

Hochschule Schmalkalden

Fakultät Informatik

Technische Dokumentation zum Thema:

„GUI gestützte Multiple Client Chat Applikation mittels Java-Socket-API und JavaFX“

**Vorgelegt von:**

Robert Irrgang

Martin-Luther-Ring 38

98574, Schmalkalden

**E-Mail**: [robertirrgang@outlook.com](mailto:robertirrgang@outlook.com)

**Matrikelnummer:** 308619

**Fach:** Projekt Informatik

**Fachsemester:** 5

**Studiengang:** BA Informatik

**Lehrkraft**: Prof. Dr. Thomas Heimrich

Inhaltsverzeichnis

[1. Vorwort 2](#_Toc42693693)

[2. Funktionsweise 2](#_Toc42693694)

[2.1 Server 2](#_Toc42693695)

[2.2 Client 4](#_Toc42693696)

[3. Technischer Einblick 5](#_Toc42693697)

[3.1 Model 5](#_Toc42693698)

[3.1.1 Server Klasse 6](#_Toc42693699)

[3.1.2 Client-Handler Thread 6](#_Toc42693700)

[3.1.2 Client Klasse 7](#_Toc42693701)

[3.2 Views 8](#_Toc42693702)

[3.3 Controller 8](#_Toc42693703)

[4. Fazit 8](#_Toc42693704)

[5. Quellenverzeichnis 9](#_Toc42693705)

[6. Anhang 10](#_Toc42693706)

[A. Server-Klassendiagramm 10](#_Toc42693707)

[B. Client-Klassendiagramm 11](#_Toc42693708)

# Vorwort

Dieses Projekt wurde im Rahmen des Kurses „Projekt Informatik“ entwickelt. Hierbei handelt es sich um eine „Instant Messaging“ Client-Server-Anwendung, die es den Nutzern ermöglicht innerhalb eines lokalen Netzwerkes textbasiert miteinander zu kommunizieren. Mithilfe des Frameworks „JavaFX“ und dem von Gluon entwickelten SceneBuilder wurde für die Server- und Client-Komponente eine grafische Oberfläche (GUI[[1]](#footnote-1)) entworfen und entwickelt. Die Anwendung basiert auf die „Java-Socket-API[[2]](#footnote-2)“ von Oracle®. Die Software für diese Anwendung wurde in der integrierten Entwicklungsumgebung „IntelliJ“ der Firma JetBrains entwickelt. Im folgenden Abschnitt möchte ich die Funktionsweise der einzelnen Komponenten erläutern. Anschließend folgt der Technische Einblick der verwendeten Frameworks und entwickelten Klassen. Für das bessere Verständnis wurden im Anhang des Dokuments entsprechende Klassendiagramme mit aufgeführt.

# Funktionsweise

Mithilfe dieser Anwendung wird es Nutzern eines gemeinsamen lokalen Netzwerkes ermöglicht untereinander textbasierte Nachrichten auszutauschen. Damit diese Kommunikation stattfinden kann, benötigt das Netzwerk einen sogenannten Server. Dieser ist dafür zuständig, eingehende Nachrichten an alle verbundenen Clients des Netzwerkes zu senden.

## 2.1 Server

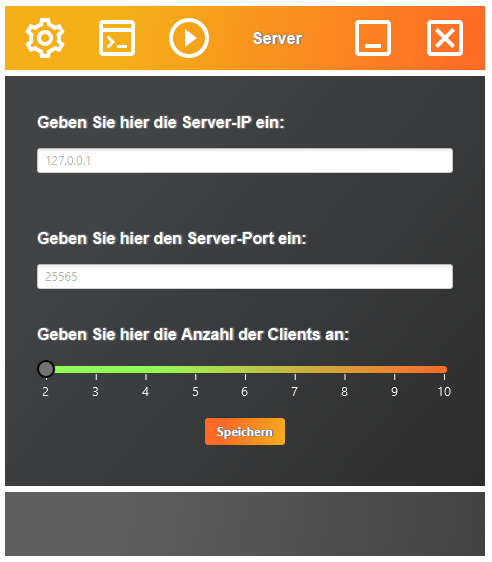
Die Aufgaben des Servers liegen darin, ankommende Nutzeranfragen zu bestätigen und diese mit dem Chat zu verbinden. Anschließend leitet er alle einkommenden Nachrichten an diesen Nutzer und alle bisherig verbundenen Clients weiter. Um den Server zu starten muss lediglich vorher eine IP und ein freier Port zugewiesen werden. Des Weiteren besteht die Möglichkeit, dass sich bis zu zehn Clients gleichzeitig miteinander verbinden und Informationen austauschen können. In der folgenden Abbildung werden die serverseitigen Einstellmöglichkeiten dargestellt.

Abbildung 1 – Server-Einstellungen

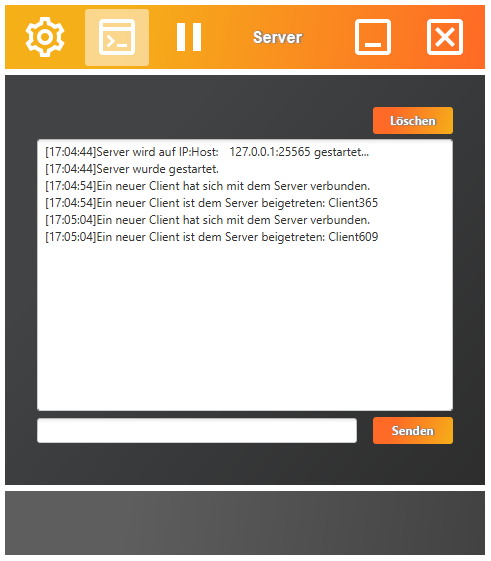
Nachdem der Server gestartet wird, wartet er auf einkommende Nutzeranfragen. Diese werden nacheinander verarbeitet bis das vorher gesetzte Limit an gleichzeitigen Nutzern erreicht ist. Danach werden weitere Anfragen abgelehnt. Weiterhin werden alle akzeptierten Verbindungen und Statusmeldungen in der Server-Konsole festgehalten. Zudem erhalten alle Clients bei dem Beitreten eines neuen Nutzers in den Chat eine entsprechende Information. Dies wird in der folgenden Abbildung veranschaulicht.

Abbildung 2 – Server-Konsole

## 2.2 Client

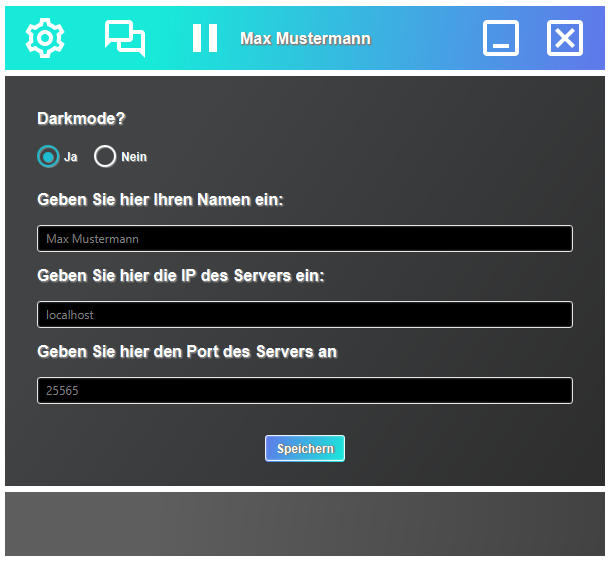
Mithilfe des Clients können sich die Nutzer mit dem Chat-Server verbinden. Hierfür wird lediglich die entsprechende IP und der offene Port des Servers benötigt. Anschließend wird der Client mit dem Server verbunden und kann mit anderen Nutzern kommunizieren. Diese Einstellungen werden ähnlich wie beim Server getroffen. Um dies noch einmal grafisch darzustellen, folgt eine Abbildung der Einstellungsmöglichkeiten des Clients.

Abbildung 3 Client - Einstellungen

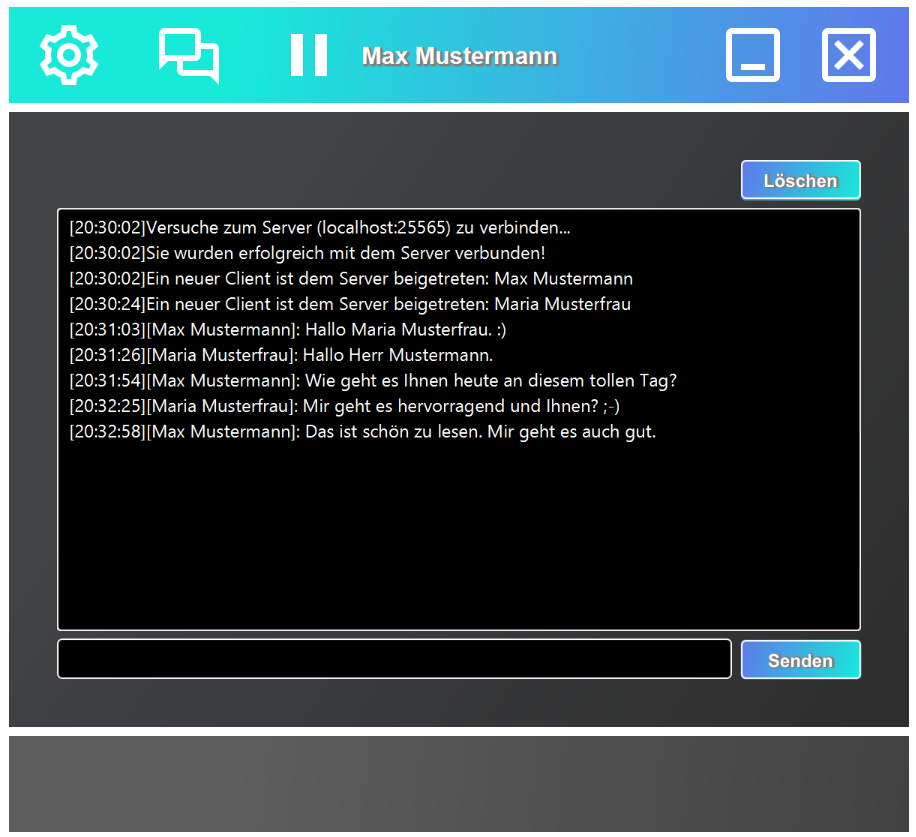
Nachdem die Verbindung zu dem Chat-Server erfolgreich war, werden entsprechende Statusmeldungen angezeigt. Anschließend kann der Nutzer im Chat mit anderen Nutzern kommunizieren. Treten Fehler bei der Verbindung auf, wird der Client mit Textmeldungen darauf hingewiesen und kann entsprechend mögliche Fehleingaben bei den Einstellungen korrigieren und einen weiteren Verbindungsversuch starten. In der nächsten Abbildung wird eine beispielhafte Konversation zwischen zwei fiktiven Charakteren dargestellt.

Abbildung 4 – Client Chat-Beispiel

# Technischer Einblick

In den folgenden Kapiteln werden nun die theoretischen Ansätze mit den jeweils dazugehörigen Quellcode detaillierter erläutert. Dabei wird noch einmal konkreter auf die Server Klassen und ihre notwendigen Threads sowie die Client Klassen eingegangen. Des Weiteren wird der grobe Aufbau der GUI erläutert und die Implementierung des MVC[[3]](#footnote-3) Modells thematisiert.

## Model

Diese Applikation verwendet das Model-View-Control Architekturmuster. Dadurch kann das eigentliche Datenmodell des Servers und des Clients unabhängig von der grafischen Oberfläche entworfen und erweitert werden. Zudem erleichtert es die Wartbarkeit des Quellcodes, da die jeweiligen Komponenten der Architektur eigenständig entwickelt werden. So kann folglich das bestehende Modell auch in anderer Präsentationsweise dargestellt werden. Dafür müssen lediglich die View und die Programmsteuerung verändert werden (Vgl. Rouse M. 2020). In den nächsten Kapiteln wird das Model des Servers und des Clients dargelegt.

### 3.1.1 Server Klasse

Der Server wurde basierend auf der Socket-API von Oracle® konzipiert. Sockets sind Endpunkte der Kommunikation zwischen zwei Maschinen oder Applikationen. Sie bilden eine Art virtueller „Steckverbindungen“, welche bidirektional Daten austauschen können (Vgl. Czeschla, JavaBeginners, 2020 ). Um eine Kommunikation zwischen mehr als zwei Maschinen zu gewährleisten, benötigt man einen Server, der mittels Multithreading die einzelnen Client-Server-Verbindungen verwaltet und abhandelt. Dafür lagert der Server das Akzeptieren von neuen Verbindungsanfragen an einen ausgegliederten Thread namens „Client-Handler“ aus. Die folgende Tabelle führt die einzelnen Methoden des Servers auf.

|  |  |  |
| --- | --- | --- |
| Anzahl | Methodenname | Funktionsweise |
| 1. | startServer() | * Startet einen neuen Server-Socket der den Server mit vorher festgelegter IP und Port öffnet * Startet einen neuen Thread (searchForClientsThread), welcher einkommende Client-Anfragen akzeptiert und die Input und Output Streams verbindet * Weiterleiten des Clients an den Client-Handler Thread |
| 2. | stopServer() | * Schließt den Server-Socket und entbindet die Input und Output Streams aller Clients |
| 3. | sendMsgToAllClients() | * Leitet einkommende Nachrichten an alle Clients weiter |
| 4. | forwardOutput() | * Leitet Nachrichten an die GUI weiter |
| 5. | injectMainController() | * Initialisiert den Main-Controller und verbindet das Server-Model mit den Steuer-Controllern |

### 3.1.2 Client-Handler Thread

Damit der Server eine Kommunikation zwischen mehreren Nutzern gewährleisten kann, speichert er die einkommenden Verbindungen vorher in einer Liste ab und reicht die „Steckverbindungen“ an ausgelagerte Threads weiter. Diese sind dafür zuständig einkommende und ausgehende Datenströme (Streams) zwischen einem Client und dem Server zu steuern. Folgende Abbildung demonstriert diese Methodik.

Abbildung 5 - Client Handler Thread (Vgl. https://www.geeksforgeeks.org/introducing-threads-socket-programming-java/)

Der Server und der Client reichen also ihre Datenströme an den Client-Handler weiter und dieser organisiert somit den Informationsaustausch beider Anwendungen. Falls die Verbindung getrennt wird, beendet der Server den erstellten Thread.

### 3.1.2 Client Klasse

Der Client wurde ebenso auf der Socket-API konzipiert. Ähnlich wie der Server lagert auch er einzelne Aufgaben in mehrere Threads aus, um somit parallel Daten senden und empfangen zu können. Des Weiteren spielt Multithreading eine große Rolle bei der Funktionsweise zwischen grafischer Oberfläche und Modell. Die folgende Tabelle führt alle Methoden des Clients auf.

|  |  |  |
| --- | --- | --- |
| Anzahl | Methodenname | Funktionsweise |
| 1. | connect() | * Versucht eine Verbindung zu einer vorher festgelegten IP und Port aufzubauen |
| 2. | disconnect() | * Schließt eine bestehende Verbindung zu einem Server |
| 3. | sendMessage() | * Sendet eine Nachricht an den Server |
| 4. | forwardOutput() | * Leitet Nachrichten an die GUI weiter |
| 5. | injectMainController() | * Initialisiert den Main-Controller und verbindet das Client-Model mit den Steuer-Controllern |

## Views

Ein weiterer Teil der MVC-Architektur ist die Präsentation der grafischen Oberfläche. Spezielle FXML Dateien beschreiben hierfür, wie die einzelnen Komponenten dargestellt werden. Sie basieren auf der XML[[4]](#footnote-4) - Markup Sprache und beschreiben den jeweiligen Szenengraphen der Anwendung (Vgl. Czeschla, Javabeginners, 2020). Die einzelnen Views wurden mittels ein von Gluon entwickelten Tools namens „SceneBuilder“ entwickelt. Je Datenmodell wurden jeweils drei Views entworfen. Als zentrale Szene dient die „MenuView“, da sie den Rahmen der Applikation beschreibt, die wichtigsten Steuerelemente beinhaltet und in ihr die zwei weiteren Views einschließt. Als weitere Szene beschreibt die „SettingsView“ alle Einstellungskomponenten der Datenmodelle. So beinhaltet sie entsprechende Formulare zur Eingabe der Verbindung (IP und Port) und speziell für den Server die Einstellmöglichkeit der maximalen Anzahl von Clients. Abschließend beschreiben die „ConsoleView“ bzw. „ChatView“ entsprechende Eingabe- und Anzeigefenster, um eine spätere Kommunikation zu gewährleisten.

## Controller

Als letzter Teil der MVC-Architektur bilden die Controller-Klassen die Programmsteuerung der Anwendung ab. Sie ermöglichen die Interaktion zwischen den Benutzer und der Applikation. So können mittels Schaltflächen Methoden der Datenmodelle ausgelöst werden. Ähnlich wie die Views besitzen die Datenmodelle drei Steuerklassen. Der „MainController“ bildet die Schnittstelle zwischen Datenmodell, Views und den beiden anderen Steuerklassen ab. Für die getätigten Einstellungen reicht der „SettingsController“ die eingegeben Daten an die Schnittstelle weiter. Ebenso leiten der „ConsoleController“ bzw. der „ChatController“ eingegebene Nachrichten an die Hauptsteuerklasse weiter, welche diese an das Datenmodell zum Senden überreicht. Mithilfe der ausgelagerten Steuerklassen wird die Wartbarkeit der Anwendung erhöht und erleichtert somit auch die Erweiterbarkeit der Applikation.

# Fazit

Während der Ausarbeitung des Projektes konnte ich das bereits im Studium erlangte Kenntnisse für die Erstellung der Benutzeroberfläche nutzen. Des Weiteren konnte ich grundlegendes Wissen bezüglich der Thematik „Computer Networking“ sowie neue Programmiertechniken, wie das „Multithreading“ in Java erlernen.

# Quellenverzeichnis

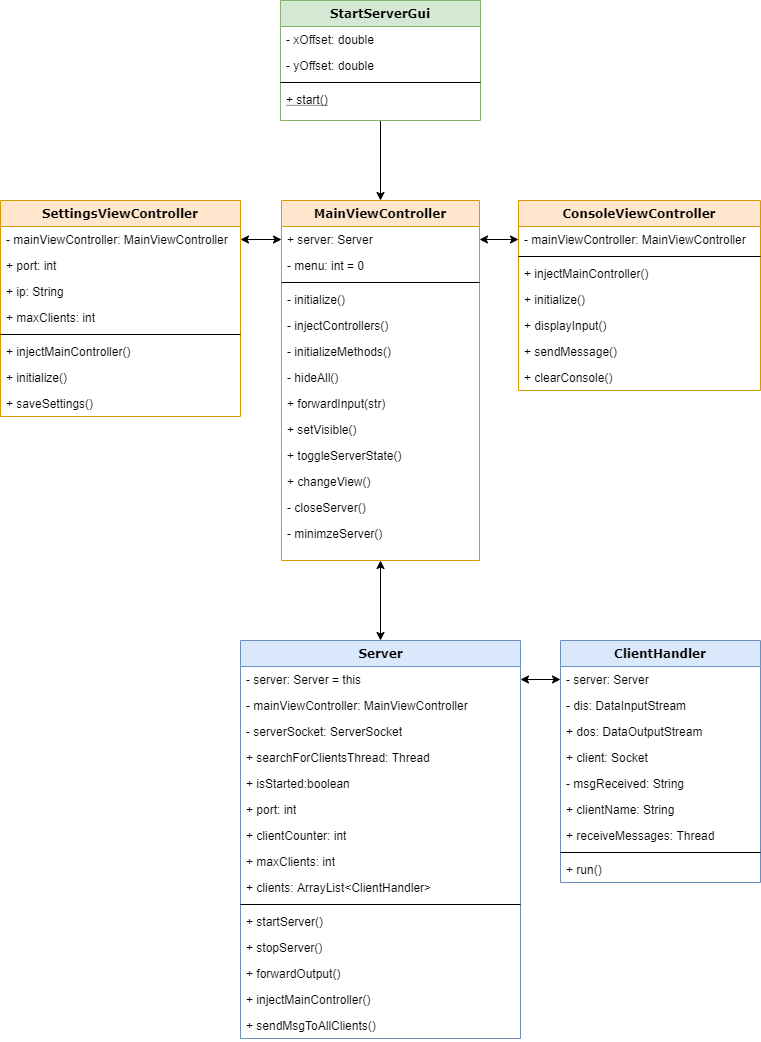
Czeschla, J. (15. März 2020). *Javabeginners*. Von Javabeginners.de: https://javabeginners.de/Frameworks/JavaFX/FXML.php abgerufen

Czeschla, J. (15. März 2020). *JavaBeginners*. Von JavaBeginners.de: https://javabeginners.de/Netzwerk/Socketverbindung.php abgerufen

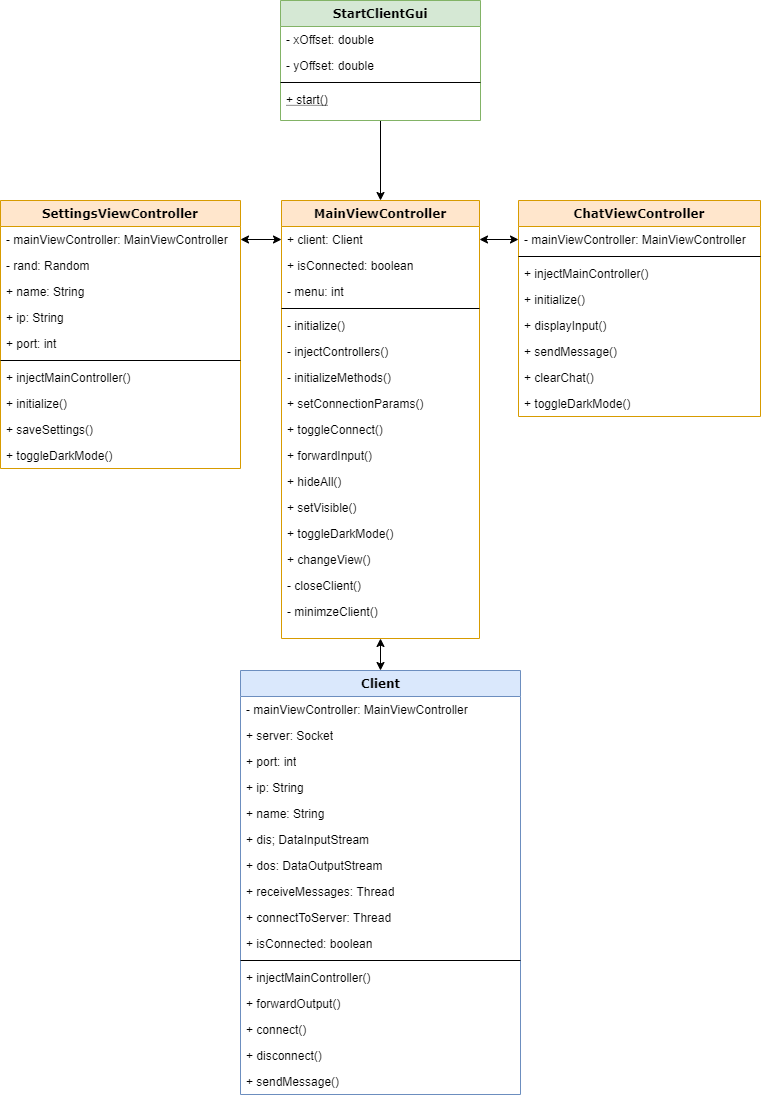
Rouse, M. (14. März 2020). *ComputerWeekly*. Von ComputerWeekly: https://www.computerweekly.com/de/definition/Model-View-Controller-MVC abgerufen

# 6. Anhang

## Server-Klassendiagramm



## Client-Klassendiagramm



1. Grafische Benutzeroberfläche = (Graphical User Interface) [↑](#footnote-ref-1)
2. Programmierschnittstelle = (Application Programming Interface) [↑](#footnote-ref-2)
3. MVC = Model View Control - Modell [↑](#footnote-ref-3)
4. XML = Extensible Markup Language (dt.: Erweiterbare Auszeichnungssprache) [↑](#footnote-ref-4)