameObject tracks the amount of seeds in it and provide the appropriate methods to get/set/increment the current seedCount. The Store and the House classes have up to now the exact same attributes and methods as their superclass.

  - *House*: The houses represent the holes on the board in which the seeds will be sown in.

  - *Store*: The store is the slightly bigger house, which stores all the seeds a player has won. Therefore, each player has only one store.

- *Board*: The board is the controller class for the mancala board. It consists the attribute Position, which is an enumeration of either "top" or "bottom". The board is able to get a house or a store by its ID so that the appropriate method (incSeedCount) 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 as a player clicks on a house. sumOfStoreAndRow shows the total amount of seeds in the store and houses according to the position. If Row is Empty, it means one of the players has won, the seeds in the houses will be collected. Seeds will be distributed as the player click on a house, using the function distributeSeeds().

- *Game*: The game is the top level controller of one instance of a game. It consists of the Board and the Players class. Additionally it has the method to reset the current game.


Game
[Not supported by viewer]
Board
[Not supported by viewer]
[Not supported by viewer]
Players
[Not supported by viewer]
2
Player
+ increaseWinCount(): void
<i>GameObject</i>
[Not supported by viewer]
<b>Store</b>
<b>House</b>

5. Implementation

5.1 Filestructure

The RESTXQ service is accessible via http://localhost:8984/. All RESTXQ annotations are assigned to the http://exquery.org/ns/restxq namespace. If a RESTXQ URL is requested, the RESTXQPATH module directory and its sub-directories will be accessed, and all XQuery files will be parsed for functions with the compatible RESTXQ annotations.

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.
- *Mancala.xqm*: Contains all methods described in the class diagram.

5.2 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).

5.3 Gameindex Database

The application stores a reference to all games in a games-database. This database is used to generate unique game-ids and keep track of the outcome of individual games. The databse has to be created once by calling */db/initgameindex*

5.4 Single Node Updates

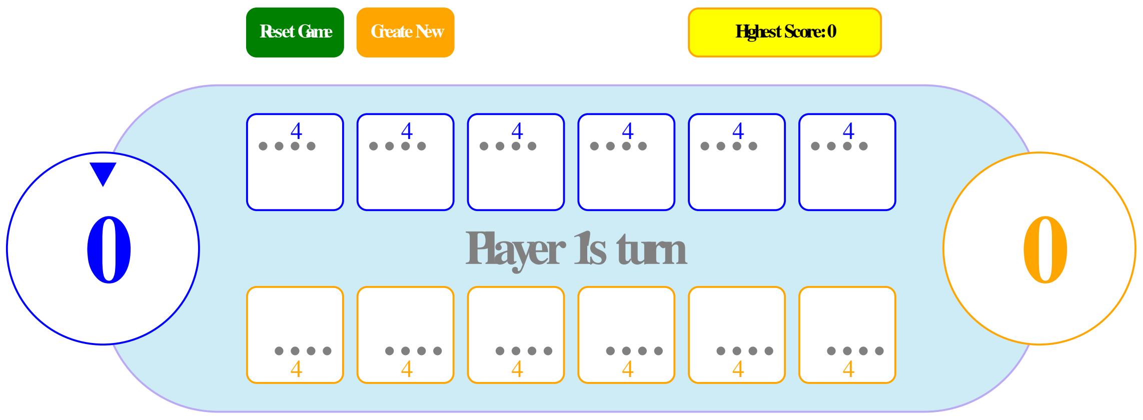
As XQuery updates are gathered and applied at once the end of the XQuery script, nodes cannot be updated multiple times. In addition, updated values are not visible until the next query. We deal with this problem in two ways: 1. Precompute values. This increases the complexity of the seed distribution, 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.

5.5 FLWOR vs. Updating-statements

XQuery FLWOR 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.

5.6 Graphical User Interface (GUI)

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

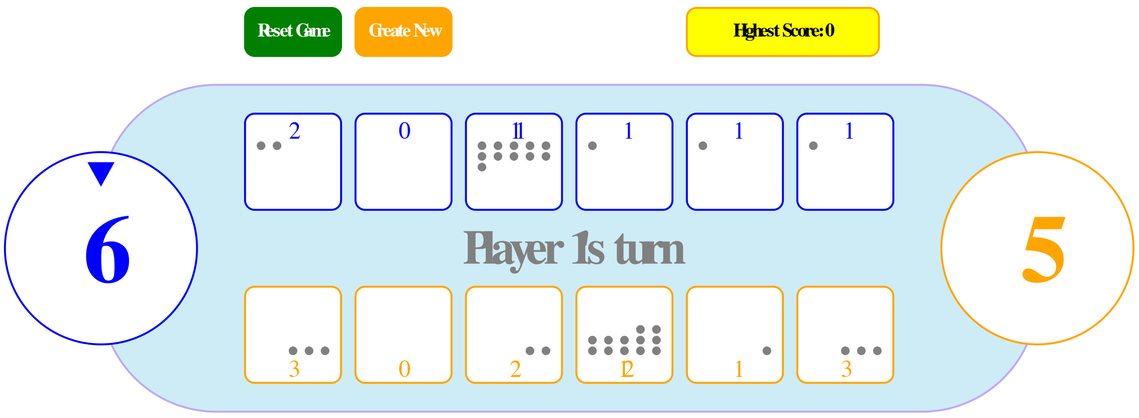


The shown Mancala board contains:

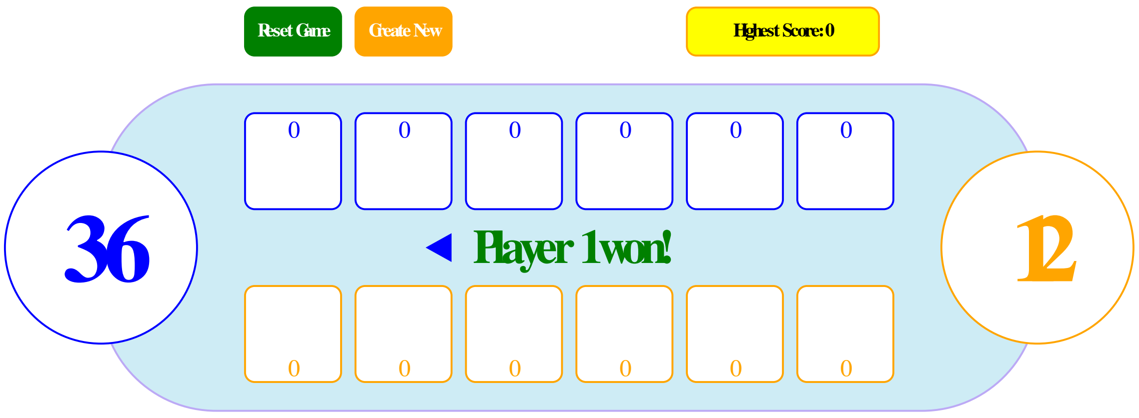
- 12 houses, 6 for each player. Player 1 owns the upper houses, player 2 the lower.
- 2 stores, 1 for each player. Player 1's store is on the left and player 2's on the right.
- Game buttons, such as:
  - *Reset Game*: the current game will be reset. As long as the game is not won (or lost), the highest score will not be updated.
  - *Create New*: creates a new game instance in DB.
  - *Highest Score:* the current highest score of all won games stored in the DB.
- The text in the middle indicates the 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 seeds. In addition to the shown digits, the corresponding amount of seeds is also displayed as a set of dots, each of which represents one seed.

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 in the second half of *Static.xml* stored in the server. This XML file contains 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 meaningful 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*: The stylesheet found in the first half of *Static.xml* 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 calculates, creates, aligns and assigns styles to all objects shown on the board. Some of the techniques which 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

6. Future Work

The game could be improved by adding the following additional features

6.1 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.

6.2 Allow multiple pairs of users to play simultaneously

The mancala server supports parallel multiple games, which are accessible from multiple clients. However, two players can only play locally against each other. Allowing a "distributed" multiplayer game where two players are not co-located on one computer faces architecture challenges, as actions performed by one player on one computer are not pushed to the other one. A page refresh is required in order to see the opponent's move, which can be implemented with the help of javascript, using automated asynchronous gamestate-refreshes. Another solution would be a websocket connection, which allows the server to push gamestate changes to the client.

6.3 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.

6.4 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.

6.5 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.

7. Reflection

The lab course introduced us to more advanced posibilities of XML that go beyond the well known markup language usage. Different aspects of XML where used to build a complete web service including database with the help of XQuery, front-end representation with SVG and backend with BaseX and RESTXQ module for the API. The task to deliver a working application contributed to the hands-on character of the class and provided us with a very realistic expirience about the challenges faced in development.

Good organisation and communication was an important factor to proceed with the project at a good pace. In order to also work efficiently as a group a certain degree of flexibility was needed, which could be accomplished by the usage of trello and git and the continuous integration approach. The documentation regarding the project was also continuously extended and updated, which helped sustain a good overview of the project.

The theoretical classes, materials and scripts provided the needed information to complete the projects without the need of additional support from the professor. One of the biggest difficulties we faced was the update of the game state because of the limitations of XQuery databases (described in section 5).