

# Task08 - RMI Auction 5AHITT

Schuschnig Tobias, Lins Tobias, Auradnik Alexander, Klune Alexander,  
Pölzbauer Patrick, Alexander Rieppel

# Inhaltsverzeichnis

<b>Task08 - RMI Auction</b>	<b>3</b>
<b>Aufgabenstellung</b>	<b>3</b>
<b>Aufgabenteilung (Vereinfacht)</b>	<b>15</b>
<b>Funktionsliste</b>	<b>16</b>
<b>Designüberlegung Allgemein (Probleme / Überlegungen)</b>	<b>17</b>
ConcurrentHashMap	17
Exceptions	17
<b>Designüberlegungen</b>	<b>18</b>
UML Klassendiagramme	18
UML Activity Diagramm	22
UML Use Case Diagramm	25
Management Client	26
<b>Abb2.: Entwickeltes Klassenkonzept</b>	<b>26</b>
Implementierung	26
Remote Message Management	27
Offene Fragen	26
Billing Server	28
Analytics Server	31
<i>Benachrichtigungen:</i>	32
Testing Component	33

Connections	<b>34</b>
Exceptions	<b>34</b>
<b>Zeitschätzung und Tatsächliche Zeit</b>	<b>35</b>
<b>Arbeitsaufteilung und Termine</b>	<b>38</b>
<b>Abnahme der Arbeitspakete:</b>	<b>40</b>
UML Klassendiagramm	<b>40</b>
<b>UML Aktivitätsdiagramm</b>	<b>40</b>
<b>UML Use Case Diagramm</b>	<b>40</b>
TDD (ersten Testfälle)	<b>40</b>
Zeitaufzeichnung und Zeitschätzung	<b>39</b>
Recherche Testfälle Designüberlegung	<b>40</b>
<b>Programmbeschreibung</b>	<b>43</b>
<b>Testing</b>	<b>43</b>
<b>Probleme</b>	<b>44</b>
Probleme 29.1.14	<b>44</b>
<b>Zeitaufzeichnung</b>	<b>45</b>

# Task08 - RMI Auction

## Aufgabenstellung

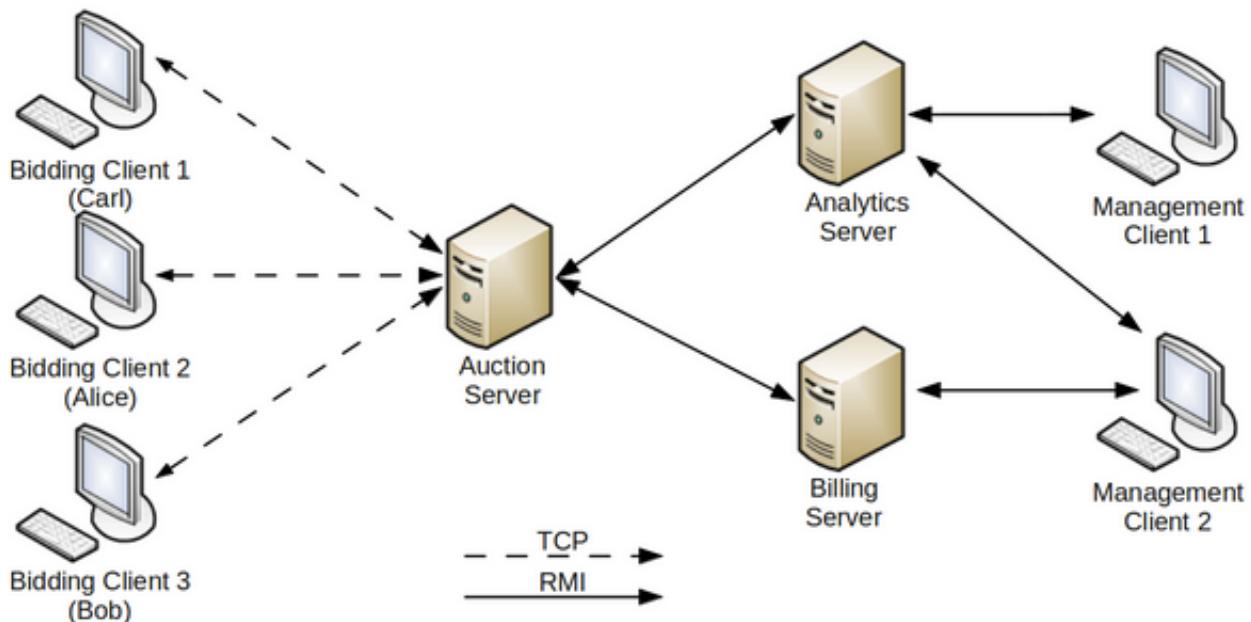
### General Remarks

- We suggest reading the following tutorials before you start implementing:
  - [Java RMI Tutorial](#): A short introduction into RMI.
  - [Distributed Computing for Java](#): RMI Whitepaper.
- Be sure to check the Hints & Tricky Parts section for questions!

### Overview

The goal of this assignment is to extend the auction system with four additional components:

- An **analytics server** receives events from the system and computes simple statistics/analytics.
- A **billing server** which manages the bills charged by the auction provider.
- A **management client** which interacts with the analytics server and the billing server.
- A **testing component** to generate client requests for automated load testing of the system.



**Figure 1:** System Components and Interactions.

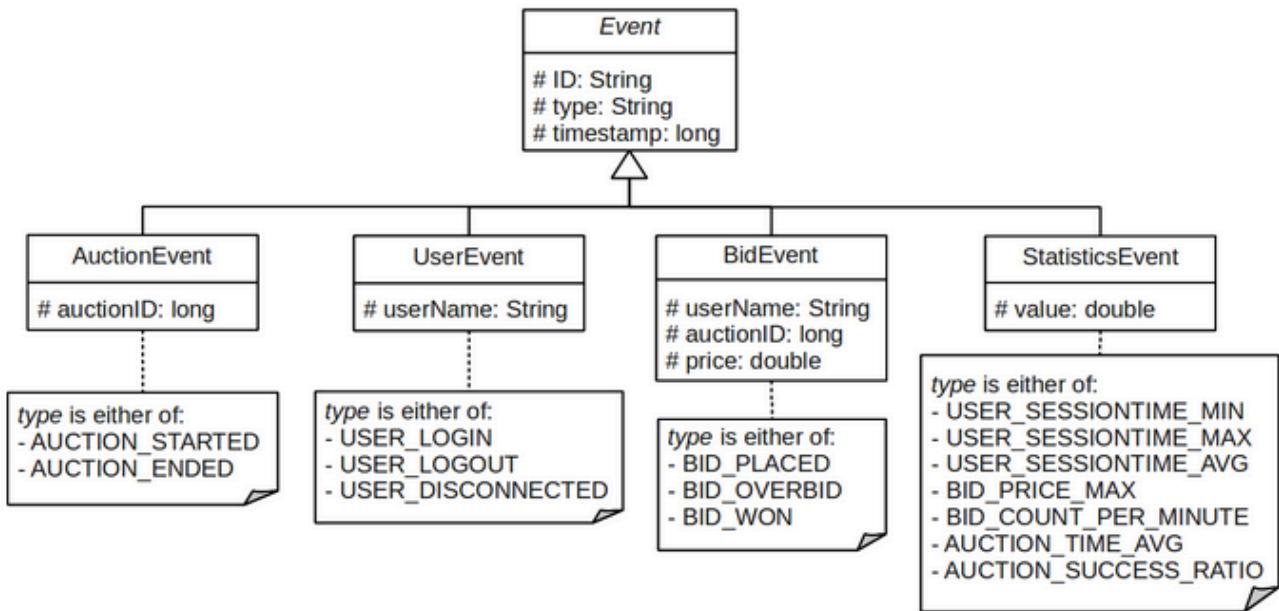
## Analytics Server

The purpose of the analytics server is to monitor the execution of the auction platform and to provide simple statistics about its usage. The analytics server should receive event updates from the auction server that you have implemented in assignment 1. Moreover, the analytics server should allow one or multiple management clients to subscribe for event notifications. To that end, the analytics server provides three RMI methods:

- A **subscribe** method is invoked by the management client(s) to register for notifications. The method must allow to specify a filter (specified as a regular expression string) that determines which types of events the client is interested in (see details further below). Moreover, the subscribe method receives a callback object reference which is used to send notifications to the clients. Study the RMI callback mechanism to solve this. The method returns a unique subscription identifier string.
- A **processEvent** method is invoked by the bidding server each time a new event happens (e.g., user logged in, auction started, ...). The analytics server forwards these events to subscribed clients, and possibly generates new events (see details further below) which are also forwarded to clients with a matching subscriptions. If the server finds out that a subscription cannot be processed because a client is unavailable (e.g., connection exception), then the subscriptions is automatically removed from the analytics server.
- An **unsubscribe** method is invoked by the management client(s) to terminate an existing event subscription. The method receives the subscription identifier which has been previously received from the **subscribe** method.

## Event Type Hierarchy

The processEvent method of the analytics server should receive an object of type **Event**. To transport the necessary event payload (user name, auction ID, bid price, ...), implement a simple event hierarchy which is illustrated in the figure below. The sub-types inherit from the abstract class **Event**, and the business logic inside the processEvent method should determine the concrete runtime type of the incoming events, in order to take appropriate action. Each event contains a unique identifier (ID), a type string, and a timestamp, and specialized event types contain additional data. For instance, the *AuctionEvent* defines a member variable *auctionID* that carries the auction identifier which this event applies to.

**Figure 2:** Event Hierarchy and Basic Event Properties.

## Statistics Events

The analytics server is responsible for processing the raw events and creating the following aggregated statistics event types:

- `USER_SESSIONTIME_MIN` / `USER_SESSIONTIME_MAX` / `USER_SESSIONTIME_AVG`: minimum/maximum/average duration over all user sessions. A user session starts with a login and ends if the user logs out (or gets disconnected unexpectedly).
- `BID_PRICE_MAX`: maximum over all bid prices.
- `BID_COUNT_PER_MINUTE`: number of bids per minute, aggregated over the whole execution time of the system.
- `AUCTION_TIME_AVG`: average auction time, aggregated overall historical auctions logged during the execution of the system.
- `AUCTION_SUCCESS_RATIO`: ratio of successful auctions to total number of finished auctions. An auction is successful if at least one bid has been placed.

Note that an incoming event (e.g., type `AUCTION_ENDED`) may trigger the generation of multiple new events (e.g., types `AUCTION_TIME_AVG`, `AUCTION_SUCCESS_RATIO`).

## Event Subscription Filter

The `subscribe` method of the statistics server should allow to set a simple subscription filter. The filter is a Java regular expression which is matched against the `type` variable of the `Event` class. For instance, to subscribe for all user-related and bid-related events, the filter "`(USER_.* | BID_.* )`" is used.

## Billing Server

The billing server provides RMI methods to manage the bills of the auction system. To secure the access to this administrative service, the server is split up into a BillingServer RMI object (which basically just provides login capability) and a BillingServerSecure which provides the actual functionality.

### Pricing Curve

Administrators can use the billing server to set the pricing curve in terms of fixed price and variable price steps. In a typical pricing curve, the auction fee percentage decreases with increasing auction price (price of the winning bid). For example, the price steps could look something like this:

Auction Price	Auction Fee (Fixed Part)	Auction Fee (Variable Part)
0	1.0	0.0 %
(0-100]	3.0	7.0 %
(100-200]	5.0	6.5 %
(200-500]	7.0	6.0 %
(500-1000]	10.0	5.5 %
> 1000	15.0	5.0 %

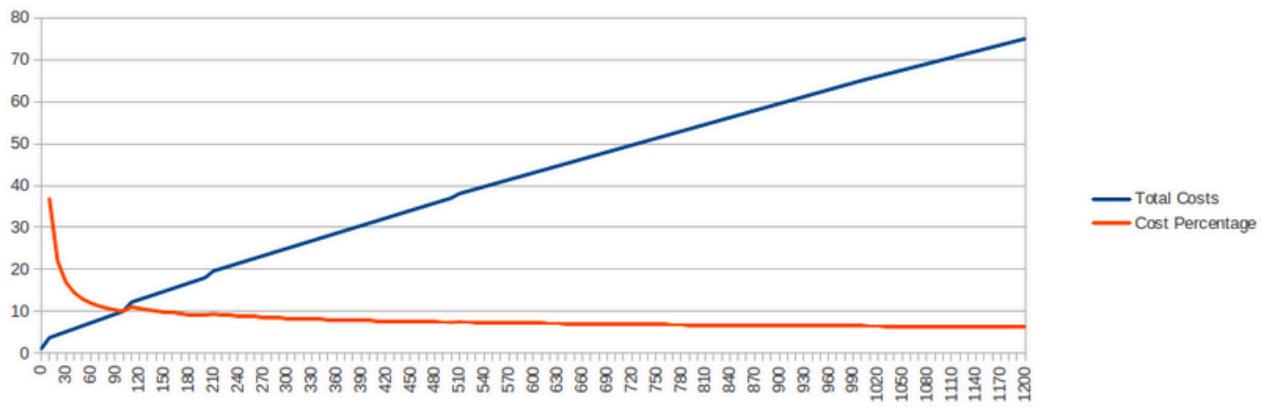


Figure 3: Pricing Curve

## Billing Server RMI Methods

The *BillingServer* class provides the following RMI methods (exception declarations not included):

- `BillingServerSecure login(String username, String password):`

The access to the billing server is secured by user authentication. To keep things simple, the username/password combinations can be configured statically in a config file

`user.properties`. Each line in this file contains an entry "<username> = <password>", e.g., "john = f23c5f9779a3804d586f4e73178e4ef0". Do not put plain-text passwords into the config file, but store the MD5 hash (not very safe either, but sufficient for this assignment) of the passwords. Use the `java.security.MessageDigest` class to obtain the MD5hash of a given password. If and only if the login information is correct, the management client obtains a reference to a *SecureBillingServer* remote object, which performs the actual tasks.

The *BillingServerSecure* provides the following RMI methods (exception declarations not included):

- `PriceSteps getPriceSteps():` This method returns the current configuration of price steps. Think of a suitable way to represent the list of price step configurations inside your `PriceSteps` class.
- `void createPriceStep(double startPrice, double endPrice, double fixedPrice, double variablePricePercent):` This method allows to create a price step for a given price interval. Throw a `RemoteException` (or a subclass thereof) if any of the specified values is negative. Also throw a `RemoteException` (or a subclass thereof) if the provided price interval collides (overlaps) with an existing price step (in this case the user would have to delete the other price step first). To represent an infinite value for the `endPrice` parameter (e.g., in the example price step "> 1000") you can use the value 0.
- `void deletePriceStep(double startPrice, double endPrice):` This method allows to delete a price step for the pricing curve. Throw a `RemoteException` (or a subclass thereof) if the specified interval does not match an existing price step interval.
- `void billAuction(String user, long auctionID, double price):` This method is called by the auction server as soon as an auction has ended. The billing server stores the auction result (data do not have to be persisted, storing in memory is sufficient) and later uses this information to calculate the bill for a user. **Note:** The auction server should use the same login mechanism as the management clients to talk to the billing server. You can add pre-defined user credentials (for instance "auctionClientUser = f23c5f9779a3804d586f4e73178e4ef0") to your properties file.
- `Bill getBill(String user):` This method calculates and returns the bill for a given user, based on the price steps stored within the billing server. Implement a class `Bill` which encapsulates all billing lines of a given user (also see sample output of management client further below). You need not implement any payment mechanism - the bill should simply show the total history of all auctions created by the user.

## Management Client

The management client communicates with the analytics server and the billing server. The client should provide all necessary commands to interact with the analytics server and the billing server. The following commands are used to interact with the billing server (please follow the output format illustrated in the examples):

- **!login <username> <password>**: Login at the billing server.

```
> !login bob BobPWD
bob successfully logged in
```

- **!steps**: List all existing price steps

```
bob> !steps
```

Min_Price	Max_Price	Fee_Fixed	Fee_Variable
0	0	1.0	0.0 %
0	100	3.0	7.0%
100	200	5.0	6.5%
200	500	7.0	6.0%
500	1000	10.0	5.5%

Note: the table columns in the output do not have to be perfectly aligned.

- **!addStep <startPrice> <endPrice> <fixedPrice> <variablePrice-Percent>**: Add a new price step.

```
bob> !addStep 1000 0 15 5
Step [1000 INFINITY] successfully added
bob> !steps
```

Min_Price	Max_Price	Fee_Fixed	Fee_Variable
0	0	1.0	0.0 %
0	100	3.0	7.0%
100	200	5.0	6.5%
200	500	7.0	6.0%
500	1000	10.0	5.5%
1000	INFINITY	15.0	5.0%

- **!removeStep <startPrice> <endPrice>**: Remove an existing price step.

```
bob> !removeStep 0 100
Price step [0 100] successfully removed
bob> !removeStep 111 222
ERROR: Price step [111 222] does not exist
```

- **!bill <userName>**: This command shows the bill for a certain user name. Print the list of finished auctions (plus auction fees) that have been created by the specified user. To de-

determine the fees of the bill, apply the current price steps configuration to the prices of each of the user's auctions.

```
bob> !bill bob
```

auction_ID	strike_price	fee_fixed	fee_variable	fee_total
1	0	1	0	1
17	700	10	38.5	48.5
19	120	5	7.8	12.8

- **!logout:** Set the client into "logged out" state and discard the local copy of the *BillingServiceSecure* remote object. After this command, users have to use the "**!login**" command again in order to interact with the billing server.

```
bob> !logout
bob successfully logged out
> !bill bob
ERROR: You are currently not logged in.
```

Provide reasonable output with all relevant information for each of the commands (as illustrated in the examples above). Also print an informative error message if an exception is received from the server (Error log messages should start with the string "ERROR:").

The following user commands are provided by the management client to communicate with the analytics server:

- **!subscribe <filterRegex>:** subscribe for events with a specified subscription filter (regular expression). A user can add multiple subscriptions.

```
bob> !subscribe '(USER_.* )|(BID_.* )'
Created subscription with ID 17 for events using filter
'(USER_.* )|(BID_.* )'
```

- **!unsubscribe <subscriptionID>:** terminate an existing subscription with a specific identifier (**subscriptionID**).

```
bob> !unsubscribe 17
subscription 17 terminated
```

Similar to the analytics server, the management client (i.e., the RMI callback object) should implement a **processEvent** RMI method. This method is invoked by the analytics server each time an event is generated that matches a subscription by this client. To display incoming events to the user, simply print the event time, event type and all additional event properties to the command line. You should support two modes of printing, namely *automatic* and *on-demand*:

- **!auto:** This command enables the automatic printing of events. Whenever an event is received by the clients, its details are immediately printed to the command line.
- **!hide:** This command disables the automatic printing of events. Incoming events are temporarily buffered and can later be printed using the "**!print**" command. This should be the default mode when the client is started.
- **!print:** Print all events that are currently in the buffer and have not been printed before (an excerpt of a possible example trace is printed below).

```

bob> !print
USER_LOGIN: 31.10.2012 20:00:01 CET - user alice logged in
USER_LOGIN: 31.10.2012 20:01:03 CET - user john logged in
BID_PLACED: 31.10.2012 20:01:50 CET - user alice placed bid
3100.0 on auction 32
BID_PRICE_MAX: 31.10.2012 20:01:50 CET - maximum bid price
seen so far is 3100.0
BID_COUNT_PER_MINUTE: 31.10.2012 20:01:50 CET - current bids
per minute is 0.563
USER_LOGOUT: 31.10.2012 20:03:11 CET - user john logged out
USER_SESSION_TIME_MIN: 31.10.2012 20:03:11 CET - minimum ses-
sion time is 62 seconds
USER_SESSION_TIME_MAX: 31.10.2012 20:03:11 CET - average ses-
sion time is 324 seconds
USER_SESSION_TIME_AVG: 31.10.2012 20:03:11 CET - maximum ses-
sion time is 778 seconds

```

Note: The value of BID\_COUNT\_PER\_MINUTE depends on the previous events which are not visible in this trace. For the example, an arbitrary value of 0.563 has been chosen for illustration. (Arbitrary values have also been chosen for the aggregated session durations.)

In both variants, it must be ensured that each distinct event is presented to the user only *once*, even if it matches multiple subscriptions. It is up to you whether you solve this requirement server-side (never send out duplicate events to any client) or client-side (receive duplicate events, but print each event only once on the command line). If you need to temporarily store a list of already published or already printed events, make sure that the events are eventually freed and that your program does not run out of memory over time.

## Load Testing Component

In practice, online auction systems are highly concurrent and should provide robustness and scalability, even for a large number of clients. Concurrency issues are hard to detect under normal operation, but generating artificial load on the system may help to detect problems and inconsistencies. Hence, you should create a load testing component to test the system's scalability and reliability. The following test parameters should be configurable in the properties file `loadtest.properties` (see template):

- *clients*: Number of concurrent bidding clients
- *auctionsPerMin*: Number of started auctions per client per minute
- *auctionDuration*: Duration of the auctions in seconds
- *updateIntervalSec*: Number of seconds that have to pass before the clients repeatedly update the current list of active auctions
- *bidsPerMin*: Number of bids placed on (random) auctions per client per minute

To place a bid, the test client selects a random auction from the list of active auctions (if any). When your test clients place bids, you need to ensure that each bid is higher than the previous bids. To achieve this, simply set the bid price that corresponds to the number of seconds that have passed since the auction was started, with decimal values (e.g., 10,342 for ten seconds and 342 milliseconds).

Moreover, the test environment should instantiate a management client with an event subscription on any event type (filter ".\*"). During the tests, the management client should be in "auto" mode, i.e., print all the incoming events automatically to the command line.

Scaling up the number of clients would block too many port numbers for the UDP notifications. Hence, for assignment 2 you should disable the UDP notifications (part of assignment 1), i.e., do not open a UDP socket on the bidding clients, and do not send UDP notifications from the auction server to the clients.

Play around with different test settings and try to roughly determine the limits of your machine. By limits we mean the configuration values for which the system is still stable, but becomes unstable (e.g., runs out of memory after some time) if any of these values are further increased (or decreased). Monitor the memory usage with tools like "top" (Unix) or the process manager under Windows, and let the test program execute for a sufficiently long time (around 5-10 minutes). Obviously, the load test can also help you identify issues in your implementation (e.g., deadlocks, memory leaks, ...).

In your submission, include a file **evaluation.txt** in which you provide your machine characteristics (operating system version, number and clock frequency of CPUs, RAM capacity) and the key findings of your evaluation (at least the limit values for all configuration parameters). **Note: Make sure that your tests terminate after a short time** (say, 8 minutes)!

## Implementation Details and Hints

### Implementation Details

RMI uses the `java.rmi.registry.Registry` service to enable application to retrieve remote objects. This service can be used to reduce coupling between clients (looking up) and servers (binding): the real location of the server object becomes transparent. In our case, the servers (analytics server and the billing server) will use the registry for binding and the client will look up the remote objects.

One of the first things the servers need to do is to connect to the `Registry`, therefore the RMI registry needs to be set up before its first usage. This can be achieved by calling the

`LocateRegistry.createRegistry(int port)` method which creates and exports a `Registry` instance on localhost. A properties file (named `registry.properties`) should be read from the classpath (see the hint section for details) to get the port the `Registry` should accept requests on. The properties file is provided in the template. It also contains the host the `Registry` is bound to. This information is vital to the client application that needs to connect to the `Registry` using the `LocateRegistry.getRegistry(String host, int port)` method. Note that a client, in contrast to the management component, need not bind any objects to the `Registry`.

After obtaining a reference to the `Registry`, this service can be used to bind an RMI remote interface (using the `Registry.bind(String name, Remote obj)` method). Remote Interfaces are common Java Interfaces extending the `java.rmi.Remote` interface. Methods defined in such an interface may be invoked from different processes or hosts. In our case, we need to bind two objects to the registry: an instance of `BillingServer` and an instance of `AnalyticsServer`. If you prefer to bind only a single object to the registry, you may choose to implement an additional "facade" object which returns instances of the two server classes (however, this is not required).

To make your object remotely available you have to export it. This can either be accomplished by extending `java.rmi.server.UnicastRemoteObject` or by directly exporting it using the static method `UnicastRemoteObject.exportObject(Remote obj, int port)`. In the latter case, use 0 as port: This way, an available port will be selected automatically.

For managing the configurable data (e.g., user credentials) you will have to read a `.properties` file. The `user.properties` file must be located in the server's classpath. A sample user properties file is provided in the template.

## Bootstrapping and Terminating the System

To allow for efficient (automated) testing of your solution, make sure that your system components can be started using both of the following command sequences:

- `ant run-billing-server`
- `ant run-analytics-server`
- `ant run-server`

or

- `ant run-analytics-server`
- `ant run-billing-server`
- `ant run-server`

That is, the billing server and the analytics server are started first (in any order), then the auction server is started. Upon instantiation of the billing server and the analytics server you should check whether the RMI registry is already available, and create a new registry instance if it does not yet exist. You can assume that there is a short delay (3 seconds) between the start of each component. To terminate any of the three components, the command `!exit` is used in the terminal in which the component is running. Terminating any of the three servers should be handled gracefully in the other components, i.e., if the auction server is unable to communicate with either the analytics server or the billing server, a warning message should be displayed, but the auction server should not terminate or raise/print an exception. Any requests targeted at an unavailable server can simply be dropped. This means that you do *not* need to store/buffer any requests or events in the auction server until the other servers become available. Simply drop requests or events (with a warning message) if they cannot be forwarded to the target server immediately.

The management clients should also be terminated using the `!exit` command. Make sure to properly clean up all resources before the client terminates.

## Important Points to Consider

Make sure to synchronize the data structures you use to manage price steps, events, etc. Even though RMI hides many concurrency issues from the developer, it cannot help you at this point. You may consult the [Java Concurrency Tutorial](#) to solve this problem.

The data needs to be persisted after shutting down the server (e.g. save data to a File). You cannot assume that the management component is up first at all, so the clients have to react on unsuccessful lookup of the management components.

## Ant template

As in the last assignment we provide a [template](#) build file (`build.xml`) in which you only have to adjust some class names, ports, and RMI names. Further on we give you the opportunity to use log4j. Do not use any other third-party libraries. Put your source code into the subdirectory "src", the `filesregistry.properties`, `loadtest.properties` and `user.properties` are already in the "src" directory (the ant compile task then copies this file to the build directory). Put the src directory including `registry.properties` and `build.xml` into your submission. Note that it's **absolutely required** that we are able to start your programs with the predefined commands!

## Hints & Tricky Parts

- To make your object remotely available you have to **export** it. This can either be accomplished by extending `java.rmi.server.UnicastRemoteObject` or directly exporting it using the static method  
`java.rmi.server.UnicastRemoteObject.exportObject(Remote obj, int port)`. Use 0 as port, so any available port is selected by the operating system.
- Before shutting down a server or client, unexport all created remote objects using the static method `UnicastRemoteObject.unexportObject(Remote obj, boolean force)` - otherwise the application may not stop.
- Since Java 5 it's not required anymore to create the stubs using the **RMI Compiler** (`rmic`). Instead java provides an automatic proxy generation facility when exporting the object.
- You should not use a Security Manager. So you do not need a policy files.
- Take care of **parameters and return values** in your remote interfaces. In RMI all parameters and return values except for remote objects are passed per value. This means that the object is transmitted to the other side using the java serialization mechanism. So it's required that all parameter and return values are serializable, primitives or remote objects, otherwise you will experience `java.rmi.UnmarshalExceptions`.
- To create a **registry**, use the static method  
`java.rmi.registry.LocateRegistry.createRegistry(int port)`. For obtaining a reference in the client you can use the static method  
`java.rmi.registry.LocateRegistry.getRegistry(String hostName, int port)`. Both hostname and port have to be read from the `registry.properties` file.
- We also provide a `registry.properties` file in the template. Make sure to set the port according to our policy.
- Reading in a **properties file** from the classpath (without exception handling):

```

java.io.InputStream is =
ClassLoader.getSystemResourceAsStream("registry.properties");
if (is != null) {java.util.Properties props = new
java.util.Properties();
try {props.load(is);
String registryHost = props.getProperty("registry.host");
...
}

} finally {is.close();}

}
} else {System.err.println("Properties file not found!");
}

```

## Further Reading Suggestions

- **APIs:**
  - RMI: [Remote API](#), [UnicastRemoteObject API](#), [Registry API](#), [LocateRegistry API](#)
  - Properties: [Properties API](#)
  - IO: [IO Package API](#)
- **Tutorials**
  - [JavaInsel RMI Tutorial](#): German introduction into RMI programming.

## 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 continuous 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(10): usecase, uml-class, activity diagrams
- Documentation(10): timetable, explanation, description(JavaDoc), protocol/test documentation
- Implementation(40): exception handling, concurrency, resource handling, performance, functional requirements
- Testing(15): unit-testing, coverage, system-testing, user-acceptance

published by: Vienna University of Technology  
Institute of Information Systems 184/1  
Distributed System Group

## Aufgabenteilung (Vereinfacht)

Für eine genaue Beschreibung bitte im Kapitel „Aufgabenteilung und Termine“ nachschauen.

Schuschnig:

- UML Klassendiagramm
- Analytics Server
- Connections
- Protokoll / Reviews
- Team Kontrolling

Lins:

- UML Klassendiagramm
- Billing Server
- Connections
- Technical Architect

Auradnik:

- UML Aktivitätsdiagramm
- Management Client

Klune:

- UML Usecasediagramm
- Billing Server
- Testing Component

Pöelzbauer:

- UML Aktivitätsdiagramm
- Testing Component
- Exceptions

Rieppel:

- UML Use Case Diagramm
- Testing (TDD)
- Management Client
- Model

## Funktionsliste

Analytics Server:

- Speichern aller Events
- Statistische Auswertung in calculate()
- Benachrichtigen mittels processEvent()
- Anmelden und Abmelden für Benachrichtigungen suscribe und unsuscribe()

Billing Server:

- Festlegen der Price Steps (Hinzufügen/Entfernen)
- Ausgeben der Price Steps
- Erstellt für jeden Benutzer Bills
- Bills ausgeben

Management Client:

- Greift auf die Funktionen von BillingServer und AnalyticsServer zu
- Benachrichtigungen mit print() oder automatisch ausgeben

Testing Component:

- Testen des gesamten Systems durch automatisches erstellen von Auktionen und Usern und automatischem bieten
- Verschiedenen Testing Einstellungen siehe Kapitel: „Testing Component“

## Designüberlegung Allgemein (Probleme / Überlegungen)

### ConcurrentHashMap

Ab jetzt werden überall ConcurrentHashMaps verwendet. Diese haben eine bessere Performance und lösen die Probleme mit Threads die in dieses Programm oft verwendet werden.

Key: Sollte dabei immer das Attribut sein nach dem meistens identifiziert wird. Zum Beispiel: Username.

Probleme: Key muss unique sein?, Wenn nach anderem Attribut identifiziert wird?

### Exceptions

Wenn es sinnvoll ist sollen Exceptions definiert werden diese werden müssen auch im UML Klassendiagramm vermerkt werden.

### Offene Fragen Management Client

- Verbindungsaufbau + Eventmanagement
- Management von gleichen Ausgaben, bei mehreren Abonneten (subscriptions). Server oder Clientseitig.
- Implementierung Command Pattern
- Zweck des TcpConnectors

## Designüberlegungen

### UML Klassendiagramme

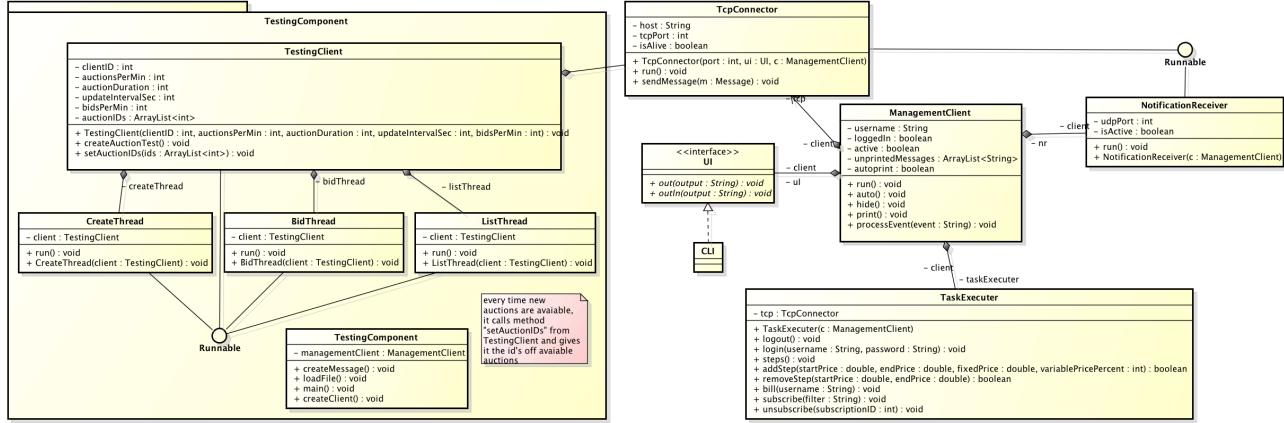


Abbildung 1: Hier sieht man den Client und die TestingComponent

Der Weitgehend unveränderte Teil aus Aufgabe07. Hier sieht man den Client wie er damals schon funktioniert hat allerdings wurden die UDP Benachrichtigungsteile gelöscht.

Des Weiteren sieht man die Erweiterung des TestingComponents dieser wird im Kapitel Testing Komponent genau beschrieben.

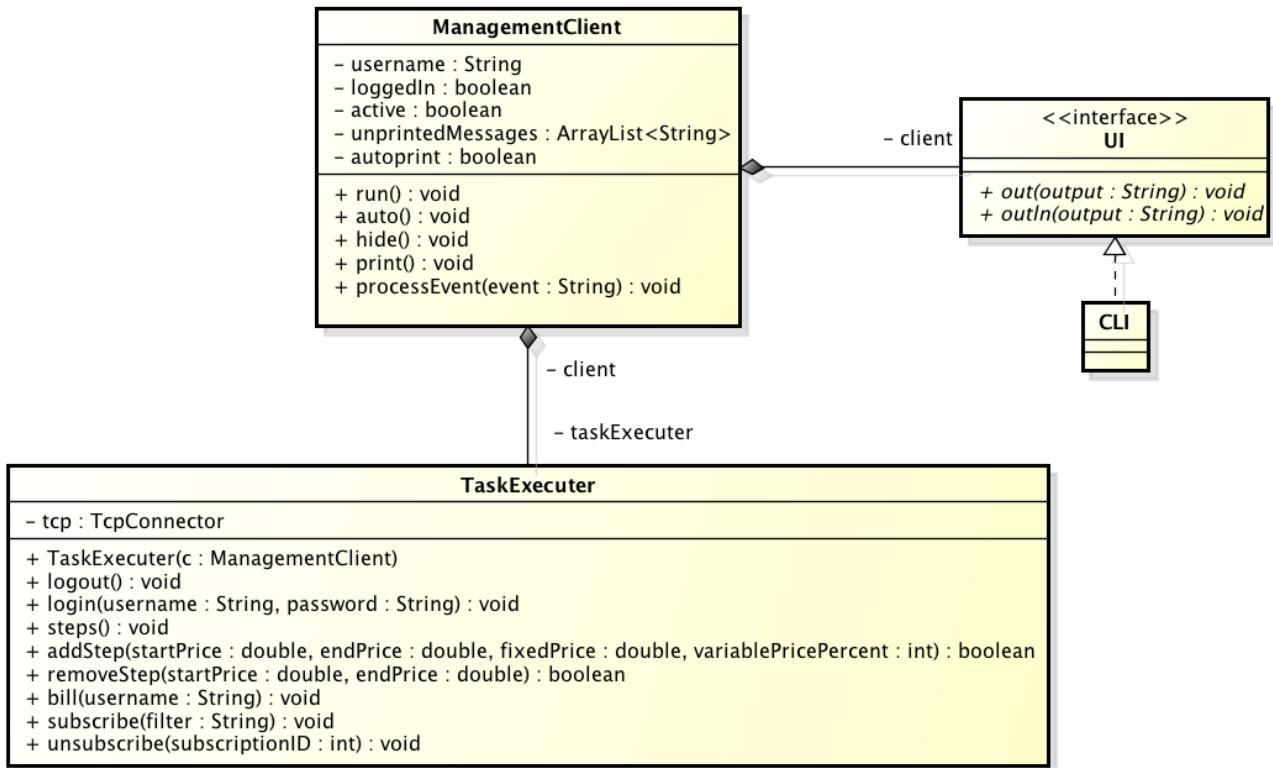


Abbildung 2: Das UML Diagramm des Management Clients

Dieser wurde im UML dem normalen Client nachempfunden. Es soll dabei an das bewährte Design des Clients festgehalten werden und dieser ähnlich programmiert werden.

Der TaskExecuter wird dabei immer aufgerufen wenn eine Methode nicht lokal ausgeführt wird sondern auf einem der Server aufgerufen wird. Aus diesem Grund verfügt der TaskExecuter für jede Funktion des ManagementClients eine Methode die mittels RMI dann die gewünschte Methode auf den Servern invoked.

Die Eingabe findet weiterhin ganz normal über die CLI statt. Dabei soll wieder das gleiche Konzept wie beim normalen Client verwendet werden

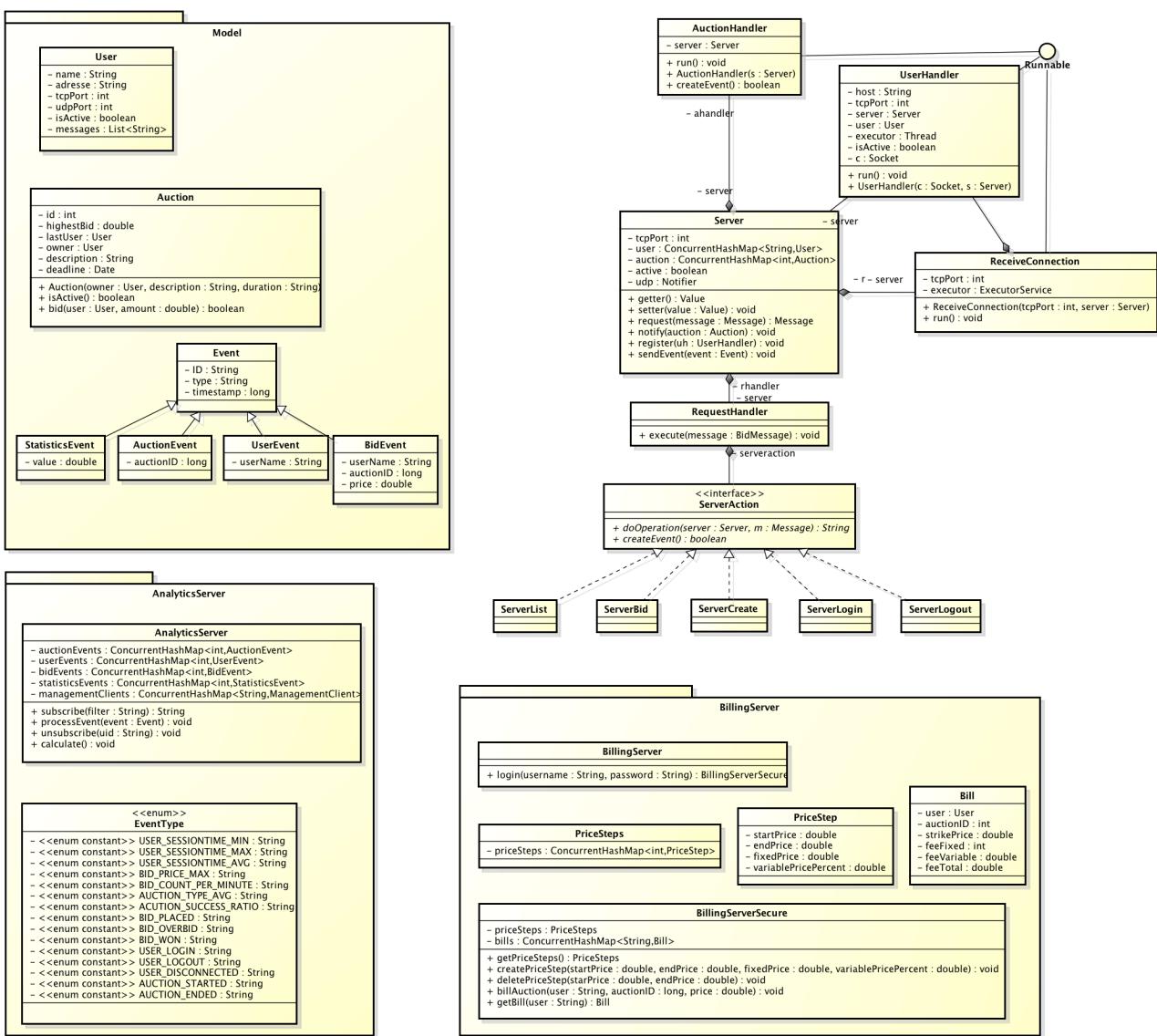


Abbildung 3: Hier sieht man alle Server im Überblick

Der Hauptserver auf dem die Auctionen laufen wurde dabei weitgehend unverändert gelassen bis auf den Unterschied, dass es keine UDP Benachrichtigungen mehr gibt.

Deshalb musste sowohl der Server als auch der AuctionHandler leicht modifiziert werden. Der Server verfügt des weiteren über zwei neue Methoden. Die Methode `createEvent()` erzeugt wann immer eine Aktion geschieht ein Event (siehe Package Model) und schickt diese mittels der Methode `sendEvent()` über RMI an den AnalyticsServer. Dieser speichert die Events und wertet sie statistisch aus.

Die restlichen Teile des Servers bleiben weitgehend unverändert da sich dieser bewährt hat.

Neu dazu kommen eine AnalyticsServer und ein BillingServer diese sind jeweils über RMI mit dem Server verbunden und eine ManagementClient kann sich ebenfalls über RMI mit diesen verbinden. (Allerdings kann er sich nicht direkt mit dem Server verbinden).

Der AnalyticServer und der BillingServer werden in den folgenden Kapiteln genauer beschrieben.

## UML Activity Diagramm

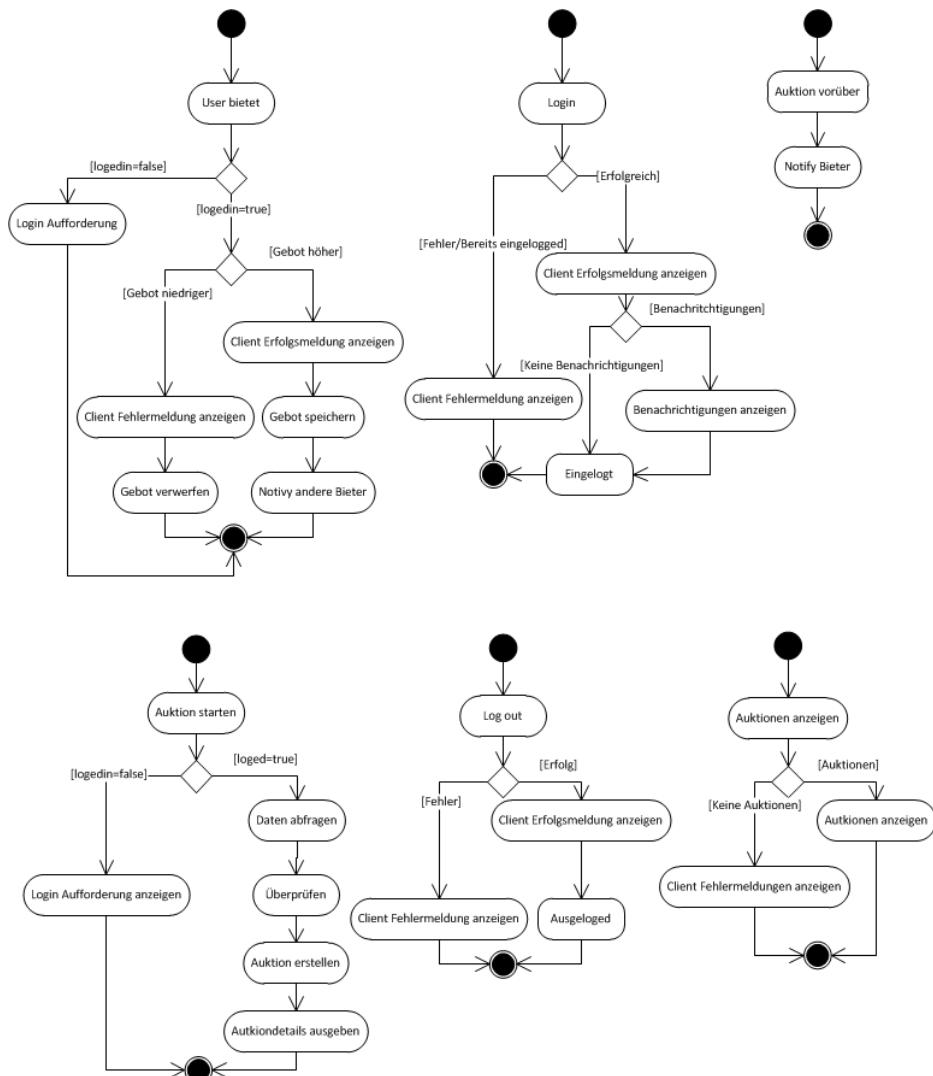


Abbildung 4: Zeigt das Activity Diagramm von Task07 mit allen Funktionen

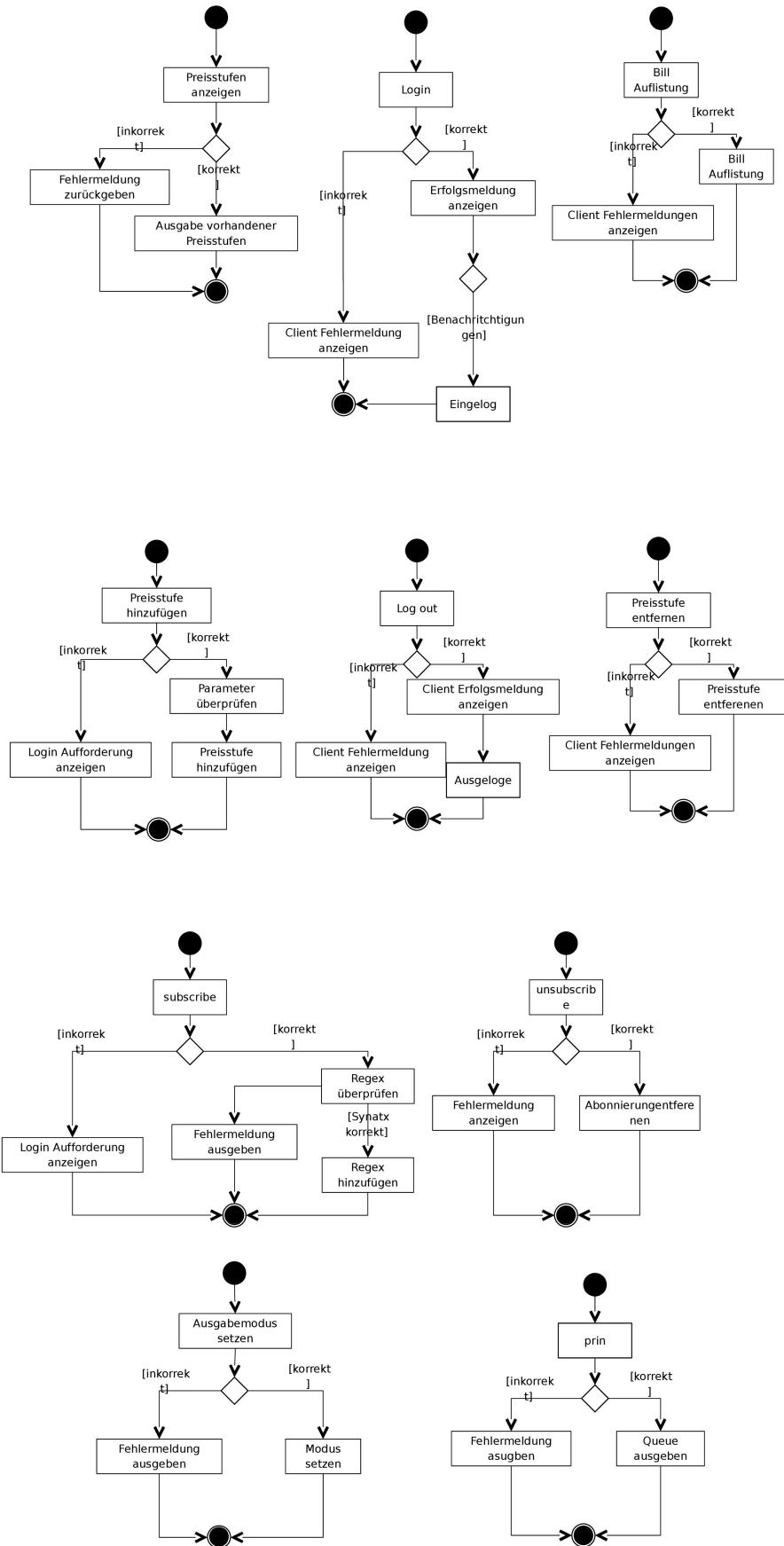


Abbildung 5: Zeigt einen Teil der neuen Funktionen aus Task08.

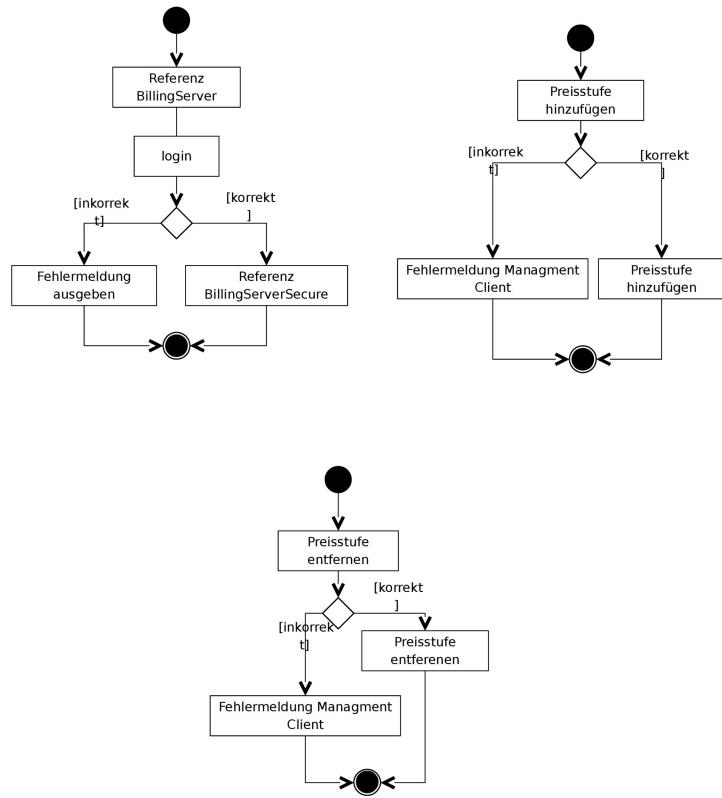


Abbildung 6: Zeigt die restlichen Funktionen aus Task08.

## UML Use Case Diagramm

Nicht vorhanden.

## Management Client

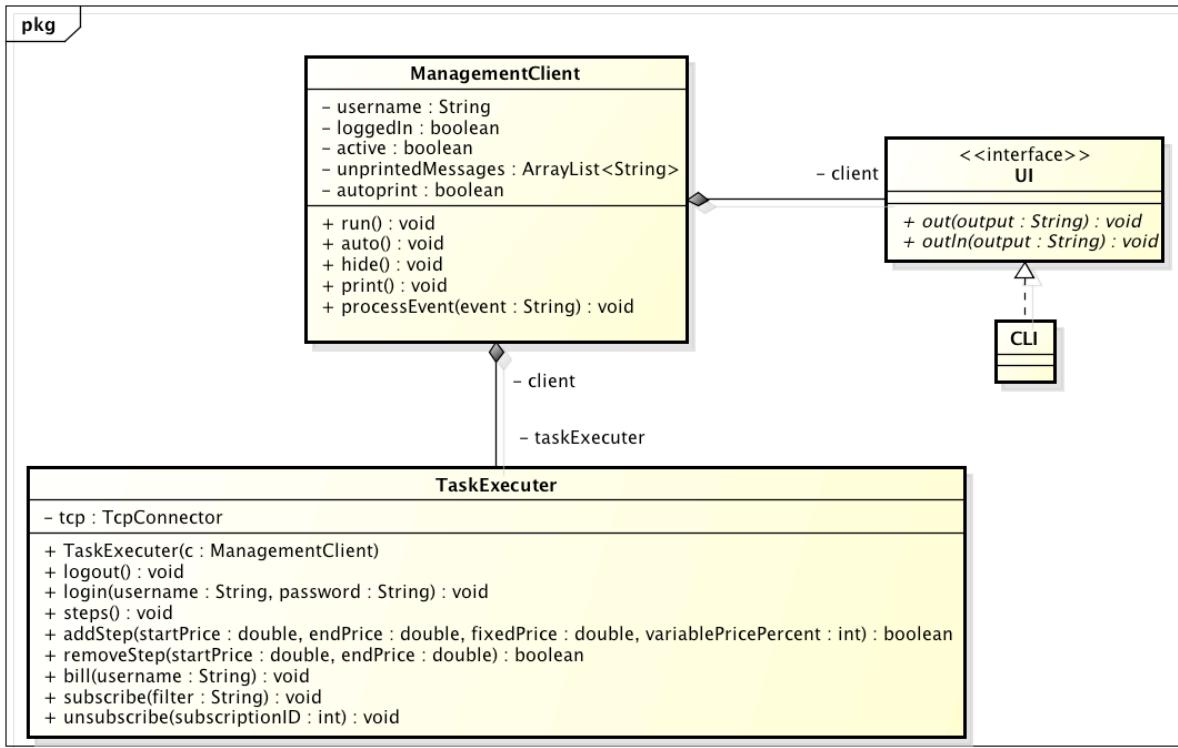


Abbildung 7.: Entwickeltes Klassenkonzept

## Implementierung

Der Management Client hat die Aufgabe, mit Billing und Analytics Server zu kommunizieren und alle nötigen Befehle zu übermitteln.

### Befehle um mit den Billing Server zu aggieren:

- !login <username> <password>: Anmelden mit spezifischen Daten
- !steps: Auflistung aller Preisstufen
- ! <startPrice> <endPrice> <fixedPrice> <variablePricePercent>: hinzufügen einer neuen Preisstufe
- !removeStep <startPrice> <endPrice>: Entfernen einer Preisstufe
- !bill <user Name>: Zeigt die Gebote des gewählten User. Die Anzeige begrenzt sich auf Auktionen die bereits abgeschlossen worden sind. Zusätzlich werden die Auktions Abgaben angegeben, welche mittels der Abgabeliste ermittelt wird.

- !logout: Setzt den Verbindungsstatus zurück. Um auf Befehle, wie ‚bill‘ zugreifen zu können, muss sich der Management Client User wieder über das BillingServiceSecure remote Object mit dem Befehl ‚login‘ Verbinden.

### Befehle um mit den Analytics Server zu aggieren:

- !subscribe <filterRegex>: Den Management ein bestimmtes Event abonnieren lassen. Mehrfache Abonnierungen möglich.

Beispiel:

```
!subscribe '(USER_.*|BID_.*'
Created subscription with ID 17 for events using filter '(USER_.*|BID_.*'
```

- !unsubscribe <subscriptionID>: Eine Abonnierung eines Events mittls ID wieder abmelden.

Befehlseingaben werden geprüft. Bei unbekannten Befehlen oder falschen Syntax wird der Client über das UI mit vordefinierten Errormessages verständigt.

### Remote Message Management

Der Management Client implemetiert die RMI Methode ‚processEvent‘. Die Methode wird von Analytic Server aufgerufen, wenn eine bestehende Event-Abonnierung (subscription) mit einem von Server generierten Event übereintrifft.

Es gibt zwei Arten die eingehenden Nachrichten anzuzeigen:

- *Automatic*
- *on-demand*

### Dadurch ergeben sich folgende Befehle, um die Moduse zu wählen bzw. zu aggiren:

- !auto: Events werden automatisch, sofort ausgegeben. (none default)
- !hide: Events werden nicht ausgegeben und im Hintergrund gespeichert.
- !print: Gespeicherte Nachrichten (durch Befehl !hide), werden ausgegeben (Reihenfolge)

## Billing Server

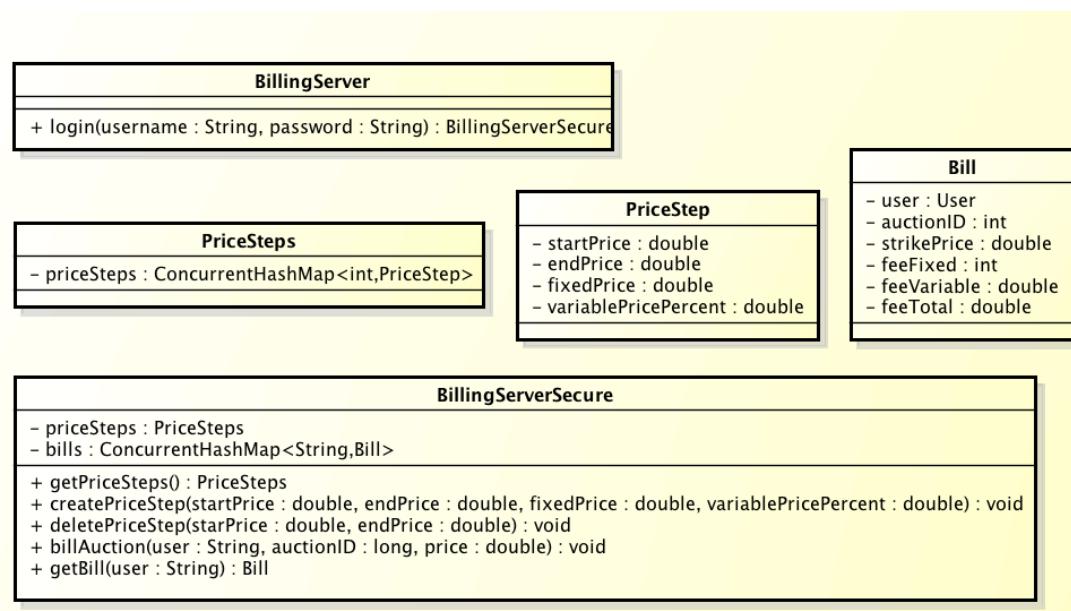


Abbildung 8: Genau veranschaulicht die für BillingServer relevanten Teile des UML's.

Der Billing Server ist für die Rechnungen des Auktion Systems verantwortlich.

Damit eine Sicherheit gegeben ist, wird es zwei RMI Objekte geben.

Das Objekt „BillingServer“ stellt uns nur eine Funktion zum login da, welche ein Management Client aufrufen kann. Falls dieser Login korrekt ist, wird die Referenz zum eigentlichen Billing Server, namens „BillingServerSecure“ weitergegeben.

Pro Auktion gibt es Gebüren. Diese Gebüren können von einem ManagementClient festgelegt werden.

Eine Gebür wird für einen bestimmten Preis-Bereich festgelegt.

Auction Price	Auction Fee (Fixed Part)	Auction Fee (Variable Part)
0	1.0	0.0 %
(0-100]	3.0	7.0 %
(100-200]	5.0	6.5 %
(200-500]	7.0	6.0 %
(500-1000]	10.0	5.5 %
> 1000	15.0	5.0 %

Abbildung 9: Auktionspreise

Auction Price: Kosten von bis

Fixed Part: Auktionskosten die fix festgelegt sind (zb 50€ -> 3€ kosten)

Variable Part: Hängt Prozentuell mit den Kosten zusammen

Je teurer die Auktion, desto weniger % zahlt man für den Variable Part.

Der BillingServer hat eine Lokale Datei namens user.properties welche die Userdaten enthält.

Das Passwort soll aus Sicherheitsgründen mit md5 gehasht sein (welches auch nicht sehr sicher ist)

Ein Eintrag sollte dann so ausschauen:

Admin=f23c5f9779a3804d586f4e73178e4ef0

Der BillingServerSecure bietet uns die Möglichkeit neue „PriceSteps“ wie oben beschrieben zu erstellen und diese zu löschen.

Desweiteren soll die Methode „billAuction“ die Relevanten informationen einer beendeten Auktion speichern, welche später für die Rechnungen verwendet werden sollen.

Eine solche Rechnung, kann dann mit der Methode „getBill(String username)“ abgerufen werden.

## Exceptions

Exceptions die hier auftreten können:

- InvalidInputException ... diese tritt auf wenn die Username Passwort Kombination falsch ist
- UserInputException ... diese tritt auf wenn kein gültiger Befehl übergeben wird.

## Ausgabe

Ausgabe der steps nach folgendem Schema:

Min_Price	Max_Price	Fee_Fixed	Fee_Variable
0	0	1.0	0.0 %
0	100	3.0	7.0%
100	200	5.0	6.5%
200	500	7.0	6.0%
500	1000	10.0	5.5%

Abbildung 10: Schema für ausgabe der Steps allerdings dann mit Ascii Zeichen gelöst.

## **login()**

Die Sicherheit wird durch den Rückgabewert bei login() erreicht.

Wenn die Username / Passwort Kombination stimmt wird die gültige Referencs auf den BillingServerSecure zurückgegeben und der ManagementClient kann arbeiten.

Andernfalls als Rückgabewert *null* und es wird eine Exception erzeugt. (InvalidInputException)

## Analytics Server

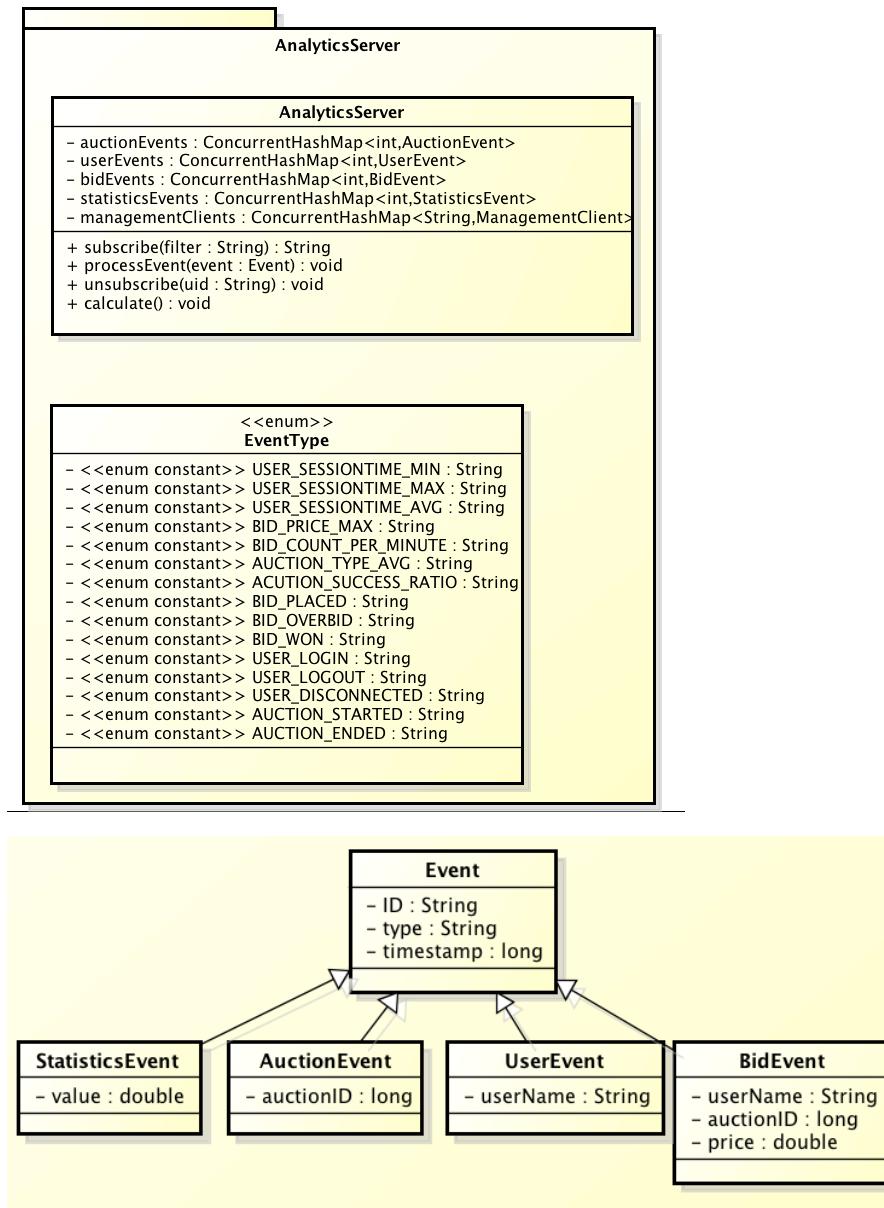


Abbildung 11: Analytics Server eventuell mit einem EnumTyp und das Model für Events

Der AnalyticsServer soll alle Aktionen die auf Auctionen passieren statistisch auswerten. Dazu werden alle Aktionen die geschehen in einem vorgefertigtem Model an den AnalyticsServer übertragen. Übertragen werden sie in sogenannten Events.

Dazu speichert er alle vom Server übertragenen Events in ConcurrentHashMap's. (Verwendung dieser siehe Designüberlegung Allgemein).

Unterschieden wird in:

- AuctionEvent: Werden bei Beginn und Ende einer Auktion erstellt  
type:
  - AUCTION\_STARTED
  - AUCTION\_ENDED
- UserEvent: Werden bei jeder User Aktivität erstellt  
type:
  - USER\_LOGIN
  - USER\_LOGOUT
  - USER\_DISCONNECTE
- BidEvent: Bei jedem bieten auf eine Auktion erstellt  
type:
  - BID\_PLACED
  - BID\_OVERBID
  - BID\_WON
- StatisticEvent: Wird auf dem AnalyticsServer erstellt aufgrund statistischer Auswertung  
type:
  - USER\_SESSIONTIME\_MIN
  - USER\_SESSIONTIME\_MAX
  - USER\_SESSIONTIME\_AVG
  - BID\_PRICE\_MAX
  - BID\_COUNT\_PER\_MINUTE
  - AUCTION\_TIME\_AVG
  - AUCTIONS\_SUCCESS\_RATO

Die einzelnen Events die am Server erstellt werden, werden an die Methode processEvent weitergeleitet diese speichert die Events in der entsprechenden Liste und ruft dementsprechend die calculate() Methode auf diese überprüft ob sich eines der statistischen Events ändert und wenn ja wird dieses neu erstellt und wieder in die Liste statisticsEvent gespeichert.

Event: Datenelement: type

Eventuell als Enum definieren da es dann nur die vorgegebenen Werte haben kann und dieses leichter eingetragen werden können. (Frage ist ob dieses als String implementiert werden muss)

### **Benachrichtigungen:**

Damit die Benachrichtigungen richtig weitergeleitet werden gibt es eine Liste mit allen ManagementClients die sich mit der subscribe Methode für Neuigkeiten angemeldet haben.

Bei dieser Map ist der Key der Filter damit leicht nachgeschaut werden kann bei wem der Filter zutrifft dieser wird dann als Unterscheidung verwendet wer die Nachrichten bekommen soll.

## Testing Component

Online-Auktionssysteme sollten in der Praxis gleichzeitig Robustheit und Skalierbarkeit bieten. Dies sollten sie auch für eine große Anzahl von Kunden gewährleistet werden.

Concurrency Probleme sind schwer zu erkennen, im Normalbetrieb, aber die Erzeugung von künstlicher Last auf dem System kann helfen, Probleme und Widersprüche zu erkennen.

Daher wird eine Lasttest-Komponente erstellt, um die Skalierbarkeit und Zuverlässigkeit des Systems zu testen.

Die folgenden Funktionen werden im Zuge dieser Lasttest-Komponente implementiert:

- Clients: Anzahl der gleichzeitig bietenden Kunden.
- auctionsPerMin: Anzahl von gestarteten Auktionen per Client per Minute.
- auctionDuration: Dauer der Auktion in Sekunden.
- updateIntervalSec: Anzahl der Sekunden die vergehen müssen bevor die Clients die Liste der aktiven Auktionen updaten.
- bidsPerMin: Anzahl der Gebote auf (zufällige) Auktionen die platziert werden, pro Client por Minute.

Um dies zu implementieren werden die Klassen CreateThread, BidThread und ListThread erstellt.

Diese werden eine Aggregation zur Klasse TestingClient haben, aus welcher die Methode setAuctionsIDs aufgerufen wird wenn neue Auktionen verfügbar sind.

Natürlich müssen diese Klassen auch Runnable implementieren.

Weiters besteht eine Aggregation zwischen TestingClient und TcpConnector.

Genaueres ist dem UML-Klassendiagramm zu entnehmen.

## Model

Nicht vorhanden.

## Connections

## Exceptions

Nicht vorhanden.

## Zeitschätzung und Tatsächliche Zeit

Tobias Schuschnig		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit
Teamkoordination	1,5	
UML Klassendiagramm	2	4
Analytics Server Designüberlegung	1	
Conections Designüberlegung	1	
Analytics Server Implementierung	2	
Conections Implementierung	1,5	
Analytics Server Fertig	3	
Conections Fertig	2	
Protokoll	1	
Anpassungen und Abstimmungen	1,5	
User Acceptance Testing	1	
<b>Summe</b>	<b>16,5</b>	<b>4</b>

Tobias Lins		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit
UML Klassendiagramm	3	6
Protokoll	1,50	
Billing Server Designüberlegung	1	
Conections Designüberlegung	1	
Billing Server Implementierung	5	
Conections Implementierung	2,50	
Billing Server Fertig	1	
Connections Fertig	1	
Anpassungen und Abstimmungen	1	
Ant	1	
User Acceptance Testing	1	
<b>Summe</b>	<b>19</b>	<b>6</b>

<b>Alexander Auradnik</b>			
Arbeitspakete		Geschätzte Zeit	Tatsächliche Zeit
UML Aktivitätsdiagramm	2	1,5	
Management Client Designüberlegung	2,5		
Management Client Implementierung	5		
Management Client Fertig	3		
Anpassungen und Abstimmungen	3		
User Acceptance Testing	2,5		
<b>Summe</b>	<b>18</b>		<b>1,5</b>

Alexander Klune		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit

Patrick Pölzlauer		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit

<b>Alexander Rieppel</b>		
Arbeitspakete	Geschätzte Zeit	Tatsächliche Zeit
Recherche Testfälle Designüberlegung	2	2
TDD (ersten Testfälle)	1	1,50
UML Use Case Diagramm	1	0
Management Client Designüberlegung	1	0
Model Designüberlegung	1	1
<b>Summe</b>	<b>6</b>	<b>4,5</b>

<b>Gesamt</b>		
Teammitglieder	Geschätzte Zeit	Tatsächliche Zeit
Tobias Schuschnig	16,5	4
Tobias Lins	19	6
Alexander Auradnik	18	1,5
Alexander Klune	0	0
Patrick Pölzelbauer	0	0
Alexander Rieppel	6	4,5
<b>Summe</b>	<b>59,5</b>	<b>16</b>

## Arbeitsaufteilung und Termine

Arbeitspakete	Schuschnig	Lins	Auradnik	Klune	Pölzlbauer	Rieppel	Termin	Erledigt (Mängel)
UML Klassendiagramm	x	x					27.01.14	x
UML Aktivitätsdiagramm			x		x		27.01.14	x (Mängel)
UML Use Case Diagramm				x		x	27.01.14	nein
TDD (ersten Testfälle)						x	27.01.14	nein
Recherche Testfälle Designüberlegung						x	27.01.14	nein
Management Client Designüberlegung			x			(x)	27.01.14	x
Billing Server Designüberlegung		x		x			27.01.14	x
Analytics Server Designüberlegung	x						27.01.14	x
Testing Component Designüberlegung				x	x		27.01.14	x (Mängel!)
Conections Designüberlegung	x	x					27.01.14	
Exceptions Designüberlegung					x		27.01.14	nein
Model Designüberlegung						x	27.01.14	nein
Zeitaufzeichnung und Zeitschätzung	x	x	x	x	x	x	27.01.14	x (Mängel: Rieppel, Klune, Poelzlbau er)
<b>Review 1 mit Protokoll</b>	<b>x</b>						<b>29.01.14</b>	
TDD Testing Fertig						x	10.02.14	
Management Client Implementierung			x			(x)	10.02.14	
Billing Server Implementierung		x		x			10.02.14	
Analytics Server Implementierung	x						10.02.14	

Testing Component Implementierung			x	x		10.02.14	
Conections Implementierung	x	x				10.02.14	
Exceptions				x		10.02.14	
Model Fertig					x	10.02.14	
<b>Review 2 mit Protokoll</b>	<b>x</b>					<b>12.02.14</b>	
Management Client Fertig			x		(x)	17.01.14	
Billing Server Fertig		x		x		17.01.14	
Analytics Server Fertig	x					17.01.14	
Testing Component Fertig			x	x		17.01.14	
Conections Fertig	x	x				17.01.14	
Exeptions Fertig				x		17.01.14	
<b>Review 3 mit Protokoll</b>	<b>x</b>					<b>19.02.14</b>	
Uni Testing					x	22.02.14	
Beschreibung Testing					x	22.02.14	
Beschreibung Applikation	x					23.02.14	
Anpassungen und Abstimmungen	x	x	x	x	x	23.02.14	
Ant		x				24.02.14	
User Acceptance Testing	x	x	x	x	x	24.02.14	
Abgabe	x	x	x	x	x	25.02.14	
<b>Finanl Review mit Protokoll</b>	<b>x</b>					<b>26.02.14</b>	

## Abnahme der Arbeitspakete:

### UML Klassendiagramm

Von Tobias Schuschnig und Tobias Lins durchgeführt keine Mängel festgestellt.

Vor dem Termin abgegeben.

### UML Aktivitätsdiagramm

Durchgeführt von Alexander Auradnik und Patrick Poelzbauer. Vorhanden mit Mängel. Schreibfehler. Nicht als .jpg sonder als .png. Des Weiteren ohne großem weißem Rand auf der Seite. Bitte nachbessern. (Bitte an Abbildung 4 richten, hier sieht man das gewünschte Format aus Task08)

### UML Use Case Diagramm

Durchgeführt von Alexander Klune und Alexander Rieppel. Nicht vorhanden.

### TDD (ersten Testfälle)

Durchgeführt von Alexander Rieppel. Die kreierten Testfälle entsprechen leider nicht der Übungsvorgabe. Es ist nicht Sinn der Sache selbstverständliche Testfälle wie getter() und setter() Methoden zu testen. Des Weiteren ist es nicht zweckmäßig TDD Testfälle für ein Package Model zu schreiben dessen Code bereits existiert und zu einer Coverage von 72.63(!) getestet wurde. Bitte überleg dir nocheinmal was du zu testen hast.

### Recherche Testfälle Designüberlegung

Durchgeführt von Alexander Rieppel. Nicht vorhanden.

## Management Client Designüberlegung

Durchgeführt von Alexander Auradnik. Gut überlegt und auf den Punkt gebracht das gesamte Arbeitspaket erledigt.

## AnalyticsServer Desingüberlegung

Durchgeführt von Tobias Schuschnig. Es wurden keine Mängel festgestellt.

## BillingServer Desingüberlegung

Durchgeführt von Tobias Lins. Kleine Mängel festgestellt allerdings schon ausgebessert.

## Testing Component Designüberlegung

Durchgeführt von Patrick Poelzbauer. Vorhanden allerdings fehlen wesentliche Teile des Arbeitspaketes. Der für diesen Teil relevante Teil des UML's sollte hier noch einmal angeführt werden. Des Weiteren fehlen Probleme die hier auftreten können. (Genauere Beschreibungen sind nicht dem UML zu entnehmen sondern sollen hier beschrieben werden). Bitte nachbessern. (Mögliche Vorlagen: AnalyticsServer Designüberlegung, ManagementClient Designüberlegung)

## Connections Designüberlegung

Durchgeführt von Tobias Lins und Tobias Schuschnig.

## Exceptions Designüberlegung

Durchgeführt von Patrick Poelzbauer. Nicht vorhanden.

## **Model Designüberlegung**

Durchgeführt von Alexander Rieppel. Leider Falsch es ist nicht gefragt das Model aus Task07 zu beschreiben sonder das aktuelle. Das Model besteht auch aus weit mehr als nur den Messages. Bitte nachbessern.

## **Zeitaufzeichnung und Zeitschätzung**

Es ist ein Zumutung ein so leichtes Arbeitspaket ungenügend abzuschliessen. Die Formatierung dieser Datei ist nicht meine Aufgabe und wäre für jeden einzelnen nicht weiter schwer gewesen. Weiters ist es auch keine Herausforderung auch in die Gesamttabelle bei seinem Namen die Gesamtzeit einzutragen.

Des Weiteren nicht erledigt von: Alexander Klune, Patrick Poelzbauer

Alexander Rieppel: Reihenfolge beachten, Summe

## Programmbeschreibung

(Laufend das Programm beschrieben.)

## Testing

(Am Ende was am Programm wie getestet wurde und wie.)

## Probleme

### Probleme 29.1.14

- Mehr Zeit als erwartet für das UML Klassendiagramm benötigt (Schuschnig/Lins)
- Auch nach dreimaliger Aufforderung falsche Gitnamen vorhanden. (Patrick Poelzlbauer)
- Formatierung der Zeitschätzungstabelle ist wünschenswert und nicht meine Aufgabe.
- Patrick Poelzlbauer ich währe dir dankbar wenn du einen ordentliches Programm verwendest das beim öffnen der .xlsx Dateien nicht die Formatierung zerstört oder bitte dich diese nach dem öffnen nicht zu speichern ich bin nicht gewillt diese wieder zu formatieren.
- Keine Fortschritt bei TDD Testfällen diese sind nicht zureichend. (Alexander Rieppel)
- Model Designüberlegung falsch. (Alexander Rieppel)
- Keine Zeitschaetzung (Alexander Klune; Patrick Poelzlbauer)
- Falsche Zeitschaetzung (Alexander Rieppel)
- Kein Use Case Diagramm (Alexander Klune; Patrick Poelzlbauer)
- Falsche Arbeitspakete Alexander Rieppel: TDD (ersten Testfälle); Recherche Testfälle Designüberlegung; Model Designüberlegung
- Fehlende Arbeitspakete Patrick Poelzlbauer: Exceptions Designüberlegung

## **Zeitaufzeichnung**

Der Übersichtlichkeitshalber der angehängten Datei Zeitaufzeichnung.xlsx zu entnehmen