Ausarbeitung

RMI Messenger

APR - Angewandte Programmierung

HÖBERT Timon | SOCHOVSKY Josef
MONGIA Himanshu | PAWLOWSKY Gabriel

5BHITS

VERSION 1.0

Last Update: 26. Februar 2013

Inhaltsverzeichnis

1 Aufgabenstellung

Description

In this assignment you will learn:

- the basics of a simple distributed object technology (RMI)
- how to bind and lookup objects with a naming service
- how to implement callbacks with RMI

Overview

In this assignment we will develop a simplified messenger application (like ICQ or MSN Messenger), which allows users to find other users, become friends and exchange messages.

First, here's a short explanation of the domain:

- For each user the following master data is captured: username, password, first name, last name, birthday, profession, email address and hobbies. The username must be unique among all users.
- Users may establish friendships. For friendships there always must be two agreeing parties, so one user has to send a friendship request to another user, who may then accept or reject it. Friendships allow users to send messages to each other; also friends of a user are notified when he/she goes online or offline or when he unregisters. Users may also stop being friends.
- The public user data (i.e. the data that may be inspected by other users) contains the whole user master data (except the password), the usernames of all friends and the current connection state (i.e. online/offline) and the time the user was last online.

For this assignment we use a conventional client-server architecture as shown in Figure 1. There is one central server who's responsible for managing the user information. For each operation (shown using solid lines) the clients have to contact the server, which implements the required behavior. Hence, in contrast to the previous assignment there's no direct user-to-user communication - every operation is propagated via the server.

Most operations invoked on the server lead to notifications (shown using dashed lines) of other users, for instance login or send messages. These event notifications will be implemented using RMI callbacks. If the recipients of any event (e.g., friendship request/acceptation, incoming messages, etc.) are offline, these events have to be preserved until the users go online the next time. Then the server has to deliver all missed events.

Most distributed object frameworks provide a naming service, which allows binding/looking up remote references to/by simple names. By this the coupling between clients (looking up) and servers (binding) can be reduced, and the real location of the server object becomes transparent. RMI provides the java.rmi.registry.Registry service (which itself is a Java RMI Remote interface) to accomplish these features. Therefore you'll have to bind your remote server object to the registry, and look it up in your client.

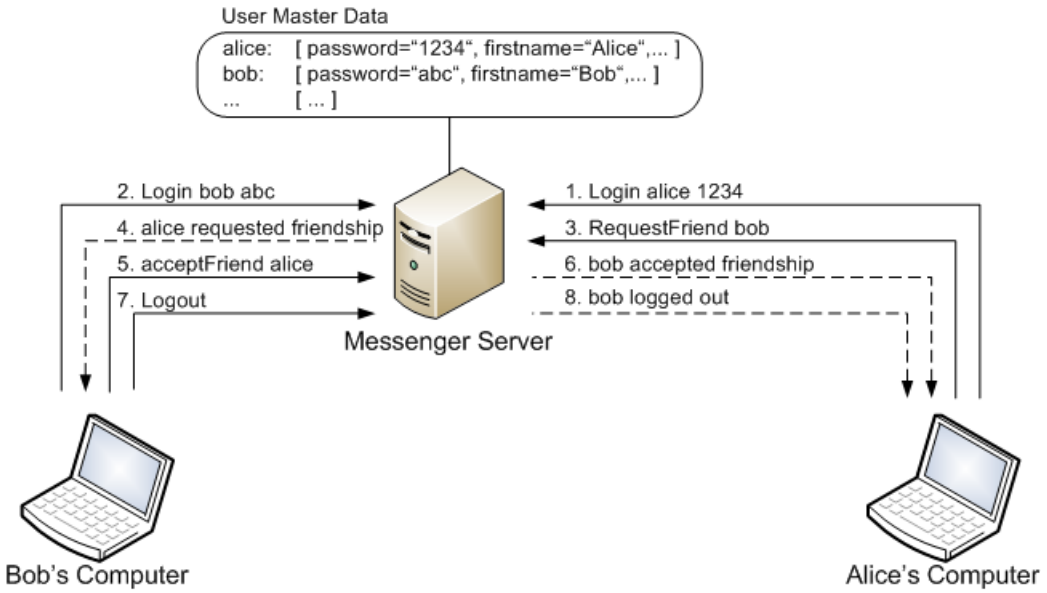

Abbildung 1: Messenger Architecture

Finally, the state of the server must not be lost when shutting down normally, so the information must be written to the file system.

Server

Arguments

Your server program does not need any command line arguments this time. Instead a properties file (named messenger.properties) should be read in from the classpath (see the hint section for details). The properties file is provided and can be downloaded here. It contains the port where a registry service has to be started, as well as the name that should be used for binding the server object. Finally the file name in which to store the data persistently is provided.

Messenger remote interface

In RMI you always have to define remote interfaces (interfaces extending java.rmi.Remote). Each method of this interface must throw java.rmi.RemoteException, so errors occurring during the remote invocation can be signaled to the caller. Define your server interface according to the description of the client interactive commands.

Implementation details

On startup you first have to read in the properties file. If the storage file with the given file name exists, you have to read it and restore the server state. See the hint section for help on how to do this. Then create a registry service, export your server object to make it remotely available and finally bind it to the specified name in the registry (help can be found here).

As in the previous assignment the server has to manage state across several threads. However, this time you don't have to create the threads yourself. Instead the RMI runtime takes over this job and transparently uses threads from a thread pool to allow concurrent access to your remote object. This again makes synchronization necessary when accessing your shared data

structures!

Note that users may go offline without informing the server appropriately (for example if the computer crashes or the connection is lost). You don't have to detect this at once, but the next time the server is trying to contact the user (for example to deliver a message): if a java.rmi.RemoteException occurs you may assume the user is offline. Therefore update the user's state, store the event for the user and inform his/her friends about it.

Your server must also be able to cope with invalid requests, therefore check that each parameter of each method is not null, and otherwise throw a java.lang.IllegalArgumentException. As you will later see the most operations require a logged in user, thus you have to make sure that the caller is really logged in and is whom he/she's pretending to be (either by using a session identifier or by letting the client submit its user name and password for each operation). Also take care in your server implementation to deal with semantically invalid requests (such as cancelling a non existing friendship, accepting a non existing friendship request, unknown login credentials etc).

The server is responsible for firing notifications if certain events occur (you find the concrete list of events here). If some recipients are not online the server must queue their events (except for "Friend went online/offline" events), and finally inform these users when they login the next time.

If your server is ready for handling requests print "Server up. Hit enter to exit." to the console and implement this behavior. On exit unbind the remote object from the registry and also unexport it, otherwise your program might not shut down - again you must not use System.exit(). Do not forget to store the server's state into the specified file. If this file does not exist yet, you have to create it (see hints for a little help).

Client

In this part you have to build an interactive command line client application.

Arguments

Again no arguments are specified on the command line; the client has to read in the same properties file as the server (messenger.properties) from the classpath. You will need the host name and port where the registry service is listening (started by the server) and the binding name of the server object.

Callback messenger remote interface

The server quite often needs to notify the clients about certain events. The server interface you are going to specify only allows clients to contact the server synchronously, but how about notifying the clients about asynchronous events such as a sent message? Polling would be an option, but a very inconvenient one. So the way to go is to define a remote interface for clients, of which the server itself can call methods remotely.

Therefore you have to provide one or several methods that can be called by the server to inform the client about the different events. To sum it up, these events may occur:

- Friendship requested/accepted/rejected/cancelled
- Message from friend
- Friend went offline/online

- Friend unregistered

If any of these events occurs you have to print an appropriate message containing all the relevant information (including the time of occurrence) to the console.

Implementation details

On startup of your program you first have to read in the properties file. Then you have to obtain a reference to the registry service and look up the server object with the specified name. Note that you can only cast the looked up java.rmi.Remote instance to the remote interface, not the implementing class, because only a stub, which is responsible for communicating with the real server object, is returned!

Now that we've got a reference to our server, the user may enter commands (see below). However, note that you have to implement the callback remote interface and export it like you exported your server object. The only difference is that you should not bind it to the registry, but instead simple pass it as parameter for the login command. Since there are quite a lot of different commands, we recommend using the Command pattern, which is described fairly well in this article.

Interactive commands

The following commands must be supported by your client application. Take care of handling invalid commands and arguments and provide usage messages in these cases. Print meaningful error messages whenever the server throws an "expected" exception (such as "Username already exists." when trying to register with an existing username). Also print success messages if an operation could be executed successfully. For all commands you first need to login (except for register, login and stop).

register <userName> <password> <firstName> <lastName> <birthday in format dd.MM.yyyy> <profession> <emailAddress> hobby

Registers the user at the server. hobby describes an optional list of hobbies, where each hobby is exactly one word. You may assume syntactically correct email addresses. However, the server has to check that no other user with the provided username yet exists. E.g.: :> register user1 password1 FirstName1 LastName1 01.01.1985 student user1@user1.at jogging swimming Registration was successful.

unregister

Unregisters the currently logged in user from the server. The server has to take care of informing all friends about the unregistration. On the client side the user must also be logged out afterwards (and the remote callback object unexported).

login <username> <password>

Logs the user in. In your client implementation this is the time to create your callback interface implementation object, export it and pass it as parameter to the server. The server has to return the user's data (name, birthday, profession, email address and hobbies), his/her direct friends' public data, pending friendship requests (outgoing as well as incoming friendship requests that have not been accepted or rejected yet) and finally all missed events (see QueuedEvents). Any friend must be informed by the server that the user has gone online now. The output should include the same information as displayed below:

logout

Logs out the currently logged in user. Any friend must be informed by the server that the user has gone offline. In your client implementation you have to unexport your previously exported callback remote object. findUser [searchString] Queries the server for finding all users matching the search string. Matching in this context means that the search string has to be case-insensitively part of the user name, first name, last name, profession, email address or any hobby. If searchString is omitted, all registered users must be found. The server then has to return the public data of all found users. Your output should include the same information as displayed below:

getUserInfo <userName>

Gets the information of a specific user. The server has to return the user's public data and the output should contain the same data as above.

requestFriend <userName>

Requests the friendship of another user. The server has to check they are not friends yet and that the other user has not requested friendship yet. Finally let the server inform the recipient of this request about this event.

acceptFriend <userName>

Accepts the requested friendship; the server has to inform the requestor appropriately. In your server you have to check that such a request really exists.

rejectFriend <userName>

Rejects the requested friendship; the server has to inform the requestor appropriately. In your server you have to check that such a request really exists.

cancelFriend <userName>

Cancels an existing friendship with the specified user. The server has to notify the other user about this event.

sendMsg <userName> <message>

Sends a message to the specified friend. The message must start and end with a quote - all characters in between (without the quotes) have to be transmitted to the server. The server has to check that the recipient is really a friend and then forwards the message to it.

stop

Stops the client application. If a user is currently logged in you have to log it out implicitly. Again you may not use System.exit(), but have to orderly close all acquired resources. Port policy

As you might have thought of your remote objects also require an unused port. But since we are using the registry for mediation purposes, this can be an anonymous (any available port selected by the operating system) port (see exporting objects for more details).

Ant template

As in the previous assignment we provide a template build file (build.xml) in which you only have to adjust some class names. Put your source into the subdirectory "src", place the messenger.properties file into the "src" directory (the ant compile task then copies this file to the build directory) move on the command line to the directory where the build file

is located and simply type ”ant” for compilation. Type ”ant run-server” to start the server, ”ant run-client” to start a client. Put the src directory including messenger.properties and build.xml into your submission. Note that it's absolutely required that we are able to start your programs with these 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. 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. 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 messenger.properties file. Reading in a properties file from the classpath (without exception handling):

```
java.io.InputStream is = ClassLoader.getSystemResourceAsStream("messenger.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!");
}
```

Listing 1: PropertiesFile reading

The easiest way for storing/loading your server state to/from the file system is by using serialization. This specification shows all the steps you need to accomplish this feature (i.e. making your object serializable by implementing java.io.Serializable, creating a java.io.FileInput/OutputStream, wrapping it using an java.io.ObjectInput/OutputStream and finally reading/writing objects). To create a file in a non existing directory, you first have to create the parent directories. Therefore check whether the file's parent file (= the directory) (java.io.File.getParentFile()) exists, and if not, call java.io.File.mkdirs() on it. Afterwards you can simply use a java.io.FileOutputStream to write to that file. Formatting dates as well as parsing date strings can be easily accomplished by using java.text.SimpleDateFormat. See the API description for more details.

```
String dateString = "10.10.2010";
java.text.DateFormat df = new java.text.SimpleDateFormat("dd.MM.yyyy");
java.util.Date date = df.parse(dateString);
String formattedDate = df.format(date);
```

Listing 2: SimpleDateFormat Verwendung

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. Java Serialization Specifications: Explains the java serialization mechanism. Serialization and Deserialization: Some important guidelines.

General Remarks

We suggest to read the following tutorials before you start implementing: Java RMI Tutorial: A short introduction into RMI. JGuru RMI Tutorial: A more detailled tutorial about RMI. RMI Callbacks: Provides a simple example for using RMI callbacks.

Submission Guide

Every group must have its own design/solution! Meta-group solutions will end in massive loss in points! After the design review with the team leader, any design changes must be approved. Write a change request and describe what parts you want to change and why you think it's necessary to adopt your design decisions. As for group work usual, a protocol with the UML-Design, the work-sharing, the timetable and test cases 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 no 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

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. 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:

Documentation(15): timetable, explanation, description(JavaDoc), protocol/tests

Implementation(40): exceptionhandling, callback, server state, conditions/login, registry/export, collections/thread safety

Design(8): uml-class, +extrapoints usecase/activity

Testing(12): ant, arguments, unit-testing, un/register, login/out, findUser, friendship, sndMsg, stop

distributed by: Vienna University of Technology

Insitute of Information Systems 184/1

Distributed System Group [1]

2 Designüberlegung

Generell besteht die Anwendung aus folgenden Hauptpaketen: dem Clientpaket, dem Serverpaket sowie einem allgemeinen Paket.

2.1 Client

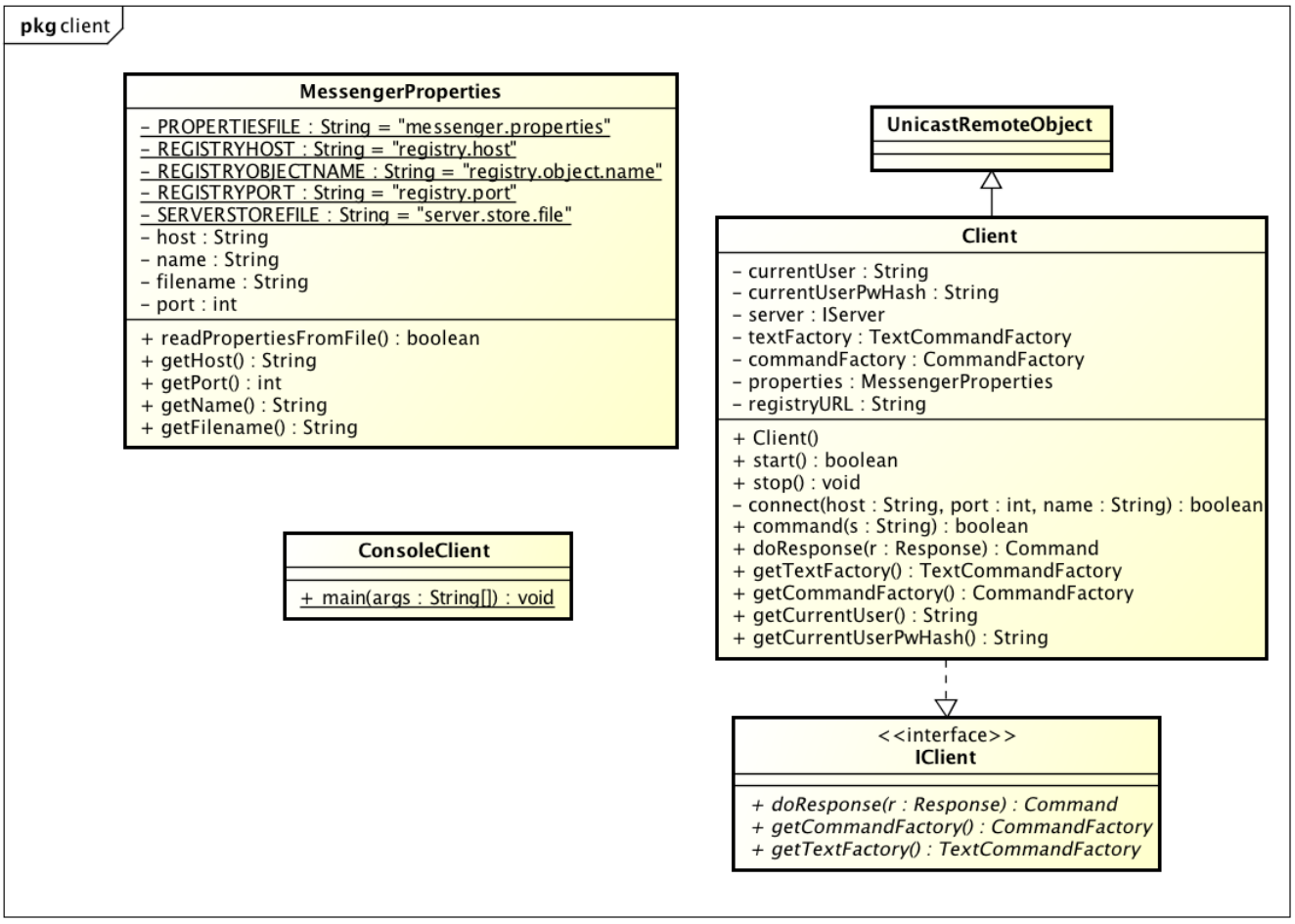

Abbildung 2: UML-Klassendiagramm Client-Paket

Prinzipiell besteht der Client aus einem Interface IClient, welches vom tatsächlichen Client implementiert wird, und im ConsoleClient via Konsole zugänglich gemacht wird. Das Interface definiert dabei eine doResponse-Methode welche via RMI asynchron vom Server für Notifications/Responses aufgerufen werden kann. Jede Response kann hierbei zusätzlich ein neues Command zurückliefern. Für RMI-Methodenaufrufe zur Serverseite wird eine Serverreferenz über die RMI-Registry in der start()-Methode geholt, wodurch nun alle Commands (z.B.: textbasiert via command-Methode) an den Server gesendet werden können.

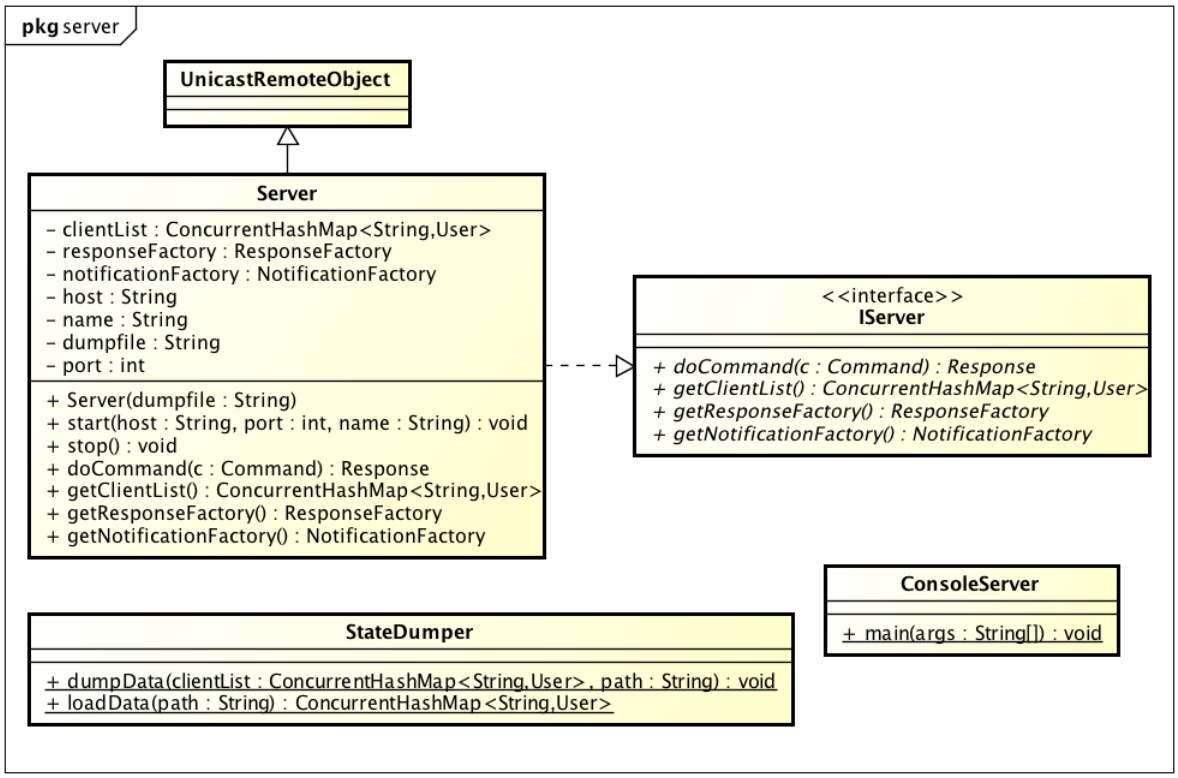

Abbildung 3: UML-Klassendiagramm Server-Paket

2.2 Server

Der Server besteht ähnlich dem Client aus einem IServer-Interface welches vom tatsächlichen Server implementiert wird, und via ConsoleServer zugänglich gemacht wird. Der Server muss hierbei von UnicastRemoteObject erben um in der start-Methode als Serviceobjekt gebindet zu werden. Das Interface definiert dabei eine doCommand-Methode welche als Gegenstück zur Clientimplementierung fungiert. Weiters werden alle User-Daten in einer ConcurrentHashMap mit dem Benutzernamen als Hashkey gepeichert. Über den im Interface deklarierten Getter können alle Commands darauf zugreifen. Zusätzlich muss diese Datenklasse in ein Dumpfile serialisiert werden könne, bzw. auch wieder zurück (siehe StateDumper).

2.3 General

Das allgemeine Paket General definiert alle notwendigen Klassen welche von Server als auch von Client verwendet werden. Zur Datenhaltung auf der Serverseite verwendeten Userklasse wird hier mit den einzelnen Attributen definiert. Interessant ist hierbei vor allem die Callback-Referenz sowie die notificationQueue.

Für die einzelnen Notifications sind die Friendship/FriendStatus-Enumdefinitionen notwendig.

Neben der MessengerHilfsklasse zum Passworthashen, bietet auch eine MessengerProper-

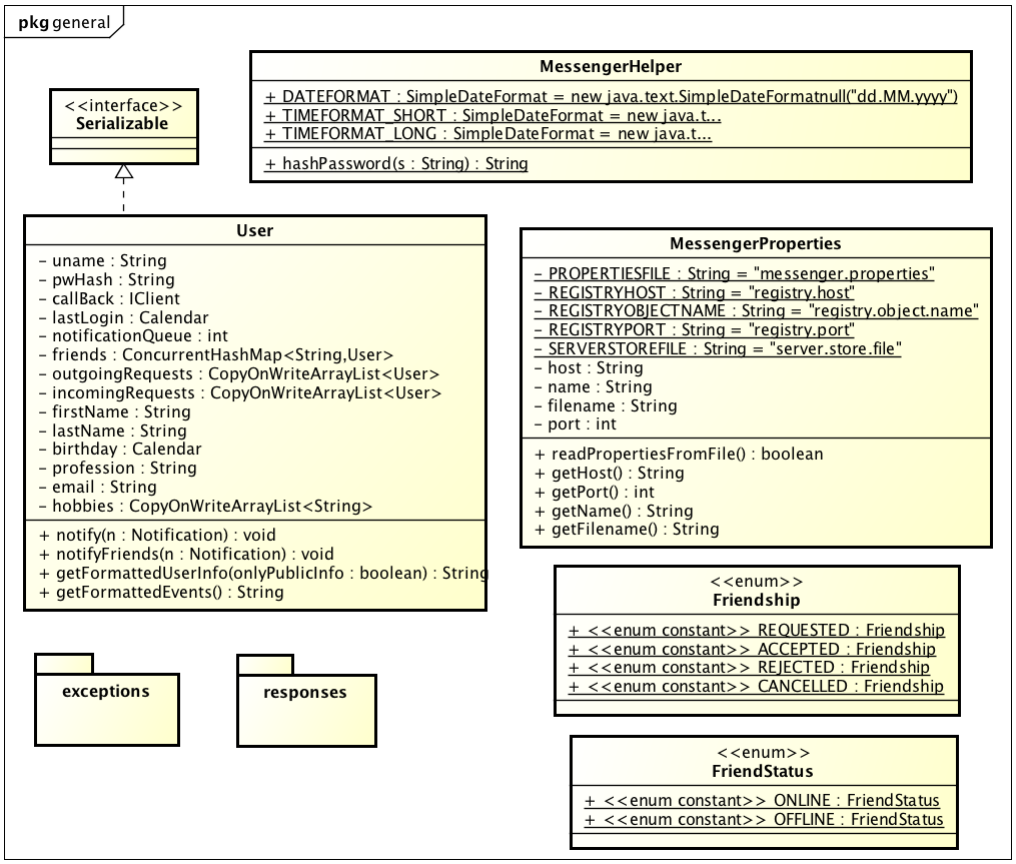

Abbildung 4: UML-Klassendiagramm General-Paket

tiesklasse die notwendigen Funktionalitäten um das Messenger.properties-File einlesen zu können.

2.4 Commands

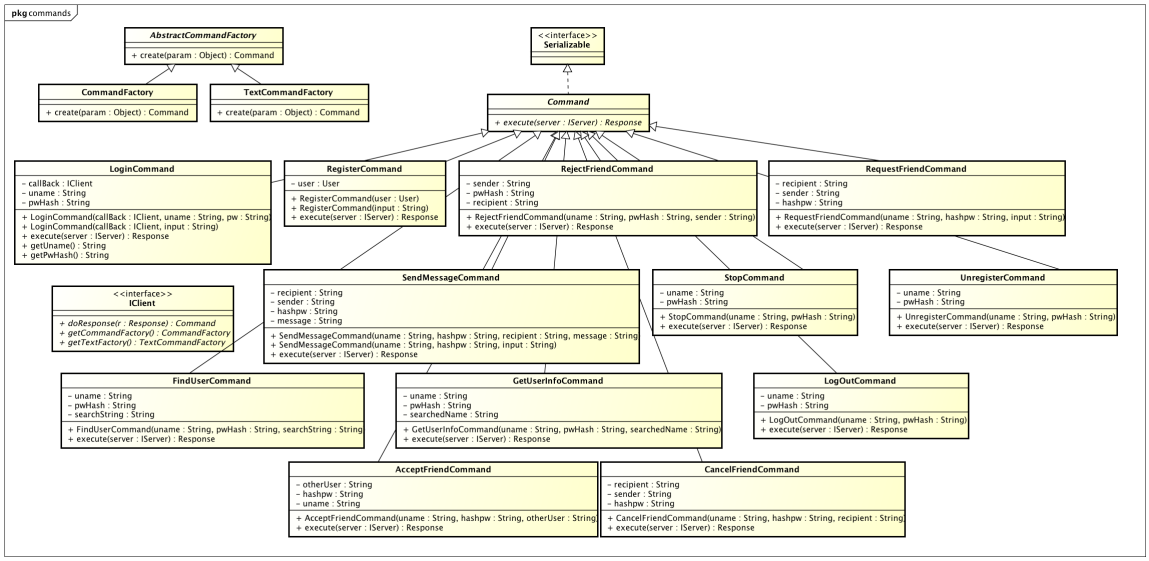

Abbildung 5: UML-Klassendiagramm Commands-Paket

Alle Serveranfragen des Clients werden funktional via Commands abgebildet. Dabei generiert der Client die jeweilige Instanz, sendet diese zum Server, wo sie via execute-Methode aufgerufen wird. Diese execute-Methode wird vom aufrufenden Server mit der eigenen Referenz ausgeführt, und kann optional eine Antwort in Form eines Response-Objektes zurückliefern. Das Markerinterface Serializable wird hierbei zum Versenden benötigt. Weiters wird in diesem Paket noch eine Factory auf Basis von Objektenparametern sowie auf Basis von Stringparametern, welche via ConsoleClient instanziert werden können.

2.5 Responses

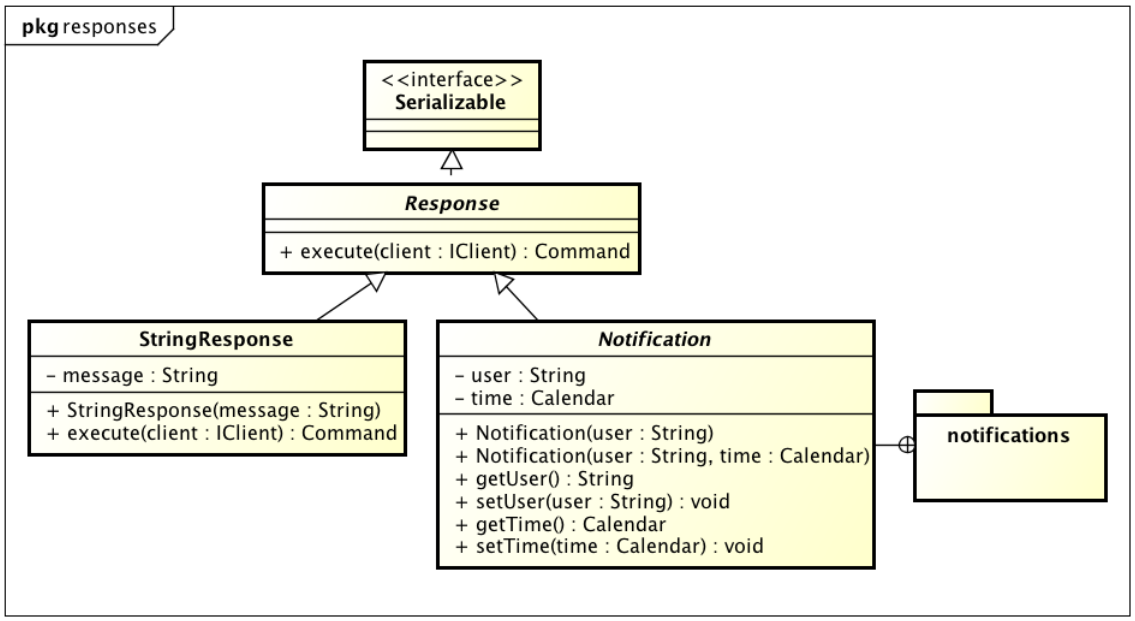

Abbildung 6: UML-Klassendiagramm Responses-Paket

Responses kann man als Gegenstück zu den Commands sehen, diese werden vom Server generiert, an den Client gesendet und dort ausgeführt. Die execute-Methode wird hierbei genauso wieder von den verschiedenen Commands verwendet, und kann eine eventuelle Antwort des Typs Command zurückliefern. Weiters wird in diesem Paket noch eine Factory auf Basis von Objektparametern implementiert.

2.6 Notifications

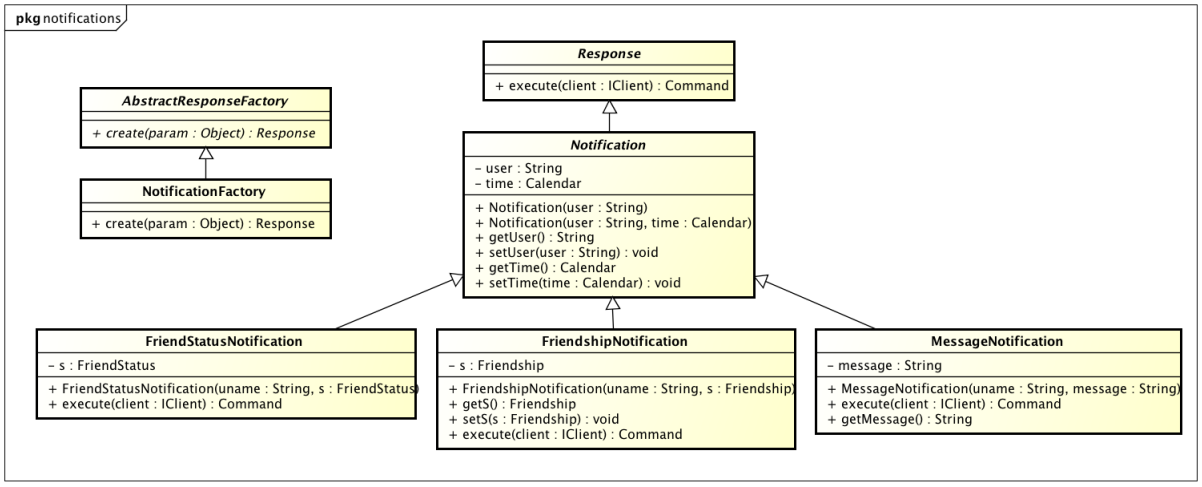

Abbildung 7: UML-Klassendiagramm Notifications-Paket

Eine Sonderform der Responses sind Notifications welche asynchron vom Server zur Benachrichtigung der Clients via Callback versendet werden. Diese Server-Client-Kommunikation ist dabei allgemein formuliert ein Response, weshalb Notification von Response erbt. Diese werden zusätzlich mit einer Timestamp sowie einem Usernamen definiert, da sich jede Notification auf einen speziellen User bezieht. Weiters wird in diesem Paket noch eine Factory auf Basis von Objektparametern implementiert.

2.7 Exceptions

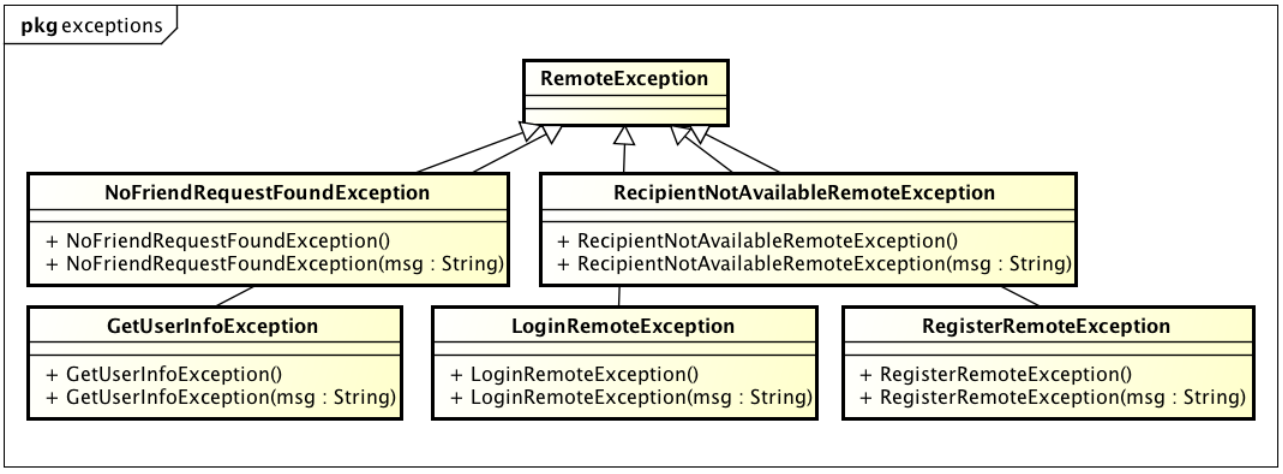

Abbildung 8: UML-Klassendiagramm Exceptions-Paket

3 Arbeitsaufteilung & Endzeitaufteilung

Pro Aufgabenpunkt wird Inline-/Protokolldokumentation sowie Testing inkludiert.

Aufgabe	**Person**	**Soll**	**Ist**
Grundgerüst	Timon	3h	9h
Callback	Timon	2h	4h
(Console-)Client	Herry	2h	4h
(Console-)Server	Herry	2h	4h
MessageHelper	Josef	1h	1h
StateDumper	Timon	2h	2h
MessengerProperties	Herry	2h	3h
Register/Unregister	Josef	2h	5h
Login/Logout/Stop	Gabriel	3h	3h
SendMessage/GetUserInfo	Josef	3h	5h
Request/Accept/Reject/Cancel Friendship	Gabriel	4h	4h
Notifications	Herry	1h	3h
SUMME Timon	Timon	7h	16h
SUMME Herry	Herry	7h	14h
SUMME Gabriel	Gabriel	7h	7h
SUMME Josef	Josef	6h	11h

Tabelle 1: Arbeitsaufteilung mit Aufwandsabschätzung & Endzeitaufteilung

4 Implementierung

4.1 Allgemein

4.1.1 Properties-Datei

Diverse Einstellungen werden über ein Properties-File eingelesen, welches zum Beispiel die Hostadresse beziehungsweise den Hostnamen, die Portnummer, den Objekt- beziehungsweise Servicenamen und für den Server enthält dieses Properties-File auch einen Pfad zum state-File, in welchem der Zustand des Servers gespeichert werden soll. Das Properties-File wird im src-Ordner abgelegt und durch den Ant-Compile Task in das build-Verzeichnis kopiert:

```
1 # the host name of the registry server
  registry.host=localhost
  # the port where the registry service is listening, replace it with a real value such as 12500
  registry.port=12345
```

```
# the name to be used for binding the server object
registry.object.name=messengerservice
# the file where to store the server's state
server.store.file=data/messenger.state
```

Listing 3: Properties File

Diese Eigenschaften werden mithilfe der Klasse **MessengerProperties** eingelesen, und danach dem Server und dem Client zur Verfügung gestellt.

Der in der Aufgabenstellung zur Verfügung gestellte Code war sehr hilfreich, und eigentlich eins zu eins übernehmbar. Folgende Zeilen sind zu beachten:

```
InputStream is = ClassLoader.getSystemResourceAsStream(PROPERTIESFILE);
```

Listing 4: PropertiesFile-Reader: ClassLoader

ClassLoader.getSystemResourceAsStream() sucht eine System-Ressource im Classpath und stellt dieser als Stream zur Verfügung.

4.2 Command

AcceptFriendCommand

Das AcceptFriendCommand soll eine bereits erzeugte Freundschaftsanfrage in eine Freundschaft konvertieren. Eine Freundschaftsanfrage besteht aus Verweisen in den Requestlisten der beiden Teilnehmer. Um nun die Freundschaft zu bestätigen werden die Verweise in eben genannten Listen gelöscht. Nur ein einziger Eintrag bei den Freundschaftsparteien wird neu erzeugt, dieser befindet sich in der Freundesliste. Nach einer vollständigen Umwandlung wird dem neu erworbenen Freund eine Notifikation gesendet.

CancelFriendCommand

Dieses Command beendet eine Freundschaft. Eine Freundschaft ist aus der Sicht des Programms ein gegenseitiger Eintrag in der Freundesliste des Freundes. Wenn eine Freundschaft beendet werden soll, wird einfach aus den zwei Freundeslisten die passenden Einträge entfernt.

FindUserCommand

Dieses Command sucht einen anderen User. Daazu muss der ausführende User einen Suchstring definieren. Alle User die in einem der folgenden: Usernamen, Vor- und Nachnamen, Hobbies, usw. mit dem Suchstring übereinstimmen werden zurückgegeben. In der Rückgabe befinden sich außerdem alle öffentlichen Informationen des Users zusammengefasst als String.

GetUserInfoCommand

Dieses Command gibt als Rückgabe alle öffentlichen Informationen eines Users, nach dem gesucht wird. Der aufrufende Sucher muss dazu nur den Usernamen des gesuchten Users angeben und bekommt all dessen öffentliche Informationen. Sowohl für dieses als auch das FindUserCommand wurde in der User-Klasse eine eigene Methode implementiert, die mittels eines Flags entweder alle öffentlichen oder alle öffentlichen und privaten Informatioenen eines Users ausgibt. Die privaten Informationen kann ein User dabei aber immer nur von sich selbst anzeigen lassen.

LoginCommand

Bei einem LoginCommand handelt es sich um die Authentifikation beim Server, und außerdem um den Austausch des Callback-Interfaces. Ein User ist dann authentifiziert, wenn sein Name und der Hash seines Passworts übereinstimmen. Ist dies geklärt wird noch das Callback-Objekt überprüft. Wenn alles stimmt wird nun am Server das Callback-Objekt gespeichert.

RegisterCommand

Das RegisterCommand meldet einen neuen Benutzer im System an. Nach der Angabe muss ein User mit folgenden Angaben registriert werden: <userName> <password> <firstName> <lastName> <birthday in dd.MM.yyyy> <profession> <emailAddress> hobby (mit , getrennt) Natürlich darf der Username mit dem man sich registrieren möchte noch nicht existieren. Gibt es einen Fehler bei der Registratur wird der Nutzer über das Auftreten einer RegisterRemoteException benachrichtigt. Für den Nutzer sieht es dabei nur so aus als ob die Registrierung nicht erfolgreich war.

RejectFriendCommand

Dieses Command soll eine bestehende Freundschaftsanfrage löschen ohne eine Freundschaft daraus zu erstellen. Eine Freundschaftsanfrage besteht aus Referenzen der User in den Requestlisten der jeweilig anderen Partei. Bei einer erfolgreichen Ausführung dieses Commands werden alle Verweise gelöscht, somit sind die beiden Benutzer wieder vollkommen ohne Verbindung, nachdem alle Einträge gelöscht worden sind wird der Anfrager noch über die missglückte Freundschaftsanfrage informiert.

SendMessageCommand

Bei dem SendMessageCommand ist die Aufgabe sehr simpel, einem User wird eine Nachricht übermittelt. Beim Übermitteln ist die Entscheidung des Servers wichtig. Diese Entscheidung ist je nachdem ob ein Callback-Objekt gespeichert wurde. Eine direkte Übermittlung kann nur passieren wenn ein solches Objekt momentan gespeichert ist. Wird kein Callback-Objekt gefunden muss die Nachricht in die Message-Queue des Empfängers übergeben werden.

StopCommand

Das StopCommand beendet die ClientSoftware. Dabei wird zuerst überprüft, ob der User, der seine Applikation beenden möchte, noch eingeloggt ist und loggt ihn gegebenenfalls unter Zuhilfenahme des LogOutCommands aus. Anschließend wird der ConsoleClient geschlossen, indem seine Methode auslaufen gelassen werden. Dieses Auslaufen lassen der Methoedn ist nicht im Command selbst, sondern im ConsoleClient implementiert.

UnregisterCommand

Dieses Command löscht einen registrierten User-Account wieder aus dem System. Dabei wird ebenfalls zuerst überprüft, ob der User, der gelöscht werden soll, noch eingeloggt ist. Ist dies der Fall wird er zuerst ausgeloggt. Dann wird aus dem Server der komplette Eintrag dieses Benutzers gelöscht und der Account existiert nicht mehr. Man kann sich mit diesem Account also auch nicht mehr einloggen.

4.3 Responses

Prinzipiell könnte jedes Command über den Rückgabewert eine Response als Antwort liefern. Dafür könnten nicht nur Ausgaben sondern auch komplexe clientseitige Verarbeitungen ausgefürt werden. In diesem Fall werden jedoch nur Textnachrichten versendet welche als StringRespone definiert wurden. Eine spezielle Form der Responses sind die Notifications welches sich ähnlcih den StringResponses mit Textnachrichten beschäftigen.

4.3.1 Notifications

Die meisten Operationen/Commands wie Login oder SendMsg welche am Server durchgeführt oder verarbeitet werden, führen zu Notifications, welche mithilfe vom Callback-Interface am Client ausgegeben werden können. Falls der Empfänger dieser Notifications offline sind, werden diese gespeichert und bei der nächsten Anmeldung des Users ausgegeben. Folgende Notifications können erfolgen:

- Friendship requested/accepted/rejected/cancelled
- Message from friend
- Friend went offline/online
- Friend unregistered

Jede Notification erweitert die abstrakte Klasse **Notification** welcher wiederum das Interface **Response** implementiert. Die abstrakte Klasse speichert ein String-Attribut- den Usernamen - und ein Calendar-Objekt - damit der Zeitpunkt der Notification festgehalten werden kann. Durch das Implementieren des Response-Interfaces erhält jede Notification eine **execute**-Methode, in welcher eine passende Ausgabe erfolgt.

Weiters wurde eine **NotificationFactory** implementiert, welche das Interface **AbstractResponseFactory** implementiert und die Erzeugung von den obigen 4 Notifications ermöglicht.

4.4 MessageHelper

Der MessageHelper hat einen simplen Zweck. Er soll einen Hash aus der Passworteingabe generieren, damit es nicht nötig sein soll eine Klartextübertragung des Passworts zu vollziehen. Bei diesem Programm wird ein SHA-1-Hash erzeugt. Dieser wird mit einem byte-Array generiert. Wenn der Hash generiert worden ist, wird das resultierende byte-Array wieder in ein String umgewandelt.

4.5 Server

4.5.1 Allgemein

Der Server liest im Konstruktor ein Zustandsfile ein und liest den vorherigen Status aus, um den letzten Zustand wiederherzustellen. Wenn nötig wird eine ClientList initialisiert und

danach werden Response und NotificationFactories initialisiert. Mithilfe der Parameter im PropertiesFile wird eine Registry erzeugt, wenn nicht schon vorhanden. Wenn die Registry erzeugt wird, kann man sich mithilfe der statischen Methode **Naming.rebind()** in dieser eintragen.

```
String registryURL = "rmi://" + host + ":" + port + "/" + name;
Registry registry = null;
try {
    registry = LocateRegistry.getRegistry(port);
    // Dieser Aufruft wirft eine Exception, falls die Registry nicht bereits existiert
    registry.list();
} catch (RemoteException ex) {
    // Keine Registry am Port vorhanden:
    registry = LocateRegistry.createRegistry(port);
}
Naming.rebind(registryURL, this);
```

Listing 5: Server-Registry

Da der Server das Remote-Interface IServer implementiert, implementiert er auch die Methode doCommand(), welche natürlich auch eine RemoteException werfen kann.

Beim Beenden des Servers, werden alle User nicht erreichbar gestellt und anschließend der Zustand im Zustandsfile gespeichert. Danach wird der Server aus der Registry ausgetragen und als Remote-Objekt unerreichbar gemacht.

4.5.2 StateDumper

Zum Absichern der Userdaten, falls der Server beendet werden sollte, bietet der StateDumper zwei statische Methoden zum Persistieren und Laden der Userdatenliste. Hierfür werden die Daten via ObjectSteams serialisiert und in eine Datei geschrieben bzw. daraus geladen.

4.6 Client

4.6.1 Allgemein

Der Client-Konstruktor intialisiert die TextCommandFactory, CommandFactory und die MessengerProperties-Klasse. Beim Starten des Clients werden die Eigenschaften aus den Properties-File hinausgelesen und als Attribute abgespeichert. Mit diesen Attributen verbindet sich der Client mithilfe von Naming zum Server.

Vorsicht: Das Objekt, welches von der Registry geliefert wird, kann nur als Remote-Interface gecastet und verwendet werden!

```
registryURL = "rmi://" + host + ":" + port + "/" + name;
server = (IServer) Naming.lookup(registryURL);
```

Listing 6: Client-Registry

Je nach eingehende Commands vom User-Interface werden im Client auch bestimmte Maßnahmen vorgenommen, wie das Beenden der Eingabe bei einem StopCommand, und das Speichern des Usernames und Passworthashes im Client bei einem LoginCommand etc.
Durch das Implementieren des Remote-Interfaces, implementiert auch der Client eine doResponse() Methode, welche passende Responses beziehungsweise Notifications verarbeiten kann.

5 Testbericht

Die Implementierten Testfälle können wie folgt via Ant ausgeführt werden:

```
Buildfile: rmimessenger/Abgabe/build.xml
compile:
test-server:
    [javac] rmimessenger/Abgabe/build.xml:39: warning: 'includeantruntime' was not set,
        defaulting to build.sysclasspath=last; set to false for repeatable builds
    [junit] Running FindUserCommandTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,561 sec
    [junit] Running GetUserInfoCommandTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,005 sec
    [junit] Running LogOutCommandTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,004 sec
    [junit] Running LoginCommandTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,01 sec
    [junit] Running MessengerHelperTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0,017 sec
    [junit] Running NotificationFactoryTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0,009 sec
    [junit] Running RegisterCommandTest
    [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0,01 sec
    [junit] Running SendMessageCommandTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,003 sec
    [junit] Running StateDumperTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,089 sec
    [junit] Running UnregisterCommandTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,011 sec

BUILD SUCCESSFUL
Total time: 1 second
```

Listing 7: Testsuite

Literatur

[1] Michael Borko. Task08 - rmi messenger, 2012. `http://elearning.tgm.ac.at/mod/assign/view.php?id=10423`.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings