**LK Q-II.3 Entwicklung von Client-Server-Systemen**

**2) Analyse, Modellierung und Implementierung von Netzwerkanwendungen in Client-Server-Strukturen**

**Vorbemerkung**

Für diese Unterrichtseinheit werden die Netzwerklassen Connection, Client und Server benötigt, die vom Land NRW auch beim Zentralabitur zugrunde gelegt werden. Darüber hinaus werden die Klassen Queue und List verwendet. Das Gesamtpaket der „Abiturklassen“ kann unter folgender URL heruntergeladen werden:

<https://www.schulentwicklung.nrw.de/lehrplaene/lehrplannavigator-s-ii/gymnasiale-oberstufe/informatik/hinweise-und-beispiele/index.html>

**Gliederung**

(a) Nutzung einfacher Server-Dienste mittels Client

Nutzung von Server-Diensten mithilfe der Klasse Connection

* DAYTIME-Dienst
* QODT-Dienst

Nutzung von Client-Diensten mithilfe der Klasse Client

* Echo-Dienst

(b) Anbieten von Diensten mittels Server

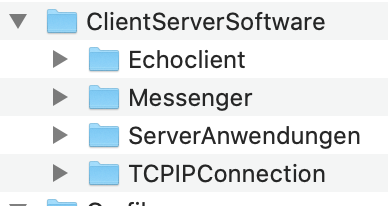
Anbieten von Server-Diensten mithilfe der Klasse Server

* Echo-Server

3) Entwicklung eines vollständigen Client-Server-Systems

Modellierung und Implementierung eines Messenger-Dienstes

* Protokoll
* Messenger-Server
* Messenger-Client



**Software**

Alle hier entwickelten bzw. erläuterten Klassen einschließlich der Netzklassen sind in den Download-Materialien als BlueJ-Projekte enthalten.

(a) Nutzung einfacher Server-Dienste mittels Client

**Nutzung von Server-Diensten mithilfe der Klasse Connection**

**Dokumentation der Klasse Connection**

Objekte der Klasse Connection ermöglichen eine Netzwerkverbindung zu einem Server mittels TCP/IP-Protokoll. Nach Verbindungsaufbau können Zeichenketten (Strings) zum Server gesendet und von diesem empfangen werden. Zur Vereinfachung geschieht dies zeilenweise, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfang wird dieser entfernt. Es findet nur eine rudimentäre Fehlerbehandlung statt, so dass z.B. der Zugriff auf unterbrochene oder bereits getrennte Verbindungen nicht zu einem Programmabbruch führt. Eine einmal getrennte Verbindung kann nicht reaktiviert werden.

* **Connection(String pServerIP, int pServerPort)**Ein Objekt der Klasse Connection wird erstellt. Dadurch wird eine Verbindung zum durch pServerIP und pServerPort spezifizierten Server aufgebaut, so dass Daten (Zeichenketten) gesendet und empfangen werden können. Kann die Verbindung nicht hergestellt werden, kann die Instanz von Connection nicht mehr verwendet werden.
* **void send(String pMessage)**Die Nachricht pMessage wird – um einen Zeilentrenner ergänzt – an den Server gesendet. Schlägt der Versand fehl, geschieht nichts.
* **String receive()**Es wird beliebig lange auf eine eingehende Nachricht vom Server gewartet und diese Nachricht anschließend zurückgegeben. Der vom Server angehängte Zeilentrenner wird zuvor entfernt. Während des Wartens ist der ausführende Prozess blockiert. Wurde die Verbindung unterbrochen oder durch den Server unvermittelt geschlossen, wird null zurückgegeben.
* **void close()**Die Verbindung zum Server wird getrennt und kann nicht mehr verwendet werden. War die Verbindung bereits getrennt, geschieht nichts.

Ein Objekt der folgenden Java-Klasse TCPIPConnection stellt unter Nutzung der Klasse Connection eine Verbindung zu einem Server her. Die IP-Adresse oder die URL des Servers und der Port, auf dem die Verbindung erfolgen soll, werden als Parameter an den Konstruktor übergeben.

|  |  |
| --- | --- |
| 1 | import javax.swing.JOptionPane; |
| 2 | public class TCPIPConnection extends Connection { |
|  |  |
| 3 | public TCPIPConnection(String pServerIP, int pServerPort) { |
| 4 | super(pServerIP, pServerPort); |
| 5 | } |
|  |  |
| 6 | public void servernachrichtAusgeben() { |
| 7 | JOptionPane.showMessageDialog(null,"Server sendet:\n"  +receive()); |
| 8 | } |
| 9 | } |

Die Methode showMessageDialog der Java-Klasse JOptionPane zeigt ein Fenster mit der als Parameter übergebenen Textnachricht an.

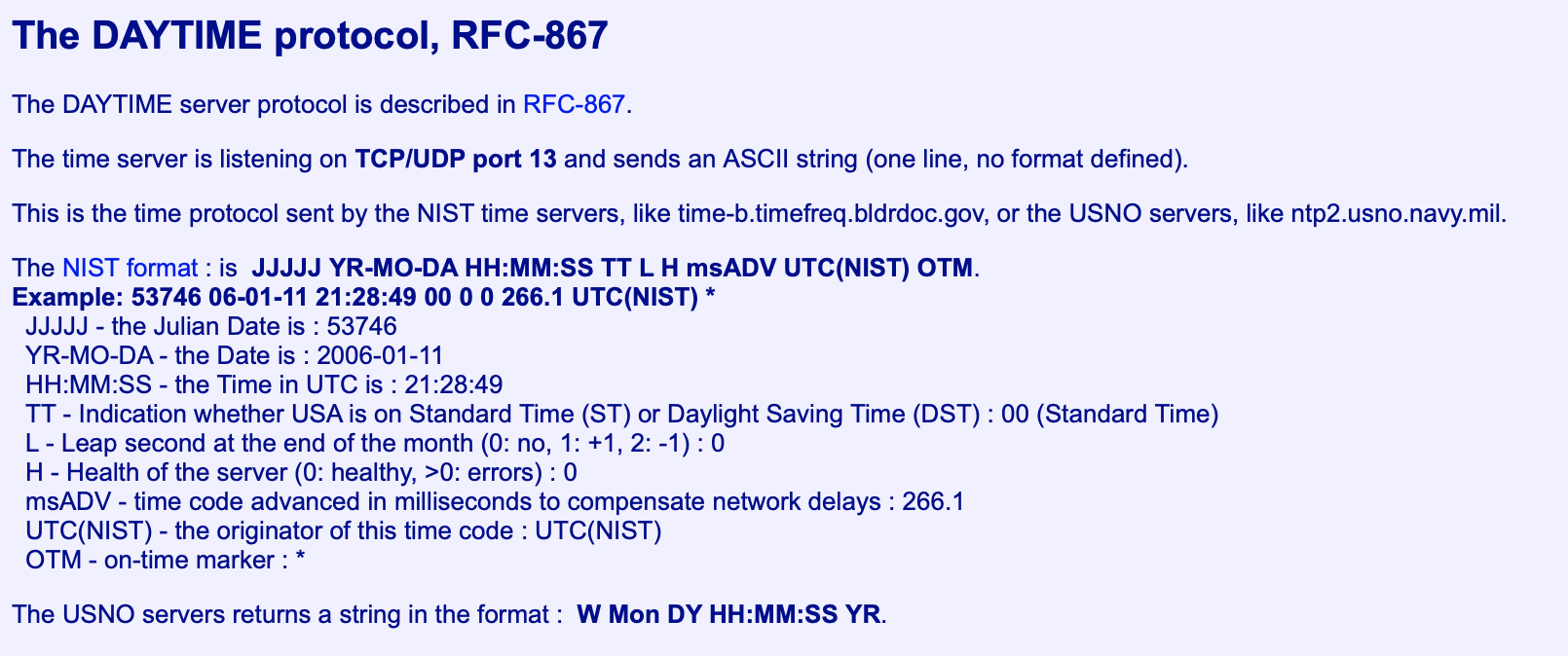
Verschiedene öffentliche Server stellen Dienste zur Verfügung, die mithilfe einer Client-Software genutzt werden können.

**Der Daytime-Dienst**

Der *Daytime*-Dienst sendet auf Anfrage das aktuelle Datum mit der aktuellen Uhrzeit. Amerikanische *Daytime*-Server senden, im Gegensatz zu ihren europäischen Pendants, vor der Übertragung von Uhrzeit und Datum noch eine einzelne Leerzeile. Da die Nachrichtenübertragung in den Netzwerkklassen von QUA-LiS NRW zeilenweise stattfinden, müssen damit konstruierte *Daytime*-Clients also in der Lage sein, ggf. zwei Nachrichten zu empfangen.

Der *Daytime*-Dienst läuft nach folgendem Protokoll ab.

|  |  |
| --- | --- |
| **Client an Server** | **Server an Client** |
| *Verbindungaufnahme* | *\*Optional:\** Leerzeile   Datum und Uhrzeit.  *Server trennt die Verbindung.* |

Der *Daytime*-Dienst läuft gemäß [RFC 867](https://tools.ietf.org/html/rfc867) im Allgemeinen auf Port 13.

Man stellt eine Verbindung zu dem im RFC-867 angegebenen Daytime-Server „time-b.timefreq.bldrdoc.gov“ mit BlueJ her, indem man ein Objekt der Klasse TCPIPConnection erzeugt, wie in Abbildung 1 zu sehen.

Nach Aufruf der Methode receive der Klasse Connection (s. Abbildung 2) wird zunächst eine Leerzeile zurückgegeben (s. Abbildung 3).

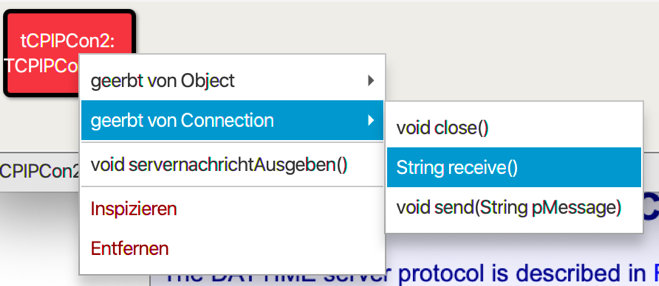


Abbildung 2

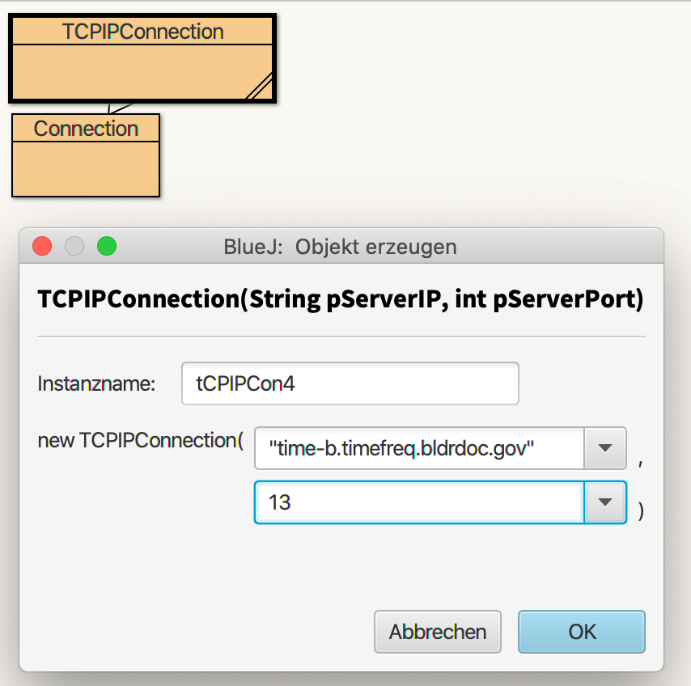


Abbildung 1

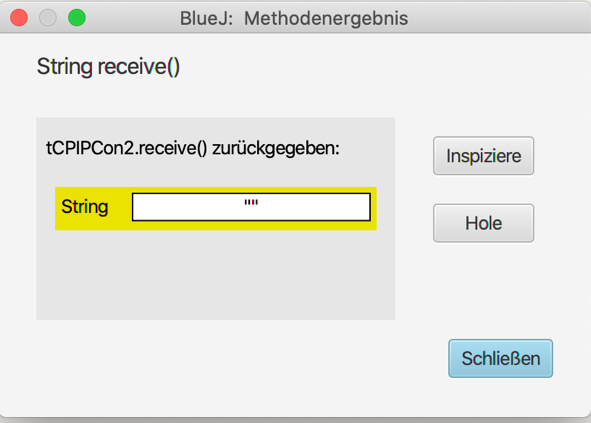


Abbildung 3

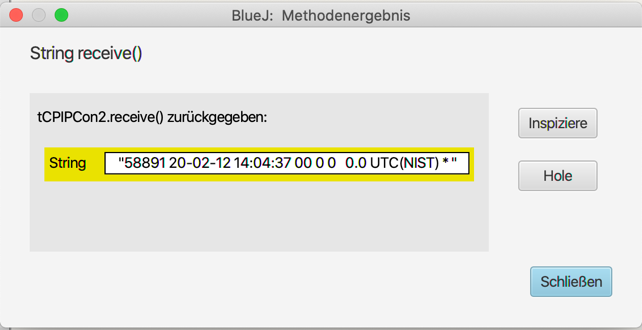


Abbildung 4

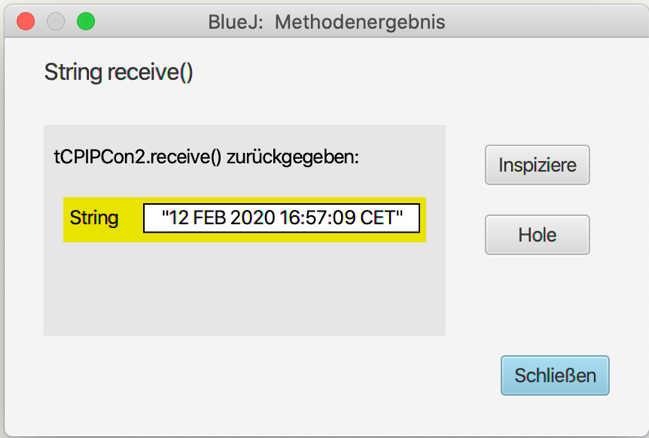
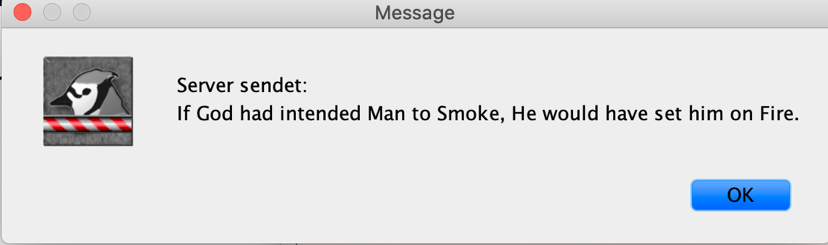


Abbildung 5

Nach erneutem Aufruf dieser Methode werden das aktuelles Datum und Zeit in dem in RFC-867 angezeigten Format angezeigt (s. Abbildung 4).

Verbindet man sich mit dem Daytime-Server der Freien Univer­sität Berlin

„time.fu-berlin.de“, erhält man direkt ohne Leerzeile das Tagesdatum mit der aktuellen mittel­europäischen Zeit in lesbarer Form (s. Abbildung 5).

**Der QODT-Dienst**

QODT-Server senden nach der Verbindung mit einem Client täglich ein neues Zitat (quote oft the day). Der QODT-Dienst arbeitet auf Port 17. Eine Liste von QODT-Servern findet man auf der Website <https://wiki.freepascal.org/QOTD>. Das Zitat in Abbildung 6 wurde vom Server „alpha.mike-r.com“ nach Aufruf folgender Main-Methode gesendet.

Abbildung 6

public static void main(String[] args) {

TCPIPConnection tcpipc = new TCPIPConnection("alpha.mike-r.com",17);

tcpipc.servernachrichtAusgeben();

}

**Nutzung von Client-Diensten mithilfe der Klasse Client**

Wenn ein Server keine Nachricht sendet, weil z.B. eine falsche Adresse oder ein falscher Port vom Client gesendet wurde, wird das Programm in der Methode receive der Klasse Connection blockiert. Um dies zu verhindern, wird im Folgenden die Klasse Client verwendet, die in einem nebenläufigen Prozess nur dann reagiert, wenn der verbundene Server eine Nachricht sendet.

**Dokumentation der Klasse Client**

Objekte von Unterklassen der abstrakten Klasse Client ermöglichen Netzwerkverbindungen zu einem Server mittels TCP/IP-Protokoll. Nach Verbindungsaufbau können Zeichenketten (Strings) zum Server gesendet und von diesem empfangen werden, wobei der Nachrichtenempfang nebenläufig geschieht. Zur Vereinfachung finden Nachrichtenversand und -empfang zeilenweise statt, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfang wird dieser entfernt. Jede empfangene Nachricht wird einer Ereignisbehandlungsmethode übergeben, die in Unterklassen implementiert werden muss. Es findet nur eine rudimentäre Fehlerbehandlung statt, so dass z.B. Verbindungsabbrüche nicht zu einem Programmabbruch führen. Eine einmal unterbrochene oder getrennte Verbindung kann nicht reaktiviert werden.

* **Client(String pServerIP, int pServerPort)**Es wird eine Verbindung zum durch pServerIP und pServerPort spezifizierten Server aufgebaut, so dass Daten (Zeichenketten) gesendet und empfangen werden können. Kann die Verbindung nicht hergestellt werden, kann der Client nicht zum Datenaustausch verwendet werden.
* **boolean isConnected()**Die Anfrage liefert den Wert true, wenn der Client mit dem Server aktuell verbunden ist. Ansonsten liefert sie den Wert false.
* **void send(String pMessage)**Die Nachricht pMessage wird – um einen Zeilentrenner ergänzt – an den Server gesendet. Schlägt der Versand fehl, geschieht nichts
* **void close()**Die Verbindung zum Server wird getrennt und der Client kann nicht mehr verwendet werden. Ist der Client bereits vor Aufruf der Methode in diesem Zustand, geschieht nichts.
* **void processMessage(String pMessage)**Diese Methode wird aufgerufen, wenn der Client die Nachricht pMessage vom Server empfangen hat. Der vom Server ergänzte Zeilentrenner wurde zuvor entfernt. Die Methode ist abstrakt und muss in einer Unterklasse der Klasse Client überschrieben werden, so dass auf den Empfang der Nachricht reagiert wird. Der Aufruf der Methode erfolgt nicht synchronisiert.

Das folgende Java-Programm ruft den Daytime-Dienst des Daytime-Servers der FU Berlin auf.

|  |  |
| --- | --- |
| 1 | import javax.swing.JOptionPane; |
| 2 | import netzklassen.Client; |
|  |  |
| 3 | public class DaytimeClient extends Client { |
|  |  |
| 4 | public DaytimeClient(String pServerIP) { |
| 5 | super(pServerIP, 13); |
| 6 | } |
|  |  |
| 7 | public void processMessage(String pMessage) { |
| 8 | JOptionPane.showMessageDialog(null,"Server sendet:\n"  + pMessage); |
| 9 | } |
|  |  |
| 10 | public static void main(String[] args) { |
| 11 | DaytimeClient dtc = new DaytimeClient("time.fu-berlin.de"); |
| 12 | } |
|  |  |
| 13 | } |

**Erläuterung:**

Die Klasse DaytimeClient ist eine Unterklasse der Klasse Client. Die ab­st­rak­te Methoden processMessage der Klasse Client muss überschrieben werden.

Die Methode processMessage (Zeilen 7 – 9) öffnet ein Nachrichtenfenster mit der Anzeige der vom Server empfangenen Nachricht. Die Methode wird automatisch aufgerufen, sobald eine Nachricht vom Server eintrifft.

In der Main-Methode wird ein Objekt der Klasse DaytimeClient erzeugt, um Datum und Zeit abzufragen.

**Aufgabe**

Implementieren Sie einen QOTD-Client mithilfe der Klasse Client.

**Der Echo-Dienst**

Ein weiterer Dienst, den öffentliche Server zur Verfügung stellen, ist der Echo-Dienst. Wenn ein Client, nachdem er eine Verbindung zum Server aufgebaut hat, eine beliebige Textzeile an den Echo-Server sendet, wird diese unverändert vom Server zurückgesandt. Die Verbindung wird anschließend vom Server nicht getrennt. Das Echo-Protokoll ist in RFC 862 spezifiziert. Port 7 ist für den Echo-Dienst reserviert.

**Protokoll für den Echo-Dienst**

|  |  |
| --- | --- |
| **Client an Server** | **Server an Client** |
| *Verbindungaufnahme*  *Textzeile* | *empfangene Textzeile* |

Das folgende Listing zeigt die Implementation eines Echo-Clients in Java.

|  |  |
| --- | --- |
| 1 | import javax.swing.JOptionPane; |
| 2 | import Client; |
|  |  |
| 3 | public class EchoClient extends Client { |
|  |  |
| 4 | public EchoClient(String pServerIP, int pServerPort) { |
| 5 | super(pServerIP, pServerPort); |
| 6 | } |
|  |  |
| 7 | public void processMessage(String pMessage) { |
| 8 | JOptionPane.showMessageDialog(null,"Server sendet:\n"  + pMessage); |
| 9 | } |
|  |  |
| 10 | public void sendMessageToServer() { |
| 11 | String lMessage; |
| 12 | lMessage = JOptionPane.showInputDialog("Bitte geben Sie eine  neue Nachricht ein:"); |
| 13 | send(lMessage); |
| 14 | } |
|  |  |
| 15 | } |

Die Methode sendMessageToServer() (Zeilen 10 – 14) fordert den Nutzer in einem modalen Java-Input-Dialog zur Eingabe einer Nachricht auf und sendet diese an den Server. Zum Versenden der Nachricht wird die Methode send(pMessage: String) der Oberklasse Client verwendet.

Die Methode processMessage (Zeilen 7 – 9) zeigt die Antwort des Echo-Servers in einem Message-Dialog an.

Um das Programm zu testen, sollte ein Echoserver auf einem der Schulrechner gestartet werden.

Da der Konstruktor Verbindungen zu einem beliebigen Server, dessen IP-Adresse bekannt ist, auf einem beliebigen Port aufbauen kann, kann das Clientprogramm universell für die Kommunikation zwischen einem Client und einem beliebigen Server verwendet werden.

(b) Anbieten von Diensten mittels Server

* Analyse vorgegebener Implementationen einfacher Server
* Modellierung und Implementierung eigener Server

Mithilfe der Java-Klasse Server lassen sich sehr einfach Server implementieren, d.h. Programme, die im Netzwerk Dienste für Clients zur Verfügung stellen.

**Dokumentation der Klasse Server**

Objekte von Unterklassen der abstrakten Klasse Server ermöglichen das Anbieten von Serverdiensten, so dass Clients Verbindungen zum Server mittels TCP/IP-Protokoll aufbauen können. Zur Vereinfachung finden Nachrichtenversand und -empfang zeilenweise statt, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfang wird dieser entfernt. Verbindungsannahme, Nachrichtenempfang und Verbindungsende geschehen nebenläufig. Auf diese Ereignisse muss durch Überschreiben der entsprechenden Ereignisbehandlungsmethoden reagiert werden. Es findet nur eine rudimentäre Fehlerbehandlung statt, so dass z.B. Verbindungsabbrüche nicht zu einem Programmabbruch führen. Einmal unterbrochene oder getrennte Verbindungen können nicht reaktiviert werden.

**Server(int pPort)**  
Ein Objekt der Klasse Server wird erstellt, das über die angegebene Portnummer einen Dienst anbietet an. Clients können sich mit dem Server verbinden, so dass Daten (Zeichenketten) zu diesen gesendet und von diesen empfangen werden können. Kann der Server unter der angegebenen Portnummer keinen Dienst anbieten (z.B. weil die Portnummer bereits belegt ist), ist keine Verbindungsaufnahme zum Server und kein Datenaustausch möglich.

**boolean isOpen()**  
Die Anfrage liefert den Wert true, wenn der Server auf Port pPort einen Dienst anbietet. Ansonsten liefert die Methode den Wert false.

**boolean isConnectedTo(String pClientIP, int pClientPort)**  
Die Anfrage liefert den Wert true, wenn der Server auf Port pPort einen Dienst anbietet. Ansonsten liefert die Methode den Wert false.

**void send(String pClientIP, int pClientPort, String pMessage)**  
Die Nachricht pMessage wird – um einen Zeilentrenner erweitert – an den durch pClientIP und pClientPort spezifizierten Client gesendet. Schlägt der Versand fehl, geschieht nichts.

**void sendToAll(String pMessage)**  
Die Nachricht pMessage wird – um einen Zeilentrenner erweitert – an alle mit dem Server verbundenen Clients gesendet. Schlägt der Versand an einen Client fehl, wird dieser Client übersprungen.

**void closeConnection(String pClientIP, int pClientPort)**Die Verbindung des Servers zu dem durch pClientIP und pClientPort spezifizierten Client wird getrennt. Zuvor wird die Methode processClosingConnection mit IP-Adresse und Port des jeweiligen Clients aufgerufen. Ist der Server nicht mit dem in der Parameterliste spezifizierten Client verbunden, geschieht nichts.

**void close()**Alle bestehenden Verbindungen zu Clients werden getrennt und der Server kann nicht mehr verwendet werden. Ist der Server bereits vor Aufruf der Methode in diesem Zustand, geschieht nichts.

**void processNewConnection(String pClientIP, int pClientPort)**Diese Ereignisbehandlungsmethode wird aufgerufen, wenn sich ein Client mit IP-Adresse pClientIP und Portnummer pClientPort mit dem Server verbunden hat. Die Methode ist abstrakt und muss in einer Unterklasse der Klasse Server überschrieben werden, so dass auf den Neuaufbau der Verbindung reagiert wird. Der Aufruf der Methode erfolgt nicht synchronisiert.

**void processMessage(String pClientIP, int pClientPort, String pMessage)**  
Diese Ereignisbehandlungsmethode wird aufgerufen, wenn der Server die Nachricht pMessage von dem durch pClientIP und pClientPort spezifizierten Client empfangen hat. Der vom Client hinzugefügte Zeilentrenner wurde zuvor entfernt. Die Methode ist abstrakt und muss in einer Unterklasse der Klasse Server überschrieben werden, so dass auf den Empfang der Nachricht reagiert wird. Der Aufruf der Methode erfolgt nicht synchronisiert.

**void processClosingConnection(String pClientIP, int pClientPort)**Sofern der Server die Verbindung zu dem durch pClientIP und

pClientPort spezifizierten Client trennt, wird diese Ereignisbehandlungsmethode aufgerufen, unmittelbar bevor die Verbindungstrennung tatsächlich erfolgt. Wird die Verbindung unvermittelt unterbrochen oder hat der in der Parameterliste spezifizierte Client die Verbindung zum Server unvermittelt getrennt, erfolgt der Methodenaufruf nach der Unterbrechung/Trennung der Verbindung. Die Methode ist abstrakt und muss in einer Unterklasse der Klasse Server überschrieben werden, so dass auf das Ende der Verbindung zum angegebenen Client reagiert wird. Der Aufruf der Methode erfolgt nicht synchronisiert.

**Hinweis:** Bei denfolgenden Beispielen enthält das Paket netzklassen, die „Abiturklassen“ Server, Client, Connection und List. Diese müssen importiert werden, wenn sie verwendet werden sollen.

**Beispiel:** Echoserver

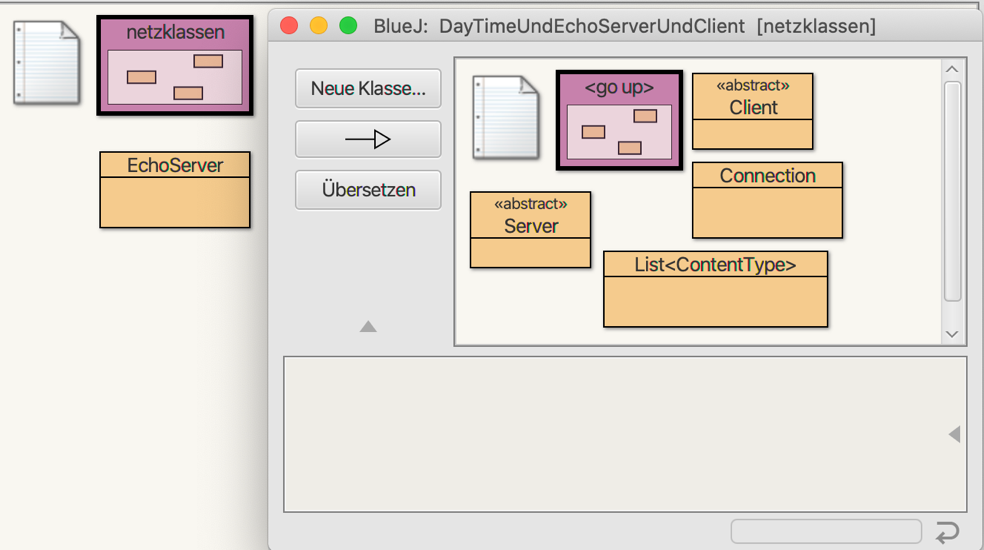


Abbildung 7: Java-Projekt Echoserver: BlueJ-Oberfläche

|  |  |
| --- | --- |
| 1 | import netzklassen.Server; |
| 2 | public class EchoServer extends Server { |
|  |  |
| 3 | public EchoServer() { |
| 4 | super(7); |
| 5 | System.out.println("Server gestartet"); |
| 6 | } |
|  |  |
| 7 | public void processNewConnection(String pClientIP,  int pClientPort) { |
| 8 | } |
|  |  |
| 9 | public void processMessage(String pClientIP, int pClientPort, |
|  | String pMessage) { |
| 10 | send(pClientIP, pClientPort, pMessage); |
| 11 | } |
|  |  |
| 12 | public void processClosingConnection(String pClientIP,  int pClientPort) { |
| 13 | } |
|  |  |
| 14 | } |

**Erläuterung**

Die Klasse Echoserver importiert die Klasse Server aus dem Paket netzklassen (Zeile 1). Echoserver ist eine Unterklasse der abstrakten Klasse Server (Zeile 2). Der Konstruktor ruft den Konstruktor der Klasse Server auf mit dem Port, auf dem der Dienst stattfindet (Zeile 4). Beim Echoserver ist das standardmäßig Port 7. Damit der Benutzer weiß, dass der Server erfolgreich gestartet wurde, wird eine entsprechende Meldung in die Konsole geschrieben.

Die drei abstrakten Methoden processNewConnection, processMessage und processClosingConnection müssen überschrieben werden. Sie werden automatisch ausgeführt, sobald sich ein Client angemeldet hat, bzw. eine Nachricht von einem Client beim Server eingegangen ist, bzw. sich ein Client abgemeldet hat. Der Client ist vom Server über die jeweils als Parameter übergebene IP-Adresse und den Port des Clients eindeutig zu identifizieren.

Beim Echoserver enthält lediglich die Methode processMessage eine Anweisung. Der Server sendet mit der Methode send an den Client den empfangenen Text zurück (Zeile 10).

Der Server startet, sobald ein Objekt der Klasse EchoServer generiert wird. Wenn mit dem weiter oben beschriebenen Programm EchoClient eine Verbindung auf Port 7 zum Echoserver aufgebaut wird und eine Nachricht an den Server versendet wird, schickt der Server diese unverändert zurück. Wenn Server und Client auf demselben Rechner ausgeführt werden, erfolgt die Verbindung über die IP-Adresse „127.0.0.1“ ansonsten über die IP-Adresse des Rechners, auf dem der Server läuft.

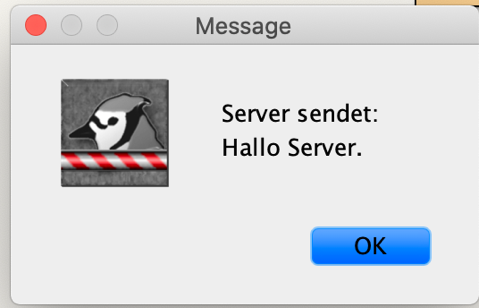


Abbildung 9: Nachricht des Servers



Abbildung 8: Nachricht an den Server

**Aufgabe**

a) Entwickeln Sie einen Daytime-Server.

b) Entwickeln Sie einen QOTD-Server, der aus einer Liste mit Zitaten berühmter Persönlichkeiten zufällig das „Zitat des Tages“ an einen QOTD-Client sendet.

Lösung:

Die Lösungen finden Sie in den Download-Materialien.

**3) Entwicklung eines vollständigen Client-Server-Systems**

In diesem Abschnitt werden die Modellierung und Implementierung eines Messengerdienstes erläutert. Mit einem Messengerdienst können mehrere Personen an verschiedenen Rechnern Textnachrichten miteinander austauschen. Die Kommunikation erfolgt über einen Server, bei dem sich die Teilnehmer anmelden. Nachrichten werden von den Teilnehmern unter Angabe des gewünschten Empfängers an den Server geschickt und von diesem an den Adressaten weitergeleitet.

Die folgende Abbildung zeigt die Kommunikation zwischen zwei angemeldeten Teilnehmern.

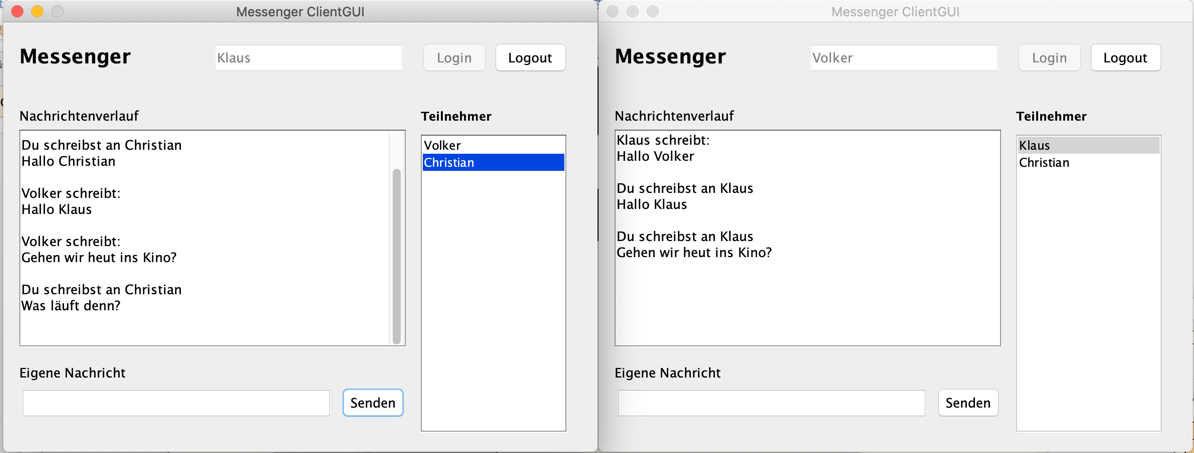


Abbildung 10: Messenger-Clients

**Funktionalität des Messenger-Dienstes**

Voraussetzung: Der Messenger-Server ist gestartet.

Ein Messenger-Client verbindet sich nach dem Start automatisch mit dem Messenger-Server. Nach Bestätigung der Verbindung durch den Server gibt der Teilnehmer einen beliebigen Benutzernamen in das Namensfeld ein und meldet sich durch Klick auf den Button „Login“ beim Messenger-Dienst an. IP-Adresse und Port für den Dienst sind als Konstanten in der Client-Software gespeichert. Da der Server die Messenger-Teilnehmer nicht speichert und auch kein Passwort eingegeben werden muss, findet keine Authentifizierung statt. Der Server verwaltet die angemeldeten Teilnehmer mit Namen, IP-Adressen und Ports in einer Liste.

Nach erfolgreicher Anmeldung erhält der Client vom Server eine Bestätigungsnachricht. Er fordert eine Liste aller angemeldeten Teilnehmer beim Server an und veranlasst die Weitergabe seines Nicknames durch den Server an die anderen angemel­deten Teilnehmer. Die Liste der angemeldeten Teilnehmerinnen und Teilnehmer wird in der Benutzungsoberfläche des Clients angezeigt.

Will ein Teilnehmer einem anderen angemeldeten Teilnehmer eine Nachricht schi­cken, geht er wie folgt vor:

1. Er schreibt den Text der Nachricht in das Eingabefeld „Eigene Nachricht“.

2. Er wählt den Empfänger in der Teilnehmerliste mit der Maus aus.

3. Er drückt den Button „senden“.

Sowohl gesendete wie auch empfangene Nachrichten werden in dem Textbereich „Nachrichtenverlauf“ angezeigt.

In der ersten Programmfassung gibt es folgende Einschränkung:

Eine Nachricht kann nur an einen Teilnehmer geschickt werden.

Die Bearbeitung des Projektes erfolgt in folgenden Schritten.

1. Entwicklung des Protokolls
2. Entwicklung des Servers

2.1 Modellierung

2.2 Implementierung

3. Entwicklung des Clients

3.1 Vorgabe der Benutzungsoberfläche

3.2 Modellierung

3.3 Implementierung

4.Test

1. **Entwicklung des Protokolls**

Die folgende Tabelle enthält eine Zusammenstellung der Nachrichten, die von einem Client an den Server und vom Server an einen oder alle Clients gesendet werden.

|  |  |
| --- | --- |
| **Client an Server** | **Server an Client** |
| *Verbindung aufnehmen* | *Verbindung bestätigen (SC\_WK)* |
| *Einloggen <Name> (CS\_AN)* | *Anmeldung bestätigen*  *(SC\_AO)*  oder  *Fehler: Name bereits vergeben (SC\_ER)*  oder  *Fehler: Teilnehmer bereits angemeldet (SC\_ER)* |
| *Teilnehmerliste anfordern (CS\_GA)* | *Teilnehmerliste <Namen aller Teilnehmer> (SC\_AT)*  oder  *Fehler: Sender nicht angemeldet*  *(SC\_ER)* |
| *Sende meinen Namen an alle Teilnehmer (CS\_NA)* | *Name des Senders wird an alle Teilnehmer gesendet <Name> (SC\_ZU)*  oder  *Fehler: Sender nicht angemeldet*  *(SC\_ER)* |
| *Text senden <Empfängername><Text> (CS\_TX)* | *Text senden an Empfänger <Absendername><Text>*  *(SC\_TX)*  oder  *Fehler: Sender nicht angemeldet*  *(SC\_ER)*  *oder*  *Fehler: Empfänger nicht gefunden*  *(SC\_ER)* |
| *Abmelden (CS\_AB)* | *Abmeldebestätigung (SC\_BY)*  *Abgemeldeten Teilnehmer an alle senden (SC\_AB)*  oder  *Fehler: Sender nicht angemeldet*  *(SC\_ER)* |

**Protokoll formalisiert: Server an Client**

|  |  |  |  |
| --- | --- | --- | --- |
| **Kurzbezeichnung** | **Servernachricht** | **Empfänger** | **Parameter** |
| SC\_WK | +WILLKOMMEN | Sender |  |
| SC\_AO | +ANMELDUNG\_OK | Sender |  |
| SC\_ZU | +ZUGANG | Alle | Nach Trennzeichen wird der Name des neu angemeldeten Teilnehmers angehängt |
| SC\_AT | +ALLE\_TEILNEHMER | Sender | Nach Trennzeichen werden Namen aller Teilnehmer, getrennt durch Trennzeichen, angehängt |
| SC\_TX | +TEXT | Adressat | Nach Trennzeichen werden der Name des Absenders und die Nachricht, getrennt durch Trennzeichen, angehängt |
| SC\_BY | +BYE | Sender |  |
| SC\_AB | +ABGANG | Alle | Nach Trennzeichen wird der Name des abgemeldeten Teilnehmers angehängt |
| SC\_ER | -FEHLER | Sender | Nach Trennzeichen wird die Fehlerursache angehängt |

**Protokoll formalisiert: Client an Server**

|  |  |  |
| --- | --- | --- |
| **Kurzbezeichnung** | **Clientnachricht** | **Parameter** |
| CS\_AN | +ANMELDEN | Nach Trennzeichen wird der Teilnehmername angehängt |
| CS\_GA | +GIB\_ALLE\_TN |  |
| CS\_NA | +SENDE\_NAME\_AN\_ALLE |  |
| CS\_TX | +TEXT | Nach Trennzeichen werden der Name des Empfängers und die Nachricht, getrennt durch Trennzeichen, angehängt |
| CS\_AB | +ABMELDEN |  |

Als Trennzeichen wird „:“ verwendet.

**2. Entwicklung des Servers**

**2.1 Modellierung**

Es muss eine Klasse MessengerServer als Unterklasse der Klasse Server implementiert werden. Die Klasse MessengerServer überschreibt die abstrakten Methoden processNewConnection, processMessage und

processClosingConnection der Klasse Server.

Wenn sich ein Client mit dem Server verbindet, wird die Methode

processNewConnection aufgerufen. In dieser Methode sendet der Server die Nachricht: +WILLKOMMEN

Geht eine Nachricht eines Clients ein, wird vom Serverobjekt automatisch die Methode processMessage aufgerufen. Die folgende Übersicht zeigt für alle Clientnachrichten an, welche Aktionen der Server durchzuführen hat und welche Nachrichten er sendet. Um die Übersichtlichkeit zu wahren, wurden die meisten Fehlerfälle nicht berücksichtigt.

+ANMELDEN:<Name>

|  |
| --- |
| falls <IP><PORT> noch nicht angemeldet  falls <Name> noch nicht vergeben  <Name> mit IP und Port an Teilnehmerliste anfügen  +ANMELDUNG\_OK -> Sender  sonst -FEHLER:Name schon vergeben  sonst -FEHLER:schon angemeldet |

+GIB\_ALLE\_TN

|  |
| --- |
| <Teilnehmer> = String mit Namen aller Teilnehmer, getrennt durch „:“.  +ALLE\_TEILNEHMER:<Teilnehmer> -> Sender |

+SENDE\_NAME\_AN\_ALLE

|  |
| --- |
| <Name> = Name des Senders aus IP und Port bestimmen  +ZUGANG:<Name> -> an alle Clients |

+TEXT:<Name>:<Text>

|  |
| --- |
| <IP> = IP zu <Name>  <Port> = Port zu <Name>  +TEXT:<Text> -> (<IP>, <Port>) |

+ABMELDEN

|  |
| --- |
| Teilnehmer aus Teilnehmerliste löschen  +BYE -> Sender  +ABGANG:<Name> -> an alle  Verbindung trennen |

Die angemeldeten Teilnehmer werden mit ihrem Namen, ihrer IP und ihrem Port in einer Liste angemeldeteTeilnehmer verwaltet. Diese verwaltet Objekte der Klasse Teilnehmer, die als Attribute den Namen, die IP-Adresse und den Port eines Teilnehmers enthalten.

Aus der Übersicht ergibt sich, dass noch folgende **privaten** Methoden in der Klasse MessengerServer implementiert werden sollten.

**boolean istTeilnehmerAngemeldet(String pClientIP,**

**int pClientPort): boolean**

Liefert true, wenn es bereits einen Teilnehmer mit der übergebenen IP und dem übergebenen Port gibt, andernfalls false.

**boolean istNameVergeben(String pName)**

Liefert true, wenn es bereits einen Teilnehmer mit dem übergebenen Namen in der Liste der angemeldeten Teilnehmer gibt.

**void meldeTeilnehmerAn(String pClientIP, int pClientPort, String pName)**

Erzeugt ein neues Objekt der Klasse Teilnehmer und hängt es an die Liste

angemeldeteTeilnehmer an.

**String gibAlleAngemeldetenTeilnehmernamen()**

Liefert alle Namen der angemeldeten Teilnehmer als String - getrennt durch das Trennzeichen „:“.

**String findeTeilnehmernameZuIPAdresseUndPort(String pClientIP, int pClientPort)**

Liefert den Namen des Teilnehmers mit der übergebenen IP-Adresse und dem übergebenen Port.

**String findeIPAdresseZuTeilnehmer(String pName)**

Liefert die Client-IP-Adresse des Teilnehmers mit dem übergebenen Name.

**int findePortZuTeilnehmer(String pName)**

Liefert den Clientport des Teilnehmers mit dem übergebenen Name.

**meldeTeilnehmerAb(String pClientIP, int pClientPort)**

Löscht den Teilnehmer mit übergebenen IP und Port aus der Liste der angemeldeten Teilnehmer.

Das Protokoll des Messenger-Dienstes wird in Konstanten in der statischen Klasse PROT gespeichert.

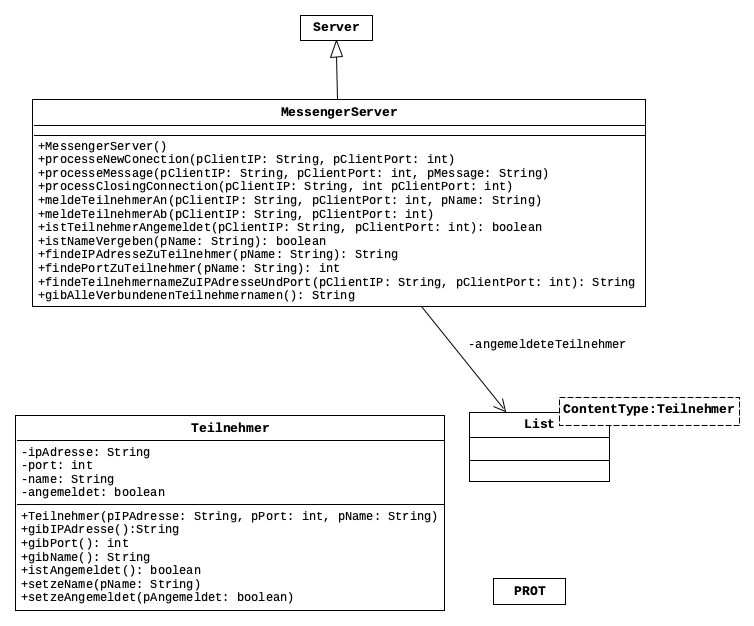
Aus diesen Überlegungen lässt sich folgendes Implementations-Diagramm für den Messenger-Server ableiten.

Abbildung 11: Implementationsdiagramm Messenger-Server

**2.2 Implementierung**

Das folgende Listing enthält den Kopf der Klasse MessengerServer, den Konstruktor sowie die Implementierung der drei Methoden processNewConnection, processMessage sowie processClosingConnection.

|  |  |
| --- | --- |
| 1 | import javax.swing.JOptionPane; |
| 2 | import netzklassen.List; |
| 3 | import netzklassen.Server; |
|  |  |
| 4 | public class MessengerServer extends Server { |
| 5 | private List<Teilnehmer> angemeldeteTeilnehmer; |
|  |  |
| 6 | public MessengerServer() { |
| 7 | super(20017); |
| 8 | if (!isOpen()) |
| 9 | JOptionPane.showMessageDialog(null, "Fehler beim Starten des  Servers auf Port 20017!"); |
| 10 | else { |
| 11 | angemeldeteTeilnehmer = new List<Teilnehmer>(); |
| 12 | System.out.println("Server ist gestartet."); |
| 13 | } |
| 14 | } |
|  |  |
| 15 | public void processNewConnection(String pClientIP,  int pClientPort) { |
| 16 | send(pClientIP, pClientPort, |
| 17 | PROT.SC\_WK + PROT.TRENNER  + "Willkommen auf dem Messenger-Server!"); |
| 18 | } |
|  |  |
| 19 | public void processMessage(String pClientIP,  int pClientPort, String pMessage) { |
| 20 | String[] pMessageZerteilt = pMessage.split(PROT.TRENNER); |
| 21 | System.out.println("S0:" + pMessageZerteilt[0] + "!"); |
| 22 | if (!istTeilnehmerAngemeldet(pClientIP, pClientPort)) { |
| 23 | if (pMessageZerteilt[0].equals(PROT.CS\_AN)) { |
| 24 | if (istNameVergeben(pMessageZerteilt[1])) |
| 25 | send(pClientIP, pClientPort, PROT.SC\_ER + PROT.TRENNER  + "Der Name ist bereits vergeben!"); |
| 26 | else { |
| 27 | meldeTeilnehmerAn(pClientIP, pClientPort,  pMessageZerteilt[1]); |
| 28 | send(pClientIP, pClientPort,PROT.SC\_AO + PROT.TRENNER  + pMessageZerteilt[1]); |
| 29 | } |
| 30 | } else { |
| 31 | System.out.println("FEHLER"); |
| 32 | send(pClientIP, pClientPort,PROT.SC\_ER + PROT.TRENNER  + "Sie sind nicht angemeldet!"); |
| 33 | } |
| 34 | } else { |
| 35 | switch (pMessageZerteilt[0]) { |
| 36 | case PROT.CS\_AN: |
| 37 | send(pClientIP, pClientPort, PROT.SC\_ER  + PROT.TRENNER + "Sie sind bereits angemeldet!"); |
| 38 | break; |
| 39 | case PROT.CS\_GA: |
| 40 | send(pClientIP, pClientPort, PROT.SC\_AT + PROT.TRENNER  + gibAlleAngemeldetenTeilnehmernamen()); |
| 41 | break; |
| 42 | case PROT.CS\_NA: |
| 43 | sendToAll(PROT.SC\_ZU + PROT.TRENNER  + findeTeilnehmernameZuIPAdresseUndPort(pClientIP, pClientPort)); |
| 44 | break; |
| 45 | case PROT.CS\_AB: |
| 46 | sendToAll(PROT.SC\_AB + PROT.TRENNER  + findeTeilnehmernameZuIPAdresseUndPort(pClientIP, pClientPort)); |
| 47 | meldeTeilnehmerAb(pClientIP, pClientPort); |
| 48 | closeConnection(pClientIP, pClientPort); |
| 49 | break; |
| 50 | case PROT.CS\_TX: |
| 51 | String empfaengerIP = findeIPAdresseZuTeilnehmer(  pMessageZerteilt[1]); |
| 52 | int empfaengerPort = findePortZuTeilnehmer(  pMessageZerteilt[1]); |
| 53 | String senderName =  findeTeilnehmernameZuIPAdresseUndPort(pClientIP, pClientPort); |
| 54 | send(empfaengerIP, empfaengerPort,PROT.SC\_TX  + PROT.TRENNER+ senderName+ PROT.TRENNER  + pMessageZerteilt[2]); |
| 55 | break; |
| 56 | } |
| 57 | } |
| 58 | } |
|  |  |
| 59 | public void processClosingConnection(String  pClientIP, int pClientPort) { |
| 60 | if (isConnectedTo(pClientIP, pClientPort)) |
| 61 | send(pClientIP, pClientPort, PROT.SC\_BY); |
| 62 | } |

**Erläuterungen**

Der Konstruktor der Klasse MessengerServer ruft den Konstruktor der Oberklasse Server auf mit einem beliebig zu wählenden Port aus dem nicht fest reservierten Bereich (1024 - 65635) als Parameter. Kommt keine Verbindung zustande, wird dies in einem Dialogfenster angezeigt, andernfalls wird die generische Liste angemeldeteTeilnehmer<Teilnehmer> zunächst als leere Liste erzeugt und in die Konsole geschrieben, dass der Server gestartet wurde und Clients sich dort anmelden können (Zeilen 6 – 14).

Die Methode processNewConnection sendet lediglich eine Willkommensmeldung an den angemeldeten Client.

Die Methode processMessage lässt sich unmittelbar mithilfe des Protokolls, der Übersicht über die Serveraktionen und den Dokumentationen der privaten Methoden herleiten.

Vor Aufruf der switch-Anweisung werden die durch ein Trennzeichen separierten Bestandteile der eingegangenen Client-Nachricht mithilfe der Java Funktion split einzeln in Komponenten des Arrays messageZerteilt gespeichert. Die Komponente mit dem Index 0 entscheidet über die Reaktion des Servers.

Das folgende Listing enthält die Implementation der privaten Methoden des Servers. Diese könnten von Schülerinnen und Schüler auf Grundlage der Dokumentation selbstständig entwickelt werden.

|  |  |
| --- | --- |
| 1 | private void meldeTeilnehmerAn(String pClientIP, int pClientPort,  String pName) { |
| 2 | Teilnehmer neuerTeilnehmer = new Teilnehmer(pClientIP, pClientPort,  pName); |
| 3 | neuerTeilnehmer.setzeAngemeldet(true); |
| 4 | angemeldeteTeilnehmer.append(neuerTeilnehmer); |
| 5 | } |
|  |  |
| 6 | private void meldeTeilnehmerAb(String pClientIP, int pClientPort) { |
| 7 | boolean gefunden = false; |
| 8 | angemeldeteTeilnehmer.toFirst(); |
| 9 | while (angemeldeteTeilnehmer.hasAccess() && !gefunden) { |
| 10 | if (angemeldeteTeilnehmer.getContent()  .gibIPAdresse().equals(pClientIP) |
| 11 | && angemeldeteTeilnehmer.getContent().gibPort() ==  pClientPort) { |
| 12 | angemeldeteTeilnehmer.getContent().setzeAngemeldet(false); |
| 13 | angemeldeteTeilnehmer.getContent().setzeName(null); |
| 14 | angemeldeteTeilnehmer.remove(); |
| 15 | gefunden = true; |
| 16 | } else { |
| 17 | angemeldeteTeilnehmer.next(); |
| 18 | } |
| 19 | } |
| 20 | } |
|  |  |
| 21 | private boolean istTeilnehmerAngemeldet(String pClientIP,  int pClientPort) { |
| 22 | boolean gefunden = false; |
| 23 | boolean angemeldet = false; |
| 24 | angemeldeteTeilnehmer.toFirst(); |
| 25 | while (angemeldeteTeilnehmer.hasAccess() && !gefunden) { |
| 26 | if (angemeldeteTeilnehmer.getContent()  .gibIPAdresse().equals(pClientIP) |
| 27 | && angemeldeteTeilnehmer.getContent().gibPort() ==  pClientPort) { |
| 28 | angemeldet = angemeldeteTeilnehmer.getContent()  .istAngemeldet(); |
| 29 | gefunden = true; |
| 30 | } else { |
| 31 | angemeldeteTeilnehmer.next(); |
| 32 | } |
| 33 | } |
| 34 | return (angemeldet); |
| 35 | } |
|  |  |
| 36 | private boolean istNameVergeben(String pName) { |
| 37 | boolean gefunden = false; |
| 38 | angemeldeteTeilnehmer.toFirst(); |
| 39 | while (angemeldeteTeilnehmer.hasAccess() && !gefunden) { |
| 40 | if (angemeldeteTeilnehmer.getContent().gibName().equals(pName)) { |
| 41 | gefunden = true; |
| 42 | } else { |
| 43 | angemeldeteTeilnehmer.next(); |
| 44 | } |
| 45 | } |
| 46 | return (gefunden); |
| 47 | } |
|  |  |
| 48 | private String findeIPAdresseZuTeilnehmer(String pName) { |
| 49 | String ipAdresse = null; |
| 50 | boolean gefunden = false; |
| 51 | angemeldeteTeilnehmer.toFirst(); |
| 52 | while (angemeldeteTeilnehmer.hasAccess() && !gefunden) { |
| 53 | if (angemeldeteTeilnehmer.getContent().gibName().equals(pName)) { |
| 54 | ipAdresse = angemeldeteTeilnehmer.getContent().gibIPAdresse(); |
| 55 | gefunden = true; |
| 56 | } else { |
| 57 | angemeldeteTeilnehmer.next(); |
| 58 | } |
| 59 | } |
| 60 | return (ipAdresse); |
| 61 | } |
|  |  |
| 62 | private int findePortZuTeilnehmer(String pName) { |
| 63 | int port = -1; |
| 64 | boolean gefunden = false; |
| 65 | angemeldeteTeilnehmer.toFirst(); |
| 66 | while (angemeldeteTeilnehmer.hasAccess() && !gefunden) { |
| 67 | if (angemeldeteTeilnehmer.getContent().gibName().equals(pName)) { |
| 68 | port = angemeldeteTeilnehmer.getContent().gibPort(); |
| 69 | gefunden = true; |
| 70 | } else { |
| 71 | angemeldeteTeilnehmer.next(); |
| 72 | } |
| 73 | return (port); |
| 74 | } |
| 75 | } |
|  |  |
| 76 | private String findeTeilnehmernameZuIPAdresseUndPort(String  pClientIP, int pClientPort) { |
| 77 | String gefundenerTeilnehmername = null; |
| 78 | boolean gefunden = false; |
| 79 | angemeldeteTeilnehmer.toFirst(); |
| 80 | while (angemeldeteTeilnehmer.hasAccess() && !gefunden) { |
| 81 | if (angemeldeteTeilnehmer.getContent()  .gibIPAdresse().equals(pClientIP) |
| 82 | && angemeldeteTeilnehmer.getContent().gibPort() ==  pClientPort) { |
| 83 | gefundenerTeilnehmername =  angemeldeteTeilnehmer.getContent().gibName(); |
| 84 | gefunden = true; |
| 85 | } else { |
| 86 | angemeldeteTeilnehmer.next(); |
| 87 | } |
| 88 | } |
| 89 | return (gefundenerTeilnehmername); |
| 90 | } |
|  |  |
| 91 | private String gibAlleAngemeldetenTeilnehmernamen() { |
| 92 | String teilnehmernamen = ""; |
| 93 | angemeldeteTeilnehmer.toFirst(); |
| 94 | while (angemeldeteTeilnehmer.hasAccess()) { |
| 95 | teilnehmernamen = teilnehmernamen + angemeldeteTeilnehmer  .getContent().gibName() + PROT.TRENNER; |
| 96 | angemeldeteTeilnehmer.next(); |
| 97 | } |
| 98 | if (!teilnehmernamen.equals("")) { |
| 99 | return (teilnehmernamen.substring(0, teilnehmernamen.length() - 1)); |
| 100 | } else { |
| 101 | return (""); |
| 102 | } |
| 103 | } |

**3. Entwicklung des Clients**

**3.1 Benutzungsoberfläche**

Prinzipiell lässt sich der Messenger-Dienst nach Starten des MessengerServers mit dem oben entwickelten allgemeinen Client betreiben, indem man zeilenweise Nachrichten an den Server im Dialogfenster eingibt und versendet. Die Serverantworten werden jeweils in einem Nachrichtenfenster angezeigt.

Eine grafische Benutzungsoberfläche macht die Kommunikation zwischen den Teilnehmern des Messenger-Dienstes deutlich komfortabler. Der Quelltext für die Benutzungsoberfläche des Messenger-Clients wie in Abbildung 10 sollte vorgegeben werden oder von Schülerinnen und Schülern mit einem GUI-Bilder z.B. im JavaEditor selbst zusammengeklickt werden.

Die grafische Benutzungsoberfläche ist ein Objekt der Klasse MessengerClientGUI. Die folgende Tabelle enthält die wesentlichen grafischen Objekte der Benutzungsoberfläche mit ihrer Funktionalität

| **Objektname** | **Klasse** | **Beschriftung** | **Funktion** |
| --- | --- | --- | --- |
| tfName | JTextField | eigener Name | Eingabefeld für den Anmeldenamen |
| bLogIn | JButton | Login | Drücken des Buttons: Anmeldung beim Server mit dem Namen im Textfeld tfName |
| btLogOut | JButton | Logout | Drücken des Buttons: Client meldet sich beim Server ab |
| taNachrichten | JTextArea |  | Alle gesendeten und empfangenen Nachrichten werden hier protokolliert |
| lstTeilnehmer | JList |  | Die Namen aller beim Server angemeldeten Teilnehmer werden hier aufgelistet. Ein Teilnehmer, an den eine Nachricht gehen soll, muss mit der Maus selektiert werden. |
| tfNachricht | JTextField |  | Nachrichten, die gesendet werden, müssen in dieses Textfeld geschrieben werden. |
| bSenden | JButton | senden | Die Nachricht, die im Textfeld tfNachricht steht, wird an den selektierten Teilnehmer gesendet. |

Das Objekt der Klasse MessengerClientGUI verwaltet ein Objekt der Klasse MessengerClient, die eine Unterklasse der vorgegebenen Klasse Client ist und die abstrakte Methode processMessage überschreibt.

**3.2 Modellierung**

Die folgende Übersicht zeigt für alle Servernachrichten an, welche Aktionen der Client durchzuführen hat und welche Nachrichten er an den Server sendet.

+WILLKOMMEN

|  |
| --- |
| Anzeige, dass Verbindung erfolgreich aufgebaut wurde |

+ANMELDUNG\_OK

|  |
| --- |
| Login-Button deaktivieren  Logout-Button aktivieren  Senden-Button aktivieren  +GIB\_ALLE\_TEILNEHMER -> Server  +SENDE\_NAME\_AN\_ALLE -> Server |

+ALLE\_TEILNEHMER:<Name1>:<Name2>...

|  |
| --- |
| Namen in die Liste der angemeldeten Teilnehmer einfügen und in der GUI anzeigen |

+ZUGANG:<Name>

|  |
| --- |
| Name an die Liste der angemeldeten Teilnehmer anfügen |

+ABGANG:<Name>

|  |
| --- |
| Namen aus der Teilnehmerliste der GUI löschen |

+BYE

|  |
| --- |
| Programm beenden |

Zur Modifikation der Benutzungsoberfläche ruft das Objekt der Klasse MessengerClient folgende privaten Methoden der Klasse MessengerGUI auf, deren Funktionalität sich aus dem Namen erschließt:

**private void ergaenzeNachrichten(String pNachricht)**

**private void ergaenzeTeilnehmerListe(String pName)**

**private void loescheNameAusTeilnehmerListe(String pName)**

Die Klasse MessengerClient erhält zur besseren Strukturierung des Quelltextes die folgenden privaten Methoden:

**private void anmelden(String pName)**

Sendet die Anmeldenachricht an den Server

**private void abmelden()**

Sendet die Abmeldenachricht an den Server

**private void nachrichtSenden(String pEmpfaenger,**

**String pNachricht)**

Sendet eine Textnachricht zur Weiterleitung an den Server und schreibt diese in die Textarea der Benutzungsoberfläche

Aus diesen Überlegungen ergibt sich folgendes Implementationsdiagramm für den Messenger-Client.

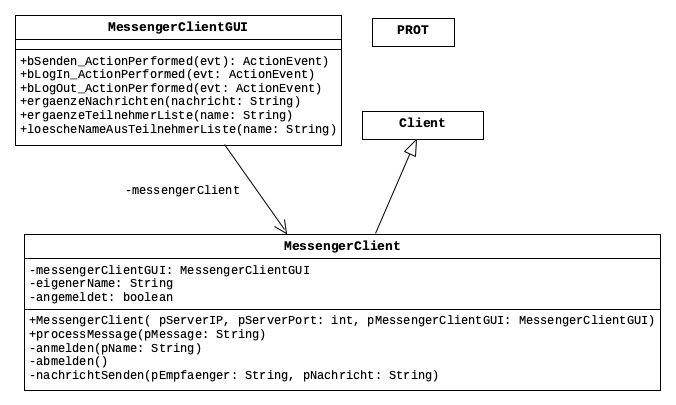


Abbildung 12: Implementationsdiagramm Messenger-Client

**3.3 Implementierung**

Das folgende Java-Listing zeigt die Implementierung der Methode processMessage der Klasse MessengerClient.

|  |  |
| --- | --- |
| 1 | public void processMessage(String pMessage) { |
| 2 | String[] pMessageZerteilt = pMessage.split(PROT.TRENNER); |
| 3 | System.out.println("C0:" + pMessageZerteilt[0] + "!"); |
| 4 | if (!angemeldet) { |
| 5 | switch (pMessageZerteilt[0]) { |
| 6 | case PROT.SC\_WK: |
| 7 | String willkommensnachricht = ""; |
| 8 | for (int index = 1; index < pMessageZerteilt.length; index++)  { |
| 9 | willkommensnachricht += pMessageZerteilt[index] + " "; |
| 10 | } |
| 11 | JOptionPane.showMessageDialog(null, willkommensnachricht); |
| 12 | break; |
| 14 | case PROT.SC\_AO: |
| 15 | eigenerName = pMessageZerteilt[1]; |
| 16 | angemeldet = true; |
| 17 | send(PROT.CS\_GA); |
| 18 | send(PROT.CS\_NA); |
| 19 | messengerClientGUI.initialisiereNachAnmeldung(); |
| 20 | break; |
| 21 | } |
| 22 | } else { |
| 23 | switch (pMessageZerteilt[0]) { |
| 24 | case PROT.SC\_TX: |
| 25 | messengerClientGUI.ergaenzeNachrichten(pMessageZerteilt[1]  + " schreibt:\n" + pMessageZerteilt[2]); |
| 26 | break; |
| 27 | case PROT.SC\_ZU: |
| 28 | if (!pMessageZerteilt[1].equals(eigenerName)) { |
| 29 | messengerClientGUI.ergaenzeTeilnehmerListe(  pMessageZerteilt[1]); |
| 30 | } |
| 31 | break; |
| 32 | case PROT.SC\_AB: |
| 33 | if (!pMessageZerteilt[1].equals(eigenerName)) { |
| 34 | messengerClientGUI.loescheNameAusTeilnehmerListe(  pMessageZerteilt[1]); |
| 35 | } else { |
| 36 | messengerClientGUI.leereNachLogout(); |
| 37 | }; |
| 38 | break; |
| 39 | case PROT.SC\_AT: |
| 40 | for (int index = 1; index < pMessageZerteilt.length;  index++) { |
| 41 | if (!pMessageZerteilt[index].equals(eigenerName)) { |
| 42 | messengerClientGUI.ergaenzeTeilnehmerListe(  pMessageZerteilt[index]); |
| 43 | } |
| 44 | } |
| 45 | break; |
| 46 | case PROT.SC\_BY: |
| 47 | eigenerName = null; |
| 48 | angemeldet = false; |
| 49 | JOptionPane.showMessageDialog(null, "Verbindung durch"  + " den Messenger-Server geschlossen.\n "  + "Das Programm wird jetzt beendet."); |
| 50 | System.exit(0); |
| 51 | break; |
| 52 | case PROT.SC\_ER: |
| 53 | JOptionPane.showMessageDialog(null,  pMessage.substring(1)); |
| 54 | break; |
| 55 | default: |
| 56 | JOptionPane.showMessageDialog(null,  "Unzulässige Anweisung empfangen: '" + pMessage + "'"); |
| 57 | break; |
| 58 | } |
| 59 | } |
| 60 | } |

Die privaten Methoden der Klasse MessengerClient können von Schülerinnen und Schülern selbstständig implementiert werden.

|  |  |
| --- | --- |
| 1 | public void anmelden(String pName) { |
| 2 | send(PROT.CS\_AN + PROT.TRENNER + pName); |
| 3 | } |
|  |  |
| 4 | public void abmelden() { |
| 5 | send(PROT.CS\_AB); |
| 6 | } |
|  |  |
| 7 | public void nachrichtSenden(String pEmpfaenger, String pNachricht) { |
| 8 | if (!pNachricht.equals("")) { |
| 9 | send(PROT.CS\_TX + PROT.TRENNER + pEmpfaenger + PROT.TRENNER |
|  | + pNachricht); |
| 10 | messengerClientGUI.ergaenzeNachrichten("Du schreibst an "  + pEmpfaenger + "\n" + pNachricht); |
| 11 | } |
| 12 | } |
|  |  |

**4. Test**

**Test auf einem Rechner**

Mit BlueJ lässt sich das Messenger-Programm mit mehreren Clients auf einem Rechner testen.

Mit folgenden Schritten kann die Anwendung mit drei Clients getestet werden:

* Die Konstante CONST\_SERVERIP in der Klasse MessengerClientGUI auf den Wert "127.0.0.1" setzen.
* Server-Programm mit BlueJ durch Aufruf der Main-Methode starten.
* Mit BlueJ drei Objekte der Klasse MessengerClientGUI erzeugen (nicht mit der Main-Methode). Abbildung 12 zeigt die Ansicht des Bildschirms (nach Verschiebung der Fenster).
* In den drei Client-Fenstern mit unterschiedlichen Namen beim Server anmelden.
* Nun können Nachrichten zwischen den drei Teilnehmern des Messenger-Dienstes ausgetauscht werden

**Test auf mehreren Rechnern**

Wenn das Programm mit **mehreren Rechnern** in einem Netzwerk getestet werden soll, muss man wie folgt vorgehen:

* Ermitteln der IP-Adresse des Rechners, auf dem das Server-Programm laufen soll.
* Die Konstanten CONST\_SERVERIP der Klasse MessengerClientGUI wird die ermittelte IP-Adresse zugewiesen.
* Das Client-Programm wird auf die Rechner kopiert, auf denen die Client-Software laufen soll.
* Der Messenger-Server wird gestartet
* Die Messenger-Clients werden mit BlueJ gestartet.

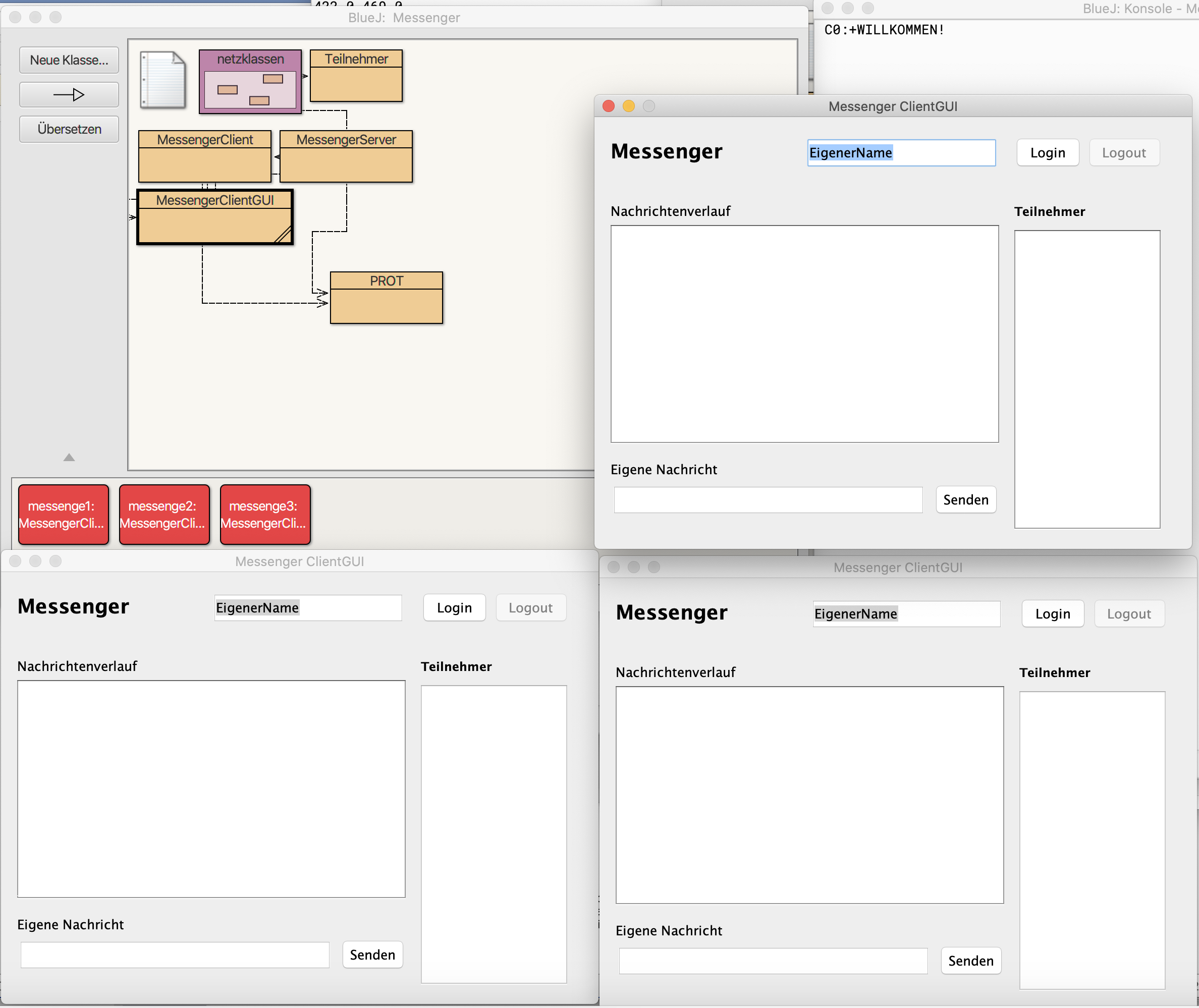


Abbildung 13: Client-Server auf einem Rechner