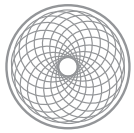


Auction System 5AHITt

Huang, Reichmann, Schuschnig, Valka

Inhaltsverzeichnis

Auction System

Angabe

Overview

You have to implement a simple auction system where multiple users bid in auctions similarly to eBay. More formally, the system will use a variant of the English auction type with only difference that an auction ends at specific time and date that is determined by the seller when the auction is created. In an English auction, all bids are open and the current highest bid is visible to all participants. Moreover, the current selling price is determined by the highest bid (and not the second highest bid like at eBay). An example of an auction is shown in Figure 1:

1. First, Alice creates an auction for her small notebook computer.
2. Then, Bob bids and sets the first selling price with a 100€ bid.
3. Later, Carl overbids Bob with a 200€ bid and sets a new selling price.
4. At last, Dave overbids Carl and sets the final selling price with a 250€ bid.

Figure 1

Design

The auction system consists of one server and multiple clients connected to it. Bidders and sellers use clients to create, list, and bid on auctions. The server coordinates execution of all commands that were received from the clients (i.e., clients are not allowed to communicate directly between each other). Because commands require proof and confirmation from the server that they have been received and executed, they must be sent reliably using TCP protocol. Moreover, the server will asynchronously notify clients with some notifications (discussed in detail further below) that are of interest to the currently logged-in user. These notifications are not core to the functionality of the server and are allowed to get lost. Thus, they will be sent unreliably from the server to client using UDP protocol.

In short, the task of the client is to allow its user to issue commands to the auction server and to display notifications from the server as they arrive. Please note that users are not anyhow uniquely connected to a client and can use different clients at different times; i.e. they can use one client and then, in some later time, after logging-out from the first client use another client by logging-in with same credentials. Moreover, the server should reject any log-in command from user that is already logged-in on a different

client. Figure 2 shows an example network overview with Alice, Bob, and an anonymous user connected to the server using three different clients.

Figure 2

After starting a client, an anonymous user (i.e., user that is not logged in) is allowed to list all currently open auctions (i.e., auction that are still not finished) with each auction displayed in a separate row. For each auction in the list, the client needs to show its identification number (abbreviated as "id" in the rest of this document), description, name of the owner, date and time when auction will end, the current highest bid amount (or 0 if no bid has been placed yet) and the name of the current highest bidder (or "none" if no bid has been placed yet). An example of listing is shown in Figure 3.

Figure 3

For bidding and creating auctions, the user has to first log into the server using a client by specifying its name. After that, the user is allowed to create an auction by specifying the duration of the auction in seconds and its description of the auction. The task of the server is to assign the new auction with a unique id number, which will be used throughout the bidding process. An example of logging in and creation of an auction is show in Figure 4.

Figure 4

Additionally, the logged in user can bid by specifying the auction's id number and a bidding amount. If the bid is higher than the previous highest bid, the bidder becomes the new highest bidder and the amount is set as the new selling price for this auction. Moreover, the previous highest bidder gets notified that he/she has been overbid. An example of the bidding process is show in Figure 5 (note that the output is displayed in a single block, while in fact it represents different terminal windows for "dave" and "carl").

Figure 5

When an auction ends, the server is responsible for notifying the owner of the auction and its highest bidder that the auction has ended. A notification example is show in Figure 6.

Figure 6

Furthermore, the user is allowed to log out from its account and then the client can be reused to log in as another user. After logging in, the server is responsible with notifying the logged in user with all pending

notifications (i.e., notifications that were created for the user by the server while user was not logged in anywhere). An example of this process is shown in Figure 7.

Figure 7

Please note that these figures are for illustration purposes only. Your output should resemble the illustrated output as much as possible (i.e., some small formatting differences are allowed, but all mentioned information in the output **must** be present except for the notifications as they are not reliably sent by the server and can get lost).

Server

The auction server application should expect the following arguments:

- `tcpPort`: TCP connection port on which the auction server will receive incoming messages (commands from) clients.

If any argument is invalid or missing print a usage message and exit.

Implementation Details

As it was explained earlier the main task of the auction server is to manage auctions created by the clients. The first thing the auction server does on startup is to open a `java.net.ServerSocket` (initialized with `tcpPort`) in order to be able to receive clients' requests. Since a`java.net.Socket`, which is returned by `ServerSocket.accept()`, provides blocking I/O-operations (via `getInputStream()` and`getOutputStream()`) and we want to serve multiple clients simultaneously each incoming request shall be handled in a separate thread.

Study the java sockets and datagrams tutorial to get familiar with these constructs. We recommend using *thread pools* (implementations of`java.util.concurrent.ExecutorService`) for implementing the described behavior. They help to minimize the overhead of creating a thread every time a request is received by reusing already existing thread instances. Java provides some sophisticated implementations that can be easily instantiated by using the static factory methods of `java.util.concurrent.Executors`. Anyway you may also manually instantiate new threads on your own without using these classes. In any case, note that threads consume memory (and CPU power), so you should take care that the threads terminate properly. Help can be found in the Java Concurrency Tutorial.

After you have performed the steps described above your auction server will be ready to serve clients' requests. Now it is time to add some business logic. The main two responsibilities of auction server are: user and auction management. User management is reduced to a minimum and you don't need to consi-

der user authentication (see `!login` command), as it only requires a single argument, i.e., a username (the authentication part, together with other security measures will be implemented in later assignments). The only "tricky" part here is how to link the notifications with users and not the physical nodes (i.e. terminals).

Considering the notifications, you will need to implement two different types: `!new-bid` and `!auction-ended` (see the detailed description of the notifications below). In order to implement the notifications you will need to use `java.net.DatagramSocket`. DatagramSocket is used to send/receive DatagramPackets. To initialize the destination of a DatagramPacket you will need client's address and an UDP port. The former can be obtained from the established connection with a client by using `socket.getInetAddress()` and the later must be provided by the client (see client description below).

In the auctions management part you will need to address the problems how to create an auction, how to list available auctions, how to place a bid and finally how too terminate an auction after it has expired. The first three parts should not be too complicated, however you should be careful, because there will be multiple clients (threads) sharing the same resources (e.g. a list of running auctions). In these situations you need to pay a special attention to the *synchronization* i.e. you need to make sure your code is *thread-safe*. Study the Java Concurrency Tutorial if you are not familiar with threading and/or synchronization.

Considering the problem of expired auction, you will need to prevent further bidding on them, but you will also need to remove them from the list of available auctions (i.e. they don't appear in the response to `!list` command). You can for example use a thread or a`java.util.Timer` in combination with a `java.util.TimerTask` to implement this kind of *garbage collector*. The choice how you want to implement it is completely up to you. However, please note that for the testing purposes the "check interval" should not be longer than 1s.

Important: During the implementation you will need to use different buffered streams (e.g. `java.io.BufferedReader`). In order not to waste the memory careful housekeeping is needed. Therefore, always close buffered streams and sockets after you have finished using them. Additionally, to improve readability and maintainability of the software take a good care of exception handling.

The auction server supports only one interactive command. That is in order to shut down the server a user needs to simply hit the enter key. Do not forget to logout each logged in user. Note that as long as there is any *non-daemon thread* alive, the application won't shut down, so you need to stop them. Therefore call `ServerSocket.close()`, which will throw a `java.net.SocketException` in the thread blocked in`ServerSocket.accept()`. All other threads currently alive should simply run out. If you are using an `ExecutorService` you have to call its shutdown() method and in case of a `Timer`, call `Timer.cancel()`. Anyway you may not call `System.exit()`, instead **free all acquired resources** orderly.

UDP notifications:

- `!new-bid <description>`

This notification is sent by the auction server to the previously highest bidder, after a new user has placed a higher bid on an auction with description `<description>`.
E.g.:

```
1    !new-bid Super small notebook
```

on carl's client:

```
1    carl> You have been overbid on 'Super small notebook'
```

- `!auction-ended <winner> <amount> <description>`

This notification is sent by the auction server to an auction owner and the auction winner after the auction has ended.
E.g.:

```
1    !auction-ended dave 250.00 Super small notebook
```

on alice's client:

```
1    alice> The auction 'Super small notebook' has ended. dave won with 250.00.
```

on dave's client:

```
1    dave> The auction 'Super small notebook' has ended. You won with 250.00!
```

Client

The client application should expect the following arguments:

- `host`: host name or IP of the auction server
- `tcpPort`: TCP connection port on which the auction server is listening for incoming connections
- `udpPort`: this port will be used for instantiating a `java.net.DatagramSocket` (handling UDP notifications from the auction server).

If any argument is invalid or missing print a usage message and exit.

Implementation Details

The main task of your client is to read user requests from standard input (`System.in`) and to communicate them to the auction server accordingly. The other important task of the client is to receive and display different notifications, sent by the auction server, to users.

One of the first things to do here is to create a `java.net.Socket` and connect to the auction server. You will need the `host` and `tcpPort`values for this. Outgoing messages are sent each time the user enters one of the interactive commands. The responses from the auction server should be handled in an own thread. Keep the connection open as long as either the client or the auction server shut down.

Another thing you will need to do in order to be able to receive the notifications from the auction server is to create a`java.net.DatagramSocket`. As the auction server needs to know the full address where to send the notifications. Therefore, your client will need to tell it the `udpPort` of the `java.net.DatagramSocket`. The best way to do this is simply to send `udpPort` as a part of `!login`message. Additionally, please note that the `DatagramSocket.receive(packet)` is a blocking operation. Therefore, using a separate thread to receive notifications should be considered.

The list of interactive commands supported by the client are listed below:

- `!login <username>`

Logs in the user <username> to the auction system.
E.g.:

```
1    > !login alice
2    Successfully logged in as alice!
```

- `!logout`

This command logs out the currently logged in user.
E.g.:

```
1    alice>: !logout
2    Successfully logged out as alice!
3    > !logout
4    You have to log in first!
```

- `> !list`

Lists all the currently active auctions in the system. The output formatting is not important, but the auction id, the auction's description, owner's username, the current highest bid amount and the current highest bidder's username must be displayed. If an auction does not currently have any bids, the listing of

that auction should output '0.00' and 'none' for the highest bid and the highest bidder's username respectively.
E.g.:

```
> !list
1. 'Apple I' wozniak 10.10.2012 21:00 CET 10000.00 gates
2. 'My left shoe' hugo 07.10.2012 12:00 CET 0.00 none
3. 'Super small notebook' alice 04.10.2012 18:00 CET 250.00 dave
```

- `> !create <duration> <description>`

With this command a user can create new auction. A short description of the auction and its duration (given in seconds) must be specified. Auction id is automatically assigned by the auction system.
E.g.:

```
> !create 25200 Super small notebook
An auction 'Super small notebook' with id 3 has been created and will end on
04.10.2012 18:00 CET.
```

- `> !bid <auction-id> <amount>`

With this command users can bid <amount> on a specific auction <auction-id>.
E.g.:

```
> !bid 3 250.00
You successfully bid with 250.00 on 'Super small notebook'.
> !bid 3 220.00
You unsuccessfully bid with 220.00 on 'Super small notebook'. Current highest bid
is 250.00.
```

- `!end`

Shuts down the client. Note, same as for the server, as long as there is any *non-daemon thread* alive, the application won't shut down, so you need to stop them. You may not call `System.exit()`, instead again **free all acquired resources** orderly.

Ant template

Ant is a Java-based build tool that significantly eases the development process. If you have not installed `ant` yet, download it and follow theinstructions.

We provide a template build file (build.xml) in which you only have to adjust some properties. Put your source into the subdirectory "`src`". To compile your code, simply type "`ant`" in the directory where the build file is located. Enter "`ant run-server`" to start a server and "`ant run-clientX`" (with X being 1 or 6) to start a respective client.

Note that it's **absolutely required** that we are able to start your programs with these predefined commands!

Also note that build files created by IDE's like Netbeans very often aren't portable, so please use the provided template.

Hints & Tricky parts

Binding problems

After starting your socket implementation, you receive a message comparable to Error 1:

Error 1

The address and port, you are trying to bind to is already in use. To solve these issues you can test your solution and don't bind to ports used by other services, or you simply change the ports. If you are still seeing this problem, it means that there is still a zombie of a previous test run of your application online. Use `ps` to find the zombie and `kill -9` it.

Further Reading Suggestions

- **APIs:**
  - IO: IO Package API
  - Concurrency: Thread API, Runnable API, ExecutorService API, Executors API
  - Java TCP Sockets: ServerSocket API, Socket API
  - Java Datagrams: DatagramSocket API, DatagramPacket API
- **Tutorials:**
  - JavaInsel Sockets Tutorial - Section 11.6, 11.7 : German tutorial for using TCP sockets.

- Java Programmierhandbuch und Referenz - Section 13.2.3 : German tutorial for using datagram sockets.

Submission Guide

Submission

- Every group **must have** its own design/solution! Meta-group solutions will end in massive loss of points!
- As for group work usual, a protocol with the UML-Design, the work-sharing, the timetable and test documentation is mandatory!
- Upload your solution as a **ZIP** file. Please submit only the sources of your solution and the build.xml file (not the compiled class files and only approved third-party libraries).
- Your submission must compile and run! Use and complete the provided ant template.
- Before the submission deadline, you can upload your solution as often as you like. Note that any existing submission will be **replaced** by uploading a new one.

Interviews

- During the implementation there will be review interviews with the teams. Please be aware that the continous implementation will be overseen and evaluated!
- After the submission deadline, there will be a mandatory interview.
- The interview will take place in the lesson. During the interview, every group member will be asked about the solution that everyone has uploaded (i.e., **changes after the deadline will not be taken into account! There will be only extrapoints for nice and stable solutions!**). In the interview you need to explain the code, design and architecture in detail.

Points

Following listing shows you, how many points you can achieve:

- Design(8): usecase, uml-class, activity diagrams
- Documentation(10): timetable, explanation, description(JavaDoc), protocol/test documentation
- Implementation(20): exception handling, concurrency, resource handling, functional requirements
- Testing(12): unit-testing, coverage, system-testing, user-acceptance

Aufgabenteilung

Tätigkeit	Huang	Reichmann	Schuschnig	Valka
Zeitaufzeichnung und Zeitschätzung	x	x	x	x
UML Klassendiagramm		x		
UML Aktivitätsdiagramm			x	
UML Use Case Diagramm				x
Testfälle	x			
Designkonzept Server			x	
Designkonzept Client				x
Designkonzept Verbindungen		x		
TDD Testfälle	x			
Implementierung Client				x
Implementierung Server			x	
Implementierung Verbindungen		x		
Usability Test	x			
Anpassung Abastimmung	x	x	x	x

Zeitschätzung Tatsächliche Zeit

Kuan Lun Huang		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit
Testfälle	1	
Zeitaufzeichnung	0,5	0
Recherche Testfälle	2	
TDD Testfälle	2	
Usability Test	5	
Anpassungen und Abstimmungen	2	
Summe	12,5	0

Daniel Reichmann		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit
UML Klassendiagramm	1	1
Zeitaufzeichnung	0,5	0,3
Designkonzepte Verbindung	1,5	2
Entwurf Verbindung	2	2
Implementierung Verbindung	5	5
Anpassungen und Abstimmungen	2	3
Summe	12	13,3

Tobias Schuschnig		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit
UML Aktivitätsdiagramm	1	0,8
Zeitaufzeichnung	0,5	0,3
Designkonzept Server	1	1,5
Entwurf Server	2	2,5
Implementierung Server	5	6
Anpassungen und Abstimmungen	2	2,2
Summe	11,5	13,3

Dominik Valka		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit
UML Use Case Diagramm	1	0,9
Zeitaufzeichnung	0,5	0,3
Designkonzept Client	0,5	1
Entwurf Client	3	2
Implementierung Client	4	5
Anpassungen und Abstimmungen	2	4
Summe	11	13,2

Gesamt		
Team Mitglied	Geschätzte Zeit	Tatsächliche Zeit
Huang	12,5	0
Valka	11	13,2
Reichmann	12	13,3
Schuschnig	11,5	13,3
Summe	47	39,8

Designkonzept allgemein

Es soll ein Auktions-System mit Server und Clients erstellt werden. Der Server verwaltet die Auktionen und die Clients bieten auf die Transaktionen.

Folgende Aufgaben haben die entsprechenden Teile:

Client

- Login am Server mittels !login <name>, zu realisieren mittels einer login(name) Methode, wobei die Syntax am Client überprüft wird und nur bei Korrekter gesendet wird. Variable loggedIn -> true
- Logout mittels !logout-> Lokales Prüfen ob eingeloggt, wenn ja senden des Befehls an Server, Warten auf Antwort, **Beenden** des Threads der auf UDP-Pakete wartet, **Beenden** Thread der die TCP-Connection zum Server instand hält.
- !list -> Auflisten aller Auktionen, läuft über die TCP-Verbindung, ohne anmelden möglich
- !bid <id> <amount> -> Bietet eine gewisse Anzahl auf eine Auktion; Bei Erfolg bekommt der Client eine gewisse Rückmeldung, bei Misserfolg ebenso

Netzwerk:

Der Client muss folgende Threads und Sockets haben, um mit dem Server kommunizieren zu können:

- Start-Thread, dieser baut eine TCP-Verbindung mit dem Server auf. Er behandelt auch die Eingaben des Users und sendet sie an den Server weiter. Während der Kommunikation kann der User nichts eingeben, da geblockt wird, dies ist ein gewünschter (!) Effekt
- Thread der auf UPD-Pakete des Servers wartet. Wird eins empfangen, werden die Daten weitergeleitet. **Problem:** Ausgabe könnte von anderem Thread besetzt sein

Beim Starten des Clients müssen folgende Argumente angegeben werden:

- `host`: host name or IP of the auction server
- `tcpPort`: TCP connection port on which the auction server is listening for incoming connections
- `udpPort`: this port will be used for instantiating a `java.net.DatagramSocket` (handling UDP notifications from the auction server).

Server

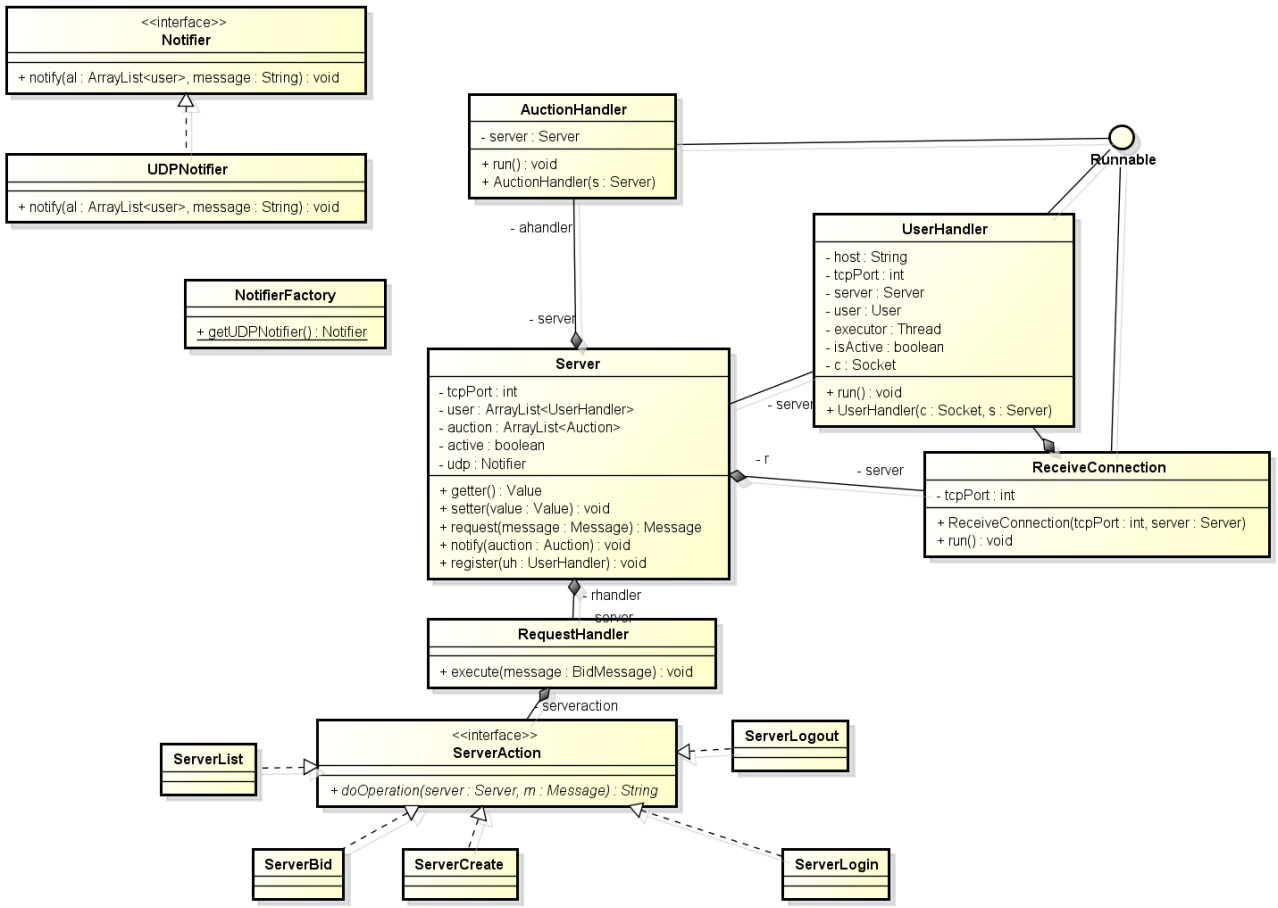
Der Server empfängt hauptsächlich Pakete von Clients und bearbeitet diese. Er benötigt folgende Komponenten:

- Liste aller User (mit IP, Port)
- Liste aller Transaktionen (wird als Objekt realisiert)
- Thread der auf eingehende Verbindungen eingeht (Realisiert Anmelden von Clients und leitet sie weiter)
- Pro Client ein Thread (aus Thread-Pool) – TCP, nimmt Anfragen des Clients war. Greift auf die Transaktionen zu -> Read/Write Lock notwendig
- Ein Thread für UDP-Verbindungen. Dieser kann UDP-Pakete seriell schicken und muss nicht durch mehrere Threads realisiert werden
- Thread pro Transaktion (?) / UDP-Thread fragt Ta ab?

Mögliche Probleme

Problem	Lösung
Auflisten von TA als nicht angemeldeter User, welcher Thread wird verwendet	Server Thraed
Nebenläufiges Schreiben von Transaktionen	Read/Write Locks
Client: Die Ausgabe könnte von einem anderen Thread besetzt sein	Ein Thread auf die Ausgabe nur
Versenden von Nachrichten an nicht angemeldete User	Ev. Im Thread warten bis User erneut angemeldet ist. Nachrichten in einer Liste vom User schreiben
User beendet ohne Logout	TCP-Timeout? Variable beim User ob aktiv, wenn nicht Nachrichten in Liste

Designkonzept Server



Server

Der Server führt alle Anfagen des Users aus. Die wichtigsten Methoden sind dabei „bid“ Mit dieser Methode kann man auf eine Auction bieten. Dabei muss zuerst überprüft werden ob das Gebot höher ist und der User eingeloged ist. „Login“ und „Logout“ öffnen bzw. schließend die bestehenden Verbindungen und setzten der User aktiv oder inaktiv. Des Weiteren gibt es noch die Methode notify mit dieser werden die User über etwaige höhere Gebote und ähnliches Informiert. Die Methode „list“ dient zur Auflistung aller vorhandene Auktionen. Diese werden in folgender Form ausgegeben: ID: "+ Die Auctions ID + "Description: " + Die Auctions Description + "Highestbid: " + Höchstes Gebot+ " from "+von wem es stammt.

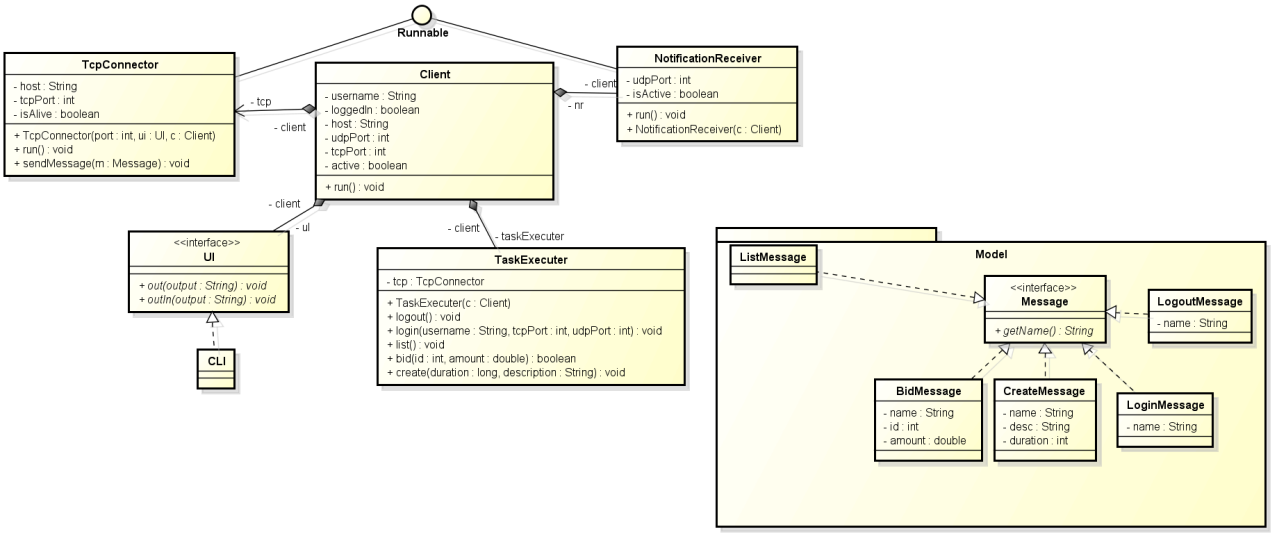
AuctionHandler

Ist eine Hilfsklasse des Servers. Die Aufgabe dieser ist es die Auctionen zu verwalten. Es handelt sich dabei um einen Threat der das Interface Runable implementiert. In diesem Threat werden laufend alle Auctionen überprüft ob diese schon abgelaufen sind. Wenn das der Fall ist werden sie beendet und und alle Bieter werden benachrichtig.

Getrennter Aufruf

Der Server verfügt nur über eine request(message : Message) Methode. Da der Server nur die Anfragen behandeln soll. Die einzelnen Funktionen sind dann in getrennten Klassen und werden von der request Methode aufgerufen dadurch können die Funktionalitäten leichter erweitert oder verändert werden.

Designkonzept Client



Client

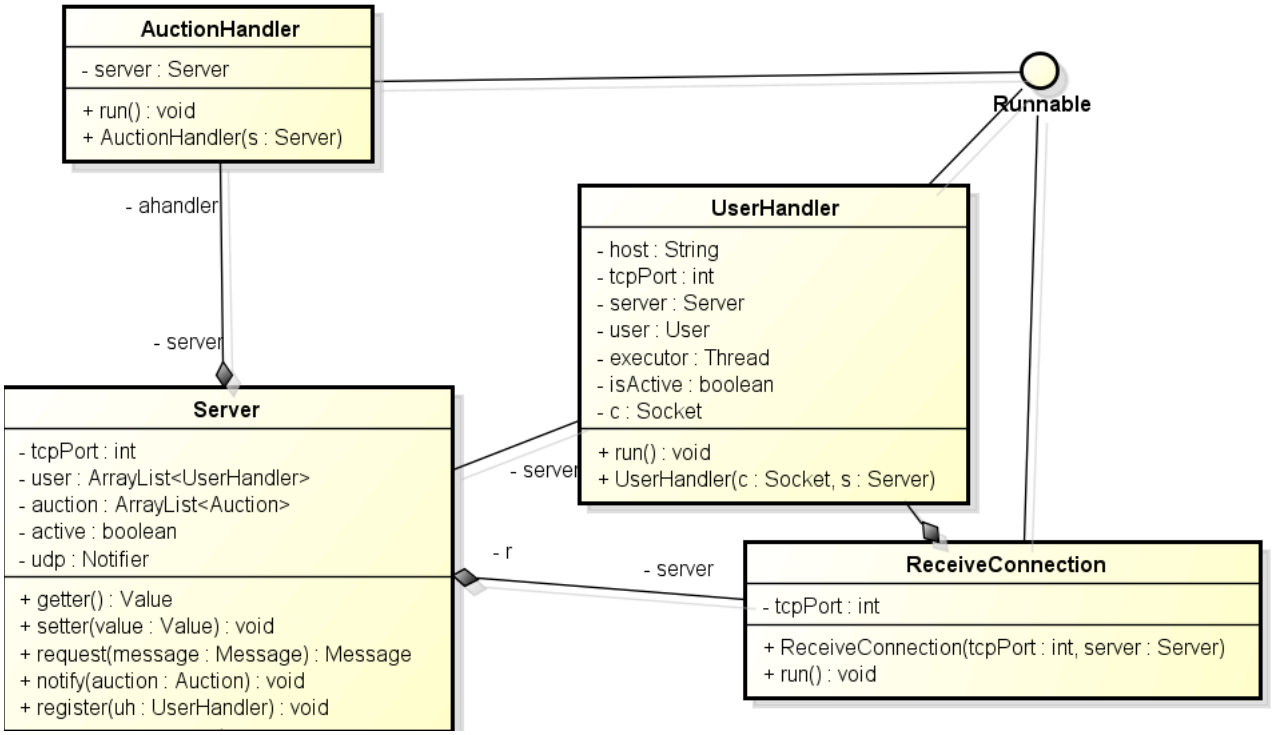
Mittels eines Threads werden die Eingaben des Users erfasst und über eine Netzwerkverbindung, an die Connector Klasse und an den Server weitergeleitet. In der run-Methode werden permanent die Eingaben des Users erfasst und die entsprechenden Methoden aufgerufen. Nach jeder User-Eingabe wird überprüft ob es Neuigkeiten gibt, z.B. ob eine Auktion endet, ob man überboten wurde usw.

Es wird Runnable implementiert, dadurch wird die run-Methode automatisch mitimplementiert, die run-Methode ist für das Erfassen der Eingaben zuständig.

Die Eingaben werden in entsprechenden Methoden als eigene Objekte verpackt, die im model für unsere Ansprüche definiert wurden. Diese Objekte werden serialisiert und mittels TCP an den Server übermittelt, den Transport übernimmt eine selbst geschriebene TCP-Verbindungsklasse.

Alle benötigten Daten werden bei jeder Anfrage an den Server weitergeleitet und die Daten somit neu geholt, es erfolgt keine Zwischenspeicherung jeglicher Daten(kein Buffering).

Designkonzept Verbindungen



UDP-Benachrichtigungen

Der Server hat die Möglichkeit auf die statische Methode **notify** der Klasse UDP-Notifier zuzugreifen. Dort gibt er an, welche User er benachrichtigen möchte und welche Nachricht er versenden möchte.

Umsetzung mittels Datagrampaket/socket.

Eingehende TCP-Verbindungen

Der Server hat einen extra Thread in **ReceiveConnection** der auf dem angegebenen tcp Port auf eingehende Verbindungen wartet. Erkennt er eine eingehende Verbindung, so wird ein neuer Socket erstellt und an den UserHandler übergeben.

Umsetzung mittels ServerSocket und Threads

User-Handler

Implementiert Runnable.

Ist ein Thread der ausschließlich für das Verwalten eines Clients mit einer TCP-Verbindung verantwortlich ist. Ihm ist egal, ob der User angemeldet ist oder nicht. Meldet sich der User an, wird das Attribut dementsprechend gesetzt. Er muss sich am Server mit dem Namen seines Users registrieren, damit dieser ihn später benachrichtigen kann. Der UserHandler interpretiert die Messages des Clients und führt entsprechende Aktionen am Server aus.

Streams

Es werden sogenannte ObjectStreams verwendet. Diese können Objekte über das Netzwerk versenden. Nötig dafür ist, dass diese Objekte das Interface Serializable implementieren. In unserem Beispiel ist das von Vorteil, da der Client Message-Objekte erstellt, welche über die Streams zum Server übertragen werden und dort interpretiert werden.

Beispiel für das Übertragen von Messages:

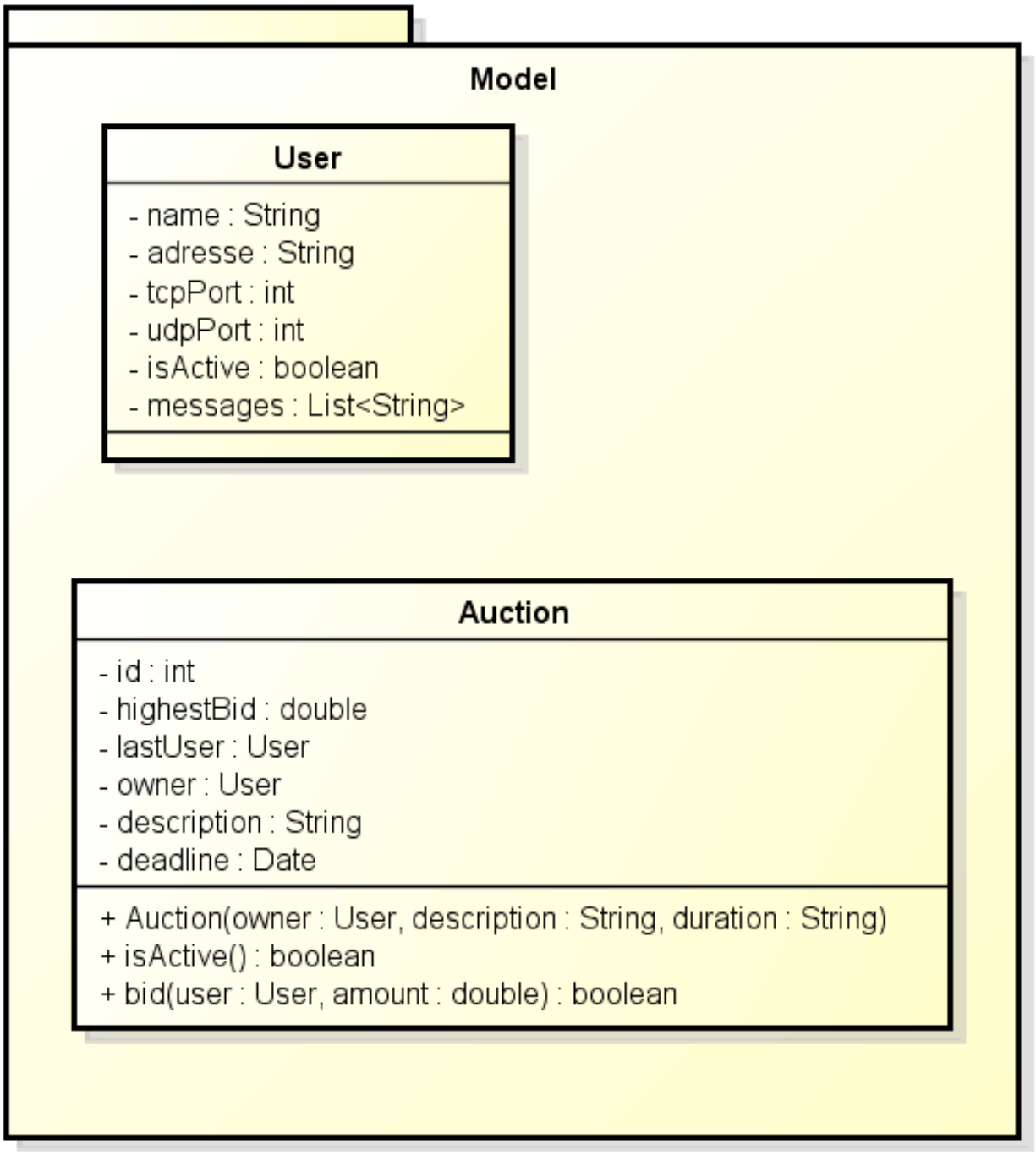
```
Object o = in.readObject();
m = (Message) o;
out.writeObject(m)
```

Empfängt eine Message und sendet diese Sofort wieder retour.

Verbindung

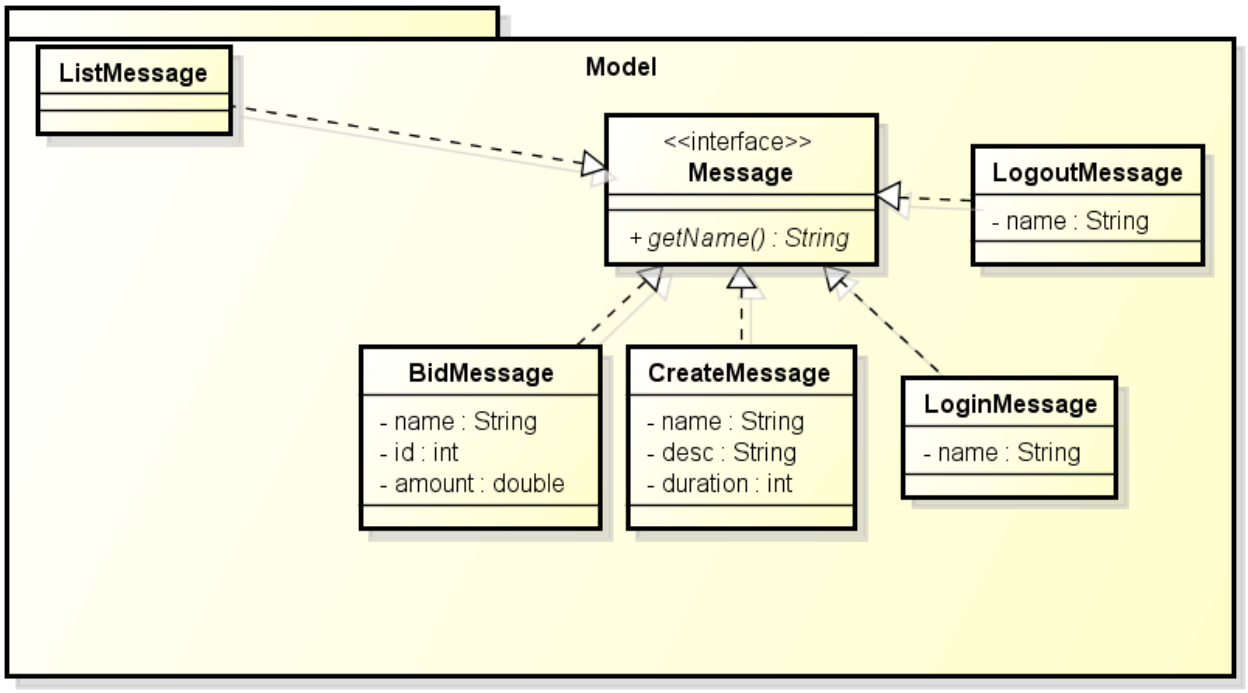
Antworten auf Nachrichten die ein Client über TCP bekommen hat, werden mittels TCP übertragen. Nachrichten die allgemein sind und eine Auktion betreffen, werden über UDP gesendet. Wird eine UDP Nachricht an einen User gesendet, der nicht aktiv ist, so wird ihm diese gespeichert. Bei einem Login werden diese Messages automatisch übergeben.

Model



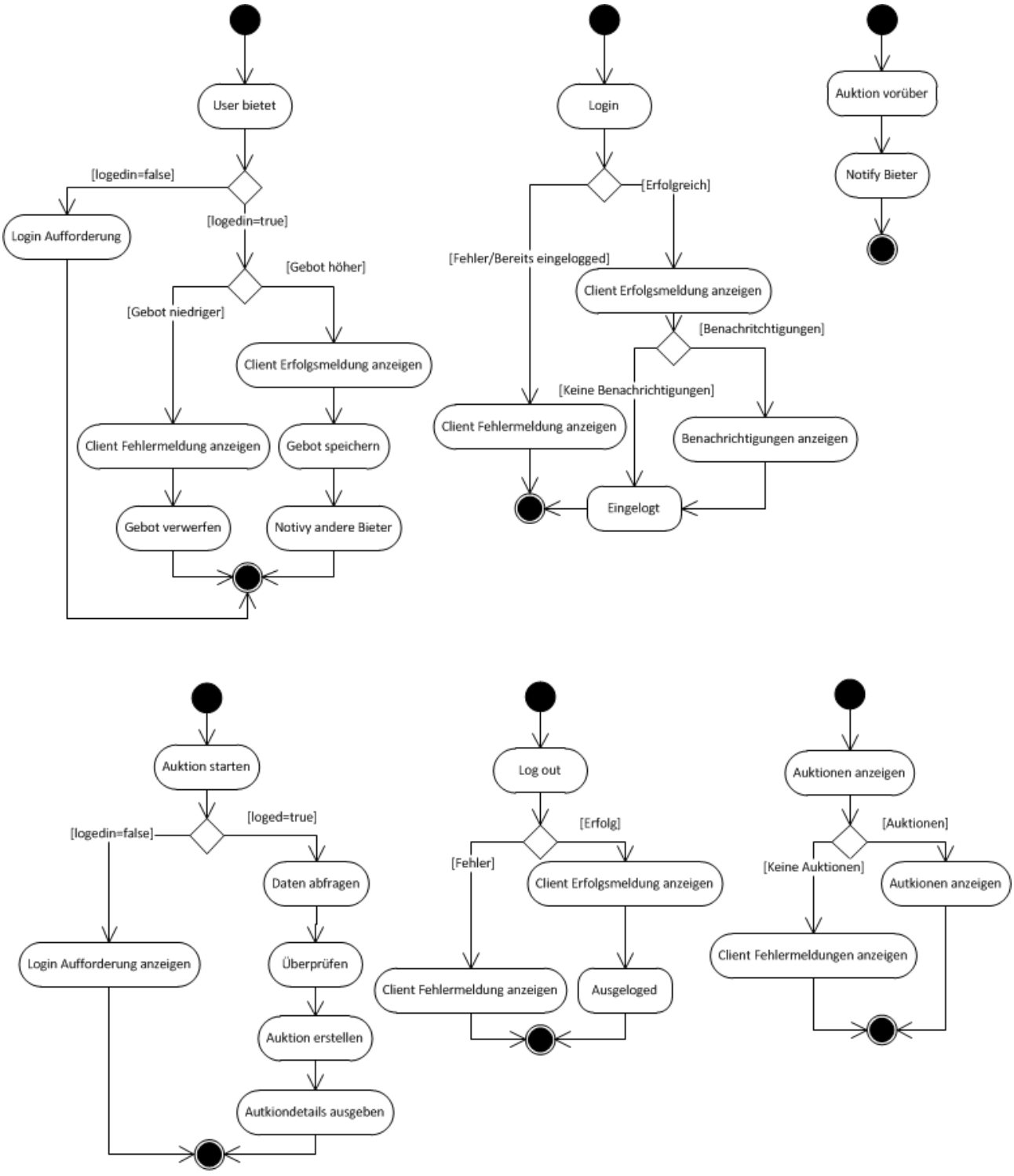
Es gibt die Klassen Auction und User diese dienen als Model hier werden alle Daten abgelegt die über den User bzw. über die Auction verfügbar sind. Diese Daten werden von den einzelnen Serverklassen bearbeitet.

Message

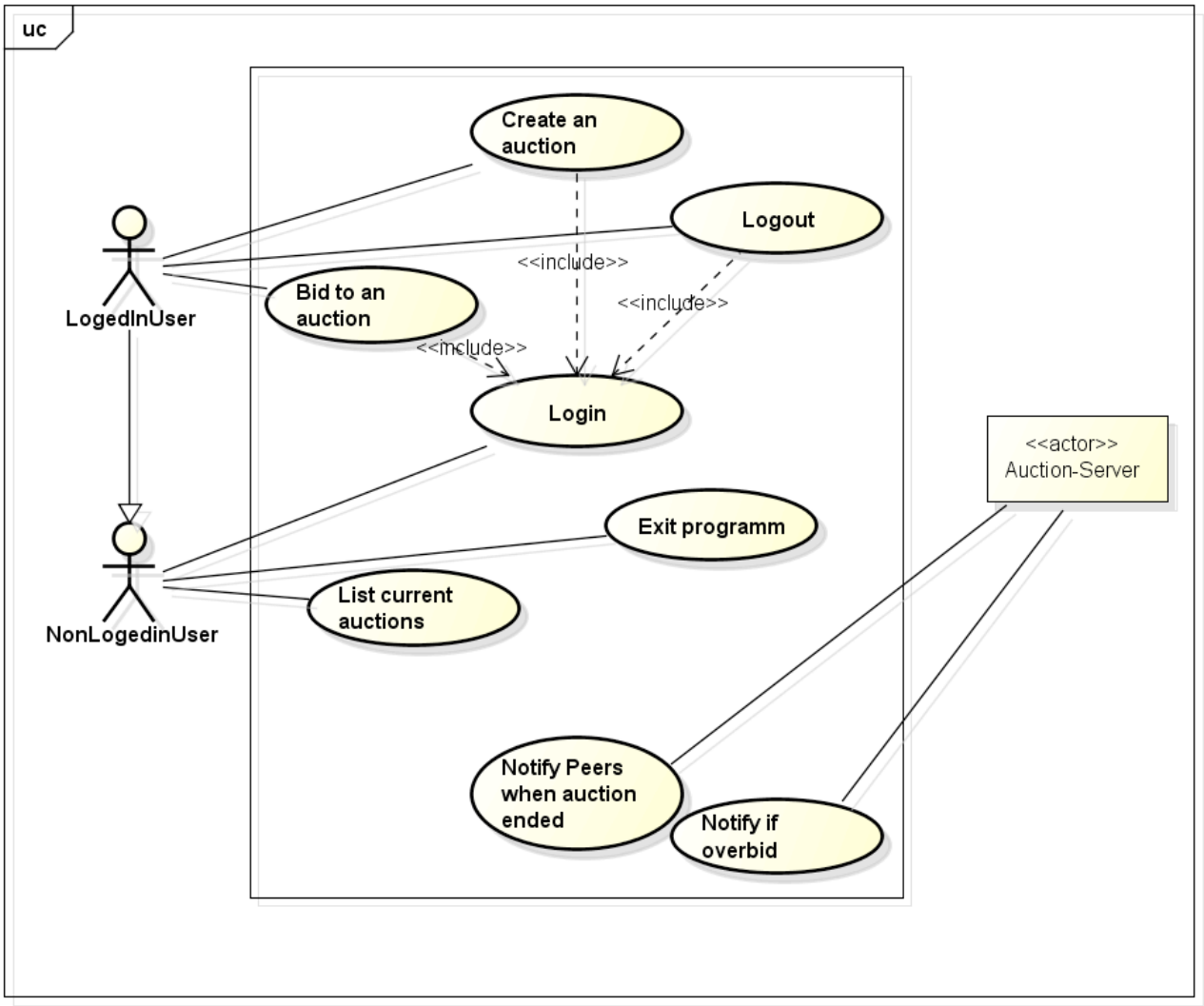


Für die Kommunikation über das Netzwerk werden alle Daten über Messgage Objekte verschickt. Message ist dabei ein Interface das alle verschiedenen Message Klassen implementieren. Dadurch können am Server die verschiedenen Anfragen von einer Methode bearbeitet werden da diese immer ein Message Objekt übergeben bekommt und je nach dem Inhalt anders handelt und eine ander Funktion aufruft.

Aktivitätendiagramm



Usecasediagramm



Funktionliste

- login: Ein login nur mit dem Namen.
- logout: Ein logout ist nur möglich mit einen loign vorher
- list: Ist sowohl eingeloged als auch ausgeloged möglich
- bid: Ist nur eingeloged möglich. Man brauch die ID der Auktion sowie einen Betrag den man bietet es findet eine Überprüfung statt ob der Betrag höher ist als der vorherige
- create: Ist nur eigeloged möglich. Man brauch eine Beschreibung und eine Deadline.
- Benachrichtigen über gewonnene und verlorene Auktionen

Probleme 11.12.13

Bis jetzt hat der Teamkollege Huang Kun Lung keine Mitarbeit gezeigt. Von Anfang an wurde wenig Interesse bemerkt. Es hat eine Woche gedauert bis er uns seine Dropbox Daten übermittelt hat und reagierte bis dahin nicht auf die Arbeitsanweisungen mündlich oder per Email.

Zum jetzigen Zeitpunkt wurden alle Teil-Arbeitspakete nicht eingereicht. Des Weiteren hat er mit uns nicht Kontakt aufgenommen zur Problembehebung.

Nicht Eingereichte Arbeitspakete:

- Arbeitszeitaufzeichnung
- Zeitschätzung/Zeitaufzeichnung
- Testfälle Designkonzept und erste Tests
- Recherche Testfälle
- TDD Testfälle

Probleme 18.12.13

Umsetzung des Strategy Patterns.

Kein Design-Approvement von Herr Professor Borko erhalten.

Konzept der Testfälle fehlt.

Die Regeln von TDD wurden nicht umgesetzt.

Unerlaubte Änderungen in bestehenden Klassen.

Recherche Testfälle fehlt.

Probleme 8.1.13

Reichmann

Der Socket schließt sich zu früh. Muss noch verbessert werden.

Valka

Keine.

Schuschnig

Probleme im AuctionHandler, dieser wirft in manchen Fällen noch eine NullPointerexception in Zeile 37 aus ungeklärten Gründen.

Huang

Reagiert auf keine Kommunikationsversuche über jegliche Medien. Dieses sind: Schul Mail, Private Mail, Facebook.

Des Weiteren hat er keinen Fortschritt der auf Git veröffentlicht wurden.

Der letzte Git zugriff ist 20 Tage her und der da publizierte Code ist fehlerhaft und es handelt sich nur um Autogenerierte Testfälle. Es wurde bis dato kein Code von Huang richtig getestet. Es wurde auch keine TDD Richtlinien wie vereinbart eingehalten.

Weiters hat Huang nicht das neue Arbeitspaket „Persistieren“ angefangen.

Deshalb fühlen wir uns in der letzten Arbeitswoche gezwungen seinen Teil vollständig zu übernehmen.

Probleme 15.1.2014

Übernahme der Arbeitspakete von Huang, da sie bis jetzt nicht erledigt wurden. Dabei sind keinen weiteren Probleme aufgetreten.

Arbeitsanweisung

Arbeitspakete	Kuan Huang Lun	Daniel Reich mann	Tobias Schusc hnig	Domini k Valka	Termi n	Erledigt	N. T.	Erledi gt
Zeitaufzeichnung und Zeitschätzung	x	x	x	x	11.12. 13	x Huang nein	18.12 .13	x
Recherche Testfälle	x				11.12. 13	nein	18.12 .13	x ducht Team
Designkonzept Verbindung		x			11.12. 13	x		
Designkonzept Server			x		11.12. 13	x		
Designkonzept Client				x	11.12. 13	x		
TDD Testfälle	x				11.12. 13	nein	18.12 .13	x durch Team
Entwurf Verbindung		x			11.12. 13	x		
Entwurf Server			x		11.12. 13	x		
Entwurf Client				x	11.12. 13	x		
Implementierung Client				x	7.1.14	x		
Implementierung Server			x		7.1.14	x		
Implementierung Verbindungen		x			7.1.14	x		
Usability Test	x				7.1.14	x durch Team		
Anpassung Abastimmung	x	x	x	x	7.1.14	x Huang nein		

Screenshots

Dieses Screenshots zeigen die Funktionstüchtigkeit des Programms und sollen einen Aufschluss darüber liefern.

Auction Time Out

Ausgabe wenn eine Auktion ausgelaufen ist und man versucht darauf zu bieten:

```
Dominik> !bid 0 100
The auction 'Super Small Notebook' is allready over you can not bid on this auct
ion anymore!
```

Bid succesfully

Die Ausgabe wenn man auf eine Auktion bietet und das Gebot erfolgreich war:

```
> !login Dominik
Successfully suscribed and loged in as: Dominik

Dominik> !bid 0 100
The auction 'Super Small Notebook' is allready over you can not bid on this auct
ion anymore!

Dominik> !bid 1 100
You successfully bid with 100.0 on 'New Auction'.

Dominik> !list
ID: 1    Description: New Auction    Highestbid: 100.0    from: Dominik
```

Create Auction

Die Ausgabe beim erfolgreichen erstellen einer Auction:

```
C:\Users\Daniel\Documents\Schule\APR\Git_Repos\Auction\AuctionSystem>java -jar l
ib\client.jar 127.0.0.1 12345 20002
Connector started

> !list
No auctions yet

> !login Daniel
Successfully suscribed and loged in as: Daniel

Daniel> !create 20 Super Small Notebook
You have succesfully created a new auction!

An auction 'Super Small Notebook' with the ID: 0 has been created and will end o
n Tue Jan 14 19:48:50 CET 2014.
Daniel>
Daniel>
The auction 'Super Small Notebook' has ended. Nobody bidded.
```

Dazu ist vorher ein login notwendig. Anschliessend sieht man auch das Ende dieser Auktion.

Unread Messages:

Beim erneuten einlogen werden einem die ungelesenen Nachrichten wie folgend angezeigt:

```
> !login Daniel
Successfully loged in as: Daniel
Unread messages: The auction 'New Auction' has ended. Dominik won with 100.0.
```