

Software-Projekt 2

Übung – Multichat



IT19a_ZH, Team 05 (02) – Einhörner

Kunsang Kündetsang

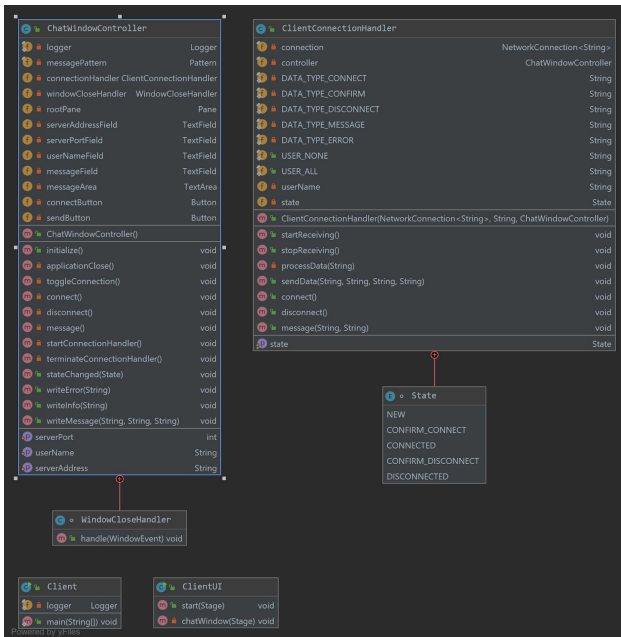
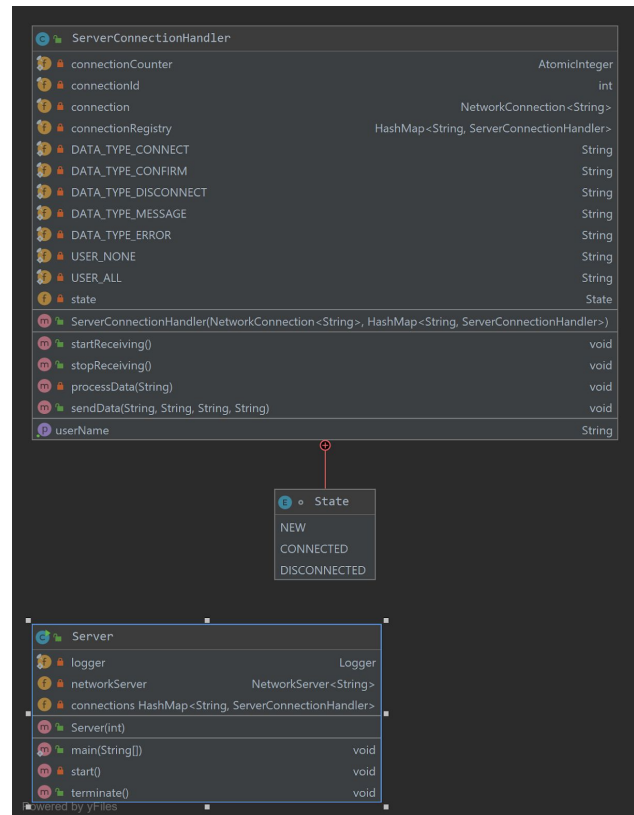
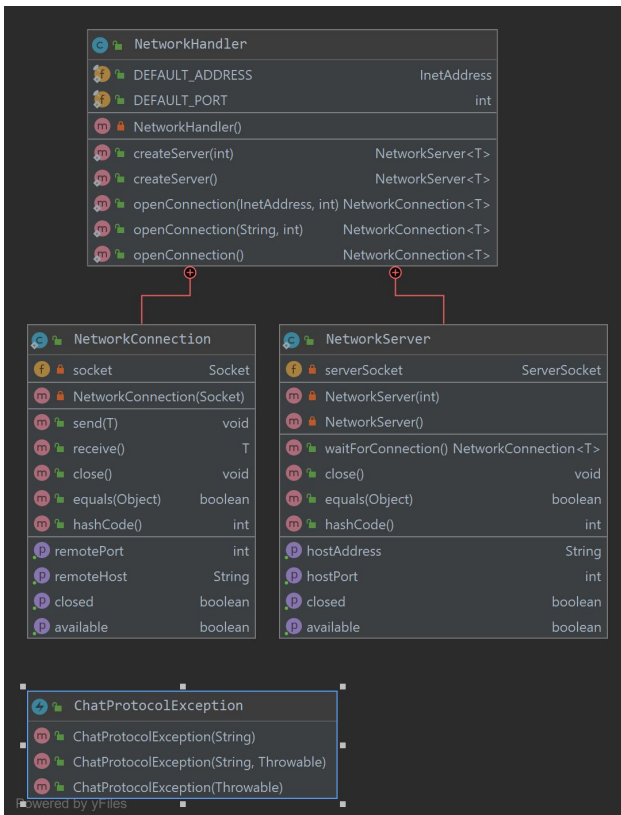
Sydney Nguyen

Inhaltsverzeichnis

Resultat der Analyse	3
1.1 Dokumentation der aktuellen Struktur (Klassendiagramm)	3
1.2 Dokumentation des Protokolls zwischen Client und Server	4
1.3 Beschreibung der gefundenen funktionalen Fehler	5
1.4 Beschreibung der gefundenen strukturellen Probleme	6
Beschreibung ihrer Lösung	7
2.1 Dokumentation der neuen Struktur	7
2.2 Verbesserung des Codes	7

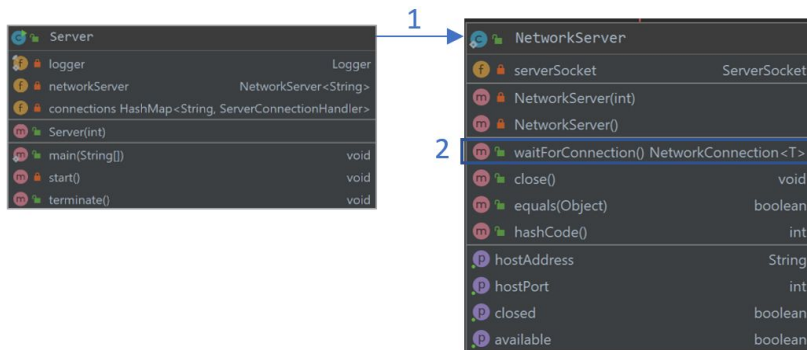
1. Resultat der Analyse

1.1 Dokumentation der aktuellen Struktur

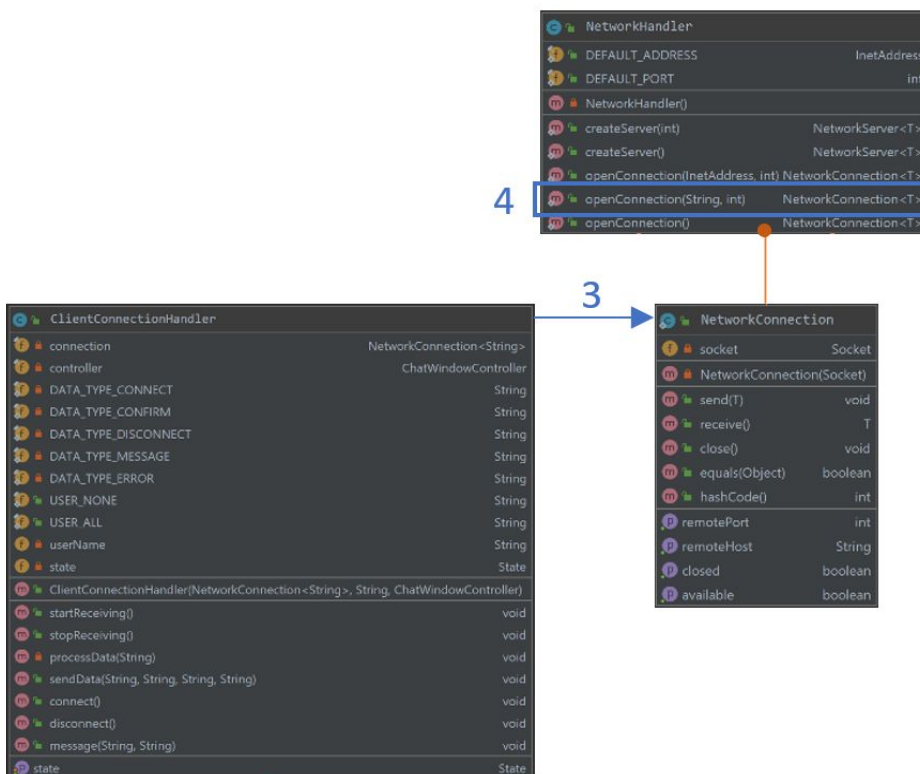


1.2 Dokumentation des Protokolls zwischen Client und Server

Ein Server kann mehrere Clients bedienen und ein Client kann sich nur mit einem Server verbinden. Wenn ein Server geschlossen wird, dann werden alle verbundenen Clients automatisch getrennt. Falls ein Client sich abmeldet, dann werden alle entsprechenden Ressourcen wieder freigegeben. NetworkServer wird auf der Serverseite verwendet, um einen Port zu öffnen und auf Verbindungsanfragen von Clients zu warten. NetworkConnection stellt eine bidirektionale Verbindung zwischen Client und Server dar, um Objekte zu senden und zu empfangen. Der typische Prozess funktioniert wie folgt:



1. Der Server erzeugt eine NetworkServer-Instanz. Ein Port wird auf allen verfügbaren Schnittstellen des aktuellen Hosts erstellt und geöffnet.
2. Sobald der Server bereit ist, Anfragen zu empfangen, ruft er die Methode `NetworkServer.waitForConnection()` auf, die blockiert und darauf wartet, dass der Client eine Verbindung öffnet.

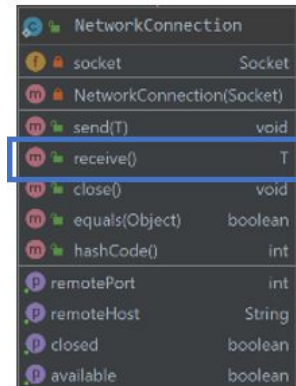


3. Auf der Clientseite wird eine neue `NetworkConnection`-Instanz erstellt.
4. Um eine Verbindung zum Server herzustellen wird `NetworkHandler.openConnection(String-Host, int-Port)` aufgerufen.

Auf der Serverseite sollte jede Interaktion mit einem spezifischen Client in einem separaten Thread behandelt werden, nach dem Start des Threads kann der Server zurückgehen und auf die nächste Verbindung warten.

Daten lesen

Aufruf `NetworkConnection.receive()`, der blockiert, bis ein Datenobjekt empfangen wird. Sobald das Objekt empfangen wurde, gibt die Methode eine Instanz des Objekts zurück. Dieses Objekt (Request) kann bearbeitet werden (auf der Server-Seite wird in der Regel eine Antwort zurückgeschickt; auf der Client-Seite wird dem Benutzer in der Regel das Ergebnis angezeigt). Nachdem die Verarbeitung abgeschlossen ist, ruft der Prozess erneut `NetworkConnection.receive()` auf, um auf die nächste Anfrage zu warten.

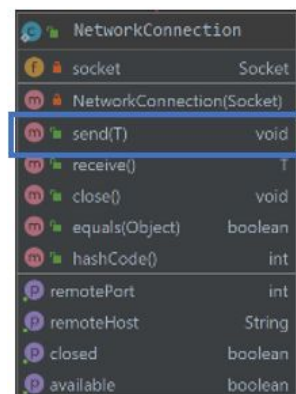


NetworkConnection	
socket	Socket
NetworkConnection(Socket)	
send(T)	void
receive()	T
close()	void
equals(Object)	boolean
hashCode()	int
remotePort	int
remoteHost	String
closed	boolean
available	boolean

Daten senden

Aufruf `NetworkConnection.send(Serialisierbare Daten)`, der das gegebene Daten Objekt an die empfangende Seite sendet. Die Methode kehrt zurück, sobald das Objekt übertragen wurde. `NetworkConnection` ist nicht thread-sicher, deshalb muss man sicher stellen, dass nur ein Thread zur gleichen Zeit Daten sendet.

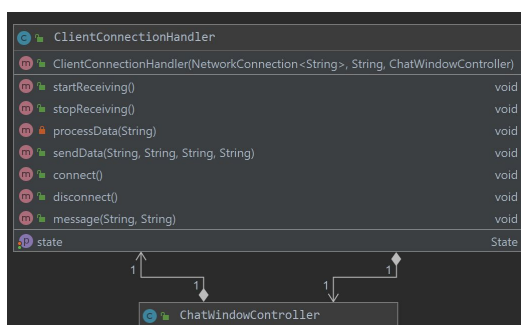
Das Senden und Empfangen von Daten ist asynchron und kann parallel erfolgen. Die Verbindung bleibt offen, bis einer der Peers sich entscheidet, sie mit `NetworkConnection.close()` zu schließen. In diesem Fall werden alle wartenden Methodenaufrufe auf der Gegenseite unterbrochen und eine `EOFException` geworfen. Auf der anderen Seite werden ebenfalls wartende Methodenaufrufe/Threads unterbrochen und eine `SocketException` wird geworfen. Um den Empfang neuer Verbindungsanforderungen auf der Serverseite zu unterbinden, kann der Server `NetworkServer.close()` aufrufen, wodurch alle derzeit geöffneten `NetworkConnection` Objekte geschlossen werden.



NetworkConnection	
socket	Socket
NetworkConnection(Socket)	
send(T)	void
receive()	T
close()	void
equals(Object)	boolean
hashCode()	int
remotePort	int
remoteHost	String
closed	boolean
available	boolean

1.3 Beschreibung der gefundenen funktionalen Fehler

Threadaufteilung



ClientConnectionHandler und ChatWindowController weisen eine zyklische Dependency auf. ClientConnectionHandler schreibt direkt auf ChatWindowController, was auf demselben Thread, dem UI Thread passiert. Dies führt zu einem Locking wobei auf eine Antwort gewartet wird, es jedoch nicht dazu kommt. Die nebenstehende Abbildung verdeutlicht dies.

Statusmeldungen

Für die Ausgabe von Status-Meldungen auf der Konsole werden `System.out.println` bzw. `System.err.println` verwendet.

1.4 Beschreibung der gefundenen strukturellen Probleme

Clean Code

ClientConnectionHandler und ServerConnectionHandler weisen duplizierten Code bei den Datenfeldern und Methoden auf. Zusätzlich ist ungenutzter Code wie in der Klasse NetworkHandler Zeile 207 `isClosed()`, in ChatProtocolException Zeile 9, 13 und ungenutzte Imports wie z.B. in ClientUI Zeile 9, 10 vorhanden. Bezüglich der Koppelung sind alle Methoden in ClientConnectionHandler und ServerConnectionHandler und die Methode `terminate()` in Server public, was eine schlechte Kapselung und eine starken Koppelung ausmacht. Die Methode `processData()` in ClientConnectionHandler und ServerConnectionHandler hat zu viele Zeilen, die mehrere logische Einheiten sind und darum eine tiefe Kohäsion aufzeigt.

MVC-Pattern

Die Controller und teilweise Model Handlungen passieren im selben Thread wie die View Handlungen. Im Multichat wären es die Methoden von der ChatWindowController Klasse, die im selben Thread laufen wie die der ClientConnectionHandler Klasse.

Tests

Es sind keine Tests vorhanden.

Javadoc

Ausser in NetworkHandler fehlen in den allen anderen Klassen Javadoc.

2. Beschreibung ihrer Lösung

2.1 Dokumentation der neuen Struktur

ClientConnectionHandler und ServiceConnectionHandler sollten von einer gemeinsamen Abstract Class erben, damit der duplizierte Code verlagert werden kann.

ClientConnectionHandler und ChatWindowController sollten keine gegenseitige Dependency bilden. Das Controlling sollte separat aufgeführt werden.

MVC

Nach Aufgabenstellung wäre ein MVC-Pattern erwünscht. In ursprünglichen Projekt wurden der Controller und die View zusammengefasst, was am Ende zu einem Programmabsturz führte. Besser wäre eine Aufteilung wie in der nächsten Abbildung dargestellt.

Die Controller Rolle wie in der Klasse ChatWindowController sollte die Änderungen durch die User Aktionen, wie Eingaben (clicks, buttons etc.) nur beobachten und diese Inputs

dem Model weitergeben, welches diese Daten festhält. Der Controller wartet, bis das Model es notifiziert. Daraufhin kann der Controller die Daten in der View abbilden. Zwischen dem Model und View sollte Data Binding passieren. In unserem Projekt wurde dies so umgesetzt, dass der ClientConnectionHandler nichts vom ChatWindowController weiss, womit die zyklische Dependancy aufgebrochen wurde.

Die Kommunikation findet nun nur noch über dünne Schnittstellen statt. Userinputs werden einerseits als ObservableValue vom ClientConnectionHandler gespeichert. Der ChatWindowController subscribed dann auf diese Observables, und wird bei Änderungen dieser Daten notifiziert (nähere Erklärungen sind in der JavaFX Dokumentation).

Gesendete Nachrichten werden in einer LinkedBlockingQueue gespeichert, welche von der Controllerseite gepostet werden, damit diese vom ClientConnectionHandler angenommen werden können, um verarbeitet zu werden. Somit wurden die MVC Aufgaben entkoppelt.

Statusmeldungen

Anstatt System.out.println und System.err.println wird für die Ausgabe von Statusmeldungen auf der Konsole die Java-Logger Funktionalität verwendet. Das ist vorteilhaft, weil jede Meldung mit einem Schweregrad versehen ist. Ausserdem kann man zentral festlegen, ab welchem Grad die Meldungen ausgegeben werden sollen.

Tests

Ausserdem fehlen sämtliche Tests. Vor allem bei UI Projekten wären automatisierte Tests notwendig. Da diese für dieses Projekt nicht gefragt wurden, wurden diese weggelassen.

Threadaufteilung

Die Controller Aufgaben im ChatWindowController laufen im UI Thread, was zum Locking und zum Abstürzen des Programms führt. Um das zu verhindern, sollten diese Prozesse auf separaten Threads ablaufen. In diesem Projekt wurde ein Sender und ein Receiver Thread im ClientConnectionHandler gestartet. Über eine LinkedBlockingQueue postet der ChatWindowController die gesendeten Nachrichten, woraufhin der der Sender Thread diese von der Queue nimmt und verarbeitet. Der Receiverthread speichert alle ankommenden Nachrichten als ObservableValue, wie bereits erläutert. Bei Änderungen, wird der UI Thread benachrichtigt, da er auf diese Observables subscribed.

Im ServerConnectionHandler wurde ein zusätzlicher Thread gestartet, der weitere Clients an den Server verbinden lässt.

