Übung – Multichat

Kontext

In dieser Übung soll ein eigentlich vollständiges, jedoch schlecht umgesetztes, fehlerhaftes und mangelhaft dokumentiertes Programm analysiert, die essenziellen Funktionen und Abläufe dokumentiert und ein Refactoring durchgeführt werden. Ziel ist, die enthaltenen Fehler zu beseitigen und eine gute Klassenstruktur und sauberen, gut dokumentierten Code zu erhalten.

In dieser Übung arbeiten Sie in Zweiergruppen. Pro Klasse ist eine Dreiergruppe möglich.

Einbettung in die Lernziele von PM2

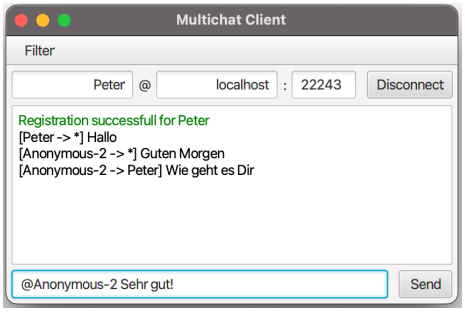
Die Übung trägt zur Erreichung der folgenden Lernziele von PM2 bei:

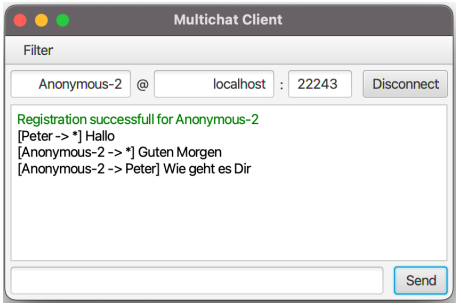
- Die Studierenden können ein bestehendes Programm strukturell und funktionell analysieren, dokumentieren und optimieren.
- Sie arbeiten in einem Team aktiv und zielführend zusammen und übernehmen dabei Verantwortung für die Erarbeitung des gemeinsamen Projektes, wie auch für den Lernfortschritt aller Teammitglieder.

Multichat

Die Anwendung Multichat implementiert einen Chat-Dienst, in welchem mehrere Benutzer einfache Textmeldungen austauschen können. Sie besteht aus einem Serverteil, der als Dienst (ohne GUI, Log-Meldungen in der Konsole) gestartet werden muss und einem JavaFX basierten Client, den mehrere Anwender starten, sich mit dem Server verbinden und Nachrichten austauschen können.

Client 1





Client 2

Server läuft im Terminal:

```
Create server connection
Listening on 0.0.0.0:22243
Server started.
Starting Connection Handler for Anonymous-1
Start receiving packages...
Connected new Client Anonymous-1 with IP:Port <localhost:64484>
Dispatching packet: Packet[sender=Peter, receiver=, type=CONNECT, payload=null]
Register connection as Peter
Registered connection for: Peter
Registration successfull.
Starting Connection Handler for Anonymous-2
Start receiving packages...
Connected new Client Anonymous-2 with IP:Port <localhost:64486>
Dispatching packet: Packet[sender=, receiver=, type=CONNECT, payload=null]
Register connection as Anonymous-2
Registered connection for: Anonymous-2
Registration successfull.
Dispatching packet: Packet[sender=Peter, receiver=*, type=MESSAGE, payload=Hallo]
Send message from Peter to all: Hallo
Sent message from Peter to Anonymous-2: Hallo
Sent message from Peter to Peter: Hallo
Dispatching packet: Packet[sender=Anonymous-2, receiver=*, type=MESSAGE, payload=Guten Morgen]
Send message from Anonymous-2 to all: Guten Morgen
Sent message from Anonymous-2 to Anonymous-2: Guten Morgen
Sent message from Anonymous-2 to Peter: Guten Morgen
Dispatching packet: Packet[sender=Anonymous-2, receiver=Peter, type=MESSAGE, payload=Wie geht es Dir]
Send message from Anonymous-2 to Peter: Wie geht es Dir
Sent message from Anonymous-2 to Peter: Wie geht es Dir
Confirm direct message for Peter back to Anonymous-2: Wie geht es Dir
```

 © ZHAW InIT

Beim Verbinden mit dem Server kann der Benutzer seinen Namen angeben (z.B. Peter). Dieser Name muss eindeutig sein. Falls man keinen Namen angibt, erstellt der Server einen eindeutigen Namen (Anonymous-<Nr>). Textmeldungen, die an den Server geschickt werden, sendet dieser normalerweise an alle verbundenen Clients weiter (broadcast). Wenn jedoch eine Meldung mit einem `@Benutzername` beginnt (z.B. @Peter) wird diese vom Server nur an den entsprechenden Benutzer weitergeleitet (direct).

Die empfangenen Text- und Statusmeldungen werden gesammelt und in einer Textbereich (TextArea) ausgegeben. Ein Filter-Feld ermöglicht diese Meldungen zu filtern und nur diejenigen anzuzeigen, welche den Suchstring enthalten.

Kommunikationsprotokoll

Ein Protokoll bestimmt die Syntax, den Ablauf und die Semantik einer Kommunikation. Es legt fest, wie eine Verbindung aufgebaut und beendet wird, welche Zustände eine Verbindung haben kann, in welcher Form die Daten ausgetauscht werden und was im Falle eines Verbindungsproblems geschehen soll. Die im Protokoll vereinbarten Regeln gelten immer für alle Teilnehmer (Client, Server).

Die Form der ausgetauschten Daten kann von einem String-Objekt (jeder Bereich im String hat eine Bedeutung) bis zu einem strukturierten Daten-Objekt (die Datenfelder sind typisiert, benannt und regeln die Bedeutung) gehen. Der Vorteil von Daten-Objekten ist, dass man einfacher auf die Datenfelder zugreifen kann und durch die Typisierung weniger Fehler passieren können. Daten-Objekte sind oft reine Daten-Container und enthalten keine Logik.

Die mitgelieferte fertige `NetworkHandler`-Klasse ermöglicht das Versenden und Empfangen von Java Daten-Objekten. Diese müssen lediglich das Marker-Interface[1] `java.lang.Serializable` implementieren. Der `NetworkHandler` sorgt dafür, dass die Objekte für die Übertragung serialisiert (d.h. in einen byte-stream codiert) und auf der Empfangsseite wieder deserialisiert (d.h. in ein Objekt zurückgewandelt) werden.

Das Protokoll für Multichat umfasst drei Hauptabläufe (CONNECT, MESSAGE und DISCONNECT), wobei bei CONNECT zwischen anonym und nicht anonym unterschieden wird und bei MESSAGE zwischen Nachricht an alle (broadcast) oder direkt an einen spezifischen Benutzer (direct).

Connect und Disconnect werden vom Server mit einer Bestätigung quittiert (CONFIRM).
Falls ein Problem auftritt, schickt der Server eine Fehlermeldung (ERROR) als Antwort an den Client.

Die Struktur der Datenpakete, die zwischen Client und Server versandt werden, ist immer gleich und besteht aus den folgenden Daten:

- **Sender** – Name des Benutzers, NONE (Leerstring) beim Anmelden anonymer Benutzer oder beim Server.
- **Receiver** – Name des Empfängers, NONE (Leerstring) für den Server, ALL ("*") für alle angemeldeten Benutzer
- **Type** – Art des Pakets (CONNECT, CONFIRM, MESSAGE, ERROR, DISCONNECT)
- **Payload** – Nachricht oder Fehlermeldung; `null` wenn nicht notwendig

[1] Ein Marker-Interface enthält keine Methoden-Spezifikationen und markiert somit nur eine Absicht.

Die Abläufe sind im folgenden Kommunikationsdiagramm (Variante eines Sequenzdiagramms) dokumentiert.

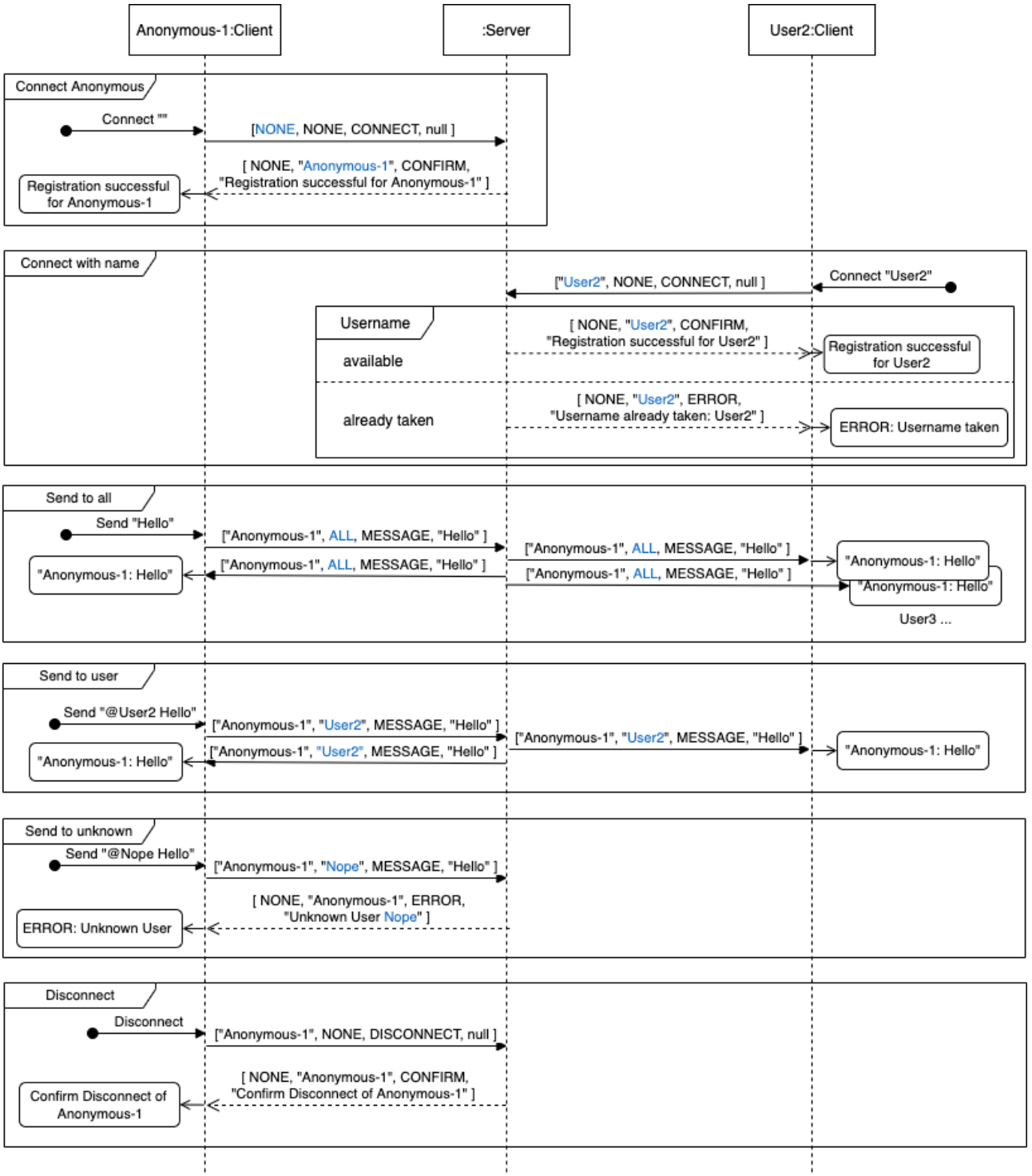


Abbildung: Multichat - Kommunikationsprotokoll

Auftrag

Erstellen Sie ein GitHub-Repository für Ihre Teilgruppe in Ihrer Klassenorganisation (Schema: **`Uebung-hk<n>-<LoginStudent1>-<LoginStudent2>`**) und fügen Sie sich als Collaborator hinzu.

Laden Sie die ZIP-Datei mit dem Ursprungsprojekt herunter und checken Sie den Inhalt in das neu erstellte Repository ein.

1. **Analysieren sie das vorhandene Ursprungsprojekt**
   - Was funktioniert in der Ursprungsversion von Client und Server nicht? Welche funktionalen Fehler sind enthalten?
     Tipp: Der Client muss gleichzeitig Nachrichten Senden und Empfangen und der Server mehrere Clients gleichzeitig bedienen können.
   - Analysieren Sie die strukturellen Probleme: Was haben Server und Client gemeinsam? Was ist unterschiedlich? Ist das Klassenmodell sinnvoll? Wo wäre der Einsatz von Vererbung sinnvoll? Ist MVC richtig umgesetzt? etc.

2. **Überarbeiten Sie die vorhandene Anwendung**, sodass:
   - die geforderte Funktion fehlerfrei erfüllt wird. Das heisst, alle Clients korrekt bedient werden und keine blockierenden Zustände auftreten können.
   - eine saubere Modul-/Klassenstruktur umgesetzt ist. Das heisst: passende Entitäten werden verwendet, Vererbung sinnvoll einsetzen, lose Koppelung zwischen den Komponenten; Einhalten von bekannten Pattern (z.B. MVC), saubere Projektstruktur, …
   - Clean Code eingehalten wird und die Klassen gut dokumentiert sind (Javadoc)

3. **Dokumentieren Sie die Analyse und das Resultat Ihrer Arbeit**.
   - Erfassen Sie mittels GitHub Issues die gefundenen funktionalen Fehler und strukturellen Probleme und dokumentieren Sie ihre Entscheidungen und Fehlerbehebung.
   - Erläutern Sie die Eigenschaften und überarbeitete Struktur Ihrer Lösung in einer kurzen Zusammenfassung und als Klassendiagramm (Top-Level Readme)

Anforderungen

Anwendung:

- Der Server muss beliebig viele Clients bedienen können (soweit es die vorhandenen Systemressourcen zulassen).
- Wenn ein Client sich verbindet, muss er korrekt eingebunden, wenn er sich abmeldet, die entsprechenden Ressourcen wieder freigegeben werden.
- Beim Beenden des Servers werden die verbundenen Clients automatisch getrennt.
- Das Protokoll zwischen Client und Server (Ablauf der Kommunikation) soll beibehalten werden. Das heisst, die Reihenfolge und Art der Nachrichten für einen bestimmten Vorgang (z.B. für die Registrierung, Senden einer Meldung, Abmeldung, …) soll nicht verändert werden. Jedoch sollten Sie die Umsetzung (Datentypen, Ablauf, Kontrolle, …) anpassen und verbessern.
- Für die Umsetzung des Protokolls zwischen Client und Server soll eine Daten-Objekt-Klasse verwendet werden.
- Der Client muss das MVC-Pattern korrekt umsetzen.
- Das Client-GUI muss verhindern, dass der Anwender unzulässige Operationen ausführen kann (z.B. senden von Nachrichten, wenn nicht verbunden).
- Beim Beenden des Clients muss er korrekt beim Server abgemeldet werden (disconnect).

- Zusätzliche Features, die über die geforderte Funktionalität hinausgehen, werden nicht bewertet.
- Der Code aller Klassen muss ausreichend dokumentiert sein (Javadoc).
- Server und Client sollen sinnvolle Statusmeldungen auf der Konsole ausgeben.
- Build-Tooling wird korrekt verwendet. Server und Client müssen je mit dem Befehl `'gradle run'` gestartet werden können.
- Die Projektstruktur muss sinnvoll sein und Dateien am richtigen Ort liegen.
- Ihr Projekt muss die im Clean Code-Handbuch definierten Regeln der Stufe L1, L2 und L3 erfüllen.

Abgrenzungen:

- Es ist nicht erforderlich, dass Ihr Client und Server mit denjenigen anderer Gruppen kommunizieren kann. Sie sind also frei in der Strukturierung der Daten, die übers Netzwerk versendet werden.

Dokumentation der Analyse und des Resultats:

Die Dokumentation besteht aus zwei Teilen:

1. **Dokumentation der gefundenen Probleme und deren Behebung mittels GitHub-Issues**.
   - Erstellen Sie **GitHub Issues** für alle gefundenen funktionale Fehler bzw. strukturellen Probleme.
   - Unterscheiden Sie mithilfe von **Labels** zwischen **funktionalen Fehlern** (Funktion ist fehlerhaft oder nicht wie erwartet) und **strukturellen Problemen** (schlechte Struktur, verletzte Pattern, Clean Code, …).
   - Die Fehler-/Problembeschreibung sollte verständlich und nachvollziehbar sein. Definieren und verwenden Sie ein Grundstruktur (GitHub Issue Templates) für die strukturierte Beschreibung von Fehlern.
   - Bei *funktionalen Fehlern* soll beschrieben werden, wie sie reproduziert werden können, sowie tatsächliches versus erwartetem Verhalten aufgezeigt werden. Gegebenenfalls kann bereits in der Meldung ein Lösungsansatz enthalten sein.
   - Bei *strukturellen Problemen*, sollten sie mehrere Lösungsoptionen aufzeigen und nachvollziehbar begründen, warum eine bestimmte Option gewählt wurde.
   - Die Commits zur Behebung des Problems sollen mit den Issues verknüpft werden.
   - Es sollten jeweils mindestens die wichtigsten 5 essenziellen unabhängigen funktionalen Fehler und 5 strukturelle Fehler dokumentiert werden.
   - Verlinken sie diese Issues im Top-Level README ihres Repositories (gruppiert nach funktionalen Fehlern, strukturellen Problemen)
2. **Beschreibung ihrer Lösung**
   - Erstellen Sie ein Klassendiagramm der überarbeiteten Struktur. Verwenden Sie einen sinnvollen Detaillierungsgrad und eine korrekte UML-Syntax gemäss der Definition, die Sie schon aus dem letzten Semester kennen (Link in der PM2 Kursinformation in Moodle).
   - Kurze Erläuterung Ihrer Lösung (circa eine halbe bis ganze Seite)
     - Warum haben Sie diese Struktur gewählt?
     - Welche speziell erwähnenswerten Eigenschaften oder Lösungsansätze machen Ihre Lösung optimal?
     - Welche sinnvollen Erweiterungsmöglichkeiten könnten ggf. basierend auf ihrer Lösung umgesetzt werden?
   - Die Beschreibung muss im Repository enthalten sein und kann entweder direkt in der **Top-Level README Datei** enthalten sein oder muss von dieser verlinkt werden.

Hinweise zum ausgegebenen Ursprungsprojekt:

Die ZIP-Datei des Ursprungsprojekts finden Sie auf Moodle.

Es ist in drei Teilmodule (server, client, protocol) gegliedert und die Gradle-Konfiguration ist vorhanden, sodass es direkt in die IDE importiert werden kann.

- **`server`**
  Dieses Modul beinhaltet den Code für den Server-Dienst. Hier finden Sie neben der Hauptanwendung `'Server'` auch den `'ServerConnectionHandler'`, welcher das Kommunikations-Protokoll auf Serverseite umsetzt.
- **`client`**
  Dieses Modul beinhaltet die JavaFX-Anwendung für den Client. Neben der Hauptanwendung `'Client'` enthält es auch die GUI-Komponenten für das Chat-Fenster (JavaFX-Application, zugehörige FXML-Datei und Controller-Klasse). Auch hier ist die Kommunikation mit dem Server in die Klasse `'ClientConnectionHandler'` ausgelagert.
- **`protocol`**
  Dieses Modul enthält die Komponenten, welche von den Modulen server und client gemeinsam genutzt werden können. Das beinhaltet zurzeit hauptsächlich die fertige Hilfsklasse `'NetworkHandler'` in welcher die Netzwerk-Kommunikation für Sie abstrahiert ist. Verwenden Sie diese und die darin enthaltenen (inneren) Klassen unverändert für die Kommunikation zwischen Client und Server (Verbindungsaufbau/-abbau und Senden von Nachrichten).
  Mithilfe der statischen Methoden von `NetworkHandler` können Sie einerseits ein `NetworkServer` Objekt erstellen, welches auf einem definierten Netzerk-Port (Standard: 22243) auf Anfragen wartet. Andererseits kann auch der Client eine `NetworkConnection` (Verbindung) zu diesem Server (`Host:Port`; Standard: `127.0.0.1:22243`) öffnen. Mittels des erzeugten Objektes vom Typ `NetworkConnection` können, solange die Verbindung offen ist, strukturierte Pakete in Form von Java-Objekten in beide Richtungen gesendet werden.
  Die Klasse verwendet Generics, d.h. der Datentyp der zu übertragenden Java-Objekte, welche übermittelt werden können, muss bei der Initialisierung angegeben werden.
  Studieren Sie die ausführliche Javadoc Dokumentation der Klasse.
  Die Klassen in der Datei `NetworkHandler.java` (inkl. innere Klassen) soll nur verwendet und nicht verändern werden.

Abgabe

Die Abgabe erfolgt bis zum im Wochenplan definierten Unterrichtstag um 23:59.

- Der erarbeitete Code muss bis zur Frist im Projektrepository auf GitHub hochgeladen sein.
- Die Issues sowie die Beschreibung ihrer Lösung (inkl. Klassendiagramm) sind im GitHub Repository vorhanden und auf der Readme-Seite verlinkt.
- Der letzte Commit im Haupt-Branch (main/master) vor diesem Zeitpunkt wird als Abgabestand verwendet.

Bewertung

In die Bewertung fliessen die folgenden Kriterien ein:

Funktionalität & Umsetzung (50%)

- Das Programm besitzt die geforderte Funktionalität. Insbesondere soll die parallele Verarbeitung funktionieren und dass das GUI nur aktuell sinnvolle Aktionen zulassen.
- Kommunikationsprotokoll zwischen Client & Server wird eingehalten. Clients werden registriert und beim Abmelden die Ressourcen wieder korrekt freigegeben.
- Die in den Anforderungen definierten Regeln zu Clean Code wurden eingehalten. Code ist sauber und umfassend dokumentiert.
- Ausnahmsweise werden Tests nicht bewertet und damit auch nicht verlangt.

Dokumentation & Struktur (50%)

- Die GitHub Fehler-Issues sind strukturiert, vollständig, verständlich und nachvollziehbar.
- Die Entscheidungen sind ersichtlich, Commits sind mit den Issues verlinkt.
- Klassendiagramm ist vorhanden, korrektes UML und haben einen sinnvollen Detaillierungsgrad.
- Die Klassenstruktur ist sinnvoll, die Koppelung ist niedrig, die Aufteilung der Klassen in Module ist sinnvoll. Vererbung ist sinnvoll eingesetzt. Pattern werden eingehalten.

 © ZHAW InIT