Software Proposal - “Watten“-Application

Alber Patrick, Moser Peter, Ranigler Matthäus

Problem:

As you probably know, most people in South Tyrol love to play the card game “Watten”.

“Watten” is a typical game for our area where normally 4 people form two teams which consist of two players. Each player is dealt 5 cards by the dealer, which is selected before the game by any method desired, but changes after every round. The dealer and the player who received the cards first, the player after the dealer, decide which cards are the best during each round. Then the player after dealer decides which number is the best and the dealer decides the suit. This decision can be done by saying their values loud or in secret by showing a card to each other at the same time. The outcome of this decision is, pretty funny, only the second best card in the game. The card above this one, is the best. After that the, best card is as mentioned before, the card they decided on, then all of the other cards with the same value, then the cards of the same suit and last but not least all of the other cards from other suits, highest to lowest. Cards of other suits can only beat cards of their own suit, only the decided suit can beat the others. By saying their decision everybody knows what is going on and the whole game is pretty easy compared to when the decision is made secretly, people have to think more mand pay more attention. During the game no one is really allowed to speak, you cant ask your team mate what he has. Doing so will end in a penalty, meaning that points are subtracted. The player after the dealer starts first by throwing a card of his own decision. Then the player together with their team mates have to try and beat the opposing team by throwing cards into the middle, which have a value higher then the opponents cards or low valued cards if their team mate has the currently highest card in the middle. Each round consists of up to 5 possibilities for a team to win a type of ¨battle¨. A round ends if a team wins 3 battles or the opposing team surrenders after a bidding. After each round the winning team is rewarded with at least 2 points, if there has been any bidding the amount of points can increase. A game ends if a team reaches or passes 18 points.

These are the basic rules of “Watten”, there are some more detailed rules that are missing and in addition every town has its own set of rules, for example the direction of play can be different from town to town.So why are we going to develop such an application, there exists a “Watten”-application on the play-store, but the type of “Watten” played is not exactly the same as we play in South Tyrol. Plus as I said before, every town has its own rules which would have to be adjustable. Another aspect of the game on the play-store that we didn't like, is the fact that it is 3D and it costs money, there is no free version.

Issues & Challenges:

- Coordination of Users and Rooms/Tables
- Minimal Communication for fast response
- Messages must arrive, a move can’t be lost
- Error handling
- Graphical Representation
- Handling of multiple request, concurrency issues

Proposed treatment:

1) Android-Application
2) Web-Service/Game-Server

Someone might be bored while he is waiting for someone/something or is in the train and wants to play a game of “Watten” but there is nobody around, he then remembers that he has some awesome app on his phone and searches the internet for some else who is bored. We will use “libGdx” as a graphical framework, since it supports Windows Desktop, Android and iOS.

Business value:

There exists a similar app on the play-store, as mentioned before, but that application is not meant for South Tyrol. So we basically have a unique application that lots of people in South Tyrol will love to use.

Providing a limited free version and a full version that costs money.Plan of work:
As we have used the agile approach before we would use it in this case too. It is best for adding new features and keeping a running prototype available all of the time.
Our first steps will include the creation of a skeleton for our card game. Trying to include all of the logic needed to play “Watten”. Next we would have to decide what goes where, selecting the parts that stay on the server/service and the parts that are done locally by the phone.

The third step should be a small graphical prototype so we can run the game on our phones.

This plan is very basic and will change pretty fast, since there are always things you don't take into account.

Actual treatment:

We decided to divide work into three main parts, such as UI, network and game logic. Since Watten is a turn based game Peter decides to implement a state machine on an iterative basic pattern, which works as follows:

Game Logic - State machine

This state machine could then easily be tested automatically, since it has not to deal with manually triggered user inputs, such as mouse clicks and keyboard short cuts. Peter implemented for this state machine three different levels of iterations, where every iteration contains several states, respectively features and constraints. Constraints mark possible transitions to other states. All called states, that are not reachable by constraint transitions from the current state produce an exception.

- Game iterator
  - state game_entry
  - state game_exit
- Round iterator
  - state round_entry
  - state round_exit
- Turn iterator (smallest)
  - state turn_entry
  - state turn_exit
  - state turn_bet
  - state turn_play_card
  - state turn_surrender_or_hold
  - state turn_select_rank
  - state turn_select_suit

Every state has been implemented within a single method, where the current state gets set if it can be reached over transitions from the current state, otherwise an error get thrown. Normally errors are considered restorable, that means the state machine remains within the same state, waiting for the next input. Handling with errors is the duty of clients.
The decision tree is too big to be presented here, please refer to the code to see which constraints follow which features. Some constraints are conditional, allowing a transition only if certain points or stiches are reached or a player decided to bet.

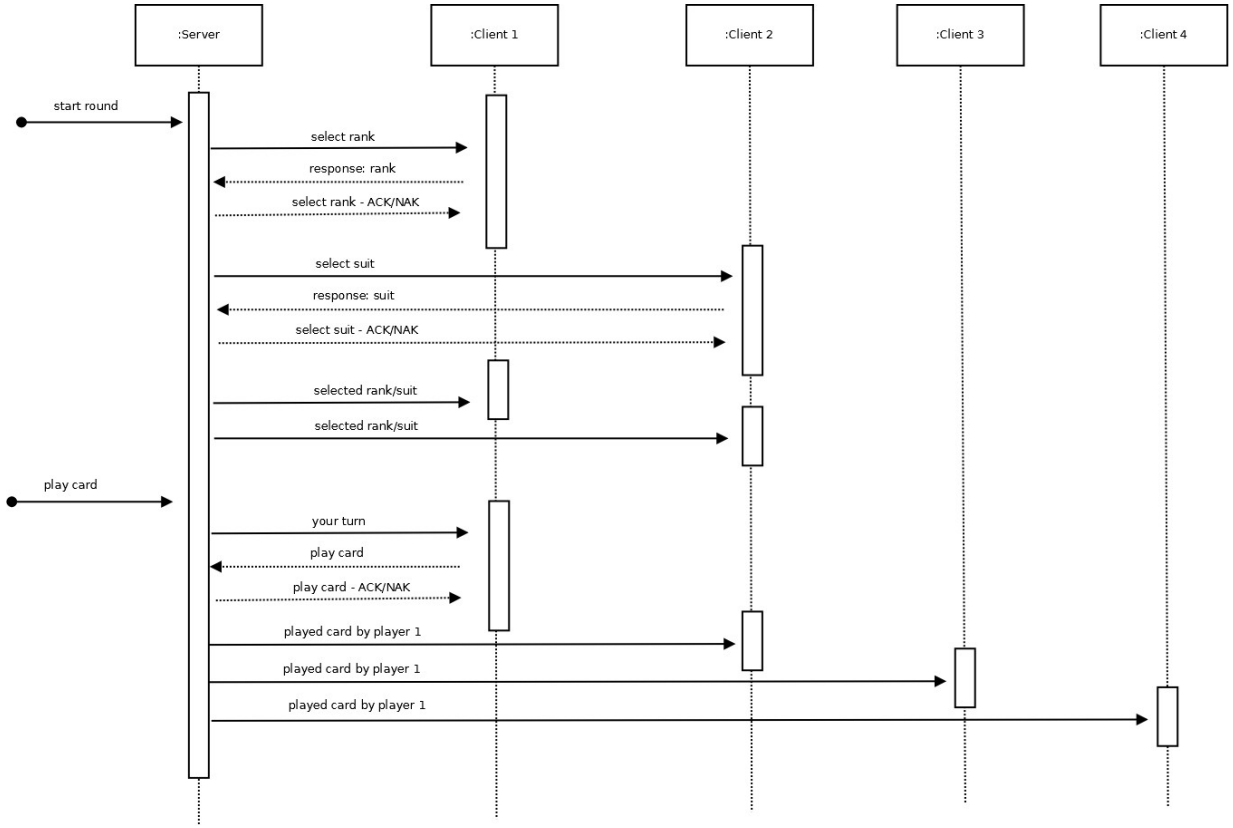
XML Network protocol

Our homegrown XML network protocol includes a simple xml parser, that allows some rudimentary xml syntax parsing and lexical analysis. It is called “simple”, because it does not allow namespaces, nor tag-attributes. However, for our purposes it was powerful enough to serialize objects and send them then as utf8 encoded strings over network TCP/IP to other clients, or respectively server. We decided to use TCP, not UDP because of its capability of reliable data transfers. XML has been chosen, because its plain text, hence easy to debug.

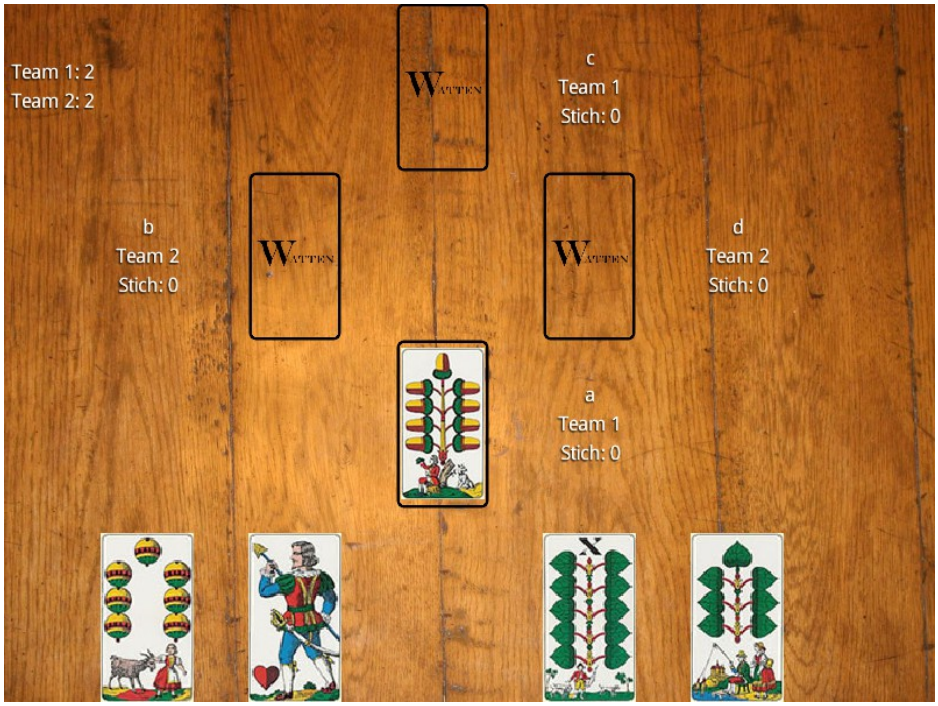
Other advantages are its nested structure where we could easily add more serialized objects later on, without changing previously implemented protocol phases. For instance, we sent player objects as XML strings and decided later to add also player's hand cards. Which has been simply added within a <cards>...</cards> tag.

Client - Server - Architecture

Patrick and Peter implemented the server, which operates as a service, providing a state machine for each game and some lobby functionality to assign players to certain games or create new games. The server uses Java sockets over the TCP/IP protocol. Currently the IP address has to be specified manually before starting the client. The different clients, for example text based clients and graphical clients sent request to the server which get responded with ACK (acknowledged) or NAK (not acknowledged) and if succeeded additional information as a XML string. This strings get then interpreted by the XML parser at client side. Please refer to the sequence diagram for an examplary data exchange. NB: This sequence diagram is an example of our network protocol. Since Watten as we play it normally always needs four players, four clients are represented here.



Graphical User Interface



Since Matthäus knew from experience, that even if the UI is often left till the end, games are something different. Your games visuals have to be constantly updated and are not as static as most non-game software is. Knowing this, a small UI with menu was created just to have a basis to work on, when Matthäus got back from the US.

After we got a simple console based prototype running and Matthäus got back from his trip, it was time to get the UI-development back on track. The main idea behind the UI is something similar to Excel, you basically divide the whole screen into a grid of cells of different sizes, by knowing how many columns and rows you need and what they will be used for. The first prototype was an attempt to create the UI by placing all of the items on screen using coordinates. This did not turn out as expected, since Matthäus hadn't done much with libGdx lately he had forgotten about the above mentioned TableLayout, but after spending way too much time with the first UI he remembered how to simplify the whole process.

The UI consists of 5 rows and seven columns(depending on the resolution, there can also be more columns as padding). Most of the visuals are achieved by simply adding and removing elements from/to the tables cells. The local player is the only one that sees a hand, the remote players are seen as a playing area with the players information next to it. This minimizes the amount of data sent.

Although the game was playable from the console, modifications to the server were necessary, since the game on the server-side changed states too fast. Additional methods were introduced, such that the state changes were accomplished by the clients requests and answers and the clients server-side thread evaluating the games status after a request. Otherwise it would have been impossible to inform the client of what was happening and thus updating the UI.

A game is started by entering a desired username, if someone is already on the server with that name a MessageDialog informs the user. If the name is accepted the user is presented with the main menu. From here he is able to create a new game or join an existing game. In both cases he must enter the games name. Also here a MessageDialog informs the user if a game he wishes to join does not exist or a game he wants to create already exists. After creating or joining a game the view will switch to the GameScreen showing the table and possibly the other players. If there are 4 players in a game, the player which joined last, is presented with a button to start the game. This button was introduced since the previous way of starting the game had some issues which will be discussed below. During every turn, each player is informed by a MessageDialog if it is his turn, if he has to select the rank or the suit, what the other person selected and the next best card(only to selecting players), who had the best card in a turn, which team won the round and which team won the game. Additionally any error sent by the server is also displayed with a MessageDialog. This concludes the information about the UI, since the rest of the gameplay follows the description at the beginning of this report.

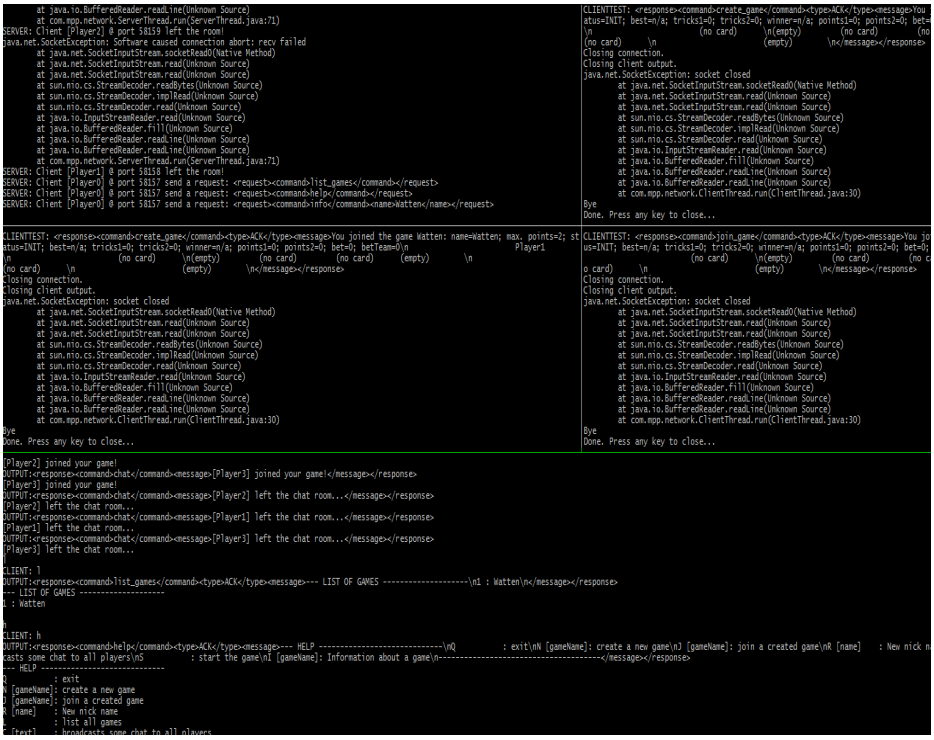
Problems and Solutions:

- States too fast
    - As mentioned above, initially the games states were continuing too fast and the client did not know what was happening.
    - *Solution*: The clients server-side thread changed the states according to requests, answers and an evaluation of the games state after a request. The evaluation being essential for informing the client of the state after a request.

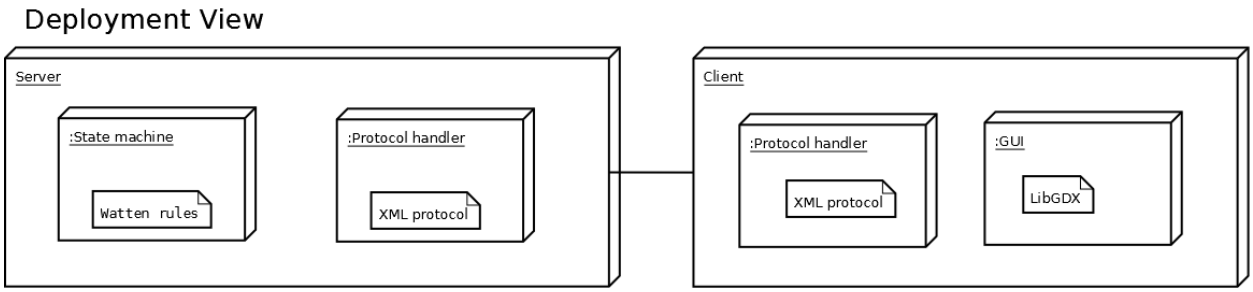
- Switch failure
    - Somehow our ClientSender had a bug, where the switch clause listening for a new request to send to the server sometimes just didn't trigger for "game_start". The server sent "game_ready" to the client that joined as 4th player and that client would have responded with "start_game". The ClientSender registered the command but the switch clause didn't trigger.
    - Solution: The client that got the "game_ready" response has a button appear on the table, which says "Start game". When he clicks the button, the "start_game" request is passed to the ClientSender and the button remains on-screen until the server returns "start_game ACK", acknowledging the games start.

- Testing
    - Testing has mainly be done with Junit test cases

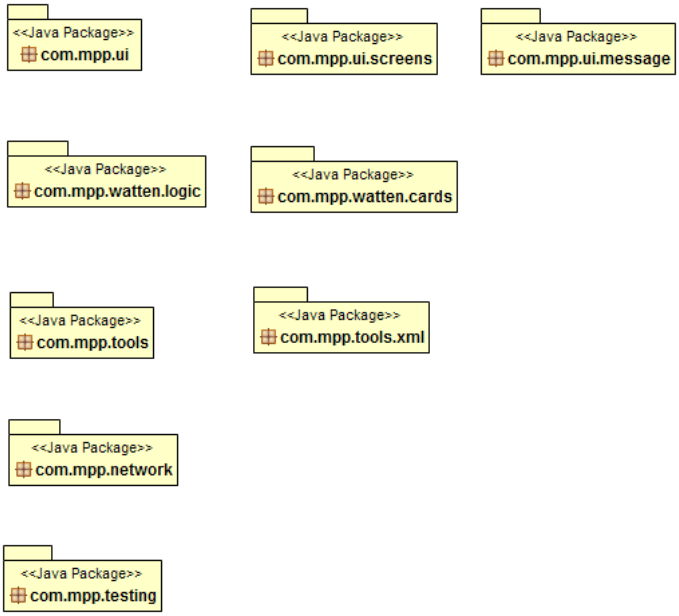
- Complex test cases have been automated with Bash scripts and tmux (a terminal multiplexer)



Additional models



Package View



UML – Class Diagram

The most important classes and dependencies.