Dokumentation - Praktikum XML-Technologie

Lovis Zenz, Xiaolin Ma, Emir Chalghaf, Rodeina Mohamed

Dokumentation - Praktikum XML-Technologie

Lovis Zenz, Xiaolin Ma, Emir Chalghaf, Rodeina Mohamed

Abstract

Im Rahmen des XML-Praktikums werden verschiedene Bestandteile der XML-Technologie von Prof. Dr. Brüggemann-Klein vermittelt und innerhalb eines Projekts in Gruppenarbeit angewandt. Beim Praktikumsprojekt handelt es sich um die Implementierung des berühmten Spiels Blackjack in einer beliebigen Variante. Dabei werden CSS, DTD, XForms, XSLT sowie XQuery und XPath eingeführt und verwendet. Durch die Gruppenarbeit und die wöchentliche Vorlesung sind die Studenten in der Lage, die XML-Technologien zu verstehen und das Projekt gemeinsam zu erschließen. Im Folgenden wird das Praktikumsspiel ausführlich vorgestellt. Das Grundkonzept und die Designentscheidungen sowie die Verwendung und Implementierung des Spiels werden geschildert respektive erklärt.

Table of Contents

List of Figures

Chapter 1. Einleitung des Spielkonzepts

Konzeptdesign

Bei Blackjack handelt es sich um ein Kartenspiel, das mit sechs französischen Decks gespielt wird. Bis zu sieben Spieler spielen gegen die Bank, vertreten durch den Croupier. Ziel der Spieler ist es, mit mindestens zwei Karten mehr Punkte als der Croupier zu erreichen, ohne 21 Punkte zu überschreiten. Umgekehrt ist es das Ziel des Croupiers, mindestens so viele Punkte wie jeder Spieler zu erreichen, ohne 21 Punkte zu überschreiten.

Zu Beginn tätigt jeder Spieler seine Einsätze. Die Einsätze sind durch das Vermögen der Spieler beschränkt. Jeder Spieler tätigt ausschließlich in seiner eigenen Box Einsätze. Jeder Spieler erhält dann zwei offene Karten, der Croupier eine offene und eine verdeckte. Zieht ein Spieler hierbei ein Ass und eine 10 bzw. Ass und König/Dame/Bube, so hat er einen Blackjack. Der Reihe nach kann dann jeder Spieler so lange weitere Karten ziehen (Take), bis er exakt 21 Punkte erreicht, 21 Punkte überschreitet, oder freiwillig seinen Zug beendet. Erreicht ein Spieler mit mehr als zwei Karten 21 Punkte, so hat er lediglich den maximal erreichbaren Punktestand, aber keinen Blackjack erreicht. Zudem kann ein Spieler, der direkt zu Beginn zwei wertgleiche Karten zieht, seine Hand in zwei Hände aufspalten (Split); für die zweite Hand ist derselbe Einsatz wie für die erste Hand zu tätigen. Ein Spieler mit mehreren Händen zieht für jede Hand Karten und kann mit beiden Händen separat gewinnen und verlieren. Haben alle Spieler und der Croupier ihre Züge getätigt, endet die Runde.

Bei der Punkteauswertung gewinnen all jene Spieler, die eine höhere Punktzahl als der Croupier erreichen. Dabei gilt außerdem, dass ein Blackjack, der nur von Spielern erreichbar ist, höher zu gewichten ist als 21 Punkte, die mit mehr als zwei Karten erreicht wurden. Außerdem erreichen all jene Spieler, die eine exakt gleiche Punktzahl wie der Croupier erzielen, ein Unentschieden. Alle übrigen Spieler verlieren. Sämtliche Spieler, die gewinnen, erhalten das Doppelte ihres Einsatzes zurück; sämtliche Spieler, die ein Unentschieden erreichen lediglich ihren Einsatz. Die Spieler, die verlieren, erhalten nichts zurück.

Implementierungsdesign

Im Folgenden erläutern wir einige Aspekte unserer Umsetzung der zuvor beschriebenen Spielidee.

Spielablauf

- Das Programm beginnt im Hauptmenü. Ein neues Spiel kann gestartet oder ein zuvor begonnenes wiederaufgenommen werden. Zudem können alte Spielstände gelöscht und die Highscores eingesehen werden.

- Wird ein neues Spiel begonnen, so können zwischen einem und sieben Spielern hinzugefügt werden, wobei Spielernamen frei wählbar sind. Anschließend kann das Spiel gestartet werden. Es können maximal neun Spielstände erzeugt werden, bevor das Löschen eines alten Spielstandes erforderlich wird.

- Im Spiel werden zunächst Einsätze beliebiger Höhe getätigt, wobei das Vermögen des jeweiligen Spielers nicht überzogen werden darf. Im darauffolgenden Durchgang erhält jeder Spieler zwei offene und der Croupier eine offene und eine verdeckte Karte. Im letzten Durchgang haben Spieler, die zuvor zwei wertgleiche Karten gezogen haben, die Möglichkeit einen Split durchzuführen. Anschließen können durch sämtliche Spielteilnehmer der Reihe nach Karten gezogen werden, wobei der Croupier so viele Karten zieht, bis er eine Punktestand von mindestens 17 erreicht. Sobald der Croupier ausreichend Karten gezogen hat, beendet er das Spiel, die Punkte werden ausgezählt und die Highscores werden aktualisiert.

- Soll ein Spiel geladen werden, kann aus einer Liste verschiedener Spielstände ausgewählt werden. Die Auswahl kann über einen Button bestätigt werden. Alternativ ist ein Zurückkehren zum Hauptmenü über einen weiteren Button möglich.

- Das Löschen eines Spielstandes erfolgt analog zum Laden.

- Die Highscores beinhalten eine absteigend sortierte Liste der Höchstpunktzahlen aller Spieler. Dabei werden die Spieler über ihre selbstgewählten Namen identifiziert. Folglich betrachtet das Spiel Namen als eindeutig und betrachtet zwei namensgleiche Spieler als identisch.

Benutzeroberfläche

Screenshots der einzelnen Spielbildschirme und Mneüs sind im Folgenden abgebildet.

Hauptmenü

"New Game" ist nur sichtbar, wenn es maximal neun gespeicherte Spielstände gibt, "Load Game" und "Delete Game" nur, wenn es mindestens einen gespeicherten Spielstand gibt, und "Show Highscores" nur, wenn Highscores in der Datenbank gespeichert sind.

Figure 1.1. Hauptmenü

Neues Spielmenü

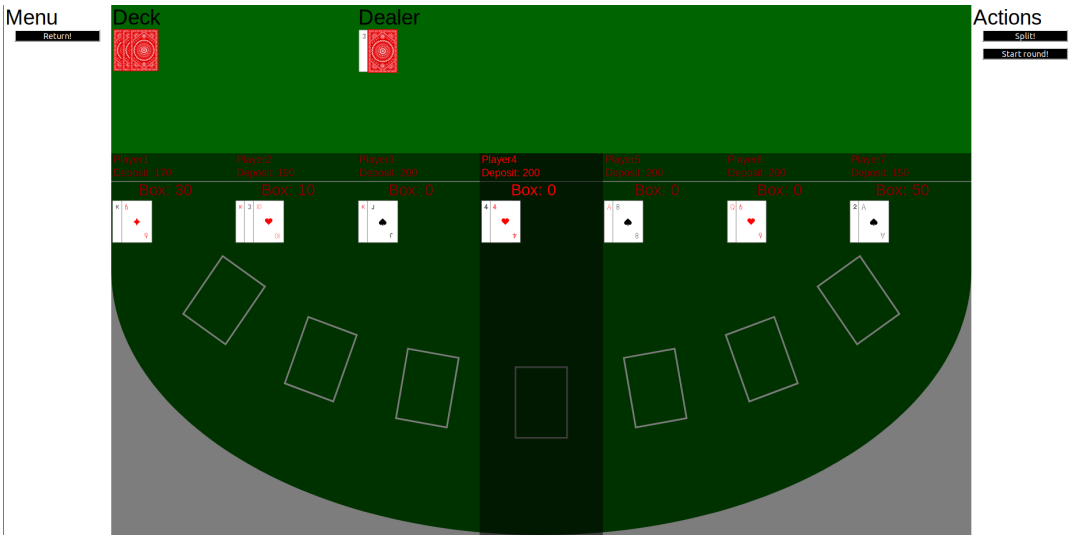
Namen können frei eingegeben werden, "Add Player" und das zugehörige Texteingabefeld sind nur sichtbar, wenn maximal sieben Spieler bereits hinzugefügt wurden, und "Start Game" ist nur sichtbar, wenn bereits mindestens ein Spieler hinzugefügt wurde.

Figure 1.2. Neues Spielmenü

Spielbildschirm

Je nach Zustand des Spiels und der einzelnen Spieler können der Croupier und die Spieler Wetteinsätze tätigen (Bet!), den Zug beenden (End turn!), den Zug beginnen (Start turn!), ihre Hand splitten (Split!), eine Karte ziehen (Draw card!), den Wetteinsatzdurchgang beenden (End betting!) oder das Spiel beenden (End game!). Dabei können die ersten vier Aktionen nur durch Spieler und die letzen zwei nur durch den Croupier initiiert werden. Zudem kann jederzeit über "Return!" zum Hauptmenü zurückgekehrt werden.

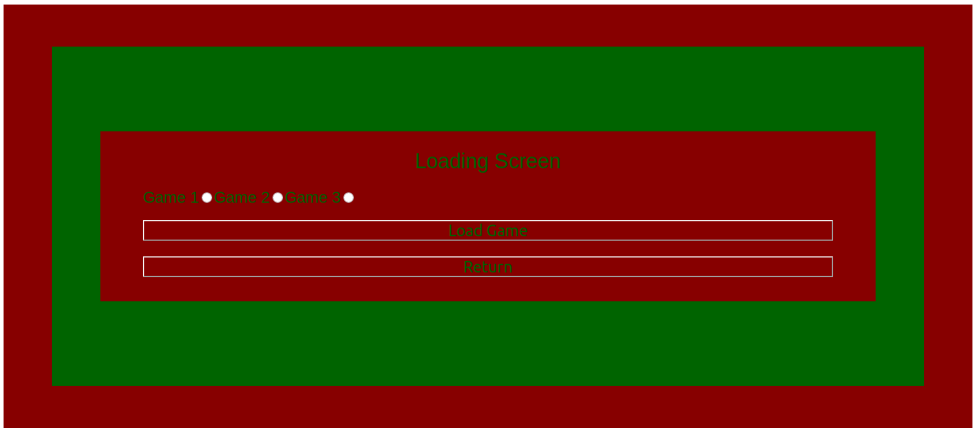
Figure 1.3. Spielbildschirm



Ladebildschirm

Es kann einer der vorhandenen Spielstände ausgewählt werden. Die Auswahl wird über "Load Game" bestätigt. Zudem kann über "Return" zum Hauptmenü zurückgekehrt werden.

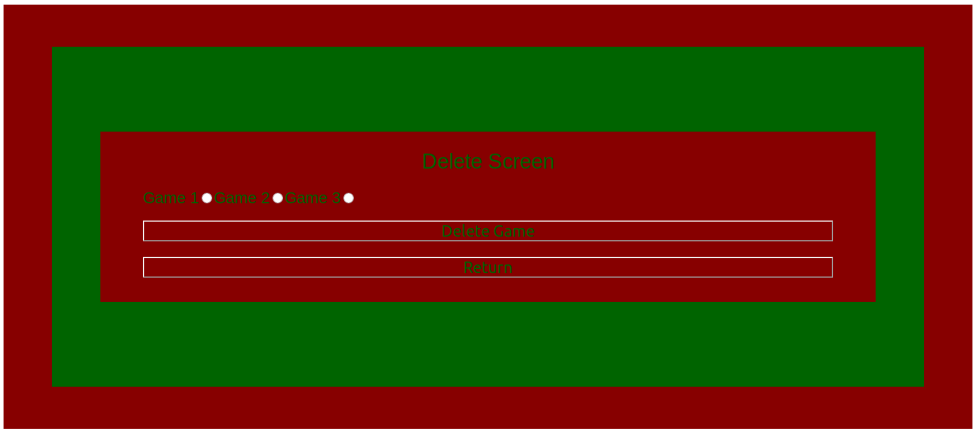
Figure 1.4. Ladebildschirm



Löschbildschirm

Es kann einer der vorhandenen Spielstände ausgewählt werden. Die Auswahl wird über "Delete Game" bestätigt. Zudem kann über "Return" zum Hauptmenü zurückgekehrt werden.

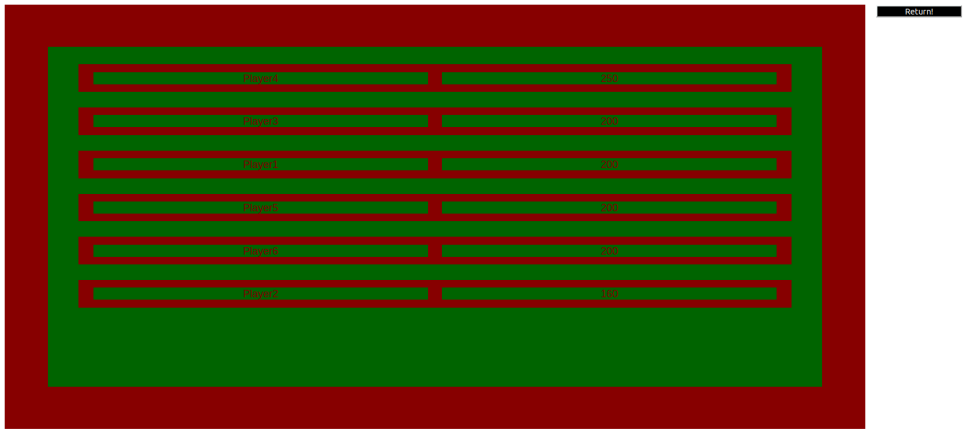
Figure 1.5. Löschbildschirm



Highscores

Für jeden Spieler, der eine komplette Spielrunde abgeschlossen hat, ist ein einziger Eintrag im Highscore zu sehen. Spieler werden dabei über ihren Namen identifiziert, weswegen EInträge aus verschiedenen Spielen fürhere Einträge überschreiben können, sofern sie den Punktestand übertreffen. Die Eintrage sind nach Punkteständen absteigend sortiert.

Figure 1.6. Highscores

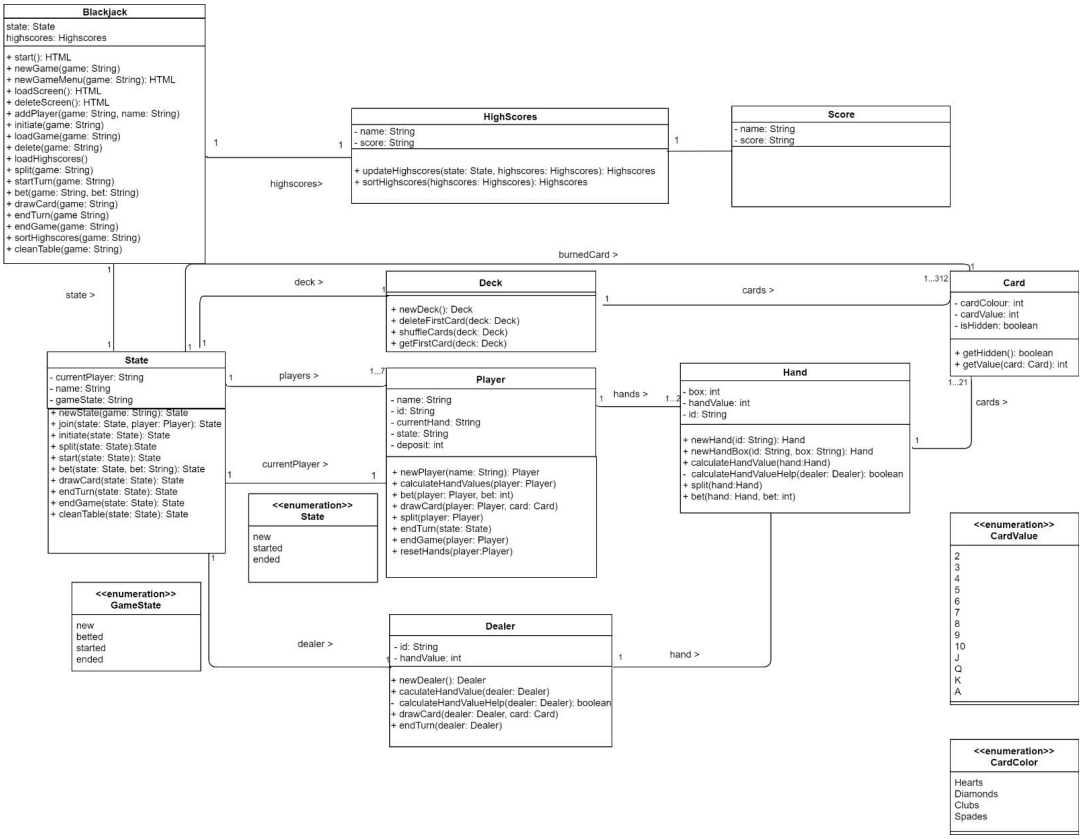


Chapter 2. Diagramme

UML-Diagramm

Um das Spiel übersichtlich darstellen zu können, ist im Folgenden ein implementierungsnahes UML-Diagramm unseres Programms zu sehen. Es ist dabei zu beachten, dass dieses Diagramm nicht vollständig mit unserer Implementierung übereinstimmt, da nicht sämtliche Funktionalitäten von XQuery in UML abbildbar sind; Beispielsweise symbolisieren wir Referenzen im UML-Diagramm durch Referenzattribute, um den Bezug zwischen verschiedenen Elementen hervorzuheben.

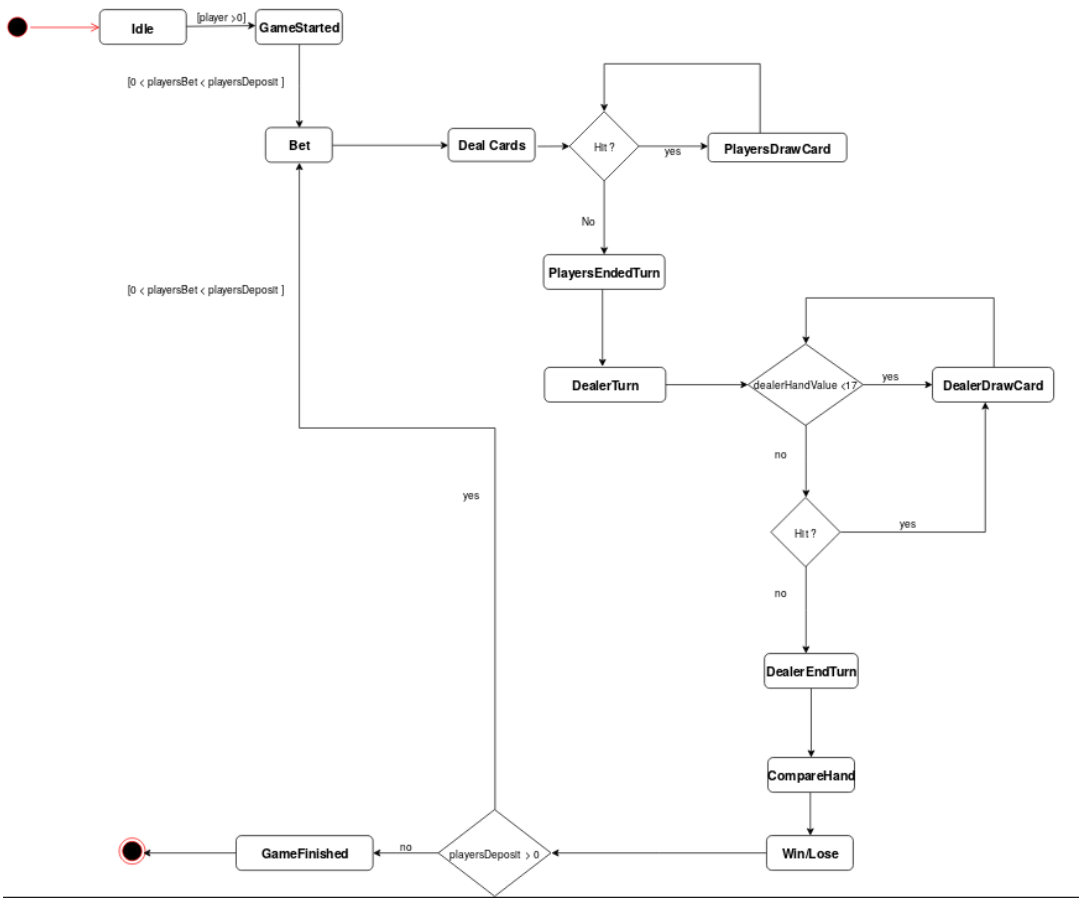
Figure 2.1. UML-Diagramm



State-Diagramm

Um Zustandsübergänge unseres Programms und damit mögliche Programmabläufe darzustellen, ist im Folgenden ein State-Diagramm abgebildet, das den beispielshaften Ablauf eines Spiels darstellt.

Figure 2.2. State-Diagramm



Chapter 3. Implementierung

Datenspeicherung: DTD & XML

Für die Implementierung des XMLs ist das DTD basierend auf dem UML-Diagramm wie folgt strukturiert: Es enthält die sechs Elemente State, Deck, Dealer, Player, Hand und Card. Alle Attribute der Elemente sind mit dem Schlüsselwort #REQUIRED deklariert, wodurch die zugehörigen Typen im XML angegeben werden müssen. CDATA bezeichnet hier "character"-Daten.

Figure 3.1. DTD

```
<!ELEMENT state (deck,dealer,player*)>
<!ATTLIST state
  currentPlayer (PER-1|PER-2|PER-3|PER-4|PER-5|PER-6|PER-7|Dealer) #REQUIRED
  name CDATA #REQUIRED
  gameState (new|betted|started|ended) #REQUIRED>

<!ELEMENT deck (card+)>

<!ELEMENT dealer (card*)>
<!ATTLIST dealer
  id (Dealer) #FIXED "Dealer"
  handValue CDATA #REQUIRED>

<!ELEMENT player (hand+)>
<!ATTLIST player
  id (PER-1|PER-2|PER-3|PER-4|PER-5|PER-6|PER-7) #REQUIRED
  currentHand (PER-1-1|PER-1-2|PER-2-1|PER-2-2|PER-3-1|PER-3-2|PER-4-1|PER-4-2|PER-5-1|PER-5-2|PER-6-1|PER-6-2|PER-7-1|PER-7-2) #REQUIRED
  name CDATA #REQUIRED
  state (new|started|ended) #REQUIRED
  deposit CDATA #REQUIRED>


<!ELEMENT hand (card*)>
<!ATTLIST hand
  id (PER-1-1|PER-1-2|PER-2-1|PER-2-2|PER-3-1|PER-3-2|PER-4-1|PER-4-2|PER-5-1|PER-5-2|PER-6-1|PER-6-2|PER-7-1|PER-7-2) #REQUIRED
  box CDATA #REQUIRED
  handValue CDATA #REQUIRED>

<!ELEMENT card EMPTY>
<!ATTLIST card
  cardColour (Hearts|Diamonds|Clubs|Spades) #REQUIRED
  cardValue (2|3|4|5|6|7|8|9|10|J|Q|K|A) #REQUIRED
  isHidden (true|false) #REQUIRED>
```

State

Das Element state enthält die drei Unterelemente deck, dealer und player. Hierbei repräsentiert deck das Kartendeck; dealer den Croupier. Das Sprachmittel + bei player stellt sicher, dass mindestens ein Spieler am Tisch sitzt. Durch das Attribut "currentPlayer" wird der Spieler referenziert, der aktuell an der Reihe ist. Das Attribut "gameState" bezeichnet den aktuellen Zustand des gesamten Spiels im Spielablauf.

Deck

Dieses Element enthält als einziges Unterelement card, wobei das Sprachmittel + anzeigt, dass das Deck aus mindestens einer, möglicherweise mehreren, Karten besteht.

Dealer

Dieses Element enthält als einziges Unterelement card mit dem Sprachmittel *, welches bedeutet, dass der Croupier keine, eine oder mehrere Karten haben kann. Dem dealer-Element ist fest die id "Dealer" zugewiesen und ein Attribut handValue, das im Spieldurchlauf konstant aktualisiert wird, speichert den Wert der enthaltenen Karten.

Player

Dieses Element besitzt als einziges Unterelement hand. Es ist durch das Sprachmittel + gekennzeichnet, welches angibt, dass der Spieler mindestens eine Hand hat. Das Element player hat zudem die

folgenden fünf Attribute: id für die Identifikation des Spielers, currentHand zur Referenzierung der aktuell aktiven Hand, name als Spielername, state als Beschreibung des aktuellen Zustands des Spielers im Spielablauf und deposit als Anzeige des Vermögens.

Hand

Dieses Element enthält als einziges Unterelement card mit dem Sprachmittel *, welches anzeigt, dass jede hand aus keiner, einer oder mehreren Karten besteht. Die drei Attribute id, box und handValue dienen der Identifikation des hand-elements, der Beschreibung des Wetteinsatzes und der Beschreibung des Werts der enthaltenen Karten.

Card

Dieses Element enthält keine Unterelemente, was durch EMPTY angezeigt wird. Die drei Attribute des Elements - cardColour, cardValue und isHidden - beschreiben die Karte, wobei letzteres anzeigt ob sie verdeckt ist.

Randbemerkung

Da wir die ids selbst festlegen, verzichten wir auf die Attributtypen ID und IDREF und verwenden stattdessen festgelegt Wertemengen.

Datenpräsentation: XSL & SVG

Zur Transformation von XML in HTML verwenden wir XSL und darin eingebettetes sowie aus externen Dateien eingebundenes SVG.

Gerüst: XSL

Wir verwenden sechs Templates für die Konvertierung der fünf einzelnen Elemente State, Deck, Dealer, Player und Card. Für Card werden zwei verschiedene Templates verwendet. Insbesondere erzeugt die Formatierung mittels Inline-CSS ein responsives Layout, d.h. die Elemente skalieren mit der Größe des Browserfensters.

Figure 3.2. XSL Überblick

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet exclude-result-prefixes="xs" version="2.0" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="state">                                                    [179 lines]
    <xsl:template match="deck">                                                     [24 lines]
    <xsl:template match="dealer">                                                   [7 lines]
    <xsl:template match="player">                                                   [47 lines]
    <xsl:template name="DealerCard">                                                [49 lines]
    <xsl:template name="PlayerCard">                                                [48 lines]
</xsl:stylesheet>
```

Das erzeugte HTML beinhaltet links ein Menü, um zum Hauptmenü zurückzukehren, mittig die Darstellung des aktuellen Spielzustandes und rechts die möglichen Aktionen, welche über Formulare und Buttons initiiert werden.

Figure 3.3. XSL State

```
<xsl:template match="state">
    <html style="height:100%; font-family: 'Arial', 'Times New Roman'; font-size: 2vmin; overflow-wrap: break-word;">
        <head>
            <title>Southpark Blackjack</title>
        </head>
        <body style="height:100%; margin: 0px;">
            <div style="width: 10%; height: 100%; float: left;">
                <div style="width: 98.572%; margin-left: 1.428%; font-size: 4vmin;">Menu</div>
                <button onclick="location.href='../htmlforms';" type="button" style="width: 80%;
                    margin: 0% 10% 0% 10%; background-color: black; color: white;"
                    onmouseover="this.style.backgroundColor='white'; this.style.color='black'"
                    onmouseout="this.style.backgroundColor='black'; this.style.color='white'">
                    Return!
                </button>
            </div>
            <div style="width: 80%; height: 100%; float: left; background-image:
                url(../static/htmlforms/svgs/playTable.svg); background-size: 100% 100%;">
                <xsl:apply-templates/>
            </div>
            <div style="width: 10%; height: 100%; float: left;">
                <div style="width: 98.572%; margin-left: 1.428%; font-size: 4vmin;">Actions</div>
                <!-- START OF BUTTON SECTION -->
                <xsl:variable name="game" select="./@name"/>
                <xsl:variable name="ifGameNew" select="./@gameState = 'new'"/>
                <xsl:variable name="ifGameStarted" select="./@gameState = 'started'"/>

                <xsl:choose>                                                                    [148 lines]
                <!-- END OF BUTTON SECTION -->
            </div>
        </body>
    </html>
</xsl:template>
```

Da die möglichen Aktionen vom Spielzustand abhängig, sind nur solche Aktionen möglich, die zum jeweiligen zeitpunkt sinnvoll sind. Alle anderen Aktionen sind ausgeblendet. Die Wahl der anzuzeigenden Elemente erfolgt mittels "xsl:choose" und spielzustandsabhängigen "xsl:variable".

Figure 3.4. XSL Dealerbuttons in State

```
<xsl:when test="./@currentPlayer = 'Dealer'">
    <xsl:choose>
        <xsl:when test="$ifGameNew">
            <button onclick="location.href='endTurn?game={$game}';" type="button" style="width: 80%;
                margin: 0% 10% 5% 10%; background-color: black; color: white;"
                onmouseover="this.style.backgroundColor='white'; this.style.color='black'"
                onmouseout="this.style.backgroundColor='black'; this.style.color='white'">
                End betting!
            </button>
        </xsl:when>
        <xsl:when test="$ifGameStarted">
            <xsl:choose>
                <xsl:when test="17 > ./dealer/@handValue">
                    <!-- if the dealer's handValue is less than 17, he must pick more cards -->
                    <button onclick="location.href='draw?game={$game}';" type="button"
                        style="width: 80%; margin: 0% 10% 0% 10%; background-color: black;
                        color: white;" onmouseover="this.style.backgroundColor='white';
                        this.style.color='black'" onmouseout="this.style.backgroundColor='black';
                        this.style.color='white'">
                        Draw card!
                    </button>
                </xsl:when>
                <xsl:otherwise>
                    <button onclick="location.href='endTurn?game={$game}';" type="button" style="width: 80%;
                        margin: 0% 10% 5% 10%; background-color: black; color: white;"
                        onmouseover="this.style.backgroundColor='white'; this.style.color='black'"
                        onmouseout="this.style.backgroundColor='black'; this.style.color='white'">
                        End game!
                    </button>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:when>
    </xsl:choose>
</xsl:when>
```

State

Im zugehörigen Template wird das Grundgerüst des resultierenden HTML-Dokuments gelegt. Der Spieltisch wird als Hintergrundgrafik eingebunden und weitere Templates werden aufgerufen.

Deck

In diesem Template werden lediglich drei Kartenrückseiten als SVGs zur Repräsentation des Kartendecks ausgegeben.

Dealer

In diesem Template wird lediglich das Template für die Karten des Croupiers aufgerufen.

Player

Zunächst werden in diesem Template der Name und das Vermögen des jeweiligen Spielers ausgegeben. Anschließend werden bis zu drei Hände ausgegeben. Die Ausgabe für jede einzelne Hand setzt sich aus der Ausgabe der zugehörigen Wette und einem Aufruf des Templates für die Karten des Spielers zusammen.

Card

Wir haben je ein XSLT-Template für die Generierung der Karten des Spielers und des Croupiers zu unserer XSL-Datei hinzugefügt. Das Template für den Croupier unterschiedet sich vom Template für den Spieler dahingehend, dass es eine Fallunterscheidung nach verdeckten und offenen Karten trifft. Für erstere wird lediglich ein Kartenhintergrund ausgegeben. Letztere werden ebenso behandelt wie es mit den Karten des Spielers geschieht: Aus Basis der Attributwerte des zugrundeliegenden XML-Elements vom Typ "card" für Kartenfarbe und Kartenwerte werden Pfade erzeugt und somit SVGs schrittweise zu einer vollständigen Karte zusammengesetzt. Hierdurch kann Code-Redundanz reduziert werden, da nicht für jede einzelne Karte ein separates Codesegment zur Generierung erforderlich ist, sondern sämtliche Karten durch ein einzelnes Codesegment generiert werden können.

Graphiken: SVG

Graphische Element werden mittels SVG erzeugt und zusammnegesetzt, um anschließend durch XSL aufgerufen und kombiniert werden zu können.

Benutzeroberfläche

Die Benutzeroberfläche des Blackjack Spiels besteht aus jeweils statischen und dynamischen Teilen.

Spieltisch

Der Spieltisch ist total statisch. Er besteht aus zwei graphischen SVG-Basiselementen wie Ellipse und Rechteck.

Karte

Um unsere Aufgaben zu erleichtern und nicht 52 separate Karten zu schreiben, definieren wir XSL, sodass alle Karte dynamisch generiert werden. Dabei verwenden wir vorallem die folgenden statischen SVGs.

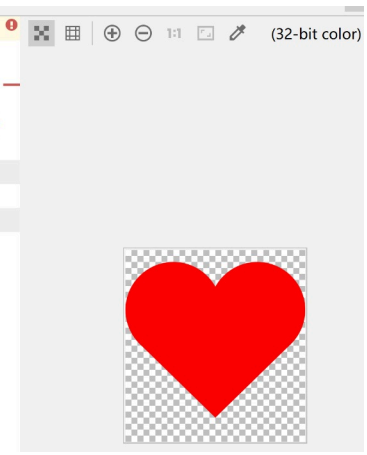
Kartenfarbe

Vier SVGs werden für Kartenfarben geschrieben, jeweils Kreuz, Pik, Herz und Karo. Solche Kartenfarben bestehen aus mehreren graphischen SVG-Basiselementen wie Kreis, Pfad, Polygonzug, etc.

```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
        "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="150" height="160" version="1.1" xmlns="http://www.w3.org/2000/svg">
    <title>heart</title>
    <circle cx="40" cy="50" r="40" stroke=""
            stroke-width="2" fill="red"/>
    <circle cx="110" cy="50" r="40" stroke=""
            stroke-width="2" fill="red"/>
    <rect x="40" y="60" width="80" height="20" fill="red"/>
    <polygon  fill="red" points="11.5,78 138.5,78 75,140"/>

</svg>
```



Kartenwert

Die Kartenwerte sind komplizierter umzusetzen als die Kartenfarben, da schwarze und rote Werte sowie Buchstaben A / J / Q / K und Zahlen von 2 bis 10 für beide Farben benötigt werden. Deswegen haben wir insgesamt 26 SVGs für die Werte geschrieben. Solche Kartenwerte bestehen auch aus mehreren graphischen SVG-Basiselementen wie Kreis, Pfad, Polygonzug usw.

Kartenhintergrund

Um die Karte hübscher zu machen, malen wir dabei auch ein hohles Rechteck durch SVG.

Kartenrückseite

Außerdem fügen wir die klassische Kartenrückseite als SVG-Bild-Element ins unser Projekt ein.

Randbemerkungen

Nachdem wir alle die erwähnten SVGs geschrieben haben, kann unser Server durch die XSLT die Karten dynamisch erstellen. Jede Karte besteht dabei aus drei Komponenten, erstens aus dem Hintergrund, zweitens aus der Kartenfarbe und zuletzt aus dem Kartenwert. Das Produkt kann mittels XSL in unsere Darstellung eingefügt werden.Somit verbinden wir die immer gleiche Darstellung typgleicher Karten mit der Vermeidung von Redundanz bei ihrer Generierung.

```xml
<svg version="1.1" viewBox="0 0 160 220" width="100%"
     xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
    <image height="220" width="160" x="0" xlink:href="../static/htmlforms/svgs/cardBackground.svg" y="0"/>
    <image height="40" width="40" x="60" xlink:href="" y="90">
        <xsl:attribute name="xlink:href">
            ../static/htmlforms/svgs/<xsl:value-of select="@cardColour"/>.svg
        </xsl:attribute>
    </image>
    <image height="40" width="40" x="10" xlink:href="" y="15">
        <xsl:attribute name="xlink:href">
            ../static/htmlforms/svgs/<xsl:choose>
            <xsl:when test="@cardColour = 'Spades'">Black</xsl:when>
            <xsl:when test="@cardColour = 'Clubs'">Black</xsl:when>
            <xsl:otherwise>Red</xsl:otherwise>
        </xsl:choose><xsl:value-of select="@cardValue"/>.svg
        </xsl:attribute>
    </image>
    <g transform="translate(30,35) rotate(180) translate(-125, -180)">
        <image height="40" width="40" x="10" xlink:href="" y="15">
            <xsl:attribute name="xlink:href">
                ../static/htmlforms/svgs/<xsl:choose>
                <xsl:when test="@cardColour = 'Spades'">Black</xsl:when>
                <xsl:when test="@cardColour = 'Clubs'">Black</xsl:when>
                <xsl:otherwise>Red</xsl:otherwise>
            </xsl:choose><xsl:value-of select="@cardValue"/>.svg
```

Datenverarbeitung: XQuery

Die Struktur der XQuery-Dateien ist weitestgehend - sofern möglich - nach dem Prinzip der Datenkapselung entworfen. Blackjack dient dabei als Einstiegspunkt des Programms, Sammelstelle für sämtliche REST-Endpunkte und Schnittstelle zur BaseX-Datenbank. Hier werden Menüs erzeugt und XML-Dokumente durch Serialisierung mit Dokumententypdefinitionen und durch Verarbeitungsinstruktionen mit XSL-Transformationsszenarios verbunden. Außerdem werden hier REST-Pfade mit Funktionen verknüpft und REST-Queryparameter ausgelesen. Zuletzt werden hier Datenbankinhalte ausgelesen, aktualisiert und neu erzeugt. All dies erfolgt an keiner anderen Stelle. Durch die Verkettung von Funktionen, endet jeder Aufruf einer Funktionen im Hauptmenü, dem Hauptspielbildschirm, dem Lade- oder Löschbildschirm, oder der Präsentation der Highscores. Folglich werden die Spielstände konstant aktualisiert und die aktualisierten Spielstände automatisch in der Datenbank gesichert, wodurch ein manuelles Speichern unnötig ist. Ebenso werden abgeschlossene Spielstände automatisch aus der Datenbank gelöscht, weswegen auch ein manuelles Löschen unnötig ist. Letzteres ist in Gegensatz zu ersterem jedoch möglich. Da es sich häufig um mit "%updating" gekennzeichnete Funktionen handelt, müssen "web:redirect"-Elemente mit "update:output"-Wrapperelementen versehen werden, die Return-Ausdrücke als Update-Ausdrücke kaschieren.

Figure 3.5. XQuery Blackjack

```
declare
    %updating
    %rest:path("/htmlforms/add")
    %rest:query-param("game", "{$game}")
    %rest:query-param("name", "{$name}")
function bj:addPlayer ($game as xs:string, $name as xs:string){
    let $state := state:join(db:open("htmlforms", fn:concat("games/", $game, ".xml"))/state, $name)
    return db:replace("htmlforms", fn:concat("games/", $game, ".xml"), $state),
    update:output(web:redirect(fn:concat("/htmlforms/newMenu?game=", $game)))
};

declare
    %updating
    %rest:path("/htmlforms/init")
    %rest:query-param("game", "{$game}")
function bj:initiate ($game as xs:string){
    let $state := state:initiate(db:open("htmlforms", fn:concat("games/", $game, ".xml"))/state)

    return db:replace("htmlforms", fn:concat("games/", $game, ".xml"), $state),
    update:output(web:redirect(fn:concat("/htmlforms/load?game=", $game)))
};

declare
    %rest:path("/htmlforms/load")
    %rest:query-param("game", "{$game}")
    %output:method("xml")
    %output:omit-xml-declaration("no")
    %output:doctype-system("../static/htmlforms/blackjack.dtd")
function bj:loadGame ($game as xs:string){
    document {
        processing-instruction xml-stylesheet {
            'type="text/xsl" href="../static/htmlforms/blackjack.xsl"'
        },
        db:open("htmlforms", fn:concat("games/", $game, ".xml"))/node()
    }
};
```

State und Highscores sind die einzigen direkt aus Blackjack erreichbaren XQuery-Dateien. Während in Highscores lediglich das Aktualisieren der Highscores zum Ende einer Spielrunde umgesetzt ist, bildet State die Schnittstelle zwischen Blackjack und Deck, Dealer und Player. Funktionsaufrufe aus Blackjack von State-Funktionen geben aktualisierte State-Elemente zurück. Hierzu werden Elemente von State mit "update" und einem Update-Ausdruck versehen und sodann zurückgegeben. Dies kaschiert "updating"-Ausdrücke als Return-Ausdrücke.

Figure 3.6. XQuery State

```
declare function state:initiate($state as element(state)) as element(state){
    if ($state/@currentPlayer eq "Dealer") then (
        state:endTurn(state:drawCard(state:drawCard($state)))
    ) else (
        if ($state/player[@id eq ../@currentPlayer]/@state eq "ended") then (
            state:initiate(state:endTurn($state))
        ) else (
            state:initiate(state:endTurn(state:drawCard(state:drawCard($state))))
        )
    )
};

declare function state:drawCard($state as element(state)) as element(state){
    if ($state/@currentPlayer eq "Dealer") then (
        $state update {
            dealer:drawCard(./dealer, deck:getFirstCard(./deck))
        } update {
            deck:deleteFirstCard(./deck)
        } update {
            dealer:calculateHandValue(./dealer)
        }
    ) else (
        $state update {
            player:drawCard(./player[@id eq ../@currentPlayer], deck:getFirstCard(./deck))
        } update {
            deck:deleteFirstCard(./deck)
        } update {
            player:calculateHandValues(./player[@id eq ../@currentPlayer])
        }
    )
};
```

Das Kaschieren ermöglicht das Mischen von Update- und Return-Ausdrücken, was unsere Art des Umgangs mit dem Update-Constraint darstellt.

Bis auf besonders einfache Updates, die in State durchgeführt werden, sind die Updates der Spielzustände in Deck, Dealer, Player, Hand und Card implementiert, wobei der Zugriff auf Hand über Player und auf Card über Deck, Hand und Dealer erfolgt. In welcher XQuery-Datei eine Update-Funktion implementiert ist, hängt von dem Unterelement ab, das den logischen Kontext der Funktion bildet. Dieses stimmt meist, aber nicht immer, mit dem Unterelement überein, das der Funktion als Parameter übergeben wird. Ein Gegenbeispiel ist die Funktion "player:endTurn", der ein State-Element übergeben wird, da nicht nur ein Player-Element, sondern auch ein Dealer-Element abgeändert wird.

Figure 3.7. XQuery Player

```
declare
    %updating
function player:endTurn($state as element(state)){
    let $previousPlayerId := $state/@currentPlayer
    let $previousPlayer := $state/player[@id eq $previousPlayerId]
    let $previousHandId := $previousPlayer/@currentHand
    let $handAmount := count($previousPlayer/hand)

    return if ($handAmount > 1 and fn:substring(xs:string($previousHandId), 7, 1) eq "1") then (
        let $currentHandId := fn:concat($previousPlayerId, "-2")

        return replace value of node $previousHandId with $currentHandId
    ) else (
        let $currentHandId := fn:concat($previousPlayerId, "-1")

        return if ($previousPlayerId eq $state/player[last()]/@id) then (
            let $currentPlayerId := "Dealer"

            return if ($state/@gameState eq "started") then (
                replace value of node $state/dealer/card[2]/@isHidden with fn:false(),
                replace value of node $previousHandId with $currentHandId,
                replace value of node $previousPlayerId with $currentPlayerId
            ) else (
                replace value of node $previousHandId with $currentHandId,
                replace value of node $previousPlayerId with $currentPlayerId
            )
        ) else (
            let $currentPlayerId := fn:concat(
                "PER-",
                xs:string(xs:integer(fn:substring(xs:string($previousPlayerId), 5, 1)) + 1)
            )

            return (replace value of node $previousHandId with $currentHandId,
            replace value of node $previousPlayerId with $currentPlayerId)
        )
    )
};
```

Chapter 4. Installationsanleitung

Das Spiel benötigt eine BaseX-Installation mit laufendem BaseX-WebServer und einer Datenbank "htmlforms". In dieser sind keine Dateien anzulegen, da dies durch das Programm vorgenommen wird. Unter "restxq" und "webapp/static" sind jeweils die zugehörigen Ordner namens "htmlforms" abzulegen. Mit Standardeinstellungen von BaseX ist unser Programm dann unter "localhost:8984/htmlforms" zu erreichen.

Chapter 5. Zielerreichung

Im Rahmen des Projektspiels hat die Gruppe Wissen über mehrere XML-Bestandteile erworben. Die Gruppe beherrscht XML, XSLT und XQuery und kann sich gut mit ihren Instanzen und Namespaces auseinandersetzen. Mithilfe der ersten Aufgaben am Anfang des Semesters kann die Gruppe XML-Dokumenten parsen und mithilfe von CSS Files in Web-Browsern visualisieren. Für den Entwurf des Spiels und für die Designentscheidung hat die Gruppe SVG-Dateien unter Verwendung von XLink implementiert. Dazu zählt auch die Dokumententypdefinition für die Modellierung der Struktur von XML-Dateien. Um über XML-Anwendungen zu navigieren hat die Gruppe mithilfe von XPath und XSLT-Transformationen der XML-Dokumenten implementiert. In unserem Projekt spielen XQuery-Funktionen eine große Rolle, um die Funktionalität des Spiels zu erreichen. Dazu zählen der Aufruf von statischen Webdateien, die Implementierung von RestXQ-Endpunkte und der Zugriff auf eine BaseX-Datenbank. In unserem Projekt verwenden wir keine XForms, da diese deutlich aufwändiger umzusetzen wären als direkte Datenbankzugriffe und somit einfache HTML-Forms ausreichend sind. Am Ende der Implementierung des Projektspiels ist die Gruppe auch in der Lage, den gesamten Projektentwurf und die gesamte Projektumsetzung mithilfe von DocBook zu dokumentieren.

Innerhalb des Praktikums sind die oben genannten XML-Technologien in Teamarbeit erarbeitet worden. Die Gruppe hat sich wöchentlich nach der Vorlesung getroffen, um den Lernstoff auszustauschen und sich auf die nächste Aufgabe vorzubereiten. In der Grupper herrschten gute Kommunikation, Spaß am Projekt und aktives Erbringen von Feedback und Ideen durch alle Gruppenmitglieder vor.

Chapter 6. Online-Literatur

- https://www.w3schools.com/xml/
- http://www.w3.org/XML/
- [https://wiki.selfhtml.org/wiki/XML]

Chapter 7. Buch-Literatur

- E.R. Harold u.a.: XML in a Nutshell. O'Reilly 2001.
- G. Kappel u.a.: Web Engineering. DPunkt 2003.
- J. Fawcett, L. Quin, D. Ayers: Beginning XML. Wrox 2012.
- Moller, M. Schwartzbach: An Introduction to XML and Web Technologies. Addison-Wesley 2006.