Task 08

RMI Auctionsystem

Krepela, Lipovits, Reichmann, Tattyrek, Traxler

29.01.2014

Inhaltsverzeichnis

General Remarks 3

Overview 3

Analytics Server 3

Billing Server 5

Management Client 7

Load Testing Component 10

Implementation Details and Hints 11

Hints & Tricky Parts 13

Further Reading Suggestions 14

Submission Guide 14

AnalyticsServer 15

Exceptions 16

ManagementClient 17

LoadTester 18

Billing Server 18

InitRMI 20

init(); 20

Lookup 20

Unexport 20

Rebind 20

Use Case 22

Klassendiagramme 33

Analytics 33

Events 33

Billing 34

Billing Model 34

Managemenclient 35

Testing Component 35

Client 36

Message-Model 36

Server 37

FileHandler 37

Model Server + FileHander 38

InitRmi 38

Zeitschätzungen und Arbeitsaufteilung 39
UnitTests 40
EventHandlerTest 40
Setup 40
TestUserLoginEvent 40
testUserLoginLogoutSessionEvent 40
testAuctionStartedEndedNoBid 40
testBidOnAuctionChangesRatio 40
testBidCountPerMinute 40
ManagementClientTest 41
LoadTest 42
BillingServer 42
Unit test of package „Client“ 43
CLITest 44
TaskExecuterTest 45
SystemTest 45
Absprachen 46

Spezifikationen

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.

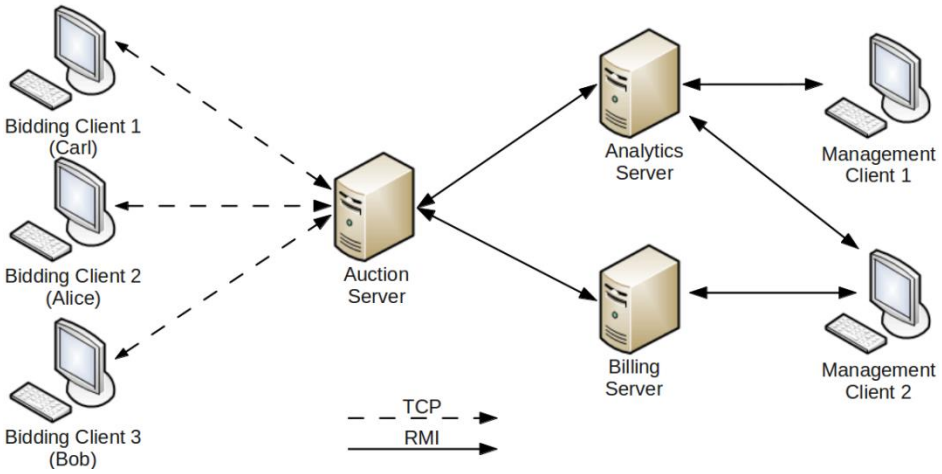


Figure 1: System Components and Interactions.

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.

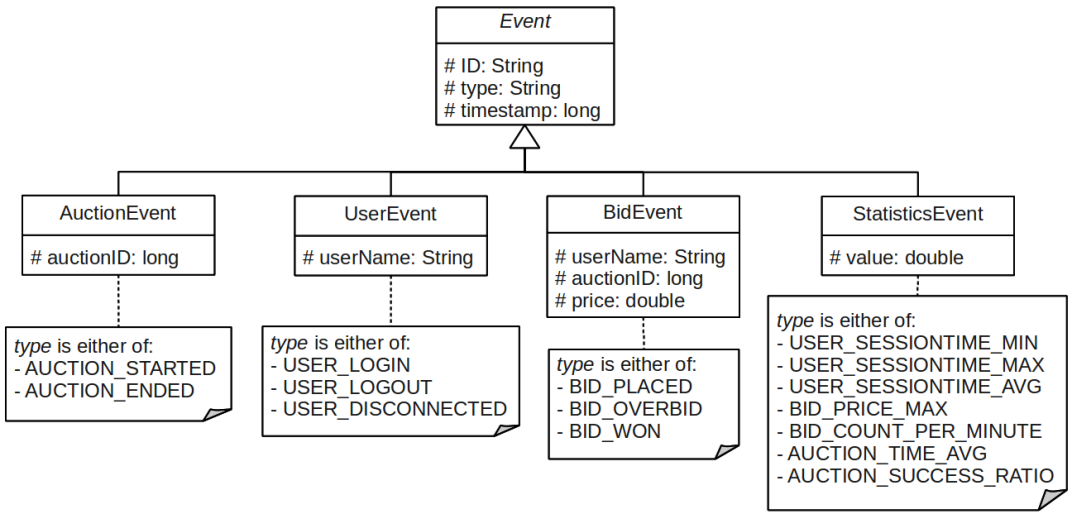


Figure 2: Event Hierarchy and Basic Event Properties.

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 %

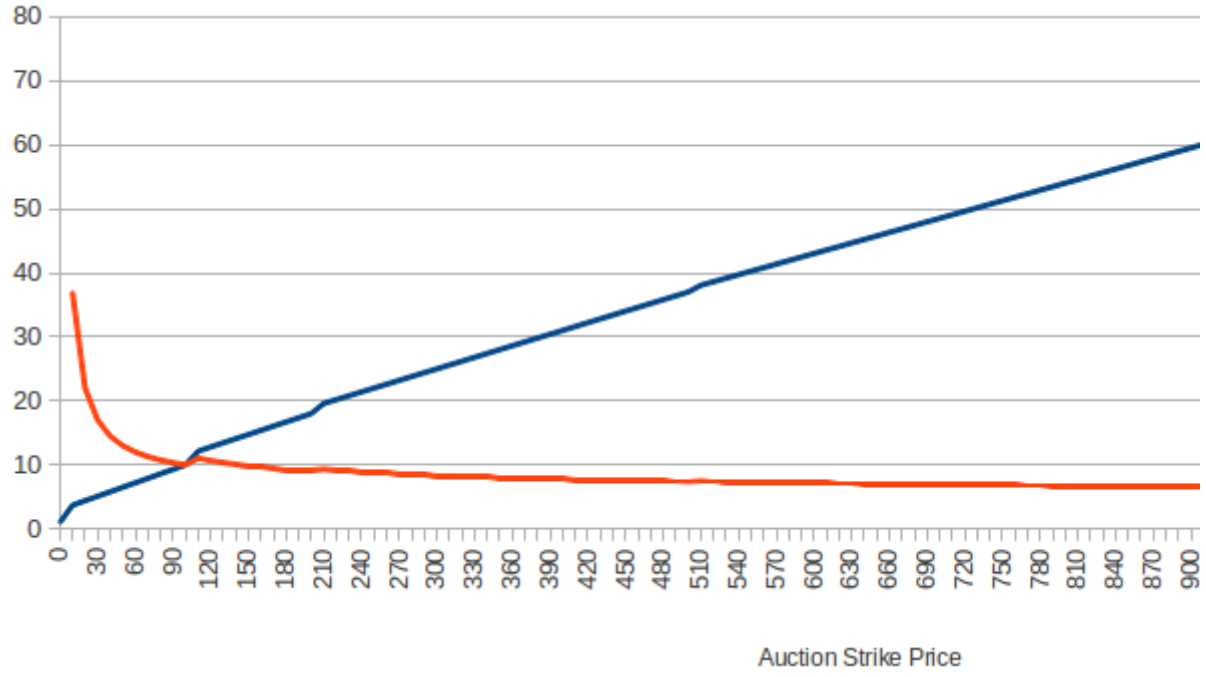


Figure 3: Pricing Curve

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 MD5 hash 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)~~ (**updated 27.01.2014**) 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> <variablePricePercent>`: 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 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
session time is 62 seconds
USER_SESSION_TIME_MAX: 31.10.2012 20:03:11 CET - average
session time is 324 seconds
USER_SESSION_TIME_AVG: 31.10.2012 20:03:11 CET - maximum
session 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 files `registry.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 by 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.UnmarshalException`s.
- 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 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(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

Designüberlegung

AnalyticsServer

Der AnalyticsServer bekommt Events vom Auction-Server und verarbeitet diese, wenn notwendig, zu weiteren Events. ManagementClients können sich mittels einer „Subscribe“-Methode beim Analyse-Server für Benachrichtigungen für bestimmte Events anmelden. Dies geschieht mit sogenanntent Regex, diese Überprüfen, ob sie mit einem Event-Typen übereinstimmen, wenn ja, bekommt der Client Benachrichtigungen darüber. Diese Benachrichtigungen werden mittels RMI Call-Back gelöst, das heißt, dass der Client beim Anmelden beim Analyse-Server sein eigenes Remote-Objekt mit der Methode „notify“ übergibt. Wird ein entsprechendes Event getriggert, so werden alle Clients für dieses Event benachrichtigt.

Intern werden die Clients in einer Map gespeichert. Diese speichert für jedes Event eine weitere Map, welche jeweils Subscription-ID und Clientinterface speichert.

Alle Events die vom Auktions-Server an den Analyse-Server geschickt werden, werden in einer ConcurrentQueue gespeichert. Die Events werden anschließend von einem Thread „EventHandler“ bearbeitet. Dieser wartet mit der Methode „queue.take()“ solange, bis in der Liste ein Event steht. Kommt ein Event rein, so wird überprüft ob es zu einem der 3 Typen gehört:

- AuctionEvent
- UserEvent
- BidEvent

Für jedes dieser Event-Arten gibt es unterschiedliche Behandlungsmöglichkeiten:

AuctionEvent:

Wenn es vom Typ Auction_Started ist, so wird das Event in einer Liste gespeichert, damit bei einem eintreffenden Event (Auction_Ended) die Dauer berechnet werden kann.

Ist es vom Typ Auction_Ended, so wird aus der Liste das Start-Event gesucht und die Dauer berechnet. Des Weiteren wird aus einer Liste mit gebotenen Auktionen gesucht, ob auf diese Auktion geboten wurde, wenn ja, dann wird ein erfolgreiches Event hinzugefügt. Danach werden folgende, weitere Events getriggert:

- AuctionSuccessRatio
- AuctionTimeAVG

UserEvent:

Ist es vom Typ User_LOGIN, dann wird das Event in einer Liste gespeichert.

Kommt ein Event USER_LOGOUT oder USER_DISCONNECTED, so wird das entsprechende USER_LOGIN Event gesucht, und folgende Events berechnet:

- USER_TIME_MIN
- USER_TIME_MAX

- USER_TIME_AVG

BidEvent:

Hier werden nur die Bid_Placed Events behandelt. Wenn ein Bid platziert wurde, so wird die Auktion ID der Liste der gebotenen Auktionen hinzugefügt, um später die erfolgreichen Auktionen zu berechnen.

Das Ursprungsevent und die berechneten Events werden in einer Queue gespeichert, die von der notify-Methode abgearbeitet wird. Diese Methode prüft auch, dass ein User nur ein Event einmal bekommt. Dies geschieht, indem man alle User die das Event schon haben in ein Set speichert. Ist der User bereits vorhanden, so bekommt er das Event nicht nochmal.

User nicht mehr vorhanden:

Wenn ein User beim notify nichtmehr vorhanden ist, so wirft die Methode eine RemoteException. Diese wird gefangen. Wenn diese Exception gefangen wird, dann wird der User aus allen Subscriptions entfernt.

Subscription-ID:

Eindeutige ID, die einer Anmeldung eines Client zugehörig ist. Mit dieser ID kann sich der Client wieder von Benachrichtigungen anmelden. Diese ist ein Integer der für jeden User hochgezählt wird.

BidCountPerMinute

Um die Bids pro Minute zu berehnen wird ein extra Timer implementiert. Dieser greift auf die Variable „bidcount“ vom EventHandler zu und berechnet durch die bereits abgelaufene Zeit (die Anzahl der Durchläufe wird als Integer gespeichert) und erstellt das entsprechende BidCountPerMinute Event.

Exceptions

AuctionStartedButDidntStartException

Exception wird geworfen, wenn eine Auktion beendet wird, die nicht gestartet wurde. Dies kann der Fall sein, wenn eine gespeicherte Auktion aus einem File gelesen wird. In diesem Fall wird die Auktion für die Statistik ignoriert.

InvalidUserLogout

Wird geworfen, wenn sich ein User ausloggt, der sich in dieser Session am AnalyticsServer nicht eingeloggt hat. Wird für diese AnalyticsSession ignoriert.

ManagementClient

Der ManagementClient baut RMI Connections zum AnalyticsServer und zum BillingServer auf. Wenn dies passiert ist können User diesen ManagementClient mithilfe folgender Kommandos bedienen:

- `!login <username> <password>`

  Loggt einen User mit username und Passwort am BillingServer ein. Nun ist das ausführen von nachfolgenden Secure-Kommandos möglich.

Secure-Kommandos an den BillingServer:

- `!steps`

  Listet alle existierenden „Price Steps“.

- `!addStep <startPrice> <endPrice> <fixedPrice> <variablePricePercent>`

  Fügt einen “Price Step” mit angegebenen Parametern hinzu.

- `removeStep <startPrice> <endPrice>`

  Löscht einen bestehenden „Price Step“

- `!bill <userName>`

  Zeigt die Rechnungen des angegebenen Users an

- `!logout`

  `Loggt den user wieder aus`

Kommandos an den AnalyticsServer:

- `!subscribe <filterRegex>`

  Meldet den User füe Events mit einem bestimmten Filter an.

- `!unsubscribe <subscriptionID>`

  Meldet den User von einer Anmeldung mit der angegebenen ID ab.

- `!auto`

  Durch diesen Befehl werden alle erhaltenen Events automatisch ausgegeben.

- `!print`

  Durch diesen Befehl werden alle erhaltenen Events gespeichert, jedoch nicht ausgegeben.

- `!hide`

  Durch diesen Befehl werden alle gespeicherten Events ausgegeben.

Die Commands wurden durch das Command-Pattern umgesetzt, darauf ergibt sich ein Interface Command, eine abstrakte Klasse SecureCommand, welche Command implementiert, eine

CommandFactory in welcher die jeweiligenCommands, abhängig vom UserInput, erstellt werden sowie alle benötigten Commands. (AddStep, RemoveStep, Steps, Logout,

Exceptions

CommandNotFoundException()
→ Thrown if a Command does not exist

IllegalNumberOfArgumentsException()
→ Thrown, if the userinput conists of a wrong number of arguments for the command

WrongInputEception()
→ Thrown, if the command exists and has the right number of arguments, but one or more arguments are of a wrong type. e.g. '!removeSteps 1 miau'

LoadTester

Der LoadtestingComponent dient dazu, die Robustheit und Skalierbarkeit des AuktionsSystems zu testen.

Es gibt hierzu ein loadtest.properties File, welches die Test-Parameter festlegt.

Bsp:

```
# TODO: adjust these values
clients = 100
auctionsPerMin = 1
auctionDuration = 2*60
updateIntervalSec: 20
bidsPerMin = 2
```

Die Properties Klasse bietet die Funktionalität, ein solches File auszulesen und ebendiese Parameter zu speichern.

Die LoadTest Klasse liest nun diese Parameter aus, und erstellt die gewünschte Menge an Clients.

Von jedem dieser Clients werden mithilfe des TaskExecutors die betreffenden Befhle in den gewünschten Intervallen ausgeführt.

FileHandler

The FileHandler is used for persistent saving of synchronized Objects. It has the ability to save and load either single objects or a whole ConcurrentHashMap.

Reading an Object

```
public Object readObject(K key) throws IOException,
```

```
                    CannotCastToMapException {}
```

Reads the saved ConcurrentHashMap out of the file and returns the Object with the specified key.

Throws IOException if any Input/Output operation fails (file not found, etc.) and CannotCastToMapException if, for whatever reason, the saved Map can't be casted.

Writing an Object

```
public boolean writeObject(K key, T value) throws IOException,
                    CannotCastToMapException {}
```

Saves the the passed Object with the specified Key to a new or existing Map to a file.

Throws IOException if any Input/Output operation fails (file not found, etc.) and CannotCastToMapException if, for whatever reason, the saved Map can't be casted.

Reading a Map

```
public ConcurrentHashMap<K, T> readAll() throws IOException,
                    CannotCastToMapException {}
```

Reads the whole saved ConcurrentHashMap from file and returns it.

Throws IOException if any Input/Output operation fails (file not found, etc.) and CannotCastToMapException if, for whatever reason, the saved Map can't be casted.

Writing a Map

```
public boolean writeMap(ConcurrentHashMap<K, T> map) throws IOException {}
```

Writes a whole ConcurrentHashMap to a file.

Throws IOException if any Input/Output operation fails (file not found, etc.)

Billing Server

Saving price steps:

```
private ConcurrentSkipListMap<CompositeKey,PriceStep> priceSteps;
```

CompositeKey has 2 attributes (pk's): start and end price. It also provides methods to compare these keys. The function overlaps() tests if they collide with each other.

Price Step contains all 4 attributes. The shutdown method saves data persistent to a file.

Saving bills:

```
private ConcurrentHashMap<String,Bill> bills;
```

If the list already contains the user (key), the bill will be added to the Bill Object. The Bill class contains a synchronized LinkedList<BillingLine>, which holds all BillingLines. The shutdown method saves data persistent to a file

Exceptions

IllegalValueException ()
→One or more arguments are invalid!
e.g. :
values below zero
endprice must be bigger than startprice

PriceStepIntervalOverlapException()
→The provided price interval overlaps with an existing price step
(delete the other price step first)!

InitRMI

InitRMI is a class that is written to encapsulate almost all uses of RMI in this Project. It has centralised Exception handling and uses an Properties object. Not Multithreadingsave.

init();

Initialises everything that's needed for further Actions and is in correlation with RMI. If any other Method (except constructors) is called before this Method, this Method will be called.

Lookup

Looks up and a returns the wanted Remote Object identified by the passed Argument.

Unexport

Unexports an Remote Object, passed as Argument.

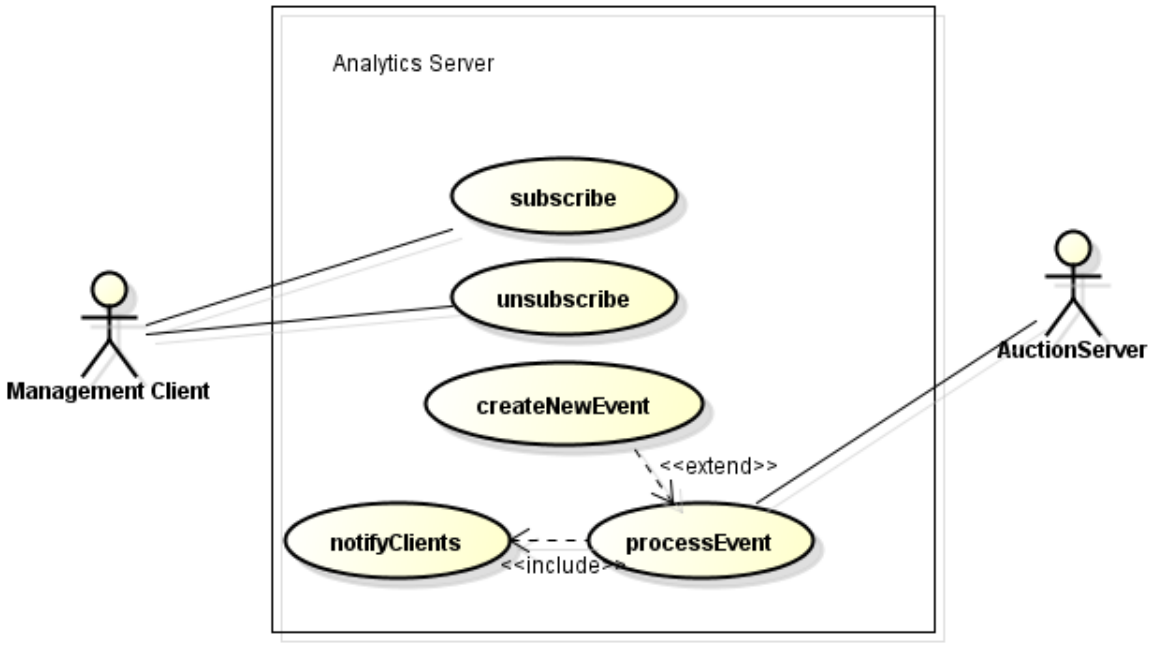
Rebind

Rebinds an Remote object to a specific identifier-string, both passed as Arguments

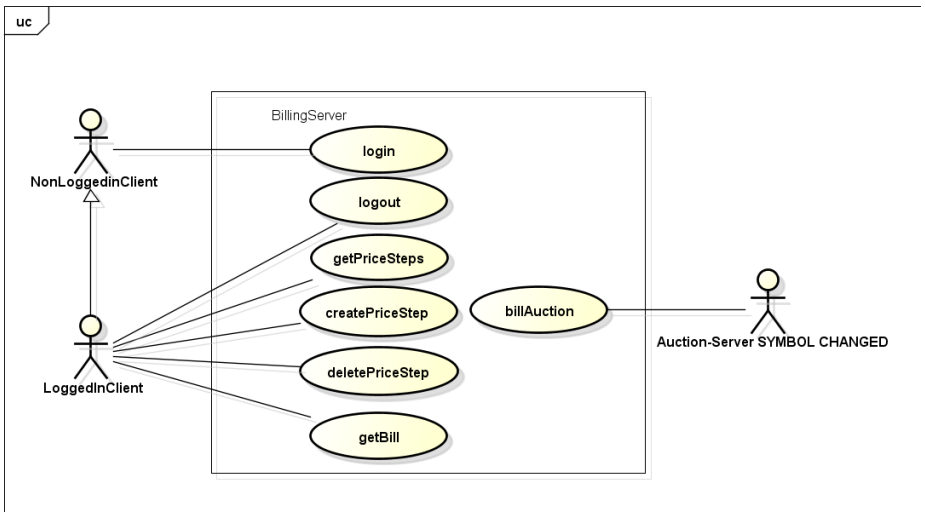
Furthermore there is a initRmi Method in all three Server and the Management-client which calls the Methods from the InitRMI-Classobject as needed for the specific case. This can be done with additional Arguments (Remote Objects or Properties) and, where possible, additionally with default values.

Use Case

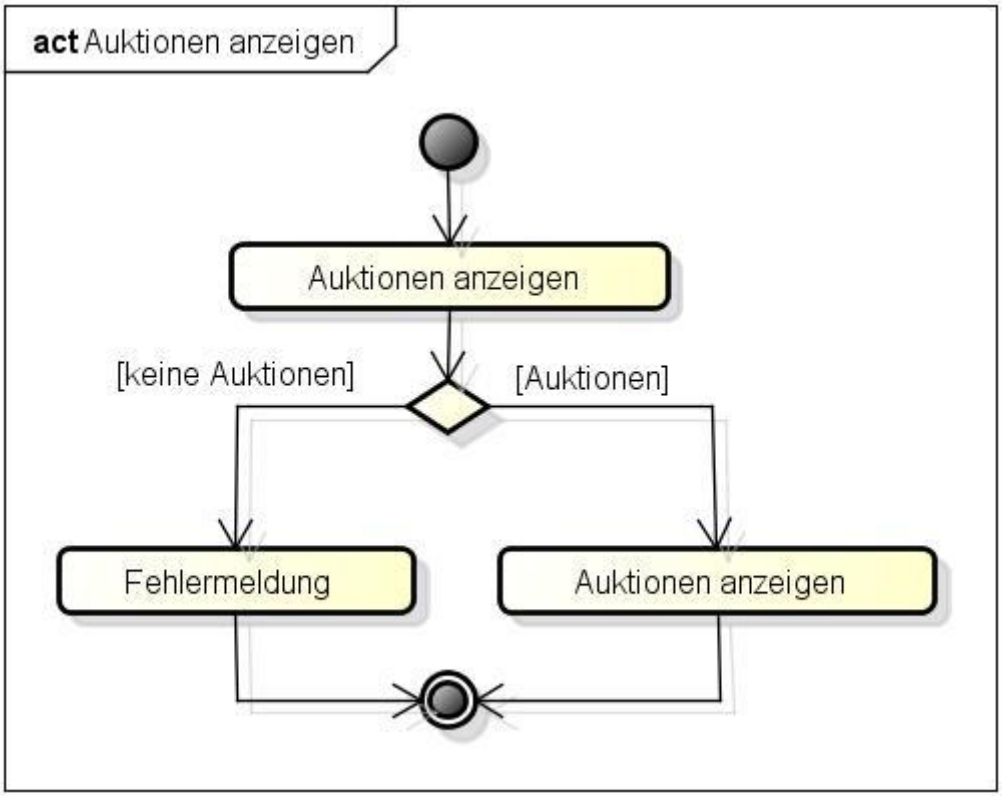
Analytic Server:

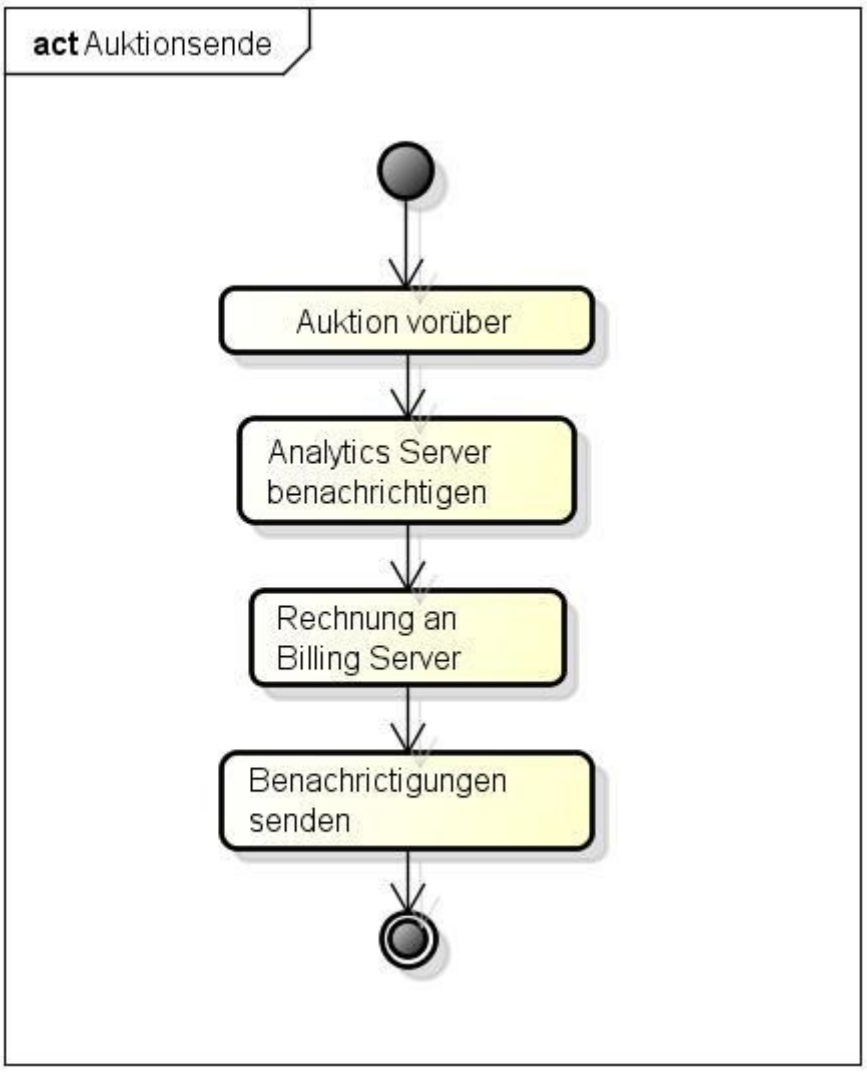


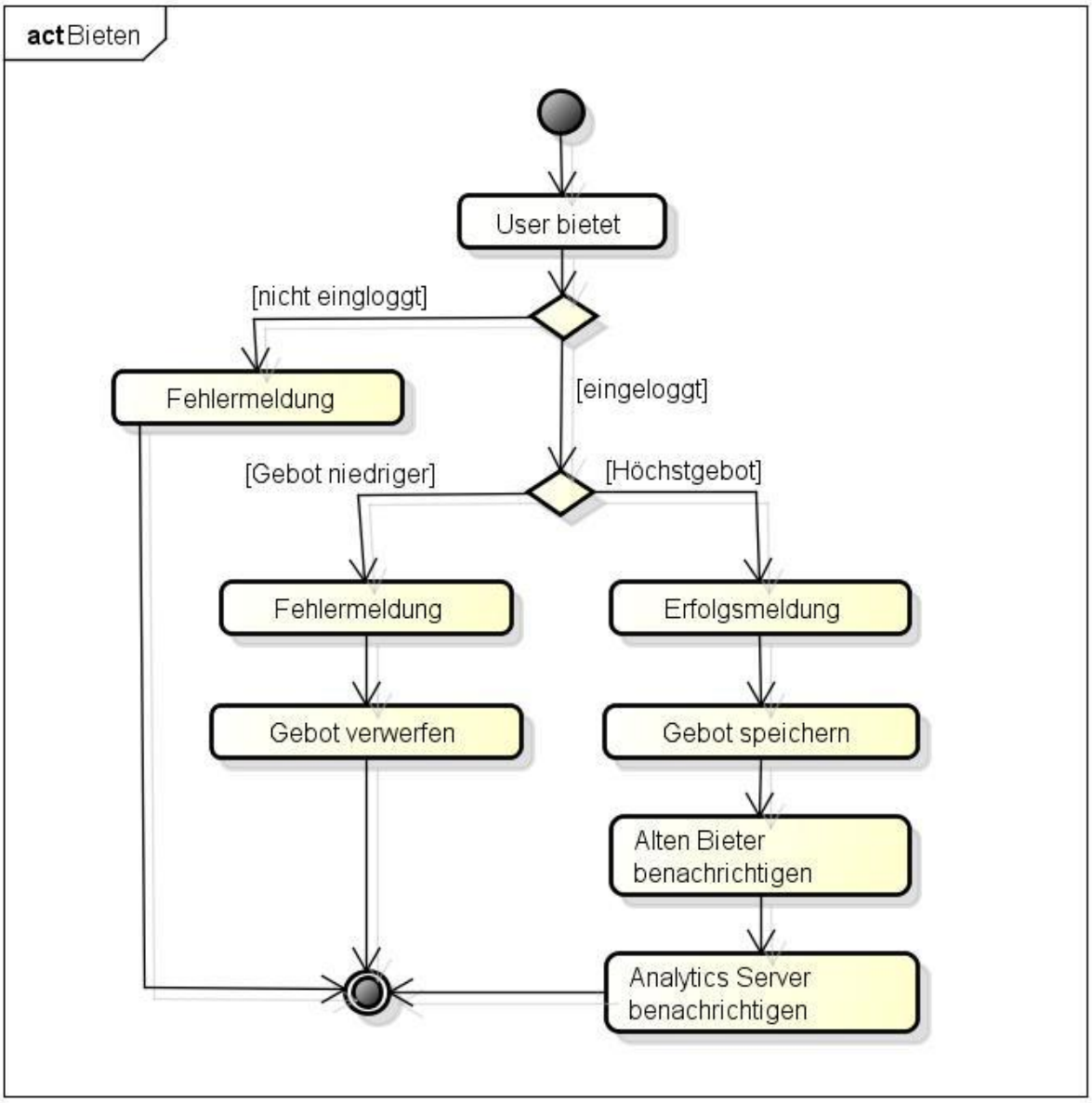
BillingServer:

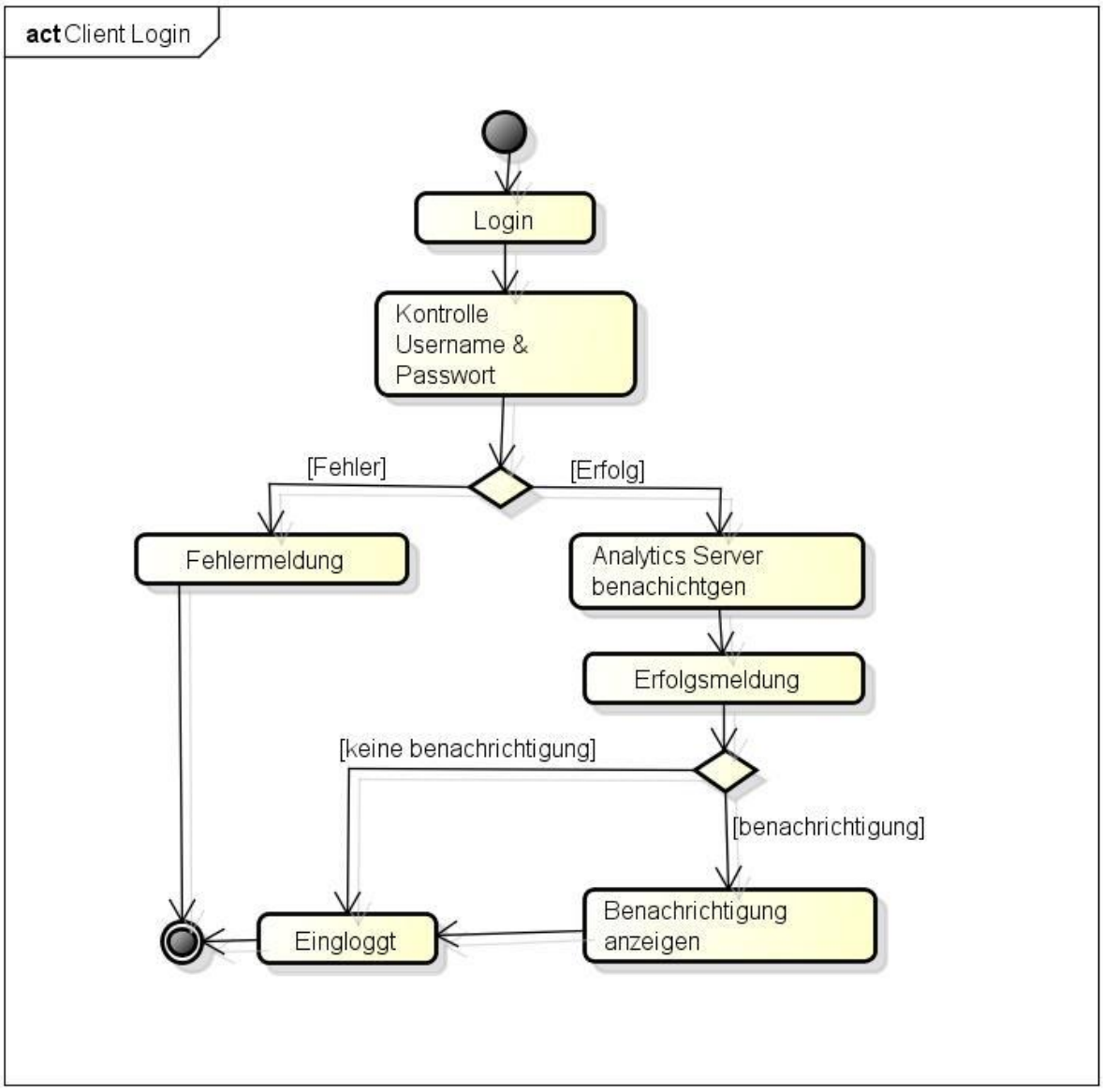


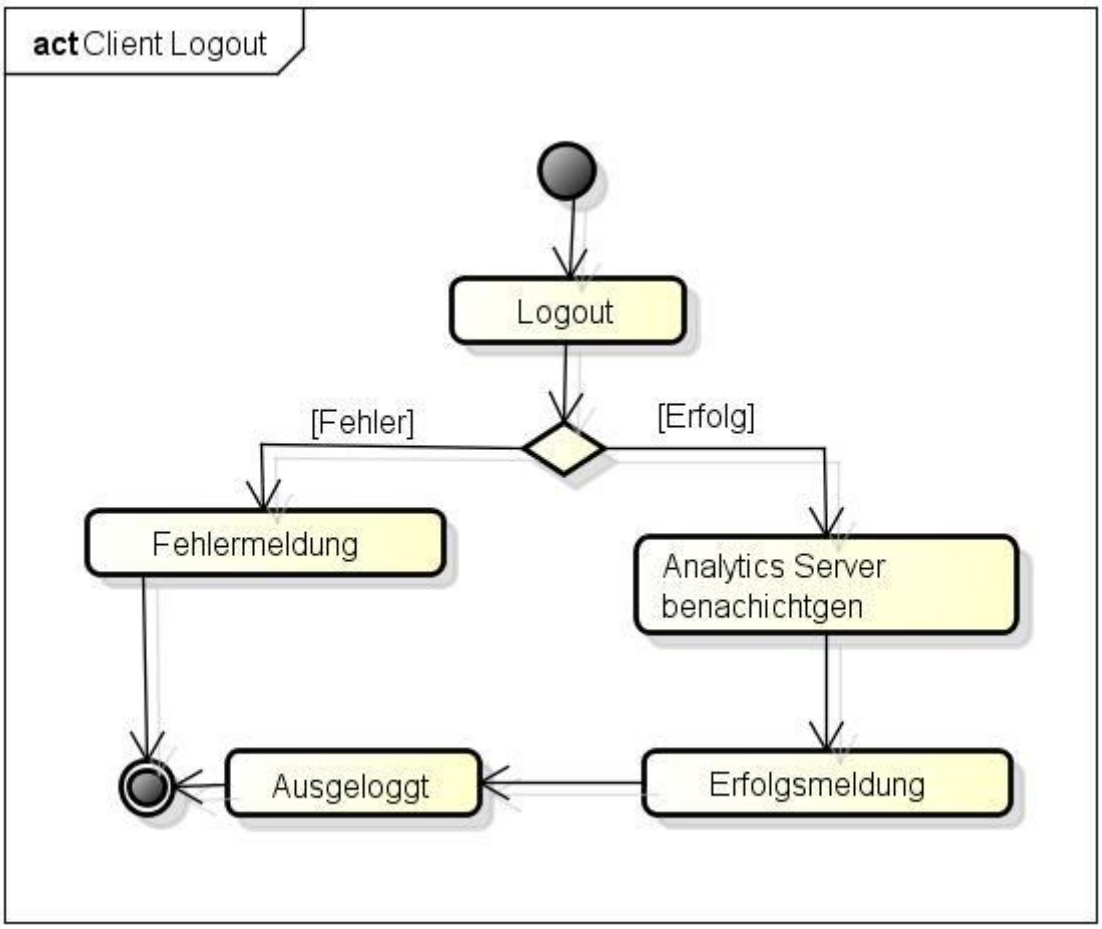
Aktivitätsdiagramme

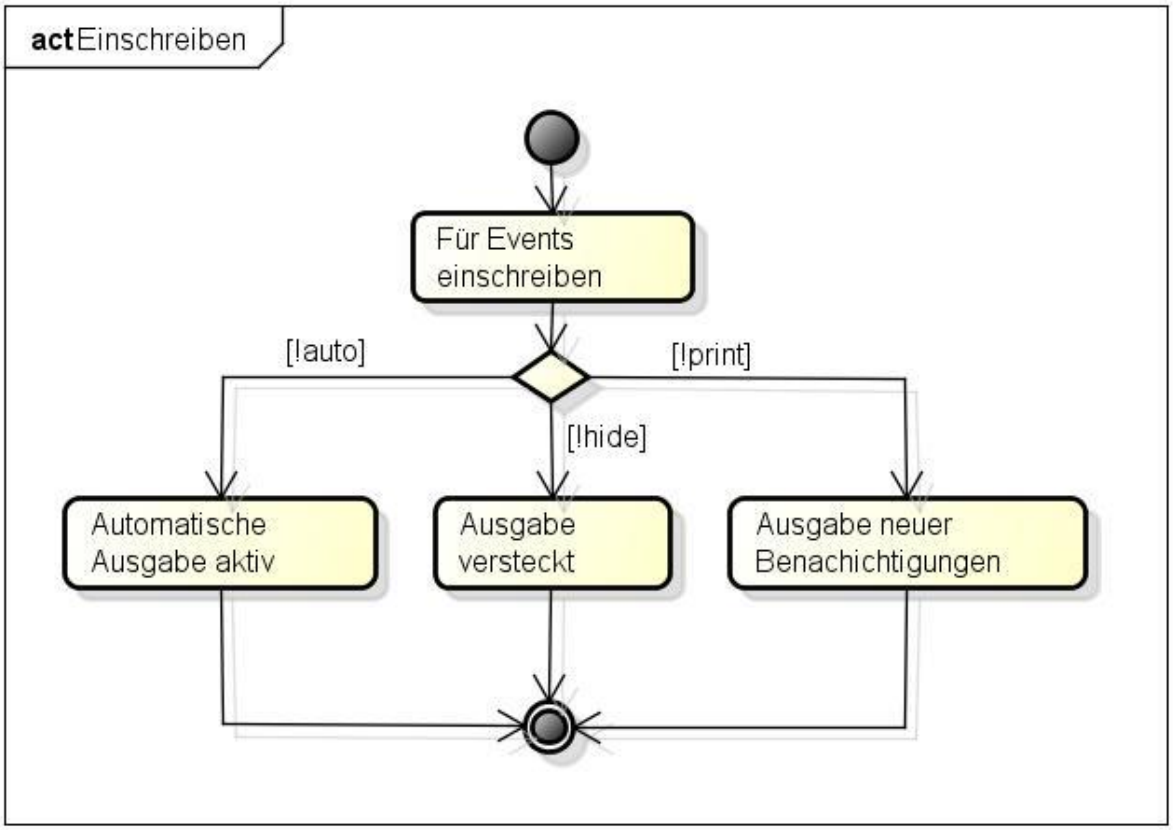


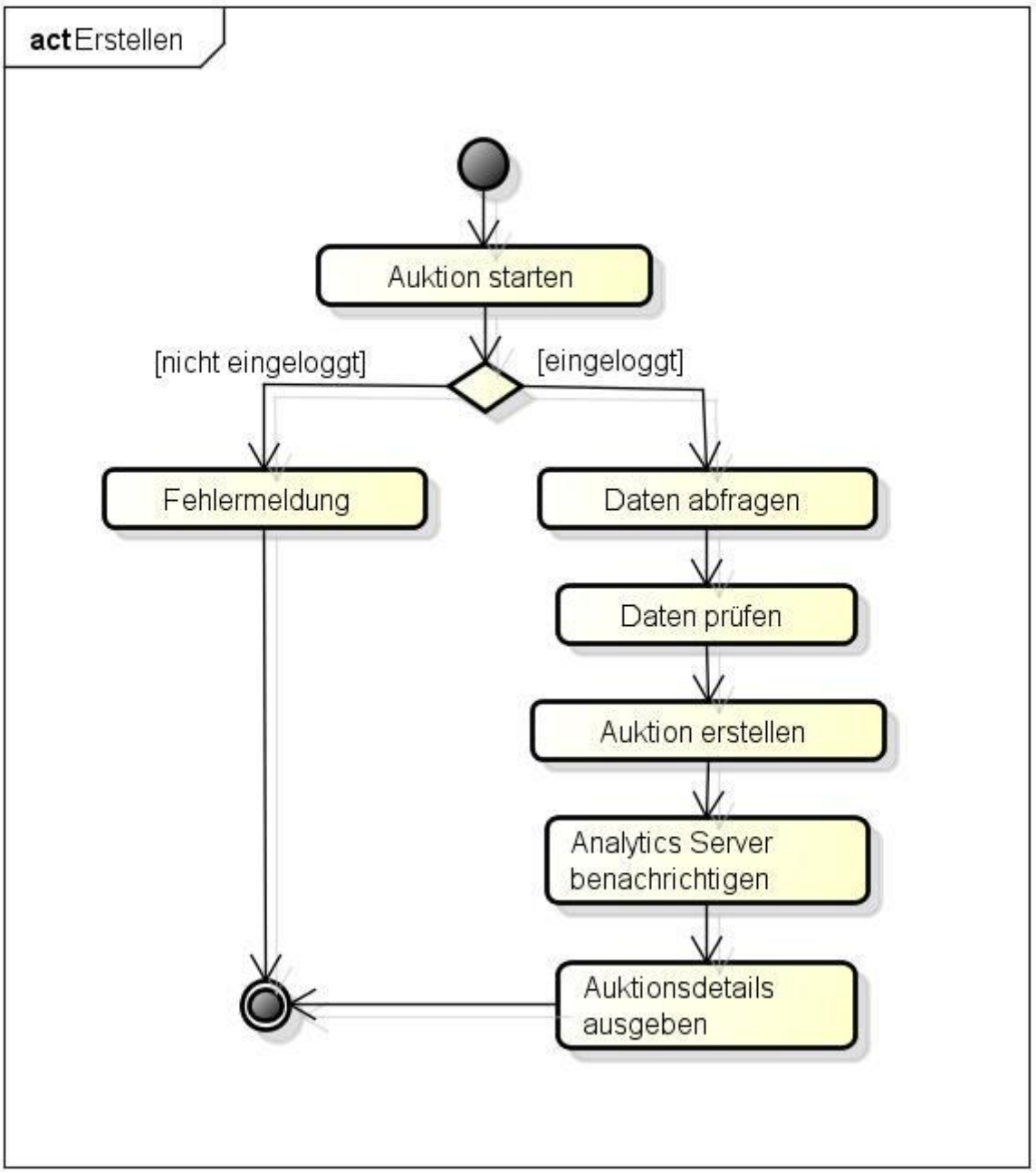

act Auktionsende
Auktion vorüber
Analytics Server benachrichtigen
Rechnung an Billing Server
Benachrictigungen senden

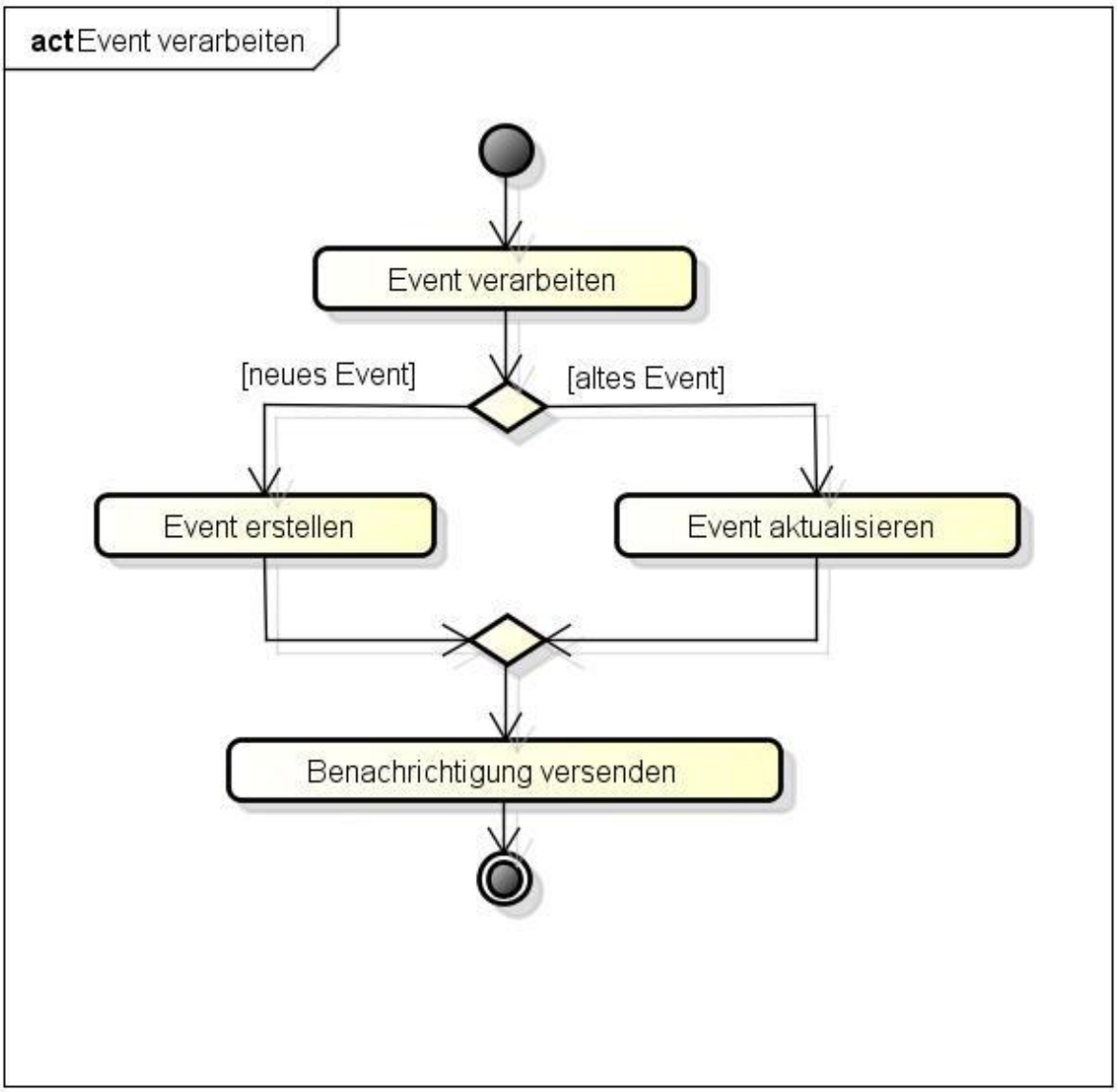

actBieten
User bietet
[nicht eingloggt]
Fehlermeldung
[eingeloggt]
[Gebot niedriger]
[Höchstgebot]
Fehlermeldung
Erfolgsmeldung
Gebot verwerfen
Gebot speichern
Alten Bieter benachrichtigen
Analytics Server benachrichtigen

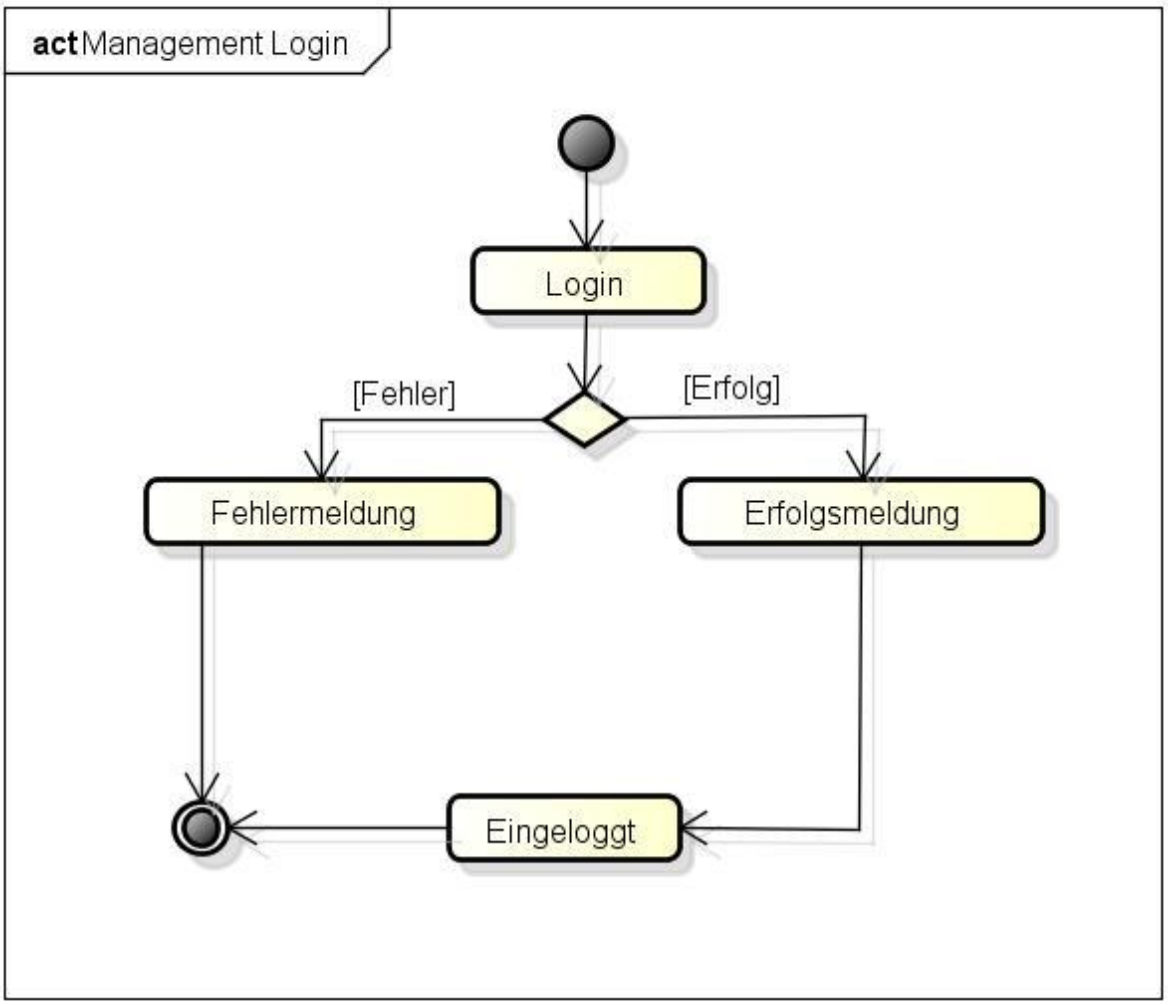

act Client Login
Login
Kontrolle
Username &
Passwort
[Fehler]
[Erfolg]
Fehlermeldung
Analytics Server
benachichtgen
Erfolgsmeldung
[keine benachrichtigung]
[benachrichtigung]
Benachrichtigung
anzeigen
Eingloggt

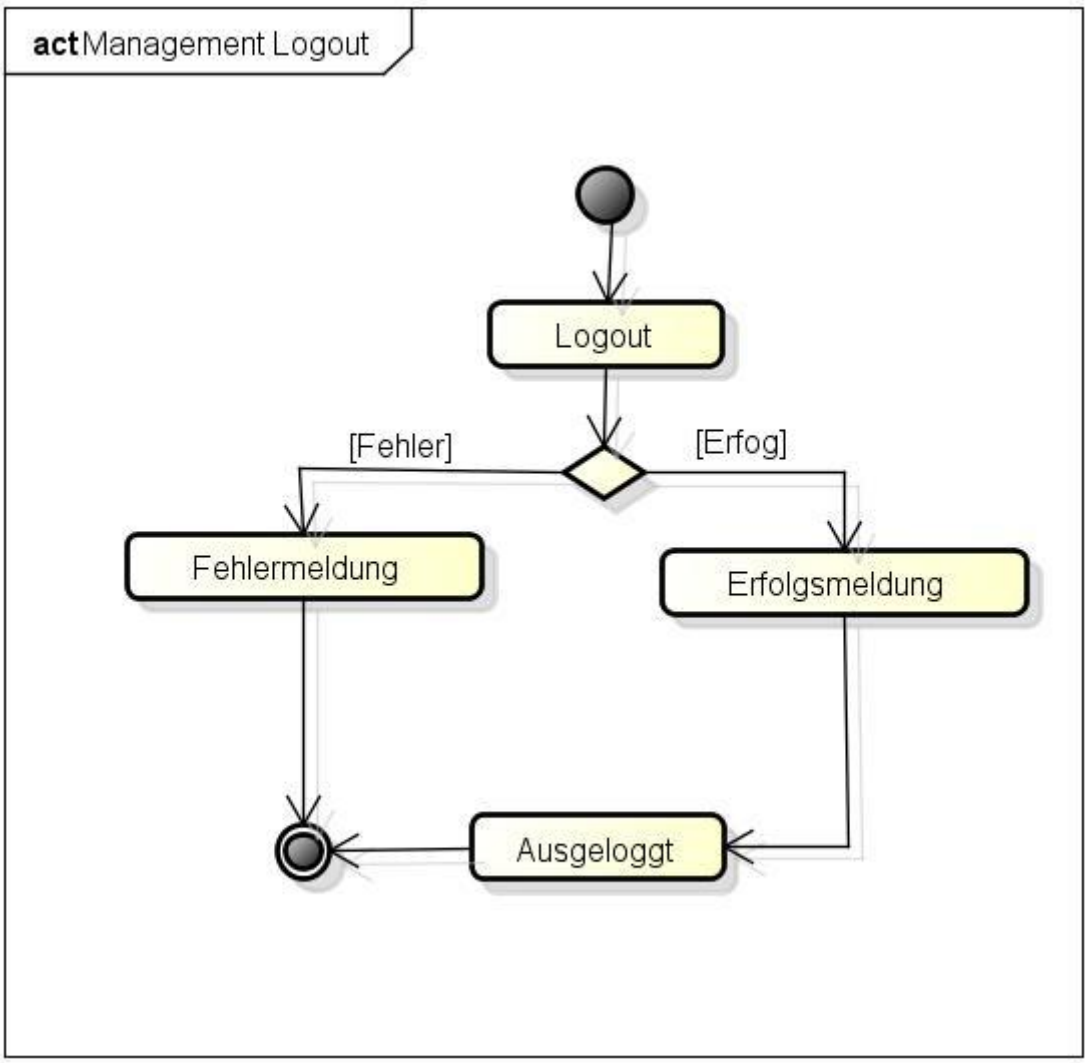

act Client Logout
Logout
[Fehler]
[Erfolg]
Fehlermeldung
Analytics Server benachichtgen
Ausgeloggt
Erfolgsmeldung

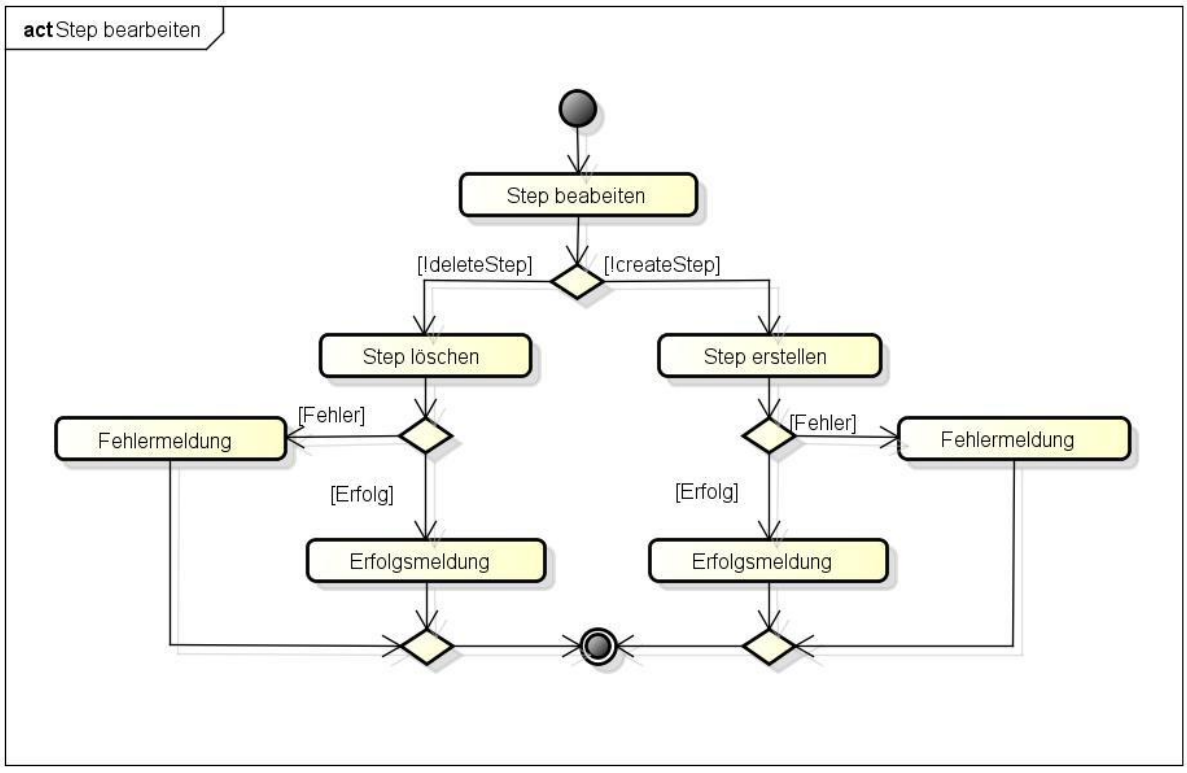

act Einschreiben
Für Events einschreiben
[!auto]
[!print]
[!hide]
Automatische Ausgabe aktiv
Ausgabe versteckt
Ausgabe neuer Benachichtigungen

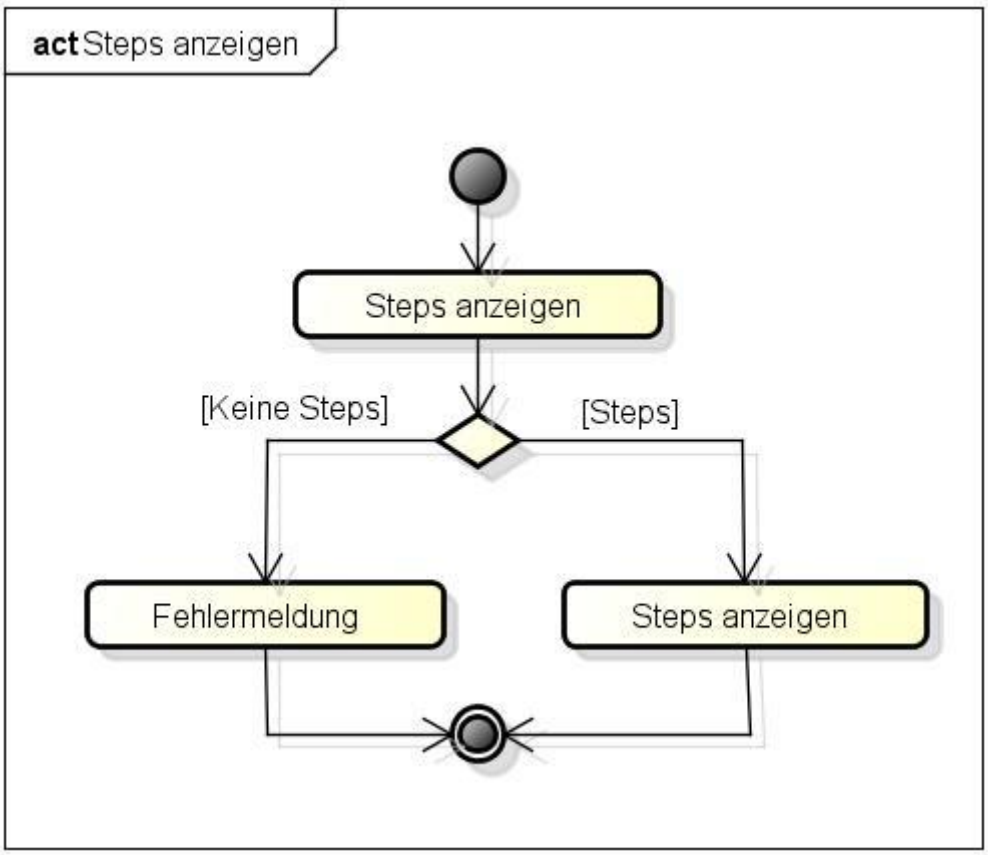

actErstellen
Auktion starten
[nicht eingeloggt]
[eingeloggt]
Fehlermeldung
Daten abfragen
Daten prüfen
Auktion erstellen
Analytics Server benachrichtigen
Auktionsdetails ausgeben


act Event verarbeiten
Event verarbeiten
[neues Event]
[altes Event]
Event erstellen
Event aktualisieren
Benachrichtigung versenden


act Management Login
Login
[Fehler]
[Erfolg]
Fehlermeldung
Erfolgsmeldung
Eingeloggt

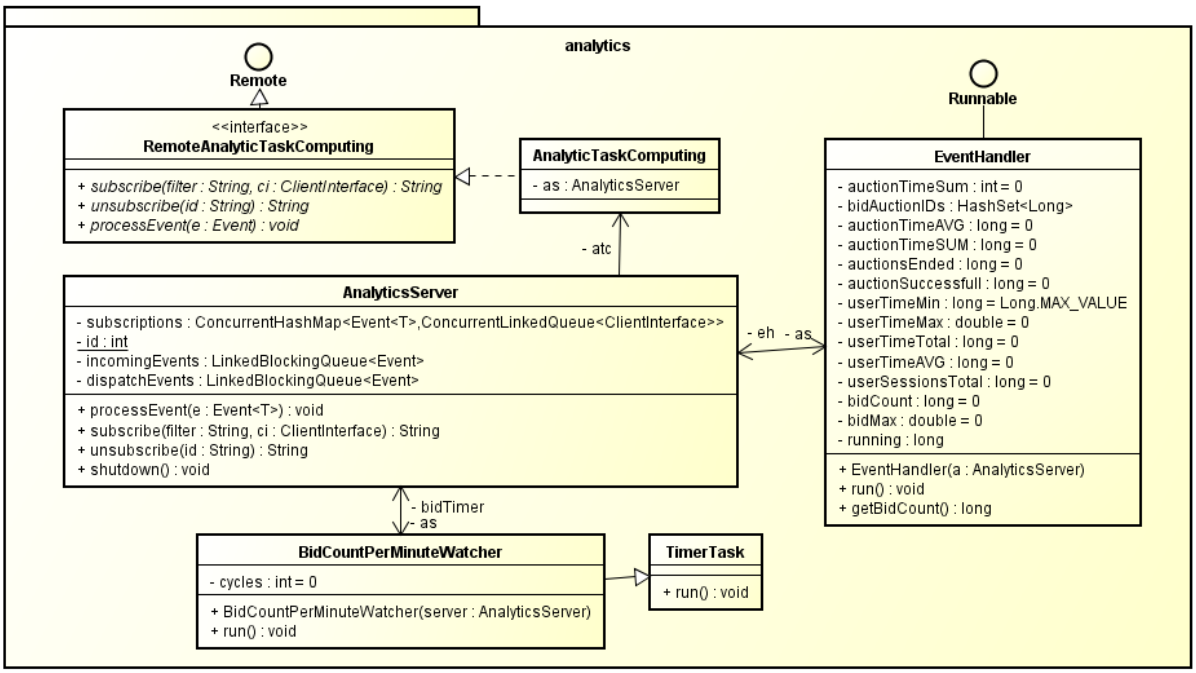

act Management Logout
Logout
[Fehler]
[Erfog]
Fehlermeldung
Erfolgsmeldung
Ausgeloggt


act Step bearbeiten
Step beabeiten
[!deleteStep]
[!createStep]
Step löschen
Step erstellen
[Fehler]
[Fehler]
Fehlermeldung
Fehlermeldung
[Erfolg]
[Erfolg]
Erfolgsmeldung
Erfolgsmeldung

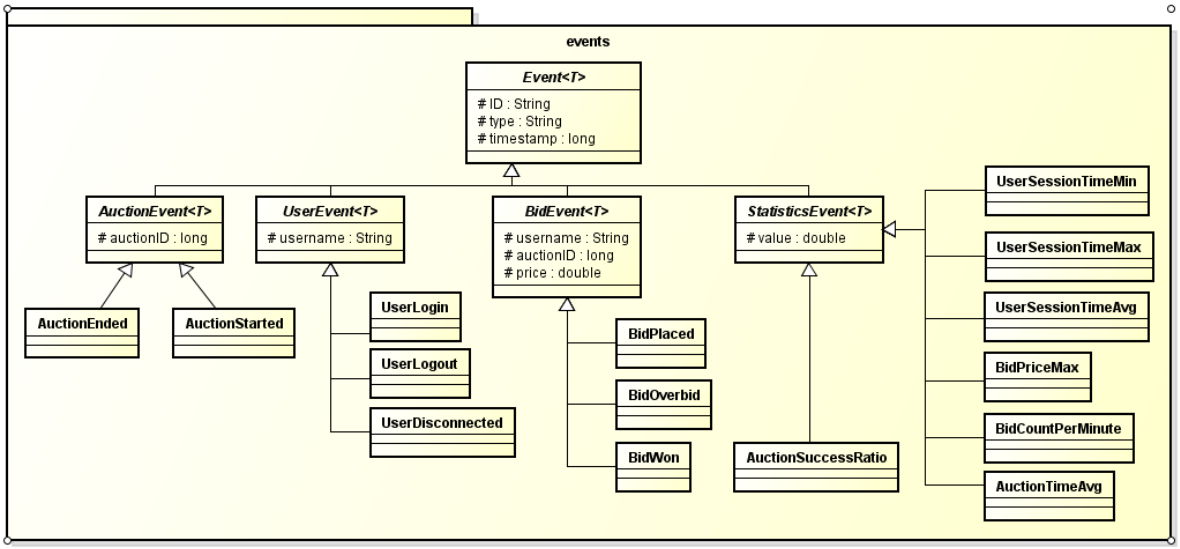

act Steps anzeigen
Steps anzeigen
[Keine Steps]
[Steps]
Fehlermeldung
Steps anzeigen

Klassendiagramme

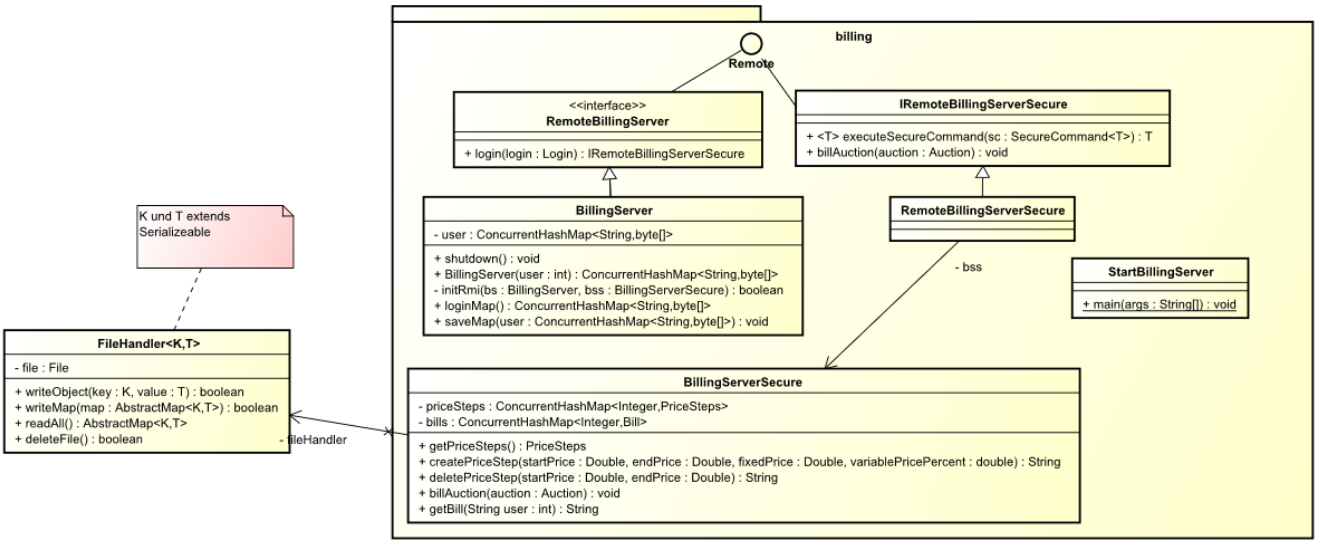
Analytics



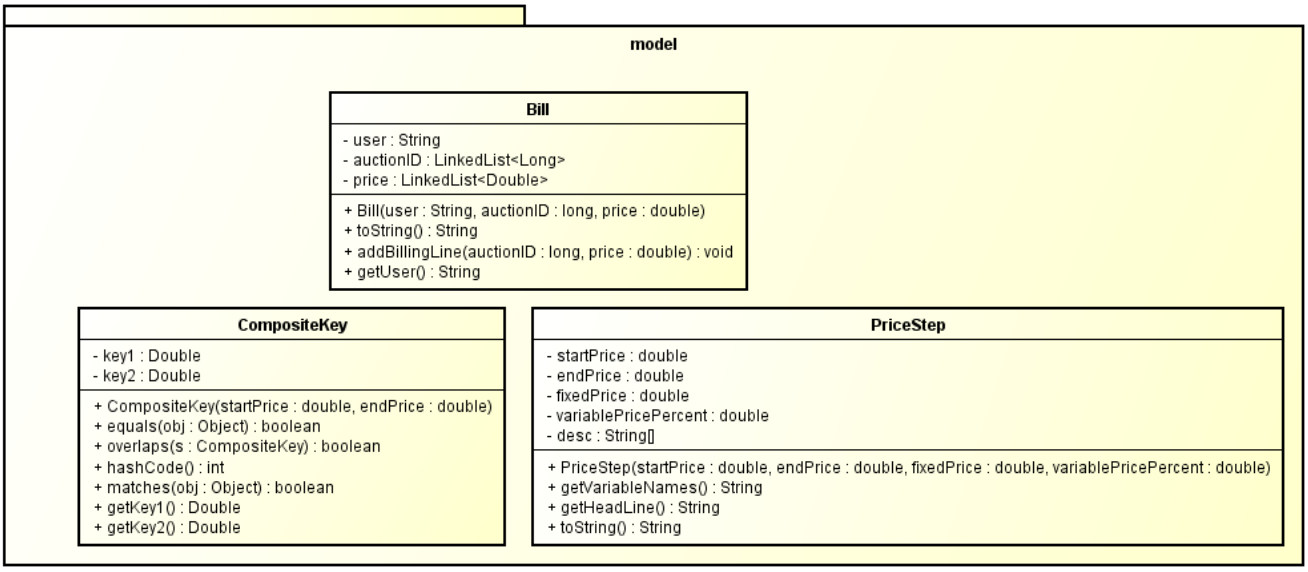
Events



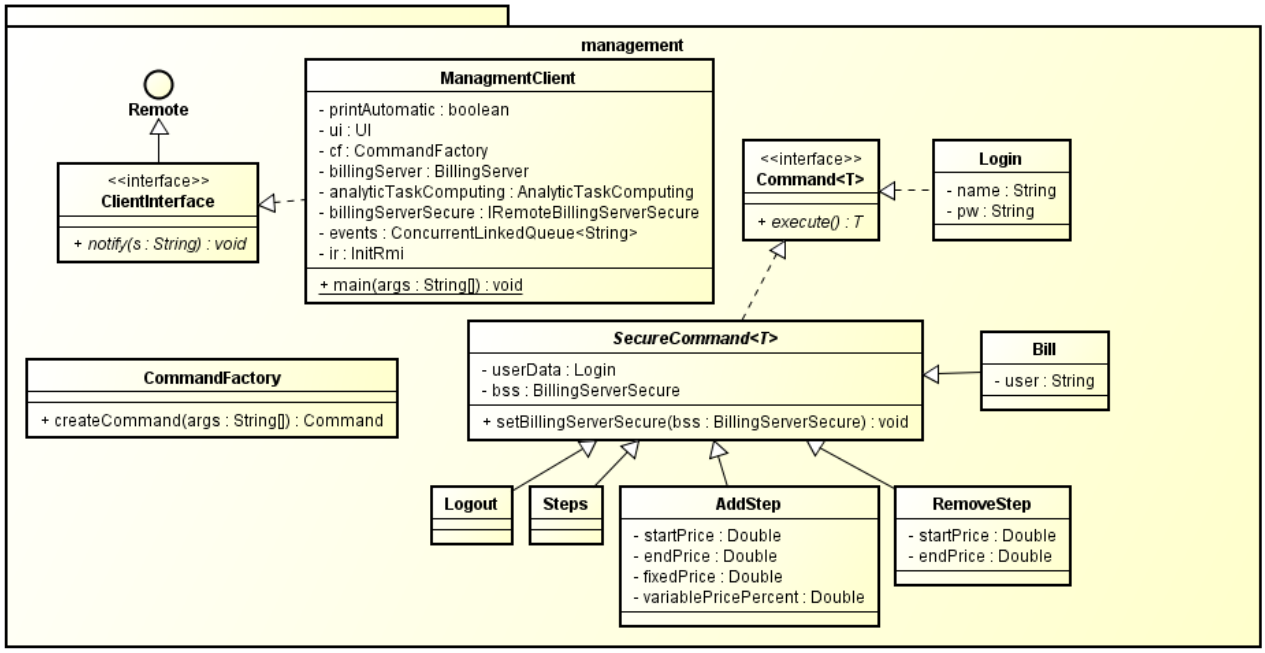
Billing



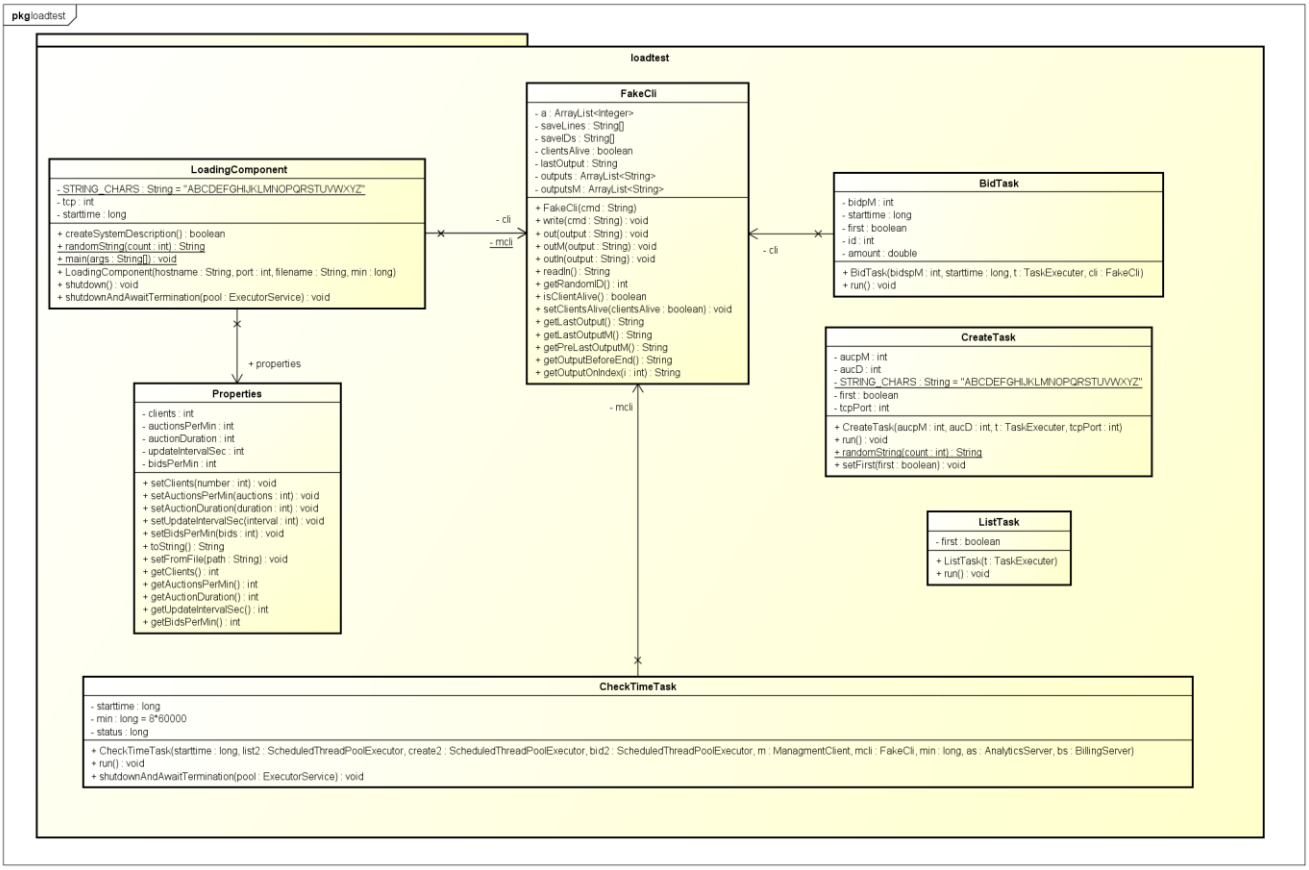
Billing Model



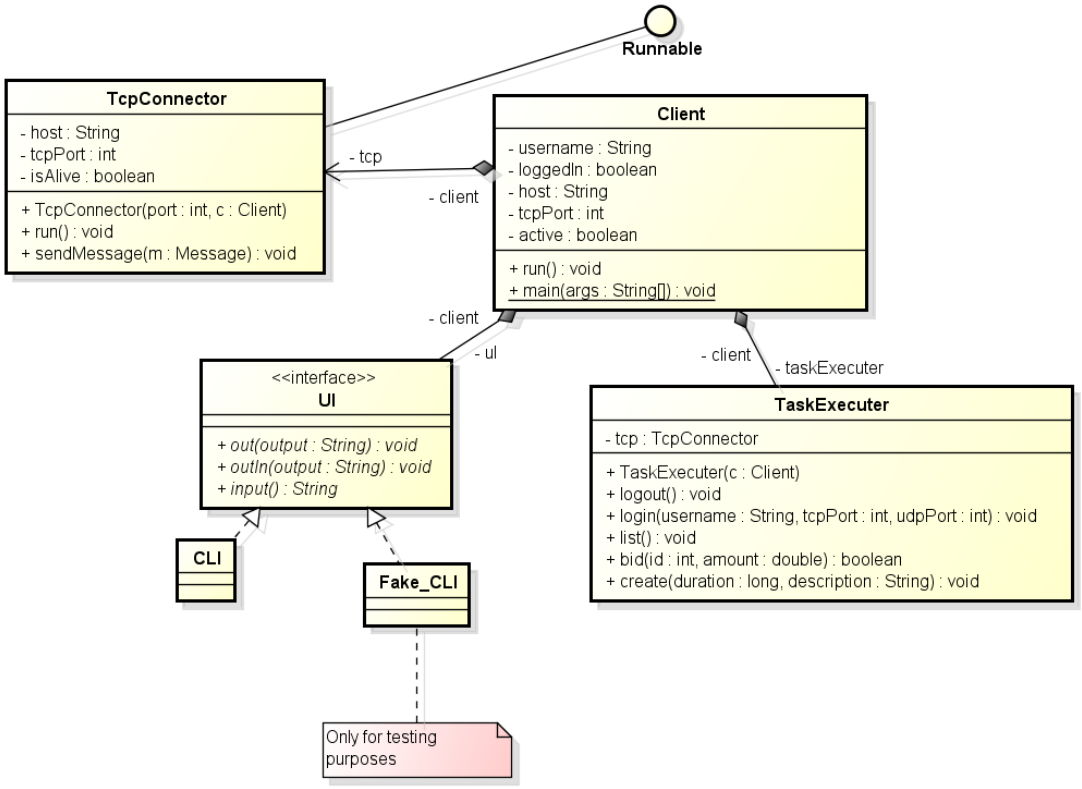
Managemenclient



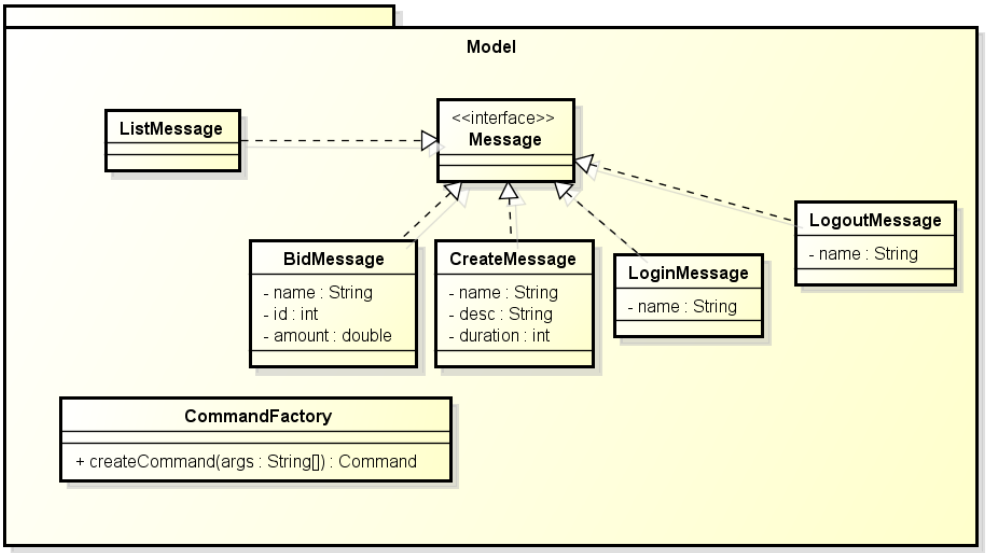
Testing Component



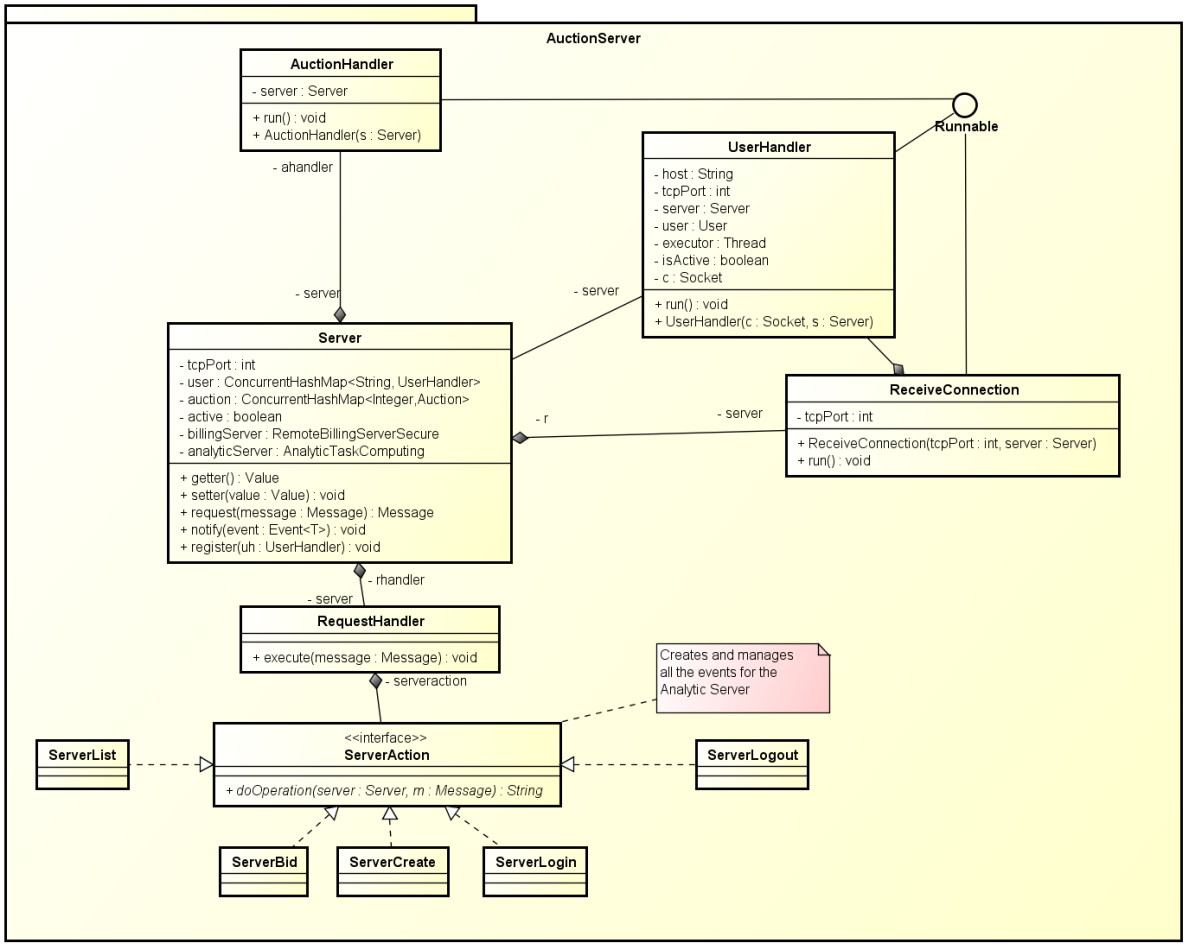
Client



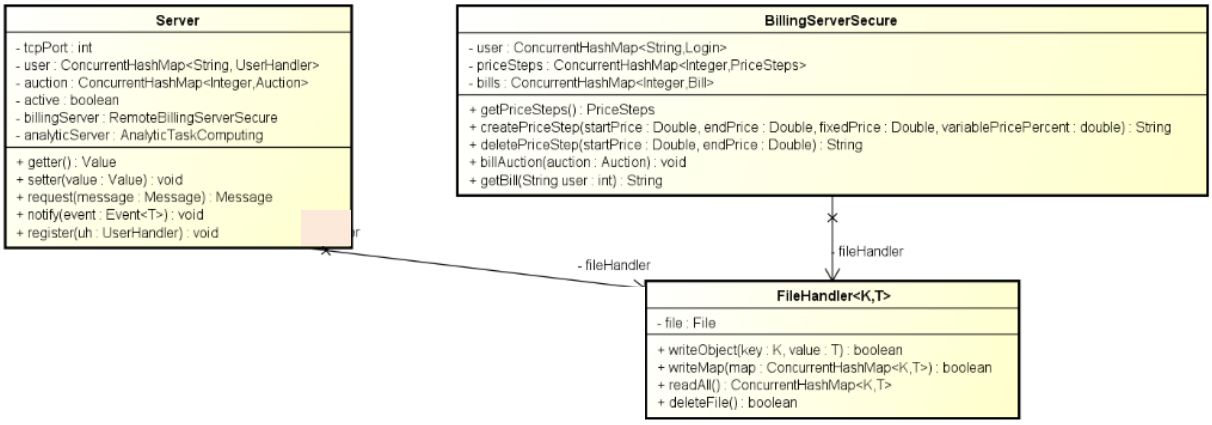
Message-Model



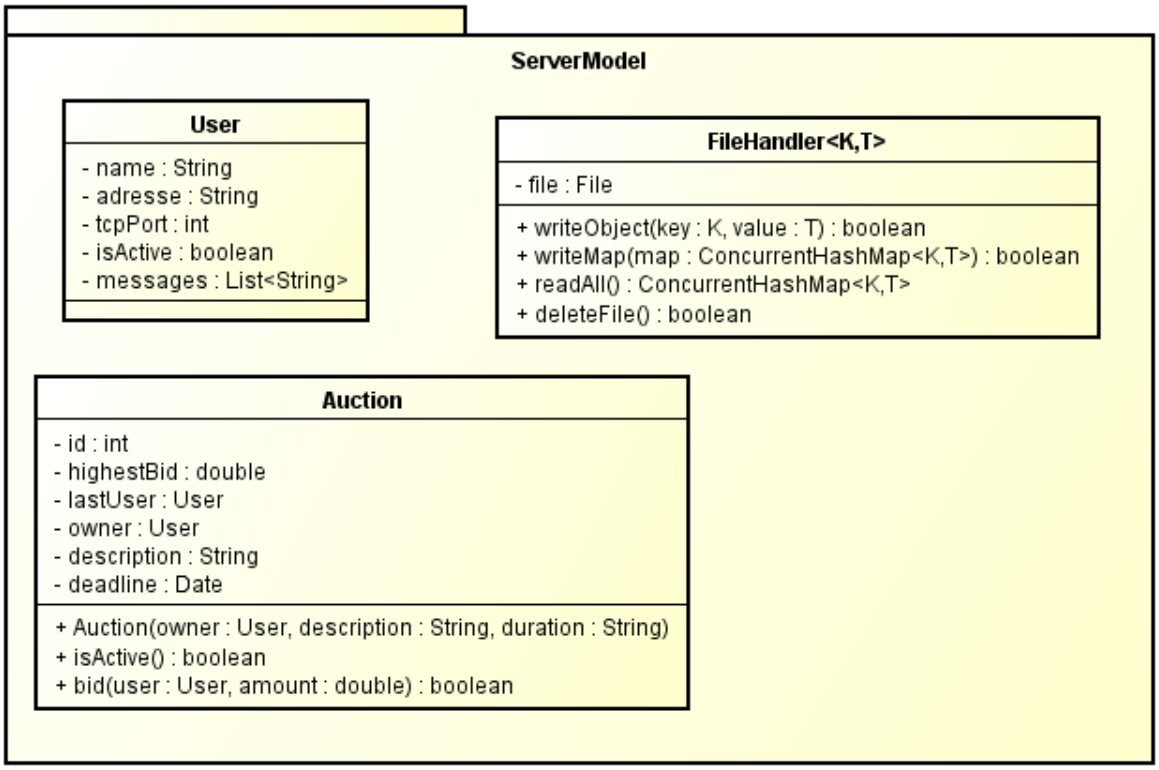
Server



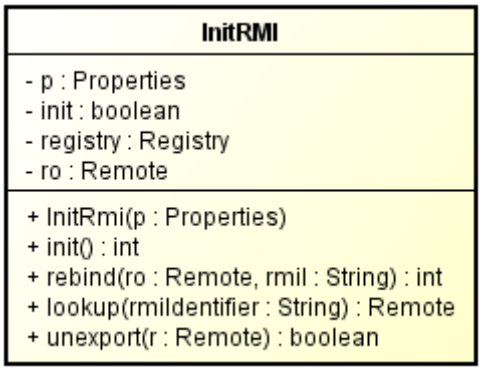
FileHandler



Model Server + FileHander



InitRmi



Zeitschätzungen und Arbeitsaufteilung

work package	Reichmann		Krepela		Lipovits		Tattyrek		Traxler	
	e.	r.	e.	r.	e.	r.	e.	r.	e.	r.
UML Klassendiagramm		2,00	2,00	2.5					2,00	4,00
UML Aktivitätsdiagramm					2,50	2,00				
UML Use-Case	1,00	0,50								
UML überprüfen							0,50	0,50		
Analytics-Server implementieren	6,00	4,00					3,00	0,50		
Billing-Server implementieren			6,00	5.5					4,00	2.5
Management-Client impl.					3,00	5,00	3,00	0,00		
Testing Component impl.					4,00	5,00				
Model-Klassen (Events, Bill, Steps)							1,50	1,00		
File-Persistence				1,00			2,00	4,00		
Refactoring old Source	2,00	2.5							2,00	1,00
RMI-Verbindungen implementieren	1,50	3,00		1,00					3,00	6,00
RMI-Verbindungen testen			1,00						1,00	1,00
Analytics Unit testen	2,00	2.5					2,00	0,00		
Billing Unit testen			2,00	2,00						
Management-Client Unit testen					2,00	4,00				
Testing Component Unit testen					2,00	1,00				
Protokoll	2,00	2.5	2,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
Dealing with testing prolems(+Ant)		8,00		4,00		3,00		2,00		3,00
total	14,50	17,50	13,00	9,00	14,50	21,00	13,00	9,00	13,00	16,00
sum					68,00					72,50

UnitTests

EventHandlerTest

In dieser Testklasse wird die Funktionsweise des EventHandler getestet. Das bedeutet, ob alle Eingangs Events die entsprechenden Ausgangsevents liefern.

Setup

Am Anfang wird ein neuer AnalyticsServer gestartet. Danach wird ein Mock-Client gestartet, welcher alle Events über die er benachrichtigt wird in einer Liste speichert. Diese können sich die Testfälle dann mittels getter + iterator holen.

TestUserLoginEvent

Ein USER_LOGIN Event wird an den EventHandler geschickt. Das erwartete Ergebnis ist ein Event USER_LOGIN, welches sich in der Queue des Clients befindet. Dieser Test ist sehr trivial und dient nur zum initialen Testen.

testUserLoginLogoutSessionEvent

In diesem Test wird ein User eingeloggt und ein User ausgeloggt. Das passiert durch das Senden eines USER_LOGIN und eines USER_LOGOUT Event. Der Timestamp wird so gesetzt, dass die SessionTime 50 ist. Nach der Verarbeitung werden folgende Events erwartet: USER_LOGIN, USER_LOGOUT, USER_SESSIONTIME_MIN (mit dem Wert 50), USER_SESSIONTIME_MAX (mit dem Wert 50) und USER_SESSIONTIME_AVG (mit dem Wert 50).

testAuctionStartedEndedNoBid

In diesem Test wird eine Auction gestartet, beendet und KEIN Bid darauf platziert. Erwartet wird eine Success_Ratio von 0.

Input: AUCTION_STARTED, AUCTION_ENDED

Output: AUCTION_STARTED, AUCTION_ENDED, AUCTION_TIME_AVG (Wert 50, da der Timestamp so gesetzt wird, AUCTION_SUCCESS_RATIO(Wert 0))

testBidOnAuctionChangesRatio

In diesem Fall wird eine Auktion gestarted, ein Bid platziert und die Auktion beendet. Erwartet wird eine SuccessRatio von 1.

Input: AUCTION_STARTED, AUCTION_ENDED

Output: AUCTION_STARTED, AUCTION_ENDED, AUCTION_TIME_AVG (Wert 50, da der Timestamp so gesetzt wird, AUCTION_SUCCESS_RATIO(Wert 1))

testBidCountPerMinute

Es wird eine Auktion gestartet, ein Bid platziert und 1 Minute gewartet. Danach wird der BidCountPerMinuteWatcher ein Event senden. Erwartet wird ein Wert 1.0

ManagementClientTest

TestCase	Input	Expected Output
loginWrongPwTest	!login test muh	“Wrong password!”
loginNoBillingTest	Bs.shutdown !login tes tes	ERROR: Connection to BillingServer lost. You have to login again!
printTest	!print	printAutomatic==False
autoTest	!auto	printAutomatic==True
hideTest	!hide	printAutomatic==False
subscribeNoAnalyticeTest	As.shutdown !subscribe .*	ERROR: AnalyticsServer not available right now. Retry after starting Analytics
subscribeAnalyticsBackTest	As.shutdown New ManagementClient New AnalyticsServer !subscribe .*	INFO: Analytics seemed to have moved. Looking up
unsubscribeAnalyticsBackTest	As.shutdown New ManagementClient New AnalyticsServer !unsubscribe 0	INFO: Analytics seemed to have moved. Looking up
unsubscribeNoAnalyticeTest	As.shutdown !unsubscribe 0	ERROR: AnalyticsServer not available right now. Retry after starting Analytics
unsubscribeExceptionTest	!subscribe	ERROR: Wrong number of arguments given!\nUsage: !unsubscribe <subscriptionID>
subscribeExceptionTest	!unsubscribe	ERROR: Wrong number of arguments given!\nUsage: !subscribe <filterRegex>
logoutNoBillingTest	Bs.shutdown !login test test	ERROR: Connection to BillingServer lost. You have to login again!
secureNoBillingTestend	!login test test Bs.shutdown !steps	ERROR: Connection to BillingServer lost. You have to login again!
endUnsecureTest	!end	Management Client is shutting down!
endSecureTest	!login test test !end	Management Client is shutting down!
endSecureNoBillingTest	!login test test Bs.shutdown !end	ERROR: Connection to BillingServer lost. You have to login again!
nonsenseTest	!muh	ERROR: This Command does not exist! Allowed commands: !login <username> <password> !logout !steps !addStep <startPrice> <endPrice> <fixedPrice> <variablePricePercent> !removeStep <startPrice> <endPrice> !bill <userName> !subscribe <filterRegex> !unsubscribe <subscriptionID> !print !auto !hide

LoadTest

TestCase	Input	Expected Output
setFromFileErrorTest	Pfad: iwo/lloadtest.properties	FileNotFoundException
setAllTest	p.setX.(10) X = alle jeweiligen attribute	10 Bei allen gesetzten Attributen

BillingServer

createPriceStepTest

Creates four pricesteps (unordered) and look with getPriceSteps() if they have been created and if they are ordered correctly.

createPriceStepTestPriceStepIntervalOverlapException

Expects: PriceStepIntervalOverlapException

Creates two PriceSteps. The end prices overlap.

createPriceStepTestPriceStepIntervalOverlapException2

Expects: PriceStepIntervalOverlapException

Creates two PriceSteps. The end prices are infinity and the start prices overlap.

createPriceStepTestPriceStepIntervalOverlapException3

Expects: PriceStepIntervalOverlapException

Creates two PriceSteps. The first price step has as end price infnity and the second start price is higher than the start price from the first price step. The price steps overlap.

createPriceStepTestIllegalValueException

Expects: IllegalValueException

Create a price step with negative value. Throws IllegalValueException.

deletePriceStepTest

Creates a price step. Deletes a price step. Then creates the same step again and some additional. No exception is called because the step was successfully deleted and could be created again.

toStringTest

Creates a price step and tests if the toString method output format is correct.

billAuctionAndGetBillTest

Creates some price steps and bills. Tests if the bill functions calculation is correct. (calls getBill).

billAuctionAndGetBillTestIntervalDoesNotExist

Creates some price steps and bills. Tests if the bill functions calculation is correct if no price step exists for the given interval. (calls getBill)

getBillTestFalse

Calls getBill for a user that does not exist. Tests if the output is correct (user gets informed that there are no bills for him)

shutdownAndLoad
Creates some price steps and bills. Calls getBill to show that the data exists in the Map. Tests if shutdown saves the data to a file and if the data is loaded again after creating a new instance. These functions work correct if the same data is loaded again. That means that the getBill method is called again after saving and loading the data.

Unit test of package „Client“

To get a code coverage of 80.5% the following Classes were tested:

- Client
- CLI
- TaskExecuter

ClientTest
This class tests the functionality of the Client. It simulates user-input with FakeCLI and checks if everything is executed properly.

Setup

The Method setUp() gets executed before every test method. It starts all needed component such the servers.

testLoginWrongNumberOfArguments
Tests if the error handling works correctly if the login command gets a wrong number of arguments.

Input: „!login test test“
Output: Error message, checks if it matches the expected error

testBidNotLoggedIn
Tests the error handling if trying to !bid without being logged in.

Input: bit command without logging in before
Output: Error message, checks if it matches the expected error

testNoSuchCommand
Tests the error handling if the specified command doesn't exist.
Input: a command that doesn't exist
Output: Error message, checks if it matches the expected error

testDoubleLogin
Tests error handling if already logged in user tries to login again.

Input: two login commands with a little pause

Output: Error message, checks if it matches the expected error

testCreateNotLoggedIn
Tests error handling if user tries to create an auction without being logged in.

Input: create command without logging in first
Output: Error message, checks if it matches the expected error

testLogoutNotLoggedIn
Tests error handling if user tries to logout without being logged in first.

Input: logout command without logging in first
Output: Error message, checks if it matches the expected error

testGetCli
Tests if getCli() works as expected.

Generate a new Client object and pass a new cli as a parameter.
Output: return value of getCli(), check if the returned object matches the passed one

testGetTcpPort
Tests if getTcpPort() works as expected.

Generate a new Client object and pass a port as parameter
Output: return value of getTcpPort(), check if the returned port matches the passed one

CLITest

This test class tests the functionality of the CLI class.

Setup
Sets up the output stream „System.out“ to print to a ByteArrayOutputStream so the prints can be matched with the passed strings.

testOut()
Tests if the out() method works properly.

Input: a String passed as parameter
Output: String „printed“ to a ByteArrayOutputStream, check if it matches the passed string + newline.

testOut()
Tests if the outLn() method works properly.

Input: a String passed as parameter
Output: String „printed“ to a ByteArrayOutputStream, check if it matches the passed string.

testReadIn
Tests if the readLn() method works properly.

Input: String read from a pre defined ByteArrayInputStream
Output: the previously read String, check if it matches the String from ByteArrayInputStream

TaskExecuterTest

This class tests the execution of the commands. Instead of simulating user input, it directly accesses and executes the command methods.
Setup

The Method setUp() gets executed before every test method. It starts all needed component such the servers.
testLogin

Executes the method for the login command with valid parameters.

testLogout
Executes the method for the logout command with valid parameters.

testBid
Executes the method for the bid command with valid parameters.

testCreate
Executes the method for the create command with valid parameters.

testList
Executes the method for the list command with valid parameters.

SystemTest

Die Ausgaben sehen aus wie die bei der Angabe. Einen umfangreichen Systemtest wird der Herr Professor Borko durchführen, daher ist dieser nicht im Protokoll

Absprachen

-> Analytic Server

-> Billing Server (Frage GRAFIK?) schritte setzen, abrechnung erstellen -> keine persistenz

-> Testing Load Client (Viele Clients machen bids etc.)

-> Management Client (Befehle für Billing Server, Benachrichtungungen etc)

Altes Programm → List-> ConcurrentHashMap, Eigene Exceptions → UDP Notification brauchen wir nicht mehr

UML-Klassendiagramm → Krepela, Traxler

Aktivitätdiagrann → Lipovits

Use Case Diagramme → Reichmann

Checker → Tattyrek

Tasks:

- RMI-Verbindungen → Traxler
- Analytics Server → Reichmann, Tattyrek
- Billing Server → Krepela
- Management Client → Lipovits
- Testing Component → Lipovits
- Model →Tattyrek (Model JUnitTests)
- Ausbesssern alten Code → Traxler, Reichmann
- Ant,Protokoll → Reichmann

JEDER TESTET SEINEN TEIL + TECHNOLOGIEBESCHREIBUNG FÜRS PROTOKOLL!!!

Arbeitspakete bis Montag, 10. Februar

Liebes Team,

damit auch was beim Projekt weitergeht, wird es notwendig die Aufgaben zu definieren:

Jeder sollte bis zum 10. Februar einen Prototypen (= 70% Funktionalität) umgesetzt haben. Dieser soll bis Montag fertig sein, da ich einen Tag benötige die Arbeitspakete zu kontrollieren und etwaige Sachen auszubessern. Ich befinde mich selbst auf Urlaub, werde jedoch per Mail erreichbar sein und kann auf auftretende Fragen antworten.

Die betreffenden Arbeitspakete pro Person könnt ihr aus der Liste auslesen. Bitte haltet euch bei der Umsetzung an das UML und schreibt mir wenn(am besten bevor) ihr Änderungen an diesem machen müsst.

Bitte sprecht euch mit eurem jeweiligen Mitarbeiter ab, wer welchen Teil übernimmt. Es gibt pro Server einen Hauptverantwortlichen der im Notfall das Sagen hat.

Dokumentiert bitte eure Vorgehensweisen, damit wir später ein gutes Protokoll haben.

Wichtig: Da wir vom Borko die Tests durchgeführt bekommen, muss die Funktionalität bis zum 17. Gegeben sein! Sonst haben wir nichts von unserem Vorteil. Unit-Tests bitte auch beachten, können aber bis zum 20. Fertiggestellt werden. Und: Ihr dürft ruhig mehr machen, dann sind wir halt schneller fertig^^

Gewünschter Fortschritt bis zur Deadline:

Management-Client: (Lipovits)

- Usereingaben werden erkannt (alle geforderten Befehle) und auf Syntax kontrolliert (Trockene Ausgabe ohne Senden)
- Bei falschen Eingaben wird eine **eigene** Exception geworden (außer nicht beötigt)

Load-Testing Component: (Lipovits)

- Properties aus dem File lesen und entsprechend speichern
- Liste an Clients (Package Client aus altem src) erstellen
- Implementieren Fake_Cli (Stichwort InputStream)

BillingServer + Secure (Krepela)

- Erstellen von PriceSteps
- Löschen von PriceSteps
- Anzeigen der PriceSteps schön formatiert
- Anmelden mittels Username + pwd, auslesen aus Property-File (siehe Angabe); Vermittlung muss noch nicht erfolgen
- Rechnung erstellen lassen

AnalyticsServer (Reichmann)

- Alle Events als Models erstellt und entsprechende HashMap angelegt

- Dokumentieren der Abhängigkeiten (bei welchem Eingehenden Event werden welche erzeugt)
- Berechnung von neuen Events
- Passende Events zu einem Regex finden
- Überlegen von Notifications zu einem Event

RMI-Verbindungen (Traxler)

- Alle Interfaces definiert
- Stubs implementiert
- Sozusagen fertig um die Componenten nurnoch verbinden zu müssen

Alter Server: (Traxler/Reichmann)

- ArrayList durch HashMap ersetzen
- Eigene Exceptions definieren
- ThreadPool einsetzen
- Ev TimerService

Diagramme: (Reichmann)

- Kein Schatten
- Include bei Use-Case entfernen
- Sichtbarer machen

Mfg Daniel

P.S.: Wenn euch die Aufgaben zu viel sind, dann bitte ich um Rückmeldung, damit ich das eine oder andere Arbeitspaket überdenken kann.

Hallo liebes Team,

wie ihr wisst, müssen wir nächste Woche mit unserer Aufgabe fertig sein, da der liebe Herr Professor Borko uns die Tests abnimmt. Ich muss ihm nurnoch seine Arbeitspakete senden (überlegen wir am Freitag)

Was heißt das für uns?
Jeder muss seinen Teil am Sonntag(! 16.02.14) fertig haben. Wie ich gesehen habe, fehlt euch allen nichtmehr viel und das sollte auf jeden Fall machbar sein. (AnalyseServer is am wenigsten weit, aber den muss sowieso ich auch machen).
Montag schau ich mir den Fortschritt an und teste euren Code.

Thomas: Bereite die Interfaces so vor, dass es funktioniert, wenn man die Teile zusammenfügt.

Nanak: Du übernimmst erstmal den FileHandler, ich überleg mir was zum AnalyseServer und sag dir wenn ich etwas brauch.

Es ist gut, wenn ihr vorher fertig seid, dann haben wir noch Zeit zum Debuggen.

Jeder schreibt bitte seine Dokumentation für seinen Part. Testing kann in der letzten Woche erfolgen, aber auch nicht bis irgendwann sondern rechtzeitig.

Das build.xml werde ich übernehmen

mfg
Daniel

P.S.: Ich bitte um Rückmeldung und schreibt mich bitte an, wenn ihr Probleme jeglicher Art habt oder Hilfe braucht. Fragen bez. UML auch an Traxler/Krepela weiterleiten.

Liebes Team,

da bis gestern alle Eigenfunktionalitäten fertigzustellen waren, möchte ich bitte von euch bis morgen, dass die Funktionalitäten zusammengefügt werden.

Das bedeutet für Traxler, dass er die RMI-Objekte überall in die Registry schreiben muss, und für alle Server + MgmtClient verfügbar machen muss.

Bitte bis morgen 19:00.
Setzt euch morgen in NWSY/Deutsch zusammen und schaut, dass ihr dann etwaige Fehler gemeinsam Lösen könnt. Am Abend werde ich mir das ganze ansehen und testen.

mfg,
Daniel

Liebes Team,

diesen Samstag 18:00 möchte ich das JEDER seine Arbeitspakete fertig hat. Ich brauch nämlich auch bisschen Zeit das ganze zu kontrollieren und will nicht soo einen Stress haben.

Daher die Arbeitspakete:

Lipovits:
- ManagementClient NamingConvention umsetzen
- Exceptions umbenennen
- Exceptions nur Message ausgegeben, nicht den StackTrace
- LoadTest Component fertigstellen
- Unittests zu: ManagemenClient, LoadtestComponent
- PoliteShutdown Loadtest
- Exceptions bei Client (alt) einbinden

Tattyrek:
- Fertigstellen Filehandler + Tests und Beschreibung!
- Unnittest Client-Package aus dem alten Server
- Eigene FileHandler-Exceptions
- Source dokumentieren

Traxler:
- Behandeln von der Exception, falls ein Server beendet wurde (Erneutes Lookup, speichern von messages in Queue und dann bearbeiten=
- UnitTest RMI-Stubs/Verbindungen
- Dokumentation der Interfaces
- NamingConventions
- Eigene RemoteExceptions

Krepela:
- Dokumentation(+ Source) BillingServer
- UnitTest alter Server (UDP-Notifier rausgeben)
- Finden eigener Exceptions alter Server

Reichmann:
- Erstellen neuer Klassendiagramme
- Dokumentation des Src + Analytics
- Finaler Test
- Eigene Exceptions definieren
- Zusammenführen aller Pakete
- XML

Diese Arbeitspakete sind zum Teil schon erledigt und bedarf nicht zuviel Aufwand.
Werden Arbeitspakete bis Samstag nicht geschafft, so fühle ich mich gezwungen das Arbeitspaket selbst umzusetzen und die Punkte auch so zu verteilen.
Wenn jemand sagt, dass er ein Arbeitspaket nicht machen kann/mag, so soll er mir das sagen und wir können das mit den Punkten regeln (ein anderer übernimmt das Paket und bekommt eine gewisse Anzahl Punkte)

Mit freundlichen Grüßen,

Daniel Reichman

Sehr geehrter Herr Professor Borko,

Wir würden Sie gerne als externen Tester in Anspruch nehmen. Bitte testen sie so, als wären Sie derjenige der das Programm abnimmt und dokumentieren Sie Ihre Tests. Bitte legen sie besonderes Augenmerk auf eine Exception "NoSuchObjectInTable" von RMI und prüfen Sie ob diese bei Ihnen auftritt. (Sollte ein Problem des Garbage-Collectors sein. Bei 4/5 des Teams tritt diese nicht auf)

Wir bräuchten Ihren Bericht bestenfalls Sonntag, damit wir noch Änderungen umsetzen können. Bitte aber auch nur, wenn es Ihre gesundheitliche Verfassung zulässt.

Wir wünschen Ihnen eine gute Besserung!

Mit freundlichen Grüßen,
Daniel Reichmann