GEWERBEOBERSCHULE “MAX VALIER” BOZEN

Blind Watten

an online game realised in JSP

Author: Stefan Peer

Tutor: Prof. Michael Wild

18.05.2007

Index of contents

Introduction

Watten is a sociable card game that was born in South Tyrol about 200 years ago[1]. Now it's more diffused and it's played also in other German speaking countries. But nowhere is it played the way it's played in South Tyrol. In every region there are different rules. Only the main idea is the same everywhere. Because I live in Tramin, I decided to realise the game the way it's played there.

Using the programming language Java Server Pages (JSP) I created a Website[2], on which "Blind Watten" can be played as an online game. It's a browser based, platform independent game, which is based upon HTML code. The web browser is used as the user interface. The web server is responsible for the course and the calculations of the game.

Each group of every 4 players takes part in a game. There is no registration or login needed to participate in games and not even to create new games. For an online game this could be useful to increase the number of visitors of a site, because users prefer playing online games, if they haven't to register for playing.

[1] Source: http://de.wikipedia.org
[2] http://watten.peerweb.it

Rules of the game

Watten is played by four people. Each two people play as a team and sit opposite each other. Each player is dealt five cards (the remainder are unused) and these are played out in tricks. The team that takes the majority of the tricks (three or more) normally wins two points. The game contains 33 cards which have a hierarchy.

All cards have one of the four suits: bells, leaves, heart and acorn. Cards do also have a value. Values are: seven, eight, nine, ten, under, over, king and ace. In addition to these 32 cards (4 suits * 8 values), there exists another card, the six of the bells called Weli.

Course

First all cards are shuffled and each player is dealt five cards. As soon as the cards are distributed, two players (called dealer and forehand) from different teams have to announce. This they can do by choosing the trump and the Schlag. One player has to choose the suit (trump), the other the value (Schlag). The special feature of Blind Watten is that the Schlag and trump are not announced out loud. Instead forehand privately shows the dealer a card whose rank is the Schlag, and the dealer then similarly shows forehand a card whose suit is trump. The partners of the dealer and forehand will only be able to deduce what the Schlag and trump are by observing the play.

Forehand, the player to dealer's left, leads to the first trick. He has to throw the first card. As soon as he has done this, the player who sits on forehand's left, must throw a card. This goes round clockwise, until every one of the four players has put down a card, so that four cards are on the table. The card with the highest rank wins a trick. The player who has thrown this card picks up all cards and then it's his turn to throw the next card. After that it's the next players turn, and so on.

The object of the play is to win three tricks. After a team has achieved this there is no need to continue the play. Any remaining cards are thrown in and the hand is scored. The team that has won three tricks normally scores 2 points. After that, all cards are shuffled and dealt again. The player who has chosen Schlag in the previous round is now the dealer and the player on his left becomes the forehand.

The game is finished, when one of the teams scores at least 18 points.

Ranking and names of cards

1) The good one – the card with the suit of trump and the value + 1 of the Schlag (is the best card in the game).
   (If ace is the Schlag, then the trump Seven is the good one)
2) The right one – the card with the suit of the trump and the value of the Schlag
3) The blinds – the cards with the value of the Schlag (first thrown before later thrown)
4) The rest of the trump cards (higher before lower)
5) All other cards (higher before lower)

Additional rules

- Weli

  The Weli is the thirty-third card in the game. Normally it is the Bell Six, but if a player chooses the Weli as Schlag, it is the second best card. The suit of the Weli becomes the colour of the trump. If Weli is announced as Schlag, the good one is the Seven in the trump suit (the right one is the Weli). The blinds do not exist.

- Demand for better cards ("Schönere")

  If one announcer has unsuitable cards, he has the possibility to ask for better cards, before Schlag and trump are chosen. The other announcer can accept or decline the query. If he refuses to ask for better cards nothing changes, otherwise both announcers receive new cards.

- Demand for exchange of Schlag ("Schlagtausch")

  Like asking for better cards, one announcer (the dealer or forehand) has the possibility to ask for exchange of Schlag. In this case, the roles of the announcers change. Forehand must choose the trump and the dealer the Schlag. After that, forehand has to throw the first card.

- Betting

  If there is no betting, the team that wins three tricks scores 2 points. During the play, a team can propose to increase the number of points by betting. At any time after the Schlag and trump have been announced, any player can bet on behalf of his team: the announcer of the other team must choose whether to give up and get for 2 points or continue playing for 3. If he decides to go on, his team can later try to increase the stake to 4, and if the original betters accept this they can later try to increase the stake to 5, and so on, up to the number of points needed to win the game. A team can only bet, if it hasn't bet previously. A bet freezes the play, until the announcer of the other team responds.

- Gestrichen

  If a team reaches 16 or 17 points it is "gestrichen". This has only consequences if the difference of points between the teams is greater than 4. In this case 4 is bet automatically after the announcement. The team that has more points has to decide whether it declines or accepts the bet. If it declines it, the other team receives 2 points. Otherwise the game is played normally and the winner scores 4 points.

- Following suit ("Trumpf zugeben")

  If one player throws a trump card as first card after a trick or after the announcement, forehand and the dealer are bound to follow with a trump or with a blind card – of course only if they own such a card. The other two players are not bound by this rule. They can play any card to any trick. If the good one is thrown as first card, the announcers don't have to follow with a suit.

What I had to realise

First of all I planned the structure of the website. I subdivided the concept of my project in two parts. The first part was the creation, participation and the maintenance of games and users. In this fragment nothing matched with the game as such. In the second part I focused on the game “Watten”, with its rules and characteristics.

I made sure that a user can participate only in one game. This made the management of the site much easier and it should also be comprehensible, because a player normally doesn’t play in more games simultaneously.

In this work I often used the word “round”. I defined that a round begins with the announcing and ends when a team has won three tricks – when it has scored some points. After a round ends, a new one begins.

Before- and after playing

Starting a new game

By selecting this option the user can create a new game. First he must insert the name of a game. If no already existing game has the same name, it will be entered in the list of all games. Selecting a game from this list other users can take part in this game.

Participating in a game

After having selected the wanted game from the game list, the user can join this game. He has to insert a unique nickname and must select the team in which he wants to play. After that, the user arrives at a “wait page”, where he has to wait until four players join this game. As soon as four players participate in the same game it can start.

Log out

If one player leaves the game or clicks on “logout”, the game enters a wait state and all players will be lead back to the wait page. Later, when four players join this game again, this can be continued with the old game state. To prevent a player from leaving the game if his team is losing, I defined that the team, in which the dropout plays, loses the actual round. So the points go to the other team. If a player wants to exit the game (because of other reasons) and he won’t give points to the other team, he has to leave the game before the announcement.

If nobody is entered in a game, this game will be deleted after 12 hours. This way I can avoid having a list of old games which are not played anymore.

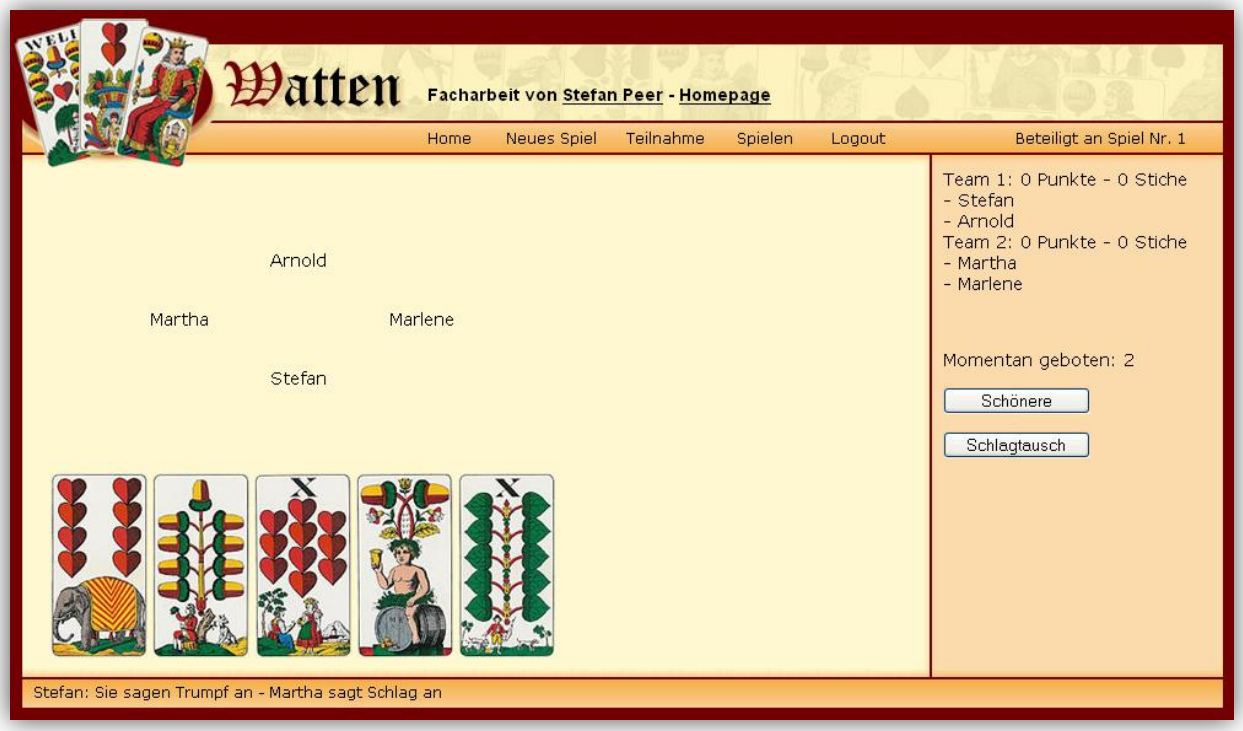
Playing

At the bottom of the page there is a state line. There you'll find the actual game state: which player has what and what he has to do. Above this line the player finds the cards he is holding. Above the cards the player can see who sits on his left, on his right and opposite. Two players who sit opposite each other play in a team. On the right there is a sidebar where you find some game information like the score, the actual bet, and so on. There the players also have the possibility to ask for better cards or for an exchange of Schlag and to bet.

At the top is the navigation menu. Using this menu the player can create new games, join games and log out from the actual game. Clicking on "Home" the player comes to the start site where he can find a link which refers to an instruction page. Clicking on "Spielen", the user comes back to the playing site (only available if he is joining a game).

Announcing

The two players who have to announce are selected by chance. Because it is "Blind Watten", the announcers have to show each other the Schlag and the trump card. This they can do by clicking on a card which they hold in their hand.



On the Sidebar are 2 buttons, with which the announcers can ask for better cards or for exchange of Schlag. These buttons are only available for them before Schlag and trump are chosen. After clicking on one of these buttons, the respective announcer receives a message which gives him the possibility to decide if he wants to accept or decline the request.

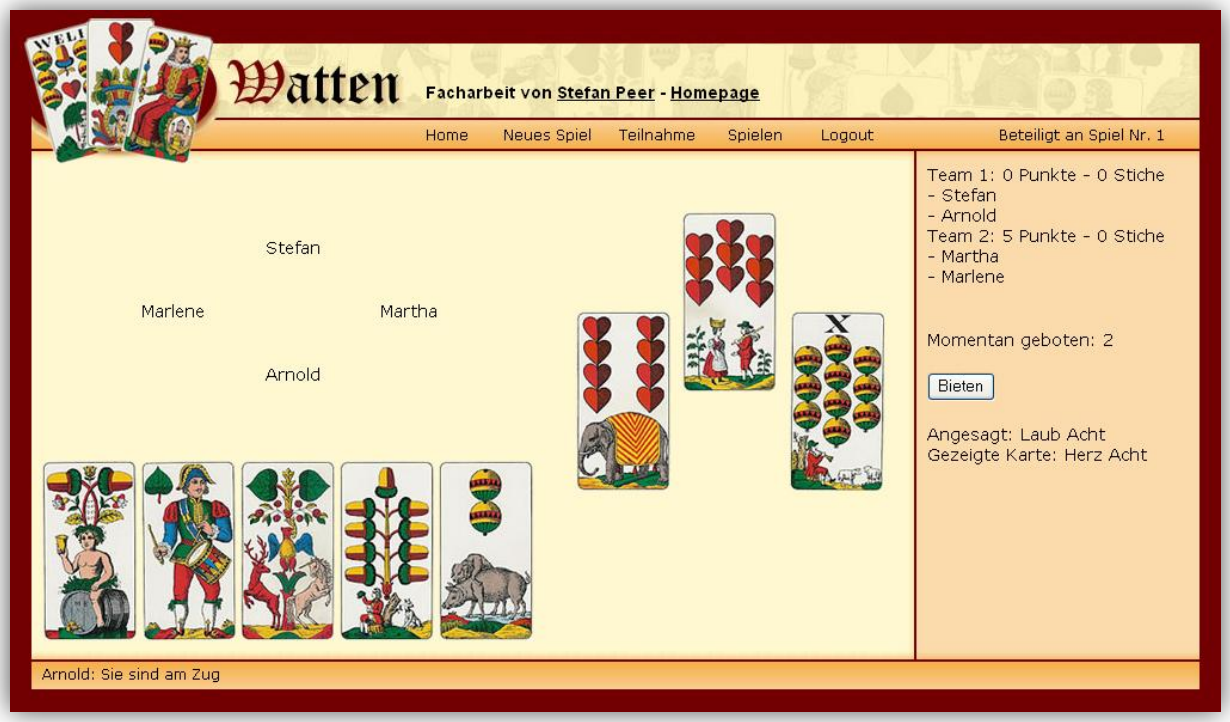
After both announcers click on the card which they want to choose as Schlag or Trump, the announcement is completed.

Throwing cards

After the announcement the announcers can see two card names on the sidebar:

- Announced – what was announced (the right one)
- Showed card – for the dealer this is the card on which forehand clicked and vice versa for forehand

Forehand now has to throw a card. This he can do by clicking on a card which he holds in his hand. After the card has been thrown, the next player has to throw a card, and so on.



The game table is positioned on the sidebars' left. There each player finds the cards which the other players have thrown.

On the sidebar there is a button for each player to bet (only visible if he's allowed to). After one player has clicked on this button, the announcer of the other team receives a message which gives him the possibility to either accept or decline the request.

As soon as one team reaches 18 points the game is finished. It then restarts.

Java beans – means to realise the game

I decided to realise the game “Blind Watten” as an online game in JSP. Java Server Pages is based on the object orientated programming language Java. Having programmed the so called Java beans (Java classes) they now take over all the management of the game. They maintain the whole administration of all games, players and the game “Watten” as such.

I structured and programmed the following Java classes, embedded in the package *wattenbeans*:

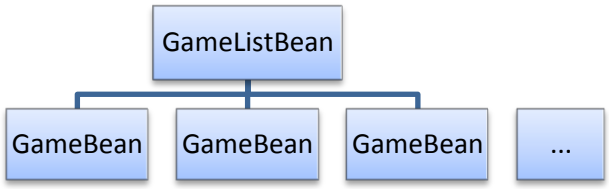
German bean name	*Translation*
KartenBean	CardBean
KartenListeBean	CardListBean
RundenBean	RoundBean
SpielBean	GameBean
SpieleListeBean	GameListBean
SpielerBean	PlayerBean

After I planned how the project should work and what it should look like, I structured the appearance of the Java classes. At first I focused only on the administration of games, not on the game “Watten” as such and then I realised it.

Administration of games and players

List of all games

The basis of the project is the *GameListBean*, which contains a list of all existing games. The list is realised by a member variable, whose type is *Vector*. In this list games are saved by their unique number. Games are implemented by the so called *GameBean*. This bean is the basis of the game concept (more about that bean later).



The *GameListBean* does also have a private member variable, which counts the number of the momentary visitors of the site (visitors who have opened the website at the moment).

The bean contains methods to add new games, to delete games and also to get games from the list. If a user wants to create a new game, he has to insert a valid name. It must be made sure, that this name is given and that it’s unique.

This class is also registered in the deployment descriptor as listener class and implements the HttpSessionListener interface (why this is necessary will be explained later).

Players

I realised a bean called *PlayerBean*. This bean stands for a user or rather a player. A player has the following properties:

- Nickname: under this name the player is logged in for a game
- Player number: 4 players can participate in a game; so each player has a different number (from 1 to 4 - #1 and #2 play in team 1, #3 and #4 play in the team 2)
- Team number: the number in which team the player plays (1, 2 or 0 if he doesn't join a game)
- Session ID: the ID which the player has on the server, required to logout a player after timeout
- Cards: a list of cards, which the player holds in his hands during the game, realised with a member variable of the type *Vector*. The *Vector* contains a list of cards, whose class is *CardBean*.
- Participating: gives the state of the player (if he is logged in for a game or not)

Moreover the *PlayerBean* has a circular reference to the *GameBean* (stands for a game), using a member variable. It links to the game, for which the player is registered. With the help of the *PlayerBean* the *GameBean* can be accessed easily (requesting the name of the game or getting information about the other players, and so on).

The bean also has a reference to the *GameListBean*. This is necessary to realise the following. If a user opens the website, the *PlayerBean* will be assigned automatically to his session object. To initiate that, I implemented in the *GameListBean* the HttpSessionListener interface. Using this interface it is also possible to log a player out from a game, as soon as his session is destroyed. So, when a player closes his browser, he will be logged out after his session-timeout runs out. I set the session-timeout to one minute, which means that the user will be logged out after one minute of inactivity.

GameBean (administration)

The *GameBean* is responsible for the course of the game, but it has also some administration to do. It can be in three different states: inactive, opened and closed. A game is inactive when there aren't players logged in for it. It's opened when fewer than four players and more than zero are logged in. If the game is called closed, then four players are logged in.

- Participation: A user can join a game by entering a valid nickname and by selecting a team, if the games state isn't closed and if he isn't already logged into another game. The bean contains also methods to validate the inserted data of the player and to set error texts.
- Logging out: A user will be logged out from a game. The games state change and the round wins the other team, in which wasn't the logged out user.
- Delete game: A game will be deleted if it has been inactive for twelve hours.

Preliminary results

Having realised this administration parts users can join games, create new ones and log out from them. I didn't have big problems to implement what I described previously and so the administration of games soon worked. At this point I had the following classes:

- GameListBean
- PlayerBean
- GameBean (partly)

In the next part of my work I focused on the programming of the card game "Watten". Now I had to concentrate on the rules and the course of the game.

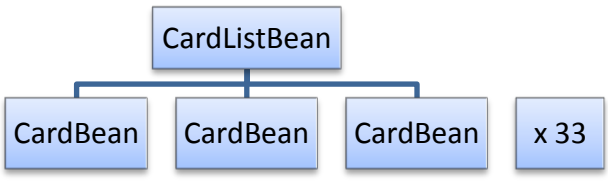
Watten in Java

Cards

Watten is played with 33 cards. I realised two beans, which concentrate on them. First bean is the *CardBean*. This bean represents one card. In principle a card consists of a number, a value and a suit. In addition the *CardBean* has a property, where the number of the player, who holds the card in his hand, is saved. Furthermore each card has a ranking, which depends on what is announced.

The *CardBean* has also a method which gets the path and name of the picture which corresponds to a card. So each card also has a picture. A problem was obtaining the pictures of the 33 cards in digital format. I didn't find them on the internet, so I had to scan in all of them.

The second class which is about cards is the *CardListBean*, comparable with a card package. Because of this, the class is extended by *Vector* and so implements a list of cards. In the constructor of the bean all cards are initialised with their numbers, suits, values and rankings and after that inserted into the list. The class has methods to get cards from the list.



Connection between two beans

The most important method in this class and also in the whole project is *setRanking*. This routine calculates the rank of the cards after announcement. If you give the cards for Schlag and Trump as parameters to this method, it resets the rank of all 33 cards corresponding to these two cards. The highest rank has the good one, the second highest the right one, and so on. It is also possible that some cards have the same rank. To determine who wins a trick isn't now more difficult (more about that later).

Main classes of Watten

The classes which regulate the course of the game “Watten” as such are the *GameBean* and the *RoundBean*. In my opinion these classes are the most complicated part in the project.

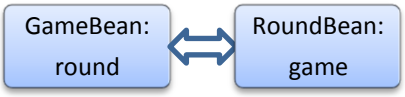
GameBean (playing part)

The *GameBean* is the central class in the whole project and represents one game. A game has the following properties:

- Number: under this number the game can be identified uniquely in the list of all games (*GameListBean*)
- Name: each game has also a unique name
- Last played time: represents the date, when the game was open or closed the last time. Asking this property a game can be deleted after twelve hours inactivity.
- Points: the score for each team is also saved in this class
- Announcers: the player numbers of the actual announcers (a member variable for the forehand and another for the dealer)

In this class the (circular) reference to all four players is also important, implemented by an array. Because of this in this class I realised the method *participate*, using which players can join a game. The method inserts the corresponding *PlayerBean* into the array of four players in the *GameBean* (method explained previously).

The most important reference in this class is that to the *RoundBean*. This bean manages each round of a game and as soon as the round is finished, the bean will be destroyed. After a round is finished a new round will be initiated. To realise that concept, the *GameBean* implements a method called *newRound*, in which is created a new instance of *RoundBean*. By setting member variables in the *GameBean,* it fixes also the forehand and the dealer. To question some properties, it is also necessary that the RoundBean has a reference to the *GameBean* (circular reference).



The *GameBean* is able to access to the *RoundBean* and reverse.

There exists also a method in the *GameBean*, to delete a round. When a round is finished, the score must be incremented (for the right team). This can be done using another method from the *GameBean*, called *incrementScore*.

A method from this class is also *initGame*. It resets some member variables, so that a game can begin. In this method for example are fixed the players, who have to announce first.

In the *GameBean* I implemented three states explained before (inactive, opened, closed). I programmed a method which finds out the actual state of a game by asking the array of players and returns it after that.

RoundBean

This class stands for a round and collaborates very closely with the *GameBean*. In the constructor of the *RoundBean* the circular reference to the game is made and the player who has to throw the first card (forehand) is set. After that, the cards are distributed.

Distribution of cards

First the card package is initialised (*CardListBean*). Then each player receives five cards randomly (by filling the Vector *cards* in the *PlayerBean*).

At first I had great problems with this method, because I forgot something very important. The problem was that sometimes some players received more than five cards. I controlled the program code often, but I couldn't find the error. Then I realised what the problem could be. All methods that access to member variables have to be synchronized. After I had synchronized the corresponding methods (also in other Java classes), all functioned properly.

Round states

I defined several states which a round can assume. The following states are fundamental:

- Announcing: which players announce
- Throwing cards: which player has to throw a card and if he has to add Trump
- Winning: which player wins a trick for his team, which team wins the round or which team wins the game
- Betting: which team bets or if the four are going
- Decline bet: which team has won the round after the decline of the bet
- Asking for better cards: which player asks for better cards
- Decline or accept the demand for better cards: if the query was declined or accepted
- Asking for exchange of Schlag: which player asks for exchange of Schlag
- Decline or accept the exchange of Schlag: if the query was declined or accepted

All these states alter during the round. I programmed a method, which returns a player specified text of his actual state. Reading the state text he knows if he has something to do or if he must wait for inputs of other players.

A little problem in some cases was the following question: How long should a state text be visible? I realised it like this: The state text is visible until all four players have called the method *getState* at least once. Then all players can read the text. The next call of the method returns the new state.

For example:
Player1 (Team1) wins a round. The state text is "Team 1 wins the round". Now this text remains until all four players have called the method getState (which returns this text). If all four have done this, they should have seen the state text. The next call of getState returns: "Player3 and Player4 have to announce".

Announcing

I defined a method, which can be called "to announce". First some things must be checked, so that a player can't manipulate the game. A player only can announce if the games state is set to "*announce"* and when nobody has asked for better cards or exchange of Schlag. Moreover must be checked that he owns the card, which he wants to announce. The method recognizes if the player is the forehand or the dealer. Then it fixes the corresponding card as Schlag or Trump in the *RoundBean*. As soon as both cards are chosen, the method which fixes the rank of all cards (*setRanking*) is called. Then the state changes to "*throwing cards*".

Throwing cards

One method I defined as "*throwCard"*: this happens when a player throws a card. Using this method some anti-manipulating things must be checked. A player can only throw a card if the games state is set to "*throwing cards*" and if it's his turn. Whose turn it is, is saved in a member variable. Naturally he can only throw cards which he owns. This method controls too, if players have to follow a suit and compels them to do so if is the case. If a card can be thrown, it is added to a *Vector* called "*thrown Cards"*. This Vector represents the game field. After that it's the next players turn, and so on. As soon as four cards are thrown, is called the method *getTrick*.

Winning a trick

A round point is scored by the player (for his team), who has thrown the card with the highest rank. The routine *getTrick* searches the owner of the card with the highest rank in the *Vector* of the thrown cards. The state of the round is set to "*winning*". The player who has won a trick is the next one to throw a card (after all players have seen the state text).

The *RoundBean* has two member variables in which are saved the number of the tricks which owns a team. The method *getTrick* increments number of tricks of the corresponding team. If a round is finished, the score of the game will be incremented in this method.

Betting

Betting is realised by two methods. By using the first method, named "*bet*", users can bet. The method controls if they are allowed to and sets then a member variable, which shows if it's already been bet or not. After betting the state is set to "*betting*", so that users must wait until the announcer of the other team processes the query. This he can do using the second method: with the method "*betAnswer*" he can accept or decline the request. The state of the game changes after this method again ("*decline bet*" or "*throwing card*").

With this part of the project I had the greatest difficulties. There are many cases when a player can bet and when he can't. For example you have to control if a team is "gestrichen", if four is being offered, and so on. It's easy to forget one case and then it doesn't work correctly. At long last, after many tests and improvements I managed to make it work properly.

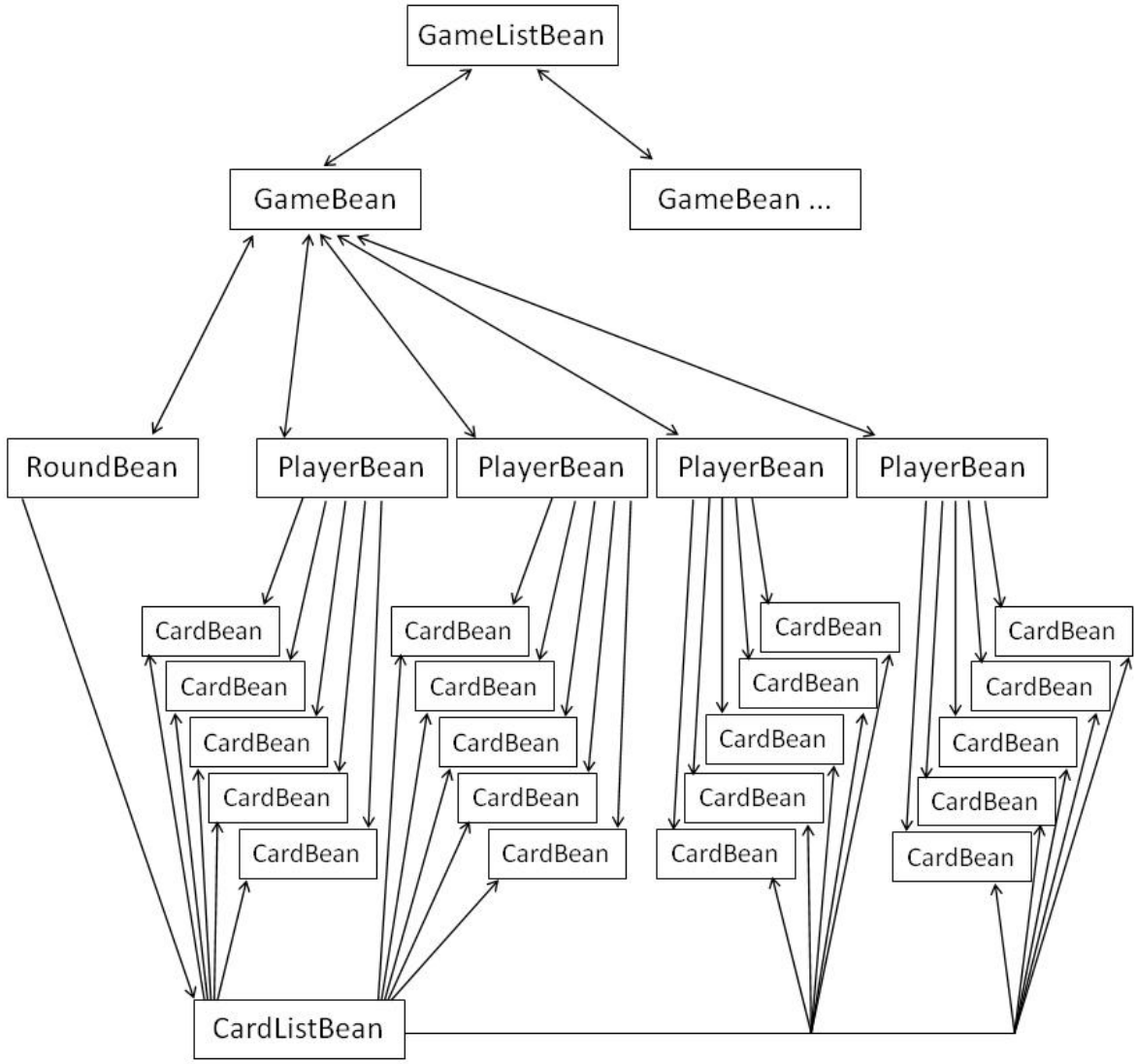
Exchange of Schlag and better cards

These two queries I realised the same way as betting. There is a method each to ask and one to answer the query. The difference is only that asking for better cards or exchange of Schlag can only be done during the announcement, while “betting” can only be done after the announcement.

Also here changes the state corresponding to what the user wants (“asking”, “decline”, “announcing”) and also here you have some member variables which save if it’s asked or not.

After having realised “betting” this wasn’t more difficult, because it has a similar logic.

Interaction of beans



Legend:

→ Simple reference to another bean (member variable)

↔ Circular reference between beans (member variables or a member variable and a list)

JSP sites

The next step was the programming of the JSP sites. During the development I made some test sites, where I tested the functions of the programmed beans, but now I had to put them together and also to enlarge them. Also the design of the web application comes now into play, which I defined using CSS.

I programmed two types of JSP sites. The first kind of sites a user can see if he doesn't join a game. In this case he can see a menu with fewer features. He's only allowed to create new games and participate in a game (like the picture above).

If a player is playing in a game, he can access also the other type of JSP sites (for example the site where it's played). If he is logged in he has more features:

In this case he can create new games or logout from the actual game and join another game or he can access the game site with all its options.

If a user wants to access a site, which he isn't allowed to see, he will be directed to an error page. For example if he wants to access the site, where some people are playing, he will be directed to such a site. This I realised by controlling in every delicate JSP site, if a player is participating to a game or not. The forwarding to the error page I realised like this:

```
<c:if test="${!spieler.bereitsBeteiligt}">
      <jsp:forward page="zugriffsberechtigungsfehler.jsp"/>
</c:if>
```

Using EL/JSTL

On the JSP sites I used the Expression Language and the JSP Standard Tag Library, provided that it was possible. Using them, the code becomes clearer and more understandable.

Structure of the JSP sites

I used tables for the most part of the page to realise the design. I needed many tables so I created a file called header and another file called foot. These files I built to put the top of the page in one and the bottom of the page in another file. Every page includes these files. So the code of the main page is clearer and more comprehensible. For example the index.jsp looks in great mode like this (picture is on the previous page):

```
<html>
<head>...<title>Blind Watten</title>...</head>
<body>
      <%@include file="header_os.jsp" %>        <%--Headerfile--%>
      ...                                       <%--Main Page --%>
      <img src="img/welcome.jpg"/>
      ...
      <%@include file="fuss_os.jsp" %>          <%--Footfile--%>
</body>
</html>
```

Because I needed two different header files (some pages are structured lightly different), both include a file called menu. In this file I put the logic of the menu (when which link should be available for the user and when not), to reduce the redundancy.

Used Beans

On the JSP sites I used only the *PlayerBean*. If a user opens a website of this project, the *PlayerBean* is assigned automatically as an attribute to his session object (how I explained previously). On the JSP site I can work with this bean using the following command:

```
<jsp:useBean id="spieler" class="wattenbeans.SpielerBean" scope="session"/>
```

On all sites the *PlayerBean* (*SpielerBean*) is used as interface to the Java beans, because this class has references to all other classes used.

State text

On each webpage I implemented a state text. This text I set using a variable (page scope):

```
<c:set var="fusszeilentext">There exists 1 game</c:set>
```

Examples of state texts:

There exists 1 game

Please wait until 4 players join the game

Stefan: You announce Schlag – Marlene announces trump

Frames

The whole project works only, if the players can always see the actual state of the game. Realising this with JSP can only be done, if the page is refreshed at periodical intervals. During these intervals the browser requests the new state of the game on the server and this sends it back to the browser, where it's shown. But it isn't useful to always refresh the whole webpage, so I used embed frames, so called iframes, to solve this problem. Iframes are frames which you can adapt better to your situation. The frame refers to the site which is refreshed periodically, or rather to the site where it's played.

In the *GameListBean* I programmed a method which calculates the number of seconds, after which the site is refreshed. This depends on the number of users which are altogether playing on the server.

At the head of the page which should be refreshed I inserted these few lines:

```
<meta http-equiv="refresh" content="${spieler.spieleListe.aktRate}">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta http-equiv="pragmas" content="no-cache">
```

The frame I created using these commands:

```
<iframe width="978" height="452" frameborder="0" src="waitframe.jsp">
      <p>Your Browser isn't able to show embed frames</p>
</iframe>
```

Logging in and out

The pages to create new games and to participate in games aren't very complicated. Their structure consists in great part of forms, where users can do their inputs (name of game, nickname, select team).

More difficult was the programming of the pages where users come to, after they have logged in. After a user is logged in for a game, he will be directed to a wait page, where he has to wait until four players join the same game. This wait page I realised with an iframe, which is refreshed periodically. On this page I put, among other things, these three lines:

```
<c:if test="${spieler.spiel.anzahlSpieler == 4}">
      <jsp:forward page="spielen.jsp"/>
</c:if>
```

The webpage will be refreshed continuously, because of the frame. As soon as the game consists of four players, they will be redirected to the page, where they can play.

Also the game page is build with a frame, but there I inserted these lines:

```
<c:if test="${spieler.spiel.anzahlSpieler != 4}">
      <jsp:forward page="waitframe.jsp"/>
</c:if>
```

As soon as one player leaves the game, the other three players are lead back to the wait page.

Announcing and throwing cards

Announcing and throwing a card can be done by clicking on the wished card. Each card is linked to a page called *cardklick.jsp* with and ID (parameter in the URL). There is controlled, what's the games state and then is called the correspond method in the *RoundBean*, accessed over the *PlayerBean*. After that is redirected back to the game page.

Betting

If a user wants to bet he can press a button on the webpage. By pressing this button a webpage named *bet.jsp* is called:

If it hasn't been bet yet, a player can try to bet:

```
<c:if test="${spieler.spiel.runde.bieten == 0}">
      <%spieler.getSpiel().getRunde().bieten(spieler.getSpielerNummer());%>
      <c:redirect url="spielen.jsp"/>
</c:if>
```

After a bet the game page causes, that the player who has to accept or to decline the bet is forwarded to the page *bet.jsp*. There is controlled that no player can bet or answer the bet if he isn't allowed. Then the page causes an output, where the user can reply. His answer is processed again by the same page (*bet.jsp*) and it is called the method *betAnswer* with the corresponding parameter:

```
<c:if test="${not empty param.ja}">
      <%-- Accept --%>
      <%spieler.getSpiel().getRunde().bietenAntwort(true); %>
      <c:redirect url="spielen.jsp"/>
</c:if>
<c:if test="${not empty param.nein}">
      <%-- Decline --%>
      <%spieler.getSpiel().getRunde().bietenAntwort(false); %>
      <c:redirect url="spielen.jsp"/>
</c:if>
```

Asking for better cards or exchange of Schlag works like betting.

Game page playing.jsp

The game page is the central page of all JSPs. It represents the game field, takes over the output of the game cards and contains also the sidebar with all its options. The page causes the creation of new rounds and the reinitialising of the game when it's finished. It forwards the player, who has to answer a query (bet, better cards, exchange of Schlag), to the corresponded JSP site.

For example: forwarding after the demand for exchange of Schlag:

```
<c:if test="${spieler.spiel.runde.schlagtausch != 0 &&
      spieler.spiel.runde.schlagtausch != spieler.spielerNummer &&
      (spieler.spiel.schlagAnsagerNr == spieler.spielerNummer ||
      spieler.spiel.trumpfAnsagerNr == spieler.spielerNummer)}">
      <jsp:forward page="schlagtausch.jsp"/>
</c:if>
```

Closing remarks

This project was a big enrichment for me. It was the first greater web project I realised and I learned much by structuring and programming the project.

It was also very instructive for me that I had to write this paper in English as it gave me a chance to improve my language stile.

Sources

- History of the game
  http://de.wikipedia.org
- Rules of the game
  http://www.pagat.com/trumps/watten.html